#### 第1章

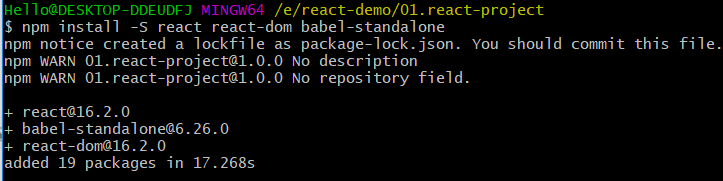
1. React安装

官网是：<https://reactjs.org/>

初始化项目：npm init //cpnm init

1. 安装依赖包

npm install react react-dom babel-standalone -S



注意使用JSX语法，因此需要安装这些依赖包。这三个都要安装才行。

react--核心库

react-dom 提供与DOM相关的功能

babel-standalone 将JSX语法转化为浏览器可以识别的ES5语法。

1. 使用JSX语法时，



注意属性的区别：class要写成className。模板后面一定要有个逗号。遇到<和>会解释成HTML，遇到大括号{和}，解释成JS。

1. 使用数组存储模板，示例如下：

**var *arr*** = [  
 <**div key="1"**>111</**div**>,  
 <**div key="2"**>222</**div**>,  
 <**div key="3"**>333</**div**>  
];  
**ReactDOM**.render(  
 <**div**>{***arr***}</**div**>,  
 **document**.getElementById(**'example'**)  
)

注意：每个实例必须有唯一一个根元素，将数组变量放在大括号中展开。

#### 第2章

1. JSX之函数

定义函数，然后调用，示例：

**function** *action*(item) {**return** item.name + **""** + item.**age**;  
}**const *user*** = {  
 **name**: **'lili'**,  
 **age**: 22  
};

**var *element*** = <**h1**>hello,{*action*(***user***)}</**h1**>;  
**ReactDOM**.render(  
 ***element***,  
 **document**.getElementById(**'example'**)  
);

1. 组件

组件定义方法：通过class继承的方式；自定义，函数。class方式，简单示例和传入属性：

**class** HelloMessage **extends React**.*Component* {  
 render() {  
 **return** <**h1**>hello, {**this**.props.name}</**h1**>  
 }  
}  
**ReactDOM**.render(  
 <**HelloMessage name="lili"**/>,  
 **document**.getElementById(**'example'**)  
);

**class** HelloMessage **extends React**.*Component* {  
 render() {  
 **return** <**h1**>hello, world</**h1**>  
 }  
}  
**ReactDOM**.render(  
 *//<HelloMessage></HelloMessage>,* <**HelloMessage**/>,  
 **document**.getElementById(**'example'**)  
);

自定义函数方式，加不上生命周期钩子的。

注意调用组件时，要关闭组件的两种方式。this表示此组件。通过自定义函数方式定义组件：



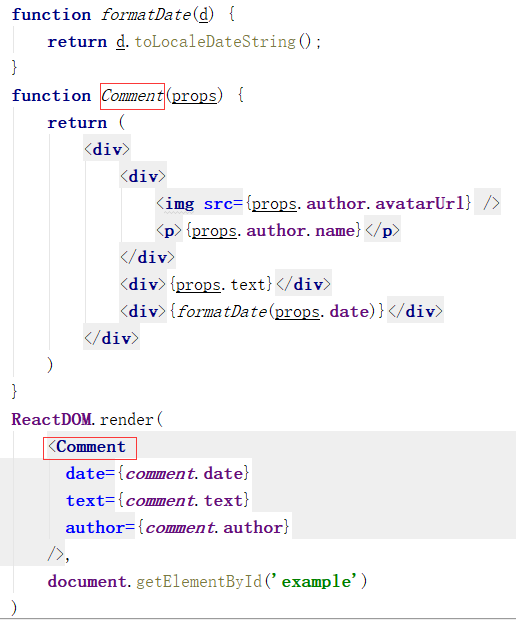
以上是两种方式定义的两个组件的混合使用。或者组件赋值给一个变量，右边示例。

组件传参，通过以下方式传参：

*//自定义组件传参，需要显式传递props(属性对象)，形参名字可以自定义***function** *Welcome*(props) {  
 **return** <**h1**>hello,{props.name},{props.**age**}</**h1**>;  
}  
**ReactDOM**.render(  
 <**Welcome name="mile" age="22"**/>,  
 **document**.getElementById(**'example'**)  
)

