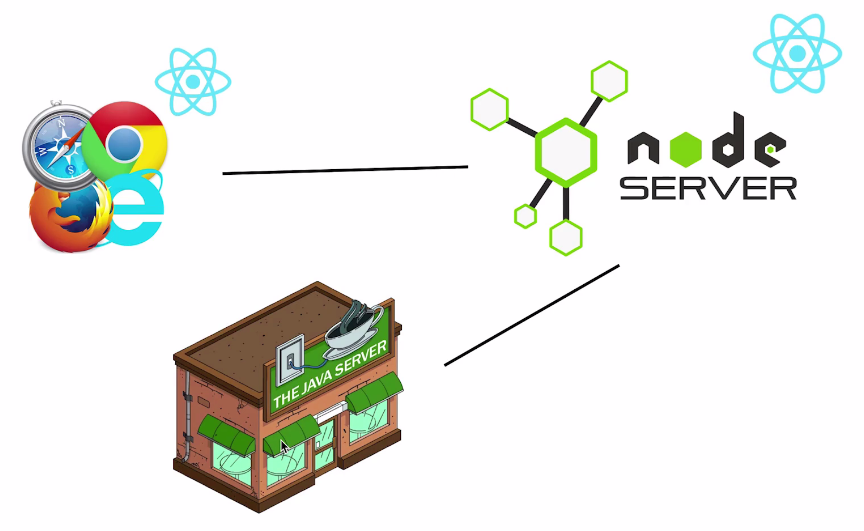
**React-ssr学习笔记**

## 项目结构：

### 先科普一下目前网站比较普遍的架构

**Node做一个中间层，数据请求放在java等语言上**



**Packjson中的script**

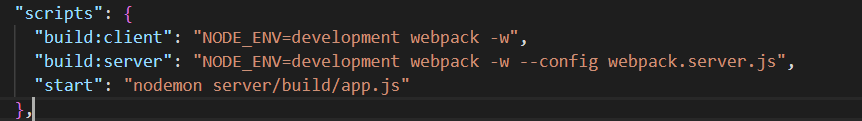
**基于webpack和nodemon实现代码自动打包和服务重启**

1. 利用webpack –watch 监听，代码更改时候重新打包。
2. 利用nodemon监听文件，文件更改时候，执行命令（即重启服务器）
3. 利用npm-run-all 并行执行多个指令

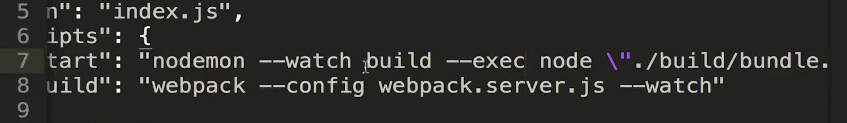
最终实现：代码更新时，能自动打包，自动重启服务。Ps：留心要先生成一次bundle，确保第一次执行时候有该文件。

