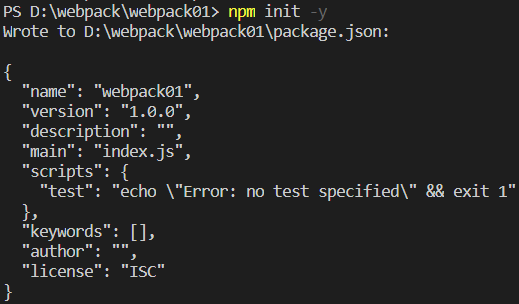
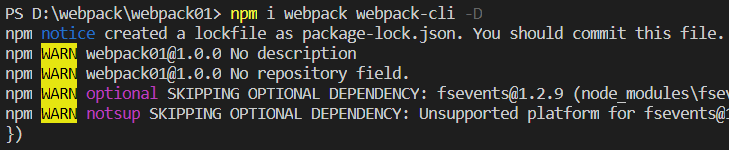
# 安装webpack

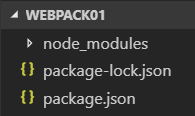
1. 初始化项目：在项目中创建package.json文件。



1. 安装webpack:
   1. 安装webpack分为本地安装与全局安装：本示例使用的是本地安装。
      1. 全局安装：当所有的项目都使用同一个版本的webpack时推荐使用(这种情况不多)，**如果项目之间使用的webpack版本不一样则不推荐使用全局安装**。
      2. 本地安装：本示例使用webpack4所以安装webpack时还需要安装webpack-cli。

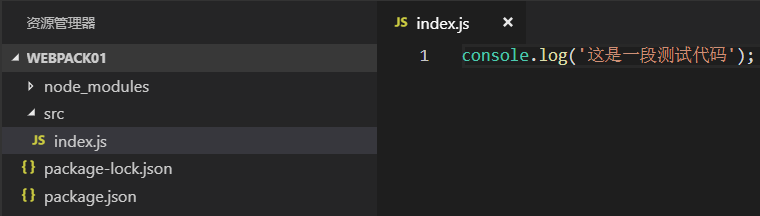


* + 1. 本地安装会安装在当前项目的node\_modules目录下。

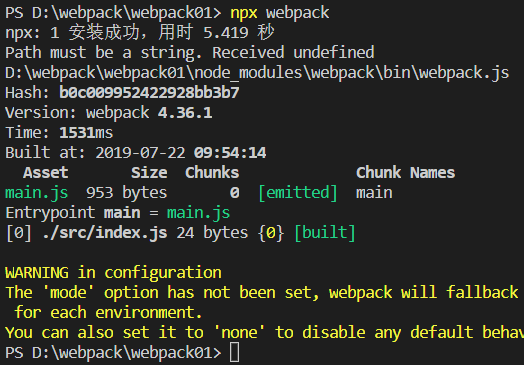


# webpack4的0配置打包

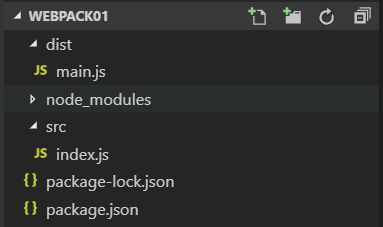
1. 在工作中不会使用到0配置打包，但通过0配置打包可以让我们了解webpack的一些配置。
2. 默认情况下，webpack会对src/index.js进行打包。所以我们需要手动创建src目录，并在src目录中创建index.js文件。
   1. 创建src/index.js文件。