注意自定义函数（组件）的形参名字props表示组件的所有属性的对象，形参名字可改变。

组件之间的嵌套：



注意这里面的App组件合成了<Welcome>组件。右边的例子使用了较复杂的传参。

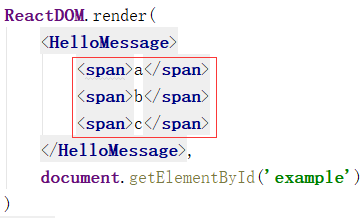
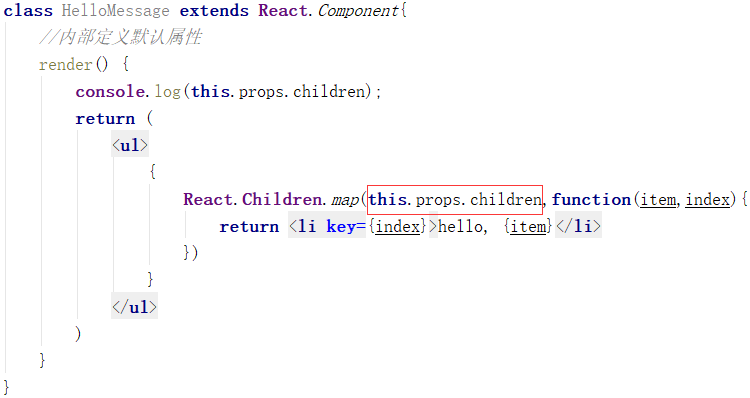
#### 第3章

1. 组件的属性默认值

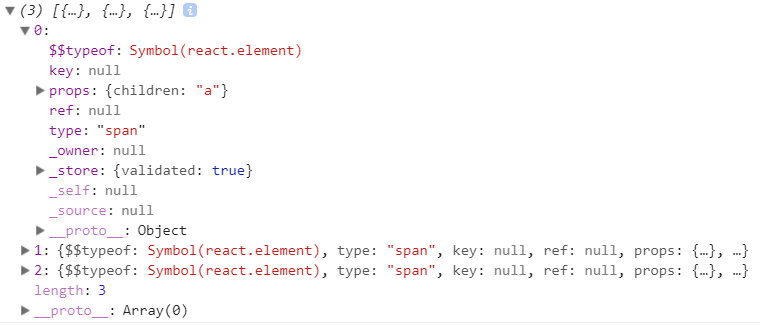


分为组件内部定义和组件外部定义两种方式。

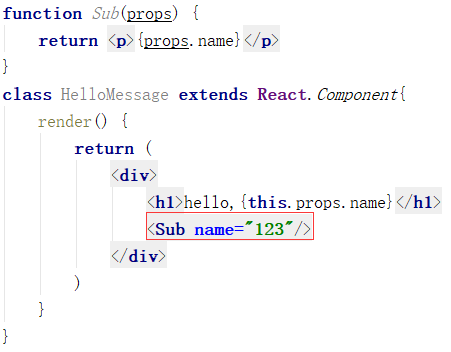
1. 获取组件的子节点，使用this.props.children:



渲染的结果和children值是：

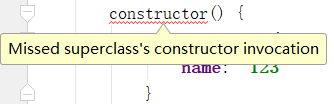
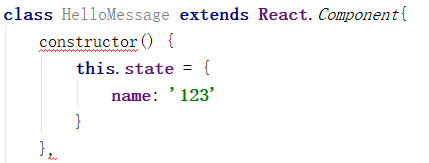


1. class语法的组件嵌套

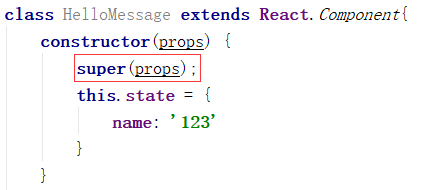


1. 状态state是可以改变的，是私有的，不能通过外部访问来修改，只能在组件内部修改，会导致组件重新render。而props主要用来传参，不直接改变它。

在构造函数中定义state，这样定义会报错：



提示必须调用父类的构造函数，正确定义如下：

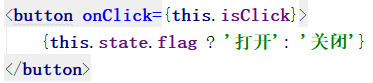
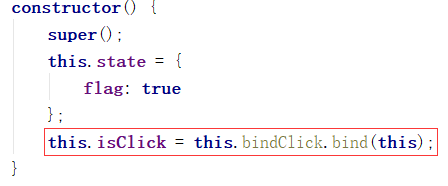


相当于子类调用父类的构造器super。super是父类构造器。注意构造器后面不要跟逗号。默认状态只能写在构造函数中。

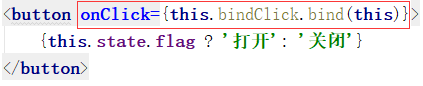
#### 事件绑定，state变化，条件渲染

1. 绑定事件有以下几种方式

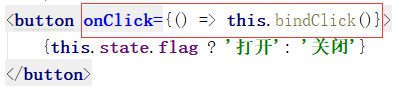
在构造器中声明一个变量，指向事件处理函数



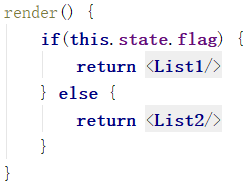
第2种是直接绑定指定的函数：



第3种方法是使用箭头函数：



1. 条件渲染



在进行条件判断时，需要写上全部的this.state.flag，不像vue那样直接写this.flag。react.js写法区分了state和props，需要显式指出，这点和vue不一样。