**步骤一：打webpack打包client端和server端文件**

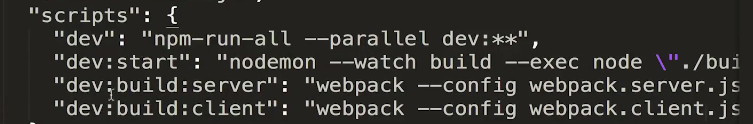
不打包的话，不能识别jsx等一些语法



**步骤二：nodemon自动重启服务**



**步骤二：npm-run-all并行执行多个命令**



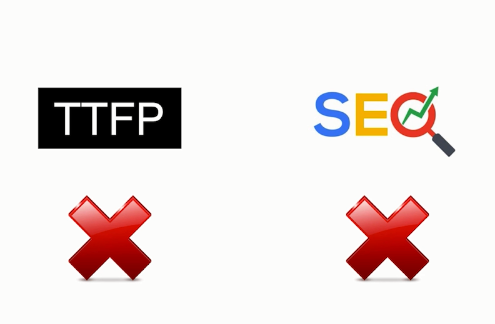
## 涉及的内容：



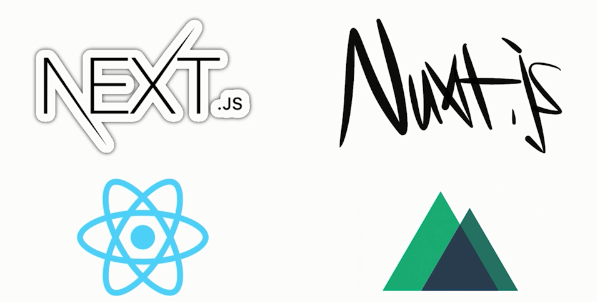


## Spa缺点

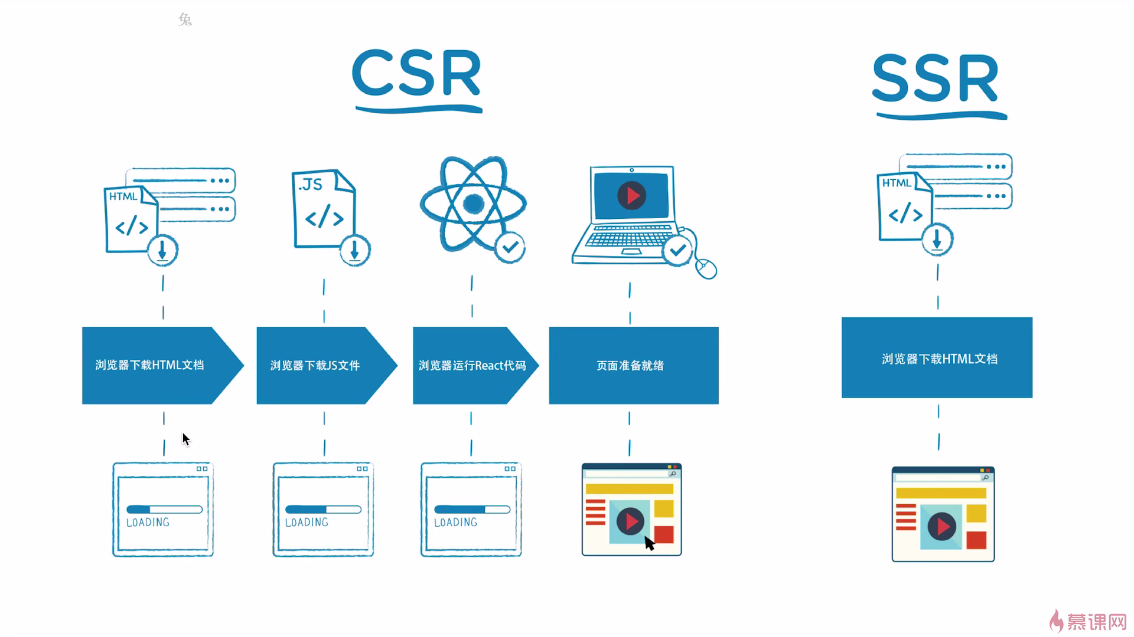
**首屏渲染ttfp慢，不利于seo**



## 常见的两个ssr框架



## SSR与CSR对比



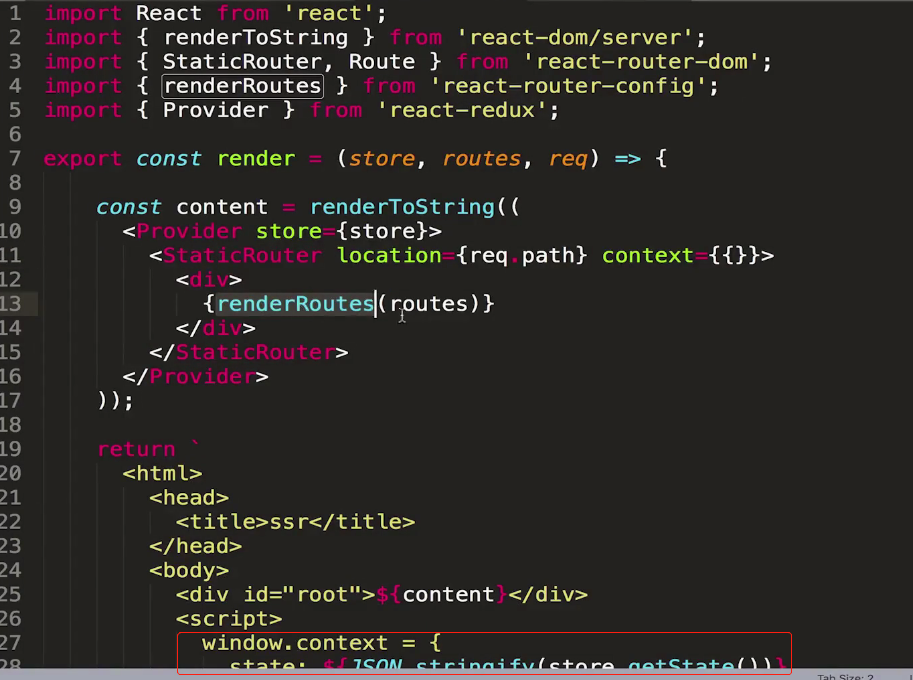
### CSR(SPA)步骤

CSR或者SPAServer返回一个很简单的html了，带有bundle.js。 bundle是webpack基于react编写的代码打包过后的文件，浏览器执行该文件，并进行路由匹配渲染出页面，用户以后进行的路由切换（通过 <Link to=”/path”>）切换页面、路由时候，都通过bunddle.js进行路由匹配和页面切换、渲染。