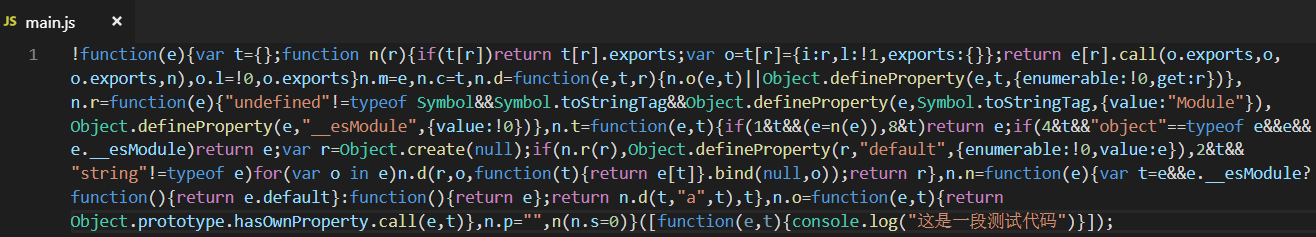
* 1. 打包：



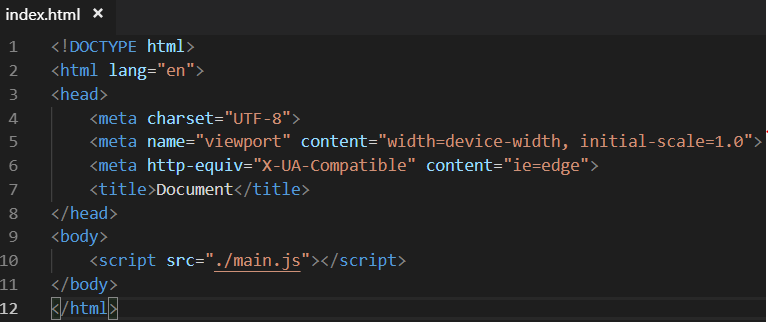
* 1. webpack会在当前项目中创建dist目录，并在dist目录中创建main.js文件。

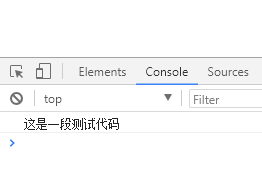


* 1. 因为使用npx命令时没有指定mode模式，所以npx默认使用的是生产环境production，所以main.js中的代码是经过压缩的。



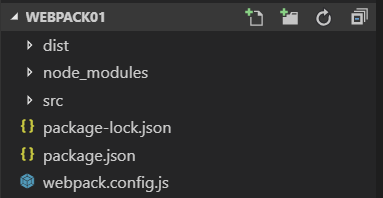
* 1. 测试0打包是否可用，在dist目录中创建名为index.html的页面，并在目录中直接运行。



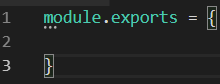


# 手动配置webpack

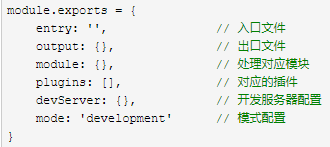
1. 在上一节中我们介绍了webpack的0配置打包。使用0配置打包时所有的内容都是默认的。我们这一节就介绍一下如何手动配置webpack的内容。
2. 在当前项目的根目录下，创建webpack的配置文件webpack.config.js。



1. webpack.config.js文件中的基本配置。
   1. 在webpack.config.js文件中配置module.exports。



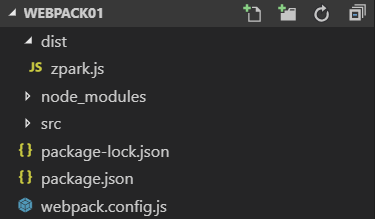
* 1. module.exports中的属性：



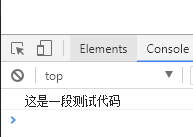
* 1. 简单配置webpack.config.js文件。



* 1. 删除当前项目之前生成的dist目录，重新打包。此时webpack会在当前项目中创建dist目录，并在该目录中创建名为zpark.js文件。此时，因为使用的是开发模式development。所以zpark.js文件中的内容没有压缩。

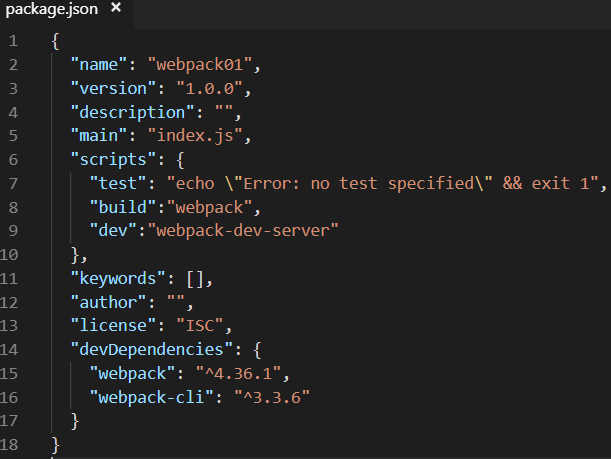


* 1. 测试：在dist目录中重新创建index.html(代码与上一节一样)，并运行。

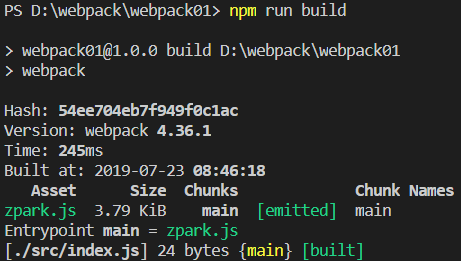


# 配置执行文件webpack.config.js

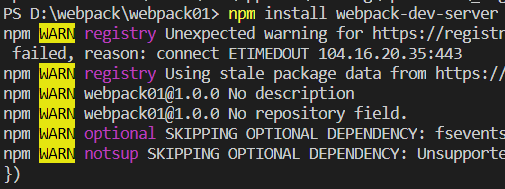
1. 在上面的示例中webpack的执行文件的名字为webpack.config.js。如果修改了配置文件的名字，webpack在编译时会发生找不到默认配置文件的情况，从而根据0配置进行打包。
2. 在package.json文件中可以配置运行环境，并指定执行文件的位置及名称。