#### tab，event事件，生命周期

1. 组件的生命周期可分为三个状态：

**Mounting**：已插入真实DOM（挂载）

componentWillMount() 组件将要挂载

componentDidMount() 组件完成挂载

**Updating**：正在被重新渲染

componentWillReceiveProps() 组件将要接收props，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。初始化时不被调用

componentWillUpdate() 组件将要更新

render() 渲染

componentDidUpdate() 组件完成更新立即调用。在初始化时不会调用

**Unmounting**

componentWillUnmount() 在组件从DOM中移除的时候立即被调用

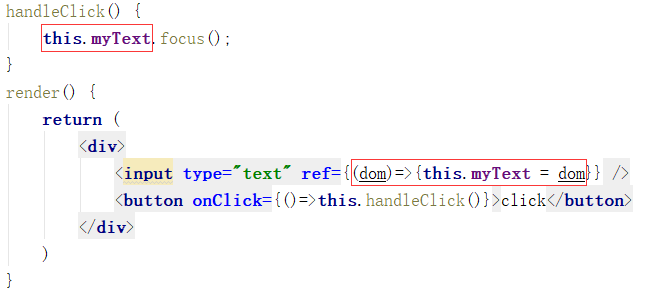
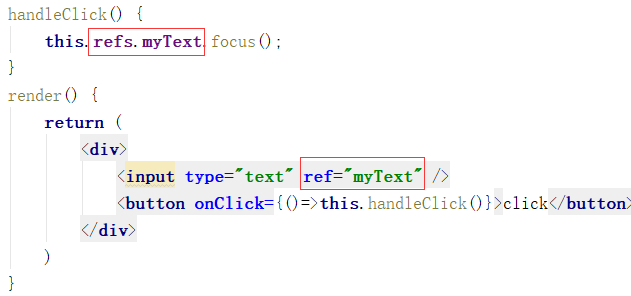
**Error** **Handling**

componentDidCatch() 错误处理

#### ref属性，表单，AJAX

1. ref属性

两种方式，一种是传字符串，用来引用这个元素；另一种是传入一个回调函数：



第二种方式，传入一个回调函数，参数表示这个元素的DOM对象，然后存储在this之中。

1. 表单（text/radio/checkbox/select）

约束性和非约束性组件（受限/非受限）

约束性组件由react管理了它的value，由状态来进行管理：

<**input type="text" value=**{**this**.state.value} **onChange=**{()=>**this**.handleChange()}/>