### SSR步骤

服务端收到请求，进行路由匹配，根据匹配的路由，渲染出html，并把html返回给浏览器，从而渲染出页面。

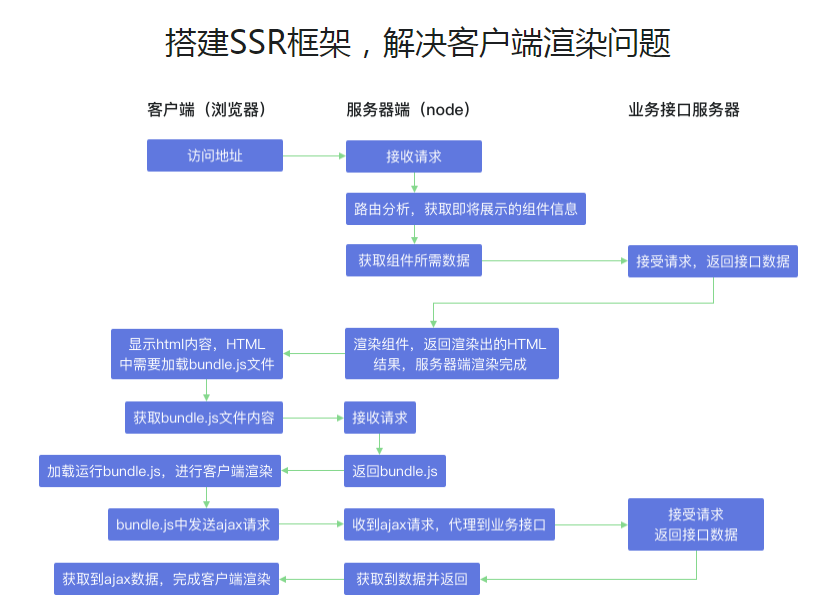


## 同构与SSR

**同构：**一套代码，在客户端执行也在服务端执行，如react写代码。在服务端通过路由匹配渲染出html，在客户端，react的代码也通过webpack打包的bundle又执行了一次。

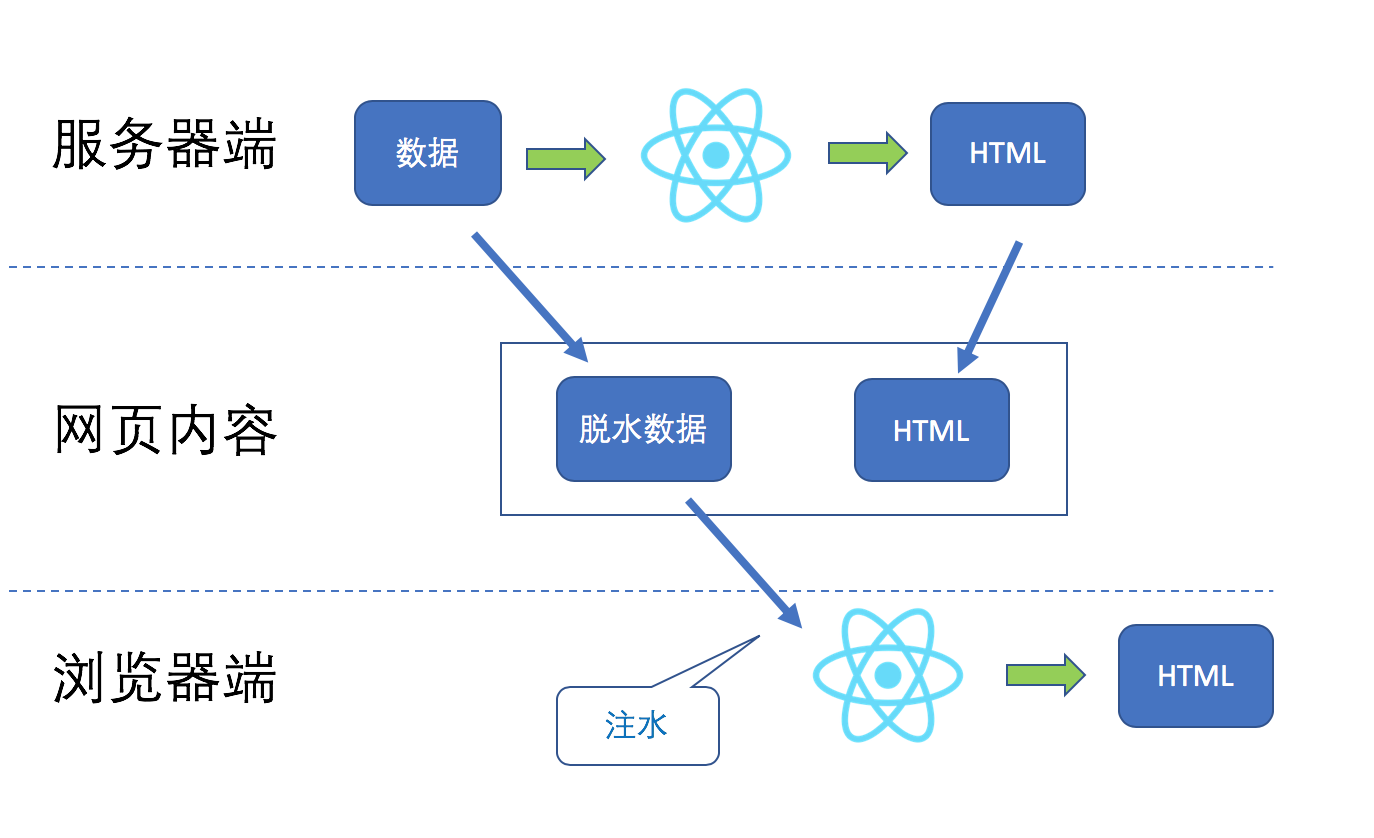
（关于为啥在客户端为啥也需要重新执行一次呢？涉及到脱水、注水的知识，后面会说到，然后在服务端是没办法注入onClick这样的组件事件，react中 render后的生命周期在服务端是不执行的，所以在客户端执行了一次bundle,重新渲染了一次，达到动态交互效果）

**SSR框架：**用户第一次请求时，服务端渲染好html，返回浏览器，并返回buddle.js,让这个bundlewen文件在浏览器也执行，用户再次进行页面请求时，不在发请求到服务端，只在浏览器端通过bundle.js进行路由匹配，也就是spa套路。



## 脱水注水

这也是ReactDom.hydrate接口的名字,前面说过同构，也就是代码会在client和server各自执行一次，如果涉及到数据请求，那么是否是服务的请求一次，客户端请求一次呢？no，在服务端执行获得的数据（数据：水），会脱水，然后也随着html返回客户端，一般可以通过挂在<script>window<script>上。然后客户端去判断如果已经拥有了就不在执行，在后面的路由切换中在通过componentDidmounted的生命周期中去重新获取数据，实际上next框架是通过组件的静态函数static async getInitialProps(){}中完成脱水注水，该函数执行一次，要么第一次渲染时候在服务端执行，要么后面的spa中在客户端执行一次。



## 服务端数据的保存和组件通讯

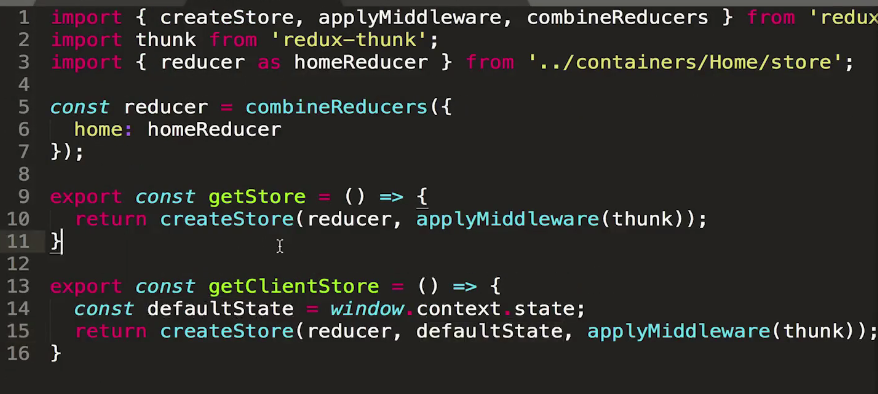
主要是基于redux 或者<.staticRouter context={{}}> 中的staticContext