当内容发生变化时，调用onChange方法。

非约束性组件，value仍由原生的DOM来进行管理，不受react管理：

<**input type="text" defaultValue="123"** />

设置了默认值defaultValue，相当于原生value属性，value的值是用户输入的值。

1. AJAX

#### 校验，组件传参

1. 校验
2. 组件传参

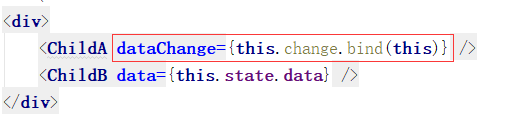
2.1 父组件-子组件

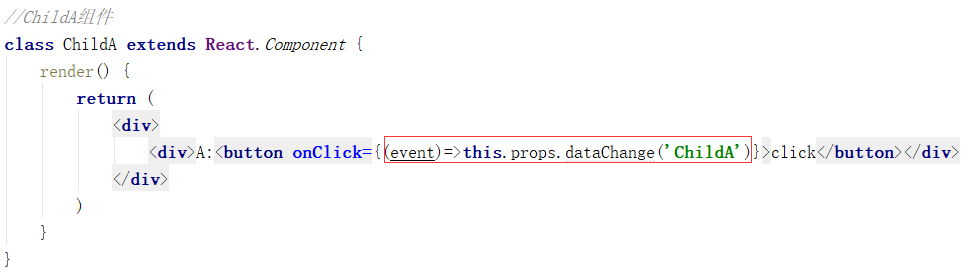
2.2 父组件-孙组件

2.3 子组件-父组件

2.4 兄弟组件

通过父组件的方法作为媒介，传给子组件的是一个方法。



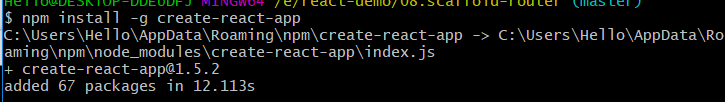


在子组件中调用的属性是一个方法dataChange，调用时传递参数。

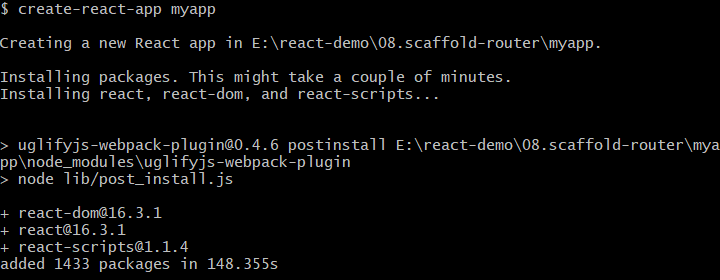
#### 第8章 react脚手架、路由

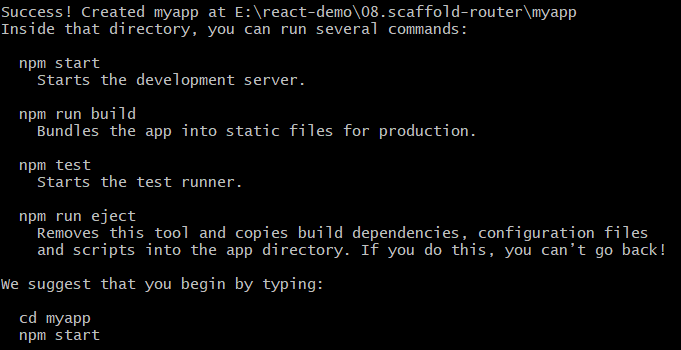
1.脚手架使用create-react-app这个npm包。

npm install -g create-react-app

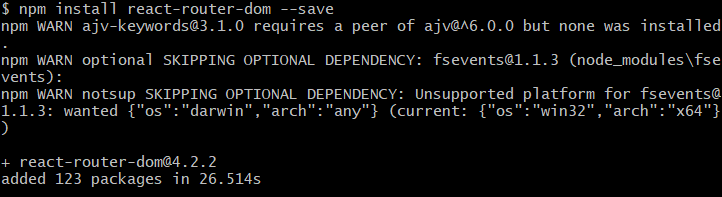


create-react-app myapp



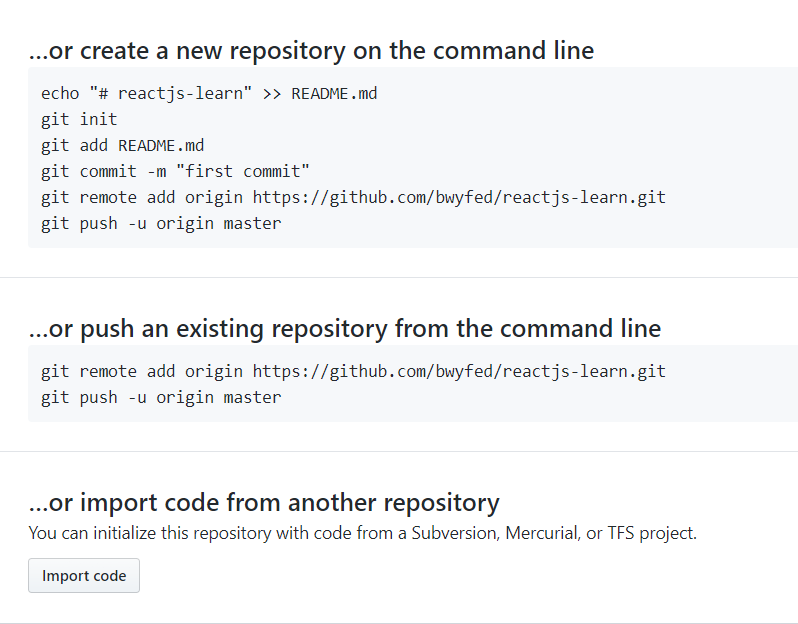


路由功能，需要安装react-router-dom：

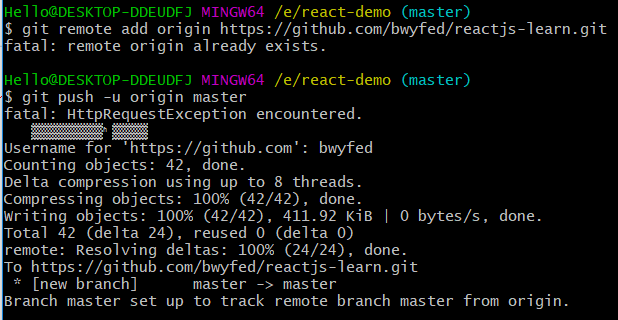


提交到github上：

github地址是：<https://github.com/bwyfed/reactjs-learn>



我在本地已经执行了git init初始化了git项目。因此使用第二种方法：

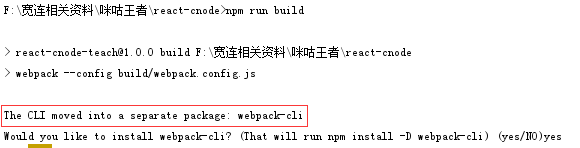


也就是这两条命令：

git remote add origin https://github.com/bwyfed/reactjs-learn.git

git push -u origin master

Webpack命令行现在已经是单独的命令行了：



因此，升级webpack-cli，从



升级为

#### Webapp的工程架构

为什么会有服务端渲染？

单页应用存在的问题：SEO不友好，首次请求等待时间较长，体验不好。

React中如何使用服务端渲染？

react-dom是React专门为web端开发的渲染工具。我们可以在客户端使用react-dom的render方法渲染组件，而在服务端，react-dom/server提供我们将react组件渲染成HTML的方法。

项目开发时常用的配置

常用配置

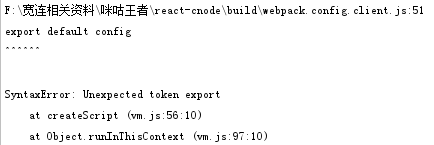
webpack-dev-server

Hot module replacement