基于redux的注意事项：

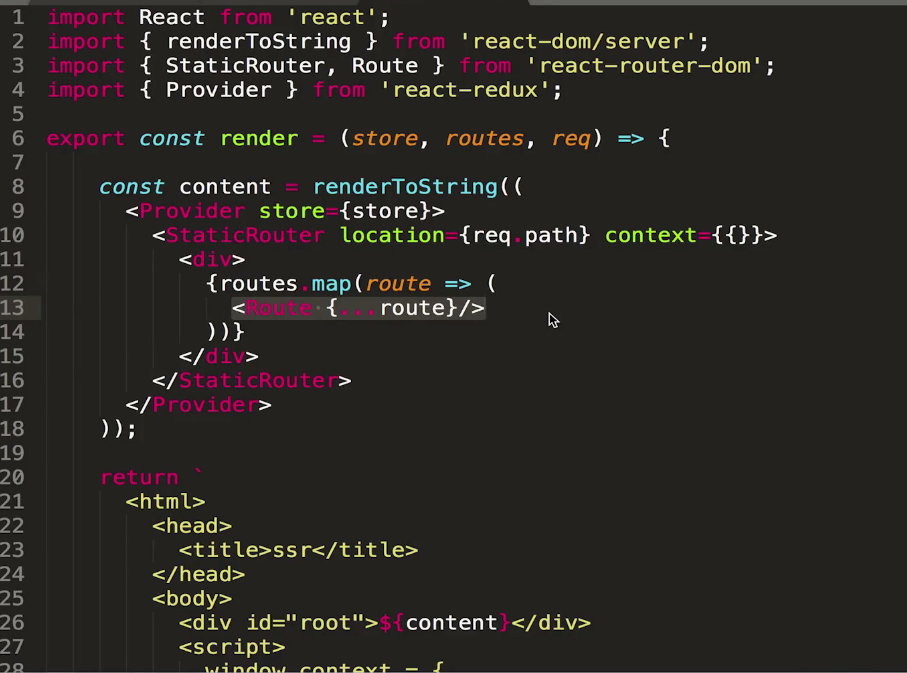


**构建store：**

为了实现脱水注水以及满足一开始说的架构可能涉及到的代理转发等 客服端和服务端对应不同的store,其中defautState就是来自服务端的“水”。



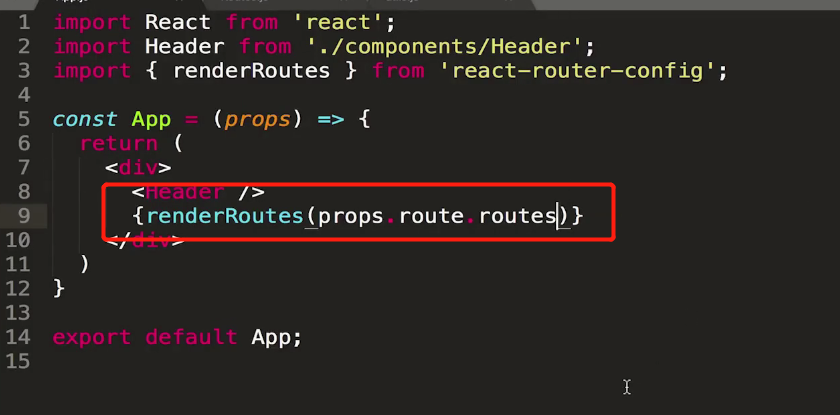
### 单级路由：

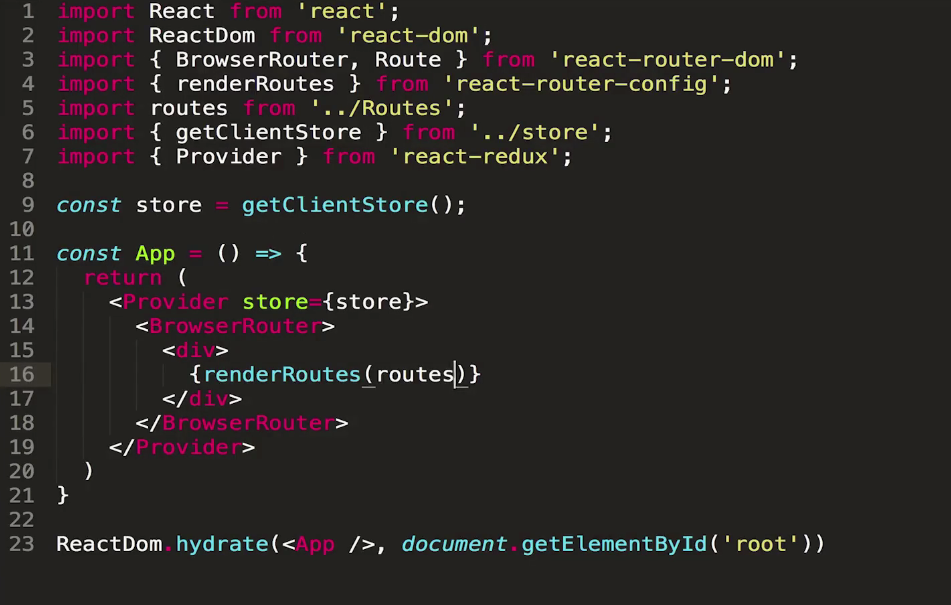


### 二级（多级）路由时候：



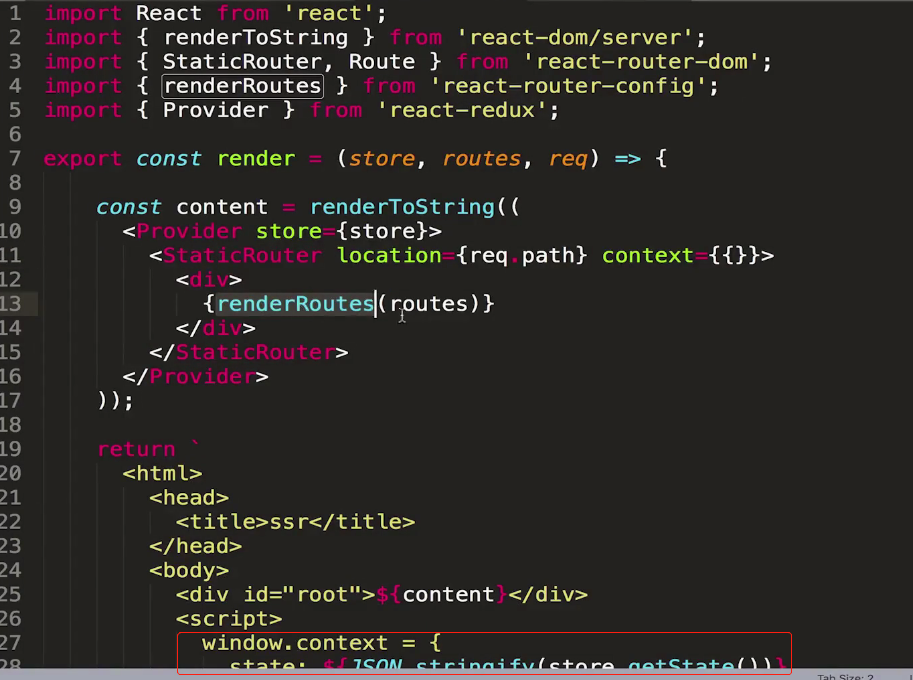
### 客户端多级路由：

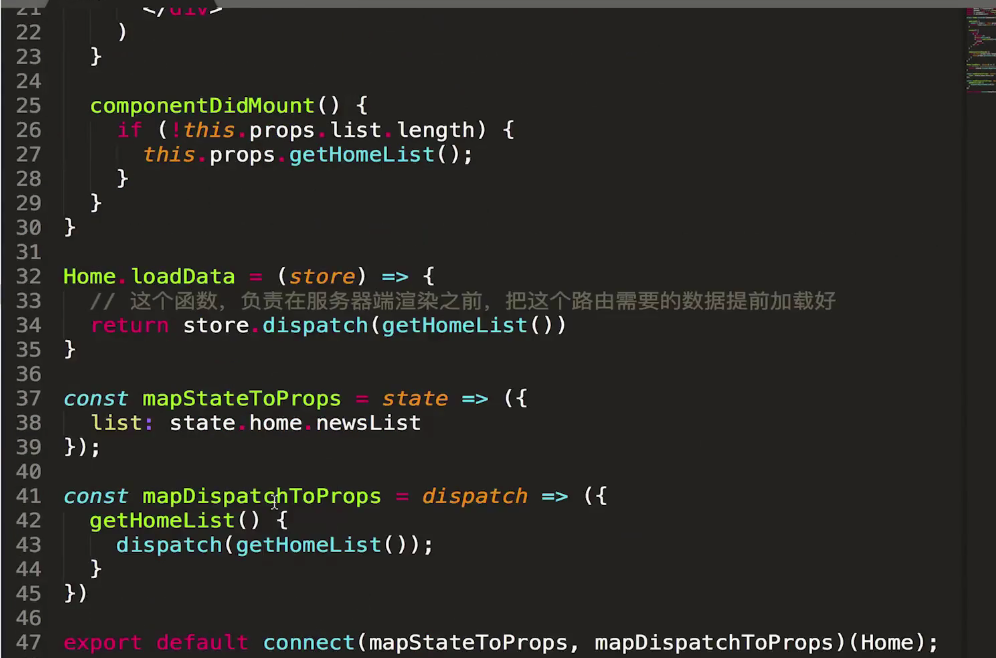