注意在webpack配置文件中，要导出配置的话，不要使用export default:



启动时会出现报错：



应该使用module.exports = config才可以。

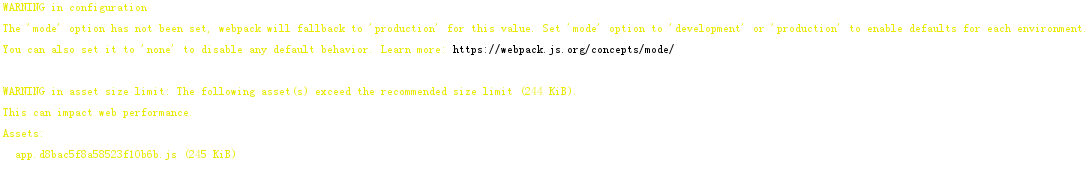
在进行如下设置：



启动脚本npm run dev:client

"dev:client": "cross-env NODE\_EVN=development webpack-dev-server --config build/webpack.config.client.js"

发现报错：



WARNING in configuration

The 'mode' option has not been set, webpack will fallback to 'production' for this value. Set 'mode' option to 'development' or 'production' to enable defaults for each environment.

You can also set it to 'none' to disable any default behavior. Learn more: <https://webpack.js.org/concepts/mode/>

后来发现即使设置了mode:’none’，还是启动不了。经过研究发现，在配置文件中：

**const** isDev = process.env.**NODE\_ENV** === **'development'**

这个isDev的值是false：



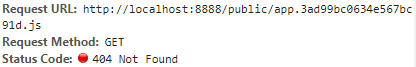
原来，是在npm脚本时输入错误了，导致无法进入了if分支：



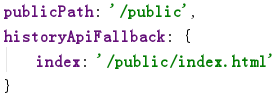
手误输错。

注意：这里要保留mode:’none’的配置。这个和视频里面的不一样。

启动之后，发现首页的index.html是可以正常访问的，但是因为js前面加了个前缀/public，访问不了：

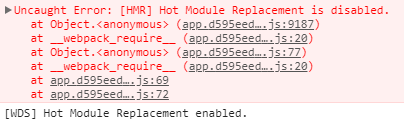


那么如何解决这个问题呢？



但是这里发现仍然是404 Not Found，如上所示。原因是首先检测硬盘上有没有这个目录，现在硬盘上的没有对应的js编译版本，所以会找不到，因此要将dist目录删掉。

删除后，发现js能正常访问了，但是js出现报错：



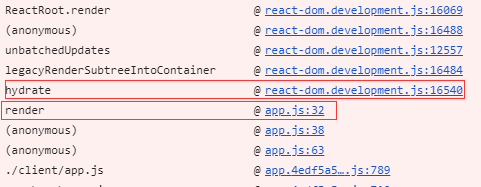
因为设置了hot:true，但是react并没有配置一些相关的模块，因此这里给hot:true去掉。重启后，发现有js报错：



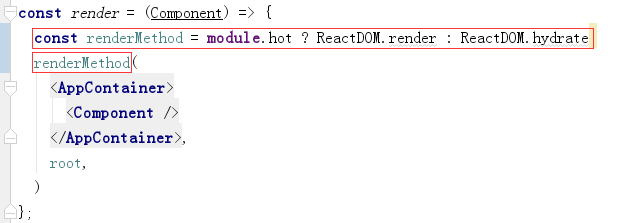
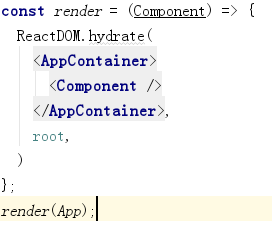
warning.js:33 Warning: Expected server HTML to contain a matching <div> in <div>.