## 服务端渲染：

通过组件的静态方法 loadData发请求



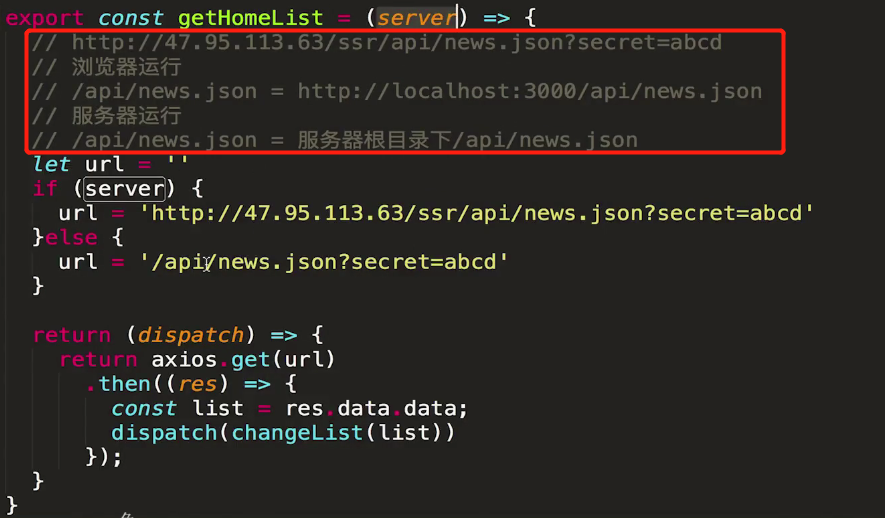


（ps：如果connect没有把静态方法放入connect新生成的组件中，可以用下面的方法：

）

**ActionCreator:**

**数据请求：基于redux与redux-thunk，在服务端和客户端（非首页）都会发请求，客户端和服务端的相对地址不一样，所以请求的url也不一样，这里的请求浏览器请求node，node利用express的一个中间件做了转发。**





### 为了更通用，抽象出两个axios实例instance:

**客服端和服务端对应不同的axios实例（baseUrl），服务端做了一次转发**



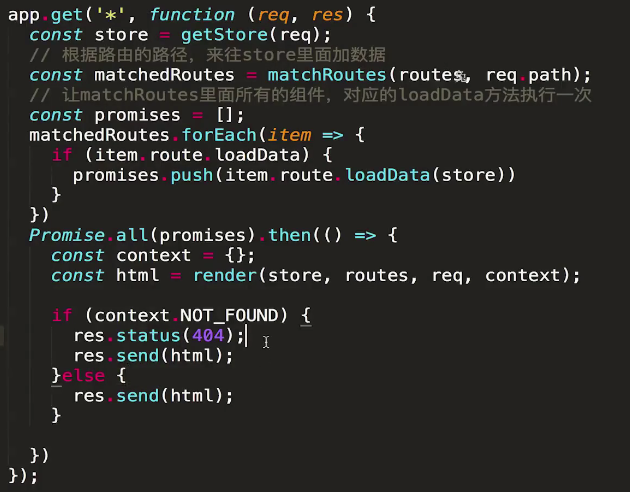
### 基于context实现 404页面：

在遇到404，也就是没有匹配的路由时候，注意要在routes中配置这个component,,path不用写

**错误的页面：必须是render前的生命周期确保在服务端能执行**



在render这个函数中 会在context添加一个NOT\_FOUND

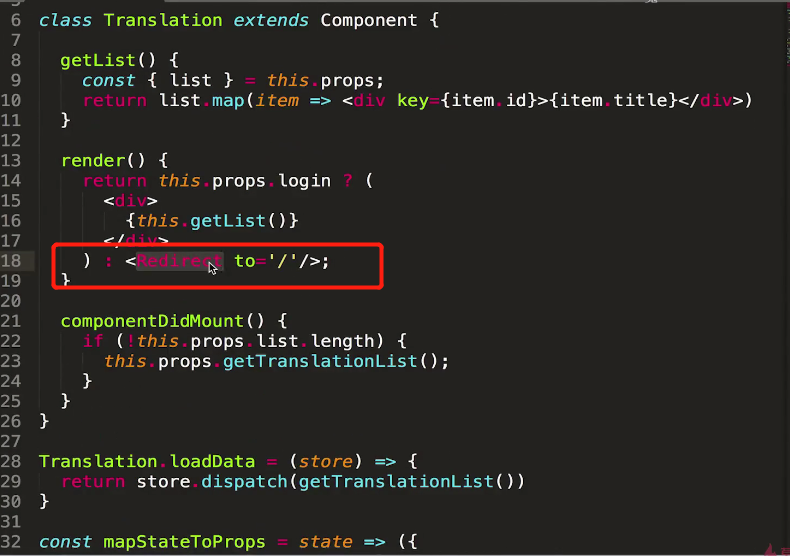


### 基于react-router-config的renderRoutes重定向

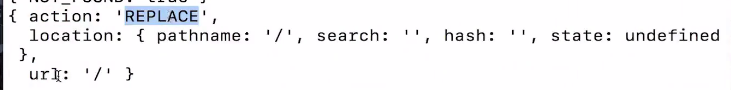
**<Redirect to>在服务端不会重定向，但react-router-config这个模块的renderRoutes在遇到<Redirect to>会在staticContext注入如下信息**