这个报错的原因，经查报错的栈结构是：





这样看报错应该是在render函数在调用hydrate时报错的，现在代码是：

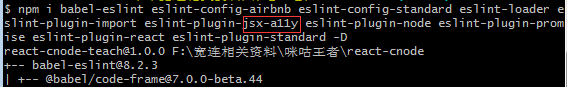


在网上搜到这样一篇帖子：<https://blog.csdn.net/ktutu/article/details/79055999>

这样修改后，错误就不再出现了。

Eslint的安装包：

一般安装eslint-config-airbnb这个包，它有很多依赖：



注意圈红的包，是eslint-plugin-jsx-a11y。

发现很多关于换行的错误：



如果想忽略这种错误，eslint是不行的，使用.editorconfig配置文件

root = true

[\*]

charset = utf-8

indent\_style = space

indent\_size = 2

end\_of\_line = lf

insert\_final\_newline = true

trim\_trailing\_whitespace = true

配置后，对App.jsx进行修改，保存后就可以看到会在文件结尾自动加新的一行。然后就会发现App.jsx的关于换行的错误就消失了。看到如下错误：



错误显示了信息和规则。这种错误原因是：



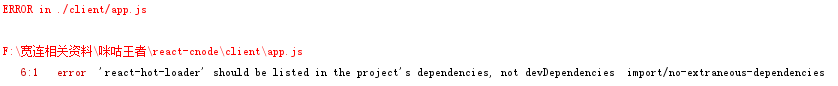
这个组件只有一个简单的render方法，没有其他定义，因此这个组件可以做如下修改：

**export default** () => {  
 **return** (  
 <**div**>This is app 123</**div**>  
 )  
}

这就变成一个纯的方法调用，不再是类的继承方式，会少了很多的初始化工作，少了很多变量的声明，节省空间。

为了使用class方式定义一个react组件，可以在class定义时加上组件生命周期方法，如componentDidMount()

在app.js中有以下错误：



因为在安装react-hot-loader是作为devDependencies去安装的，但是在app.js中又引用了它：



使用eslint的特殊注释：



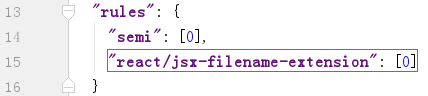


这条错误表示使用箭头函数时一定要用()包含形参：

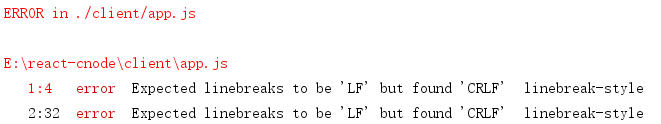




也就是说不能在js文件中引用jsx文件，这条规则没有必要的。做如下配置：

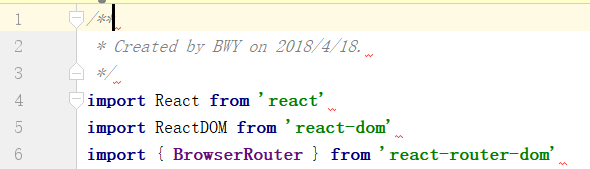


在另一台电脑从github上下载项目，会导致换行符的转换，导致eslint的检查错误：



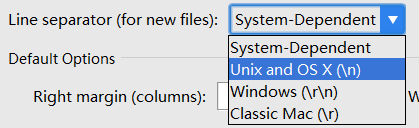
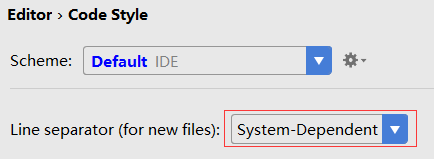
Expected linebreaks to be 'LF' but found 'CRLF' linebreak-style

在app.js中看的，每个换行的地方都有提示错误：

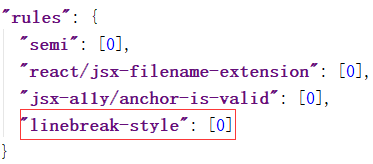


虽然在.editorconfig配置文件中配置了end\_of\_line = lf，但是还是不起作用，在网上搜索这样一篇文章：<https://blog.csdn.net/qq_22978533/article/details/78145935>