(配合StasticRouter中的context)，当识别<Redirect to=””>会为context注入如下信息：

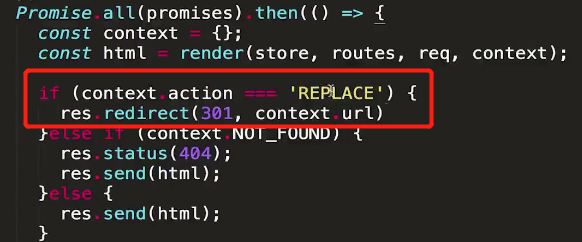
1、识别redirect



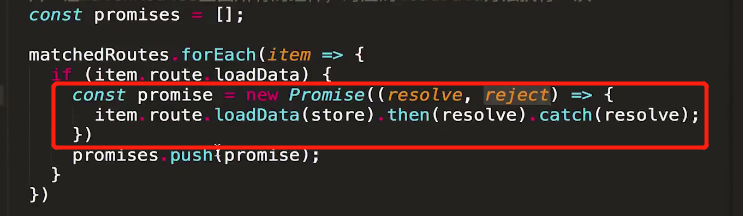
2、向context注入信息注入



识别REPLACE进行服务端重定向



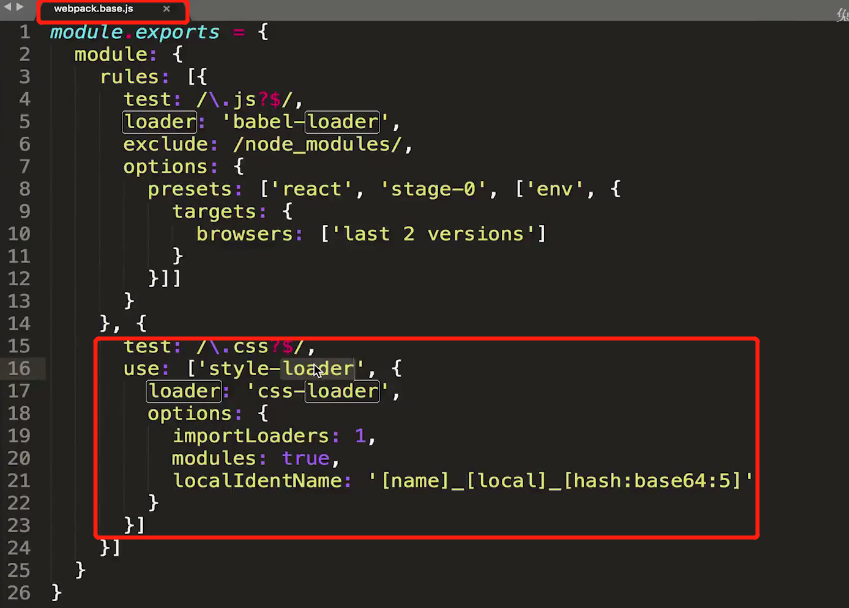
### 服务端数据请求错误，promise的容错处理：



### 处理SSR框架中的CSS样式：

利用webpack向html中加入className，

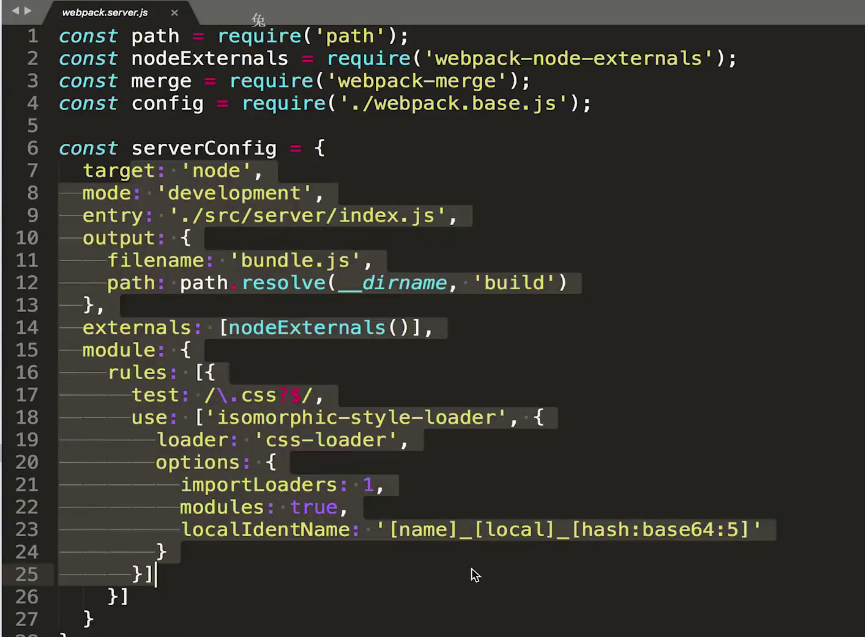
在客户端中bundle会把css注入到document上，所以如下配置在客户端是没问题的



### 但是在服务端

Style-loader会往window挂在样式，但在服务端是压根没有，所以在服务端的webpack利用下面的loader替换style-loader.。服务端和客户端在css是不同的，所以不在统一放在webpack-base.js，注意上面的modules:true,实现css模块化



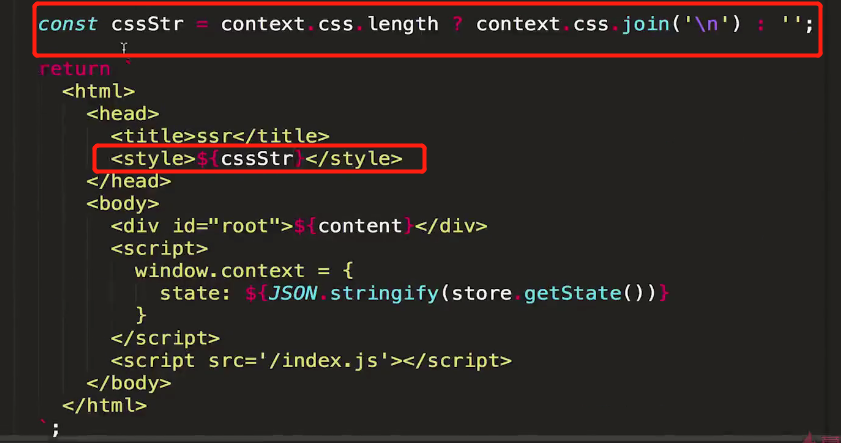


**操作步骤:**

**1、利用style.\_getCss获取样式，利用StaticContext(非路由直接组件需要利用props传递给子组件)存储css**

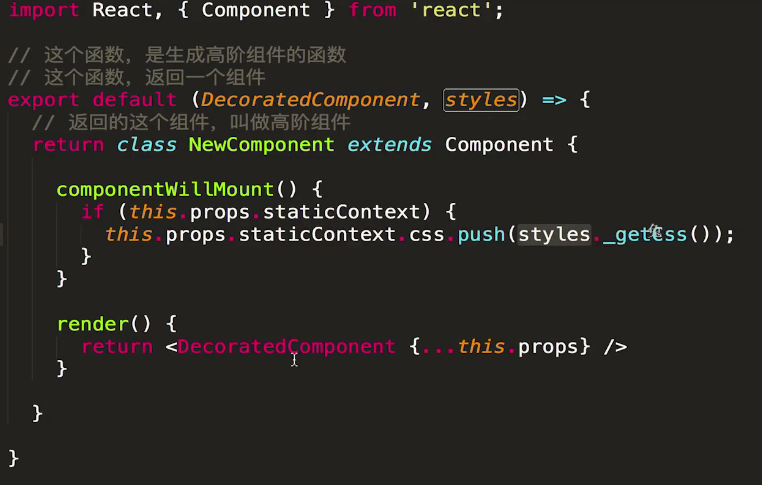


**2、注入到html中**



**利用HOC封装：**

**每一个组件都有这么一个获取并存入css样式的操作，为此把通用的功能抽象出一个HOC**



##### SEO技巧的融入



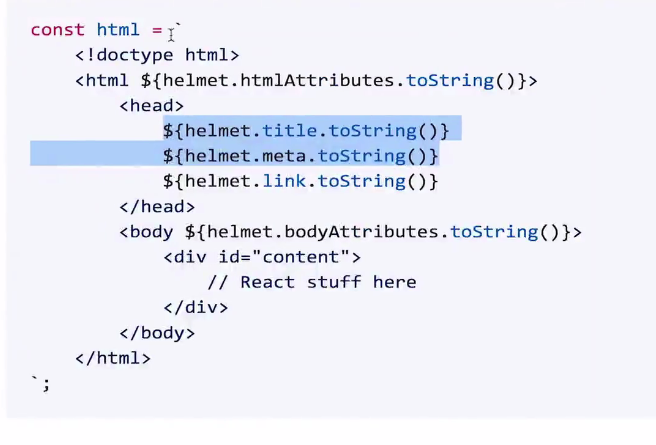
**自定义title meta 用法类似next/head**

**客户端：**



**服务端：**

**Github上的用法：**



**本实战用法：**



**使用预渲染解决SEO问题的新思路**

**预渲染**