在WS中配置，File-->Settings->Editor->Code Style->Line separator(for new lines)，当前是：

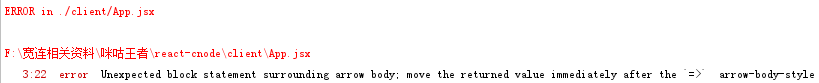


原来当前的是System-Dependent，改成Unix and OS X。后来发现还是不行。后来没办法，只能加上规则，不检测这个规则：



这样才能使这个报错消失。

再看App.jsx中的一条错误：



只有return内容时，不需要使用{}。优化如下：



app.js中有的错误：

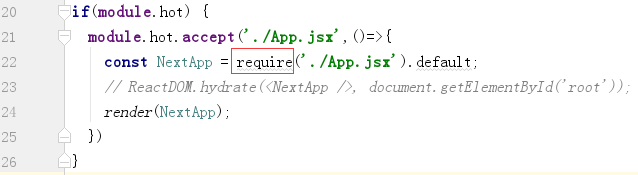


这是因为对象定义的最后一个属性，缺少逗号。修改如下：



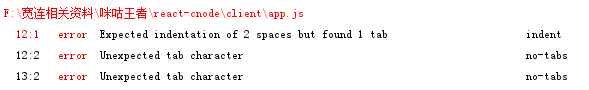


这个错误是因为在这里，没有顶层使用require：



因为这是在调试的时候使用的代码，因此不希望eslint来检查这一行，所以仍然用注释eslint-disable-line。

但是仍然发现两个源文件有很多关于tab和空格的错误：



这里首先将1个tab转换为2个space。修改方式是：

<https://blog.csdn.net/niexia_/article/details/78280569>

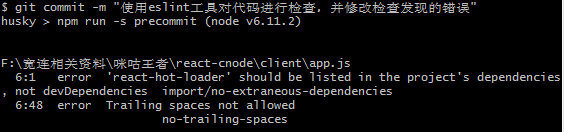
然后在需要修改的地方，逐一做换行修改。还要注意在组件结束之前，要添加个空格/>。

If语句后面要加空格，函数参数之间加空格，箭头函数=>前后都加空格，

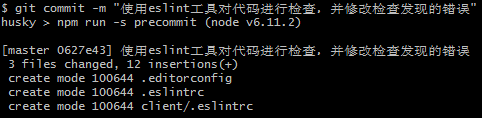
要让git在commit的时候去执行代码检查。要安装husky这个工具：npm i husky -D，然后在packagejson中增加git的commit的钩子，然后加这两个npm脚本：



然后在提交时可以看见：



这是去掉了eslint注释所致。修改错误之后提交可见：

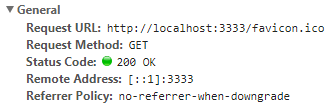


工程项目优化：

新建webpack.base.js，用于webpack配置的公共提取部分。

安装webpack-merge：npm i webpack-merge -D

发现在访问服务端时，会请求favicon.ico，但返回的是服务端渲染的代码：



这里要在服务端安装serve-favicon包，用于处理favicon.ico: npm i serve-favicon -S

在server.js中，做如下修改

**const** favicon = require(**'serve-favicon'**);

app.use(favicon(path.**join**(\_\_dirname,**'../favicon.ico'**)))

然后就可以看到可以请求到网页图标了。

使用nodemon进行服务端自动重启。安装npm i nodemon -D

在根目录下创建一个其配置文件nodemon.json:

{

**"restartable"**: **"rs"**,

"ignore": [

".git",

"node\_modules/\*\*/node\_modules",

".eslintrc",

"client",

"build"

],

"env": {

"NODE\_ENV": "development"

},

"verbose": true,

"ext": "js"

}

注意：以前使用的cross-env的方式来传递环境变量NODE\_ENV，现在通过nodemon来传递时，配置”env”选项即可，这里的npm run dev:server的脚本也要改了：

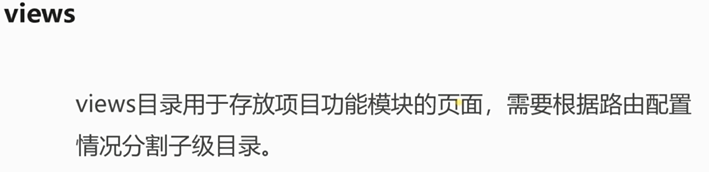


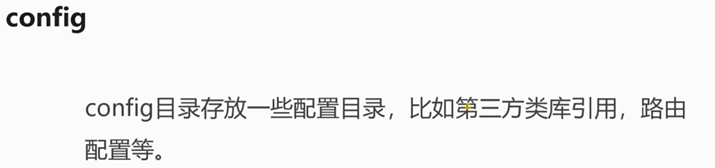
还要注意一定要配置”restable”选项，否则这个配置文件不起作用。

然后再运行npm run dev:server，就可以看到其启动了。

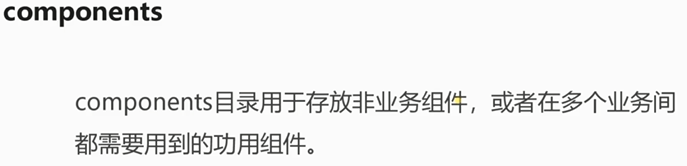
#### Webapp的项目架构

项目基本目录结构：

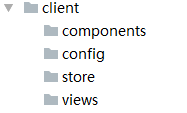








在client目录下，新建4个目录：



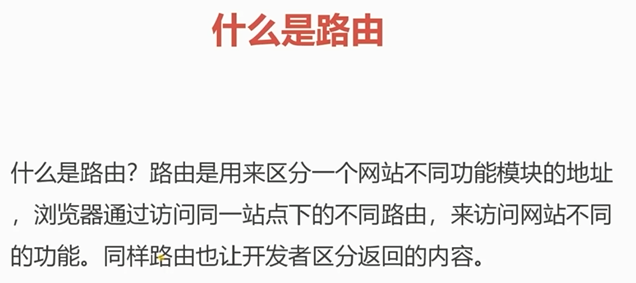
将App.jsx移入到views/目录下后，需要在app.js的3个引用的地方，都要修改其路径。然后前端启动时，出现报错：

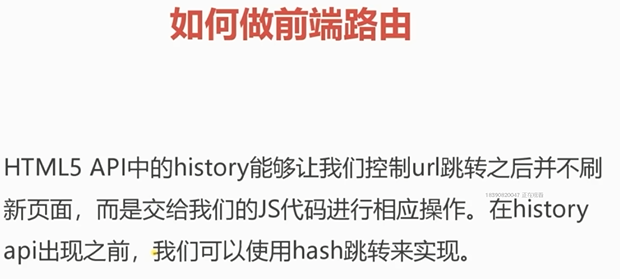


原因是在引用模块时，添加了后缀名：



路由配置







启动npm run dev:client时报错：



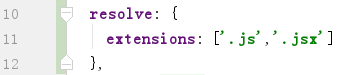
Unexpected use of file extension "jsx" for "./views/App.jsx" import/extensions

在import某个module时不希望写后缀名。然后把app.js中引用App.jsx的地方去掉后缀名.jsx。再次启动时，又发现报错：



Module not found: Error: Can't resolve './views/App'

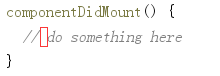
这是因为webpack默认只认识.js文件的，它会找App.js文件的，但当前只有App.jsx，因此会报找不到模块的错误。解决问题的方法是在webpack.base.js中加入：



然后重新启动npm run dev:client，eslint报错：



经研究发现，原来是在注释//之后需要一个空格之类的白字符：



报错：



这是因为第4行后面紧接着下一行export:

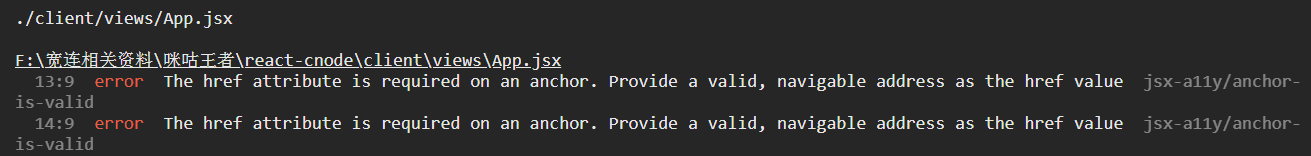


后面加个空行就行了。

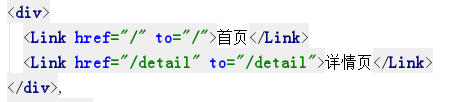
在App.jsx中使用<Link>组件：



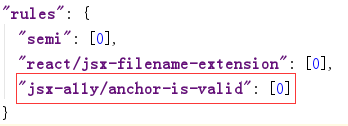
会出现报错：



使用Link标签一定要使用href属性。解决方式是，要么加上href属性：

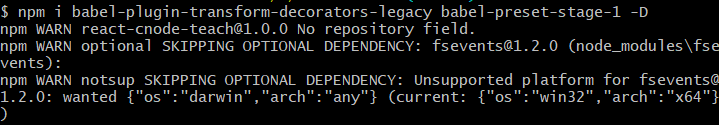


注意这里要配置href为一个路径，不能写上#。要么就在.eslint中屏蔽掉这条规则：

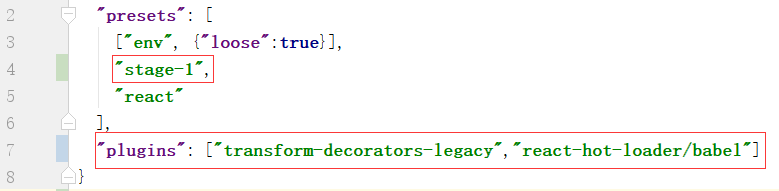


使用Mobx

安装npm i babel-plugin-transform-decorators-legacy babel-preset-stage-1 -D



然后在.babelrc中配置：



安装mobx和mobx-react: npm i mobx mobx-react -S

Cnode API代理实现

安装在node中使用的包：npm i body-parser express-session query-string -S