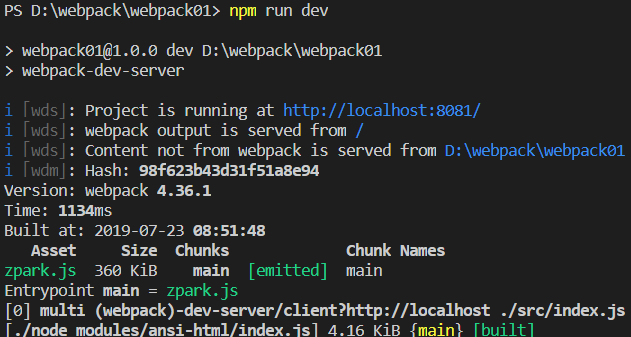
1. 此时，可以在终端使用npm run 环境运行项目。
   1. 生产环境：



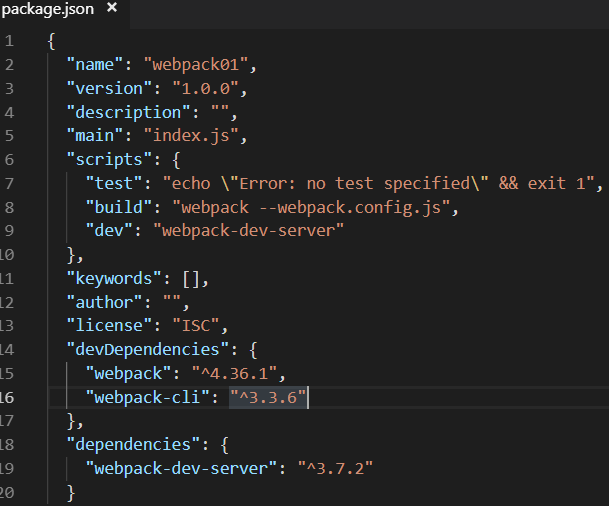
* 1. 开发环境：
     1. 必须先安装webpack-dev-server，否则webpack-dev-server命令无效。



* + 1. 再执行npm run dev命令即可，但此时会发现在项目中并没有产生dist目录及相关的文件。这是因为当前为开发环境，webpack会将这些文件存放在缓存中，而不会真正的创建文件。

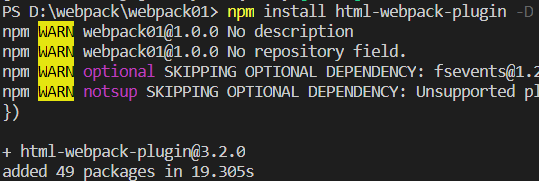


* 1. 还可以在设置环境时，设置执行文件的位置及名称。



# 配置HTML模板

1. 在上面的示例中我们一直使用手动的方式在dist目录中添加相关的html页面，但在实际开发中我们不能这样做。
2. 我们需要在打包时webpack能自动在dist中创建相关的html页面，所以下面我们介绍模板打包出html页面的功能。
3. 示例过程：
   1. 安装html-webpack-plugin插件。



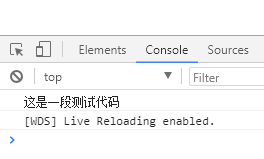
* 1. 在src目录下创建名为index.html页面。



* 1. 在webpack.config.js文件中配置插件。

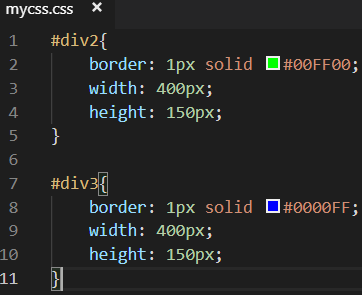


* 1. 测试：执行npm run dev



# 引用CSS

1. 在web项目中CSS是必不可少的，下面介绍一个webpack中使用CSS的步骤。
2. 示例：
   1. 在src目录中创建index.css与mycss.css文件。

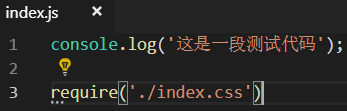




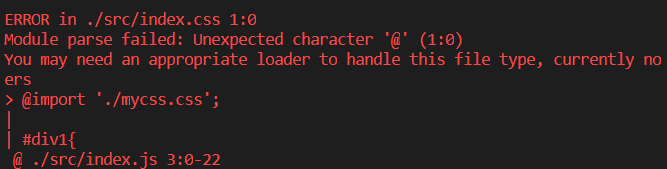
* 1. 修改src目录中index.html页面中的代码。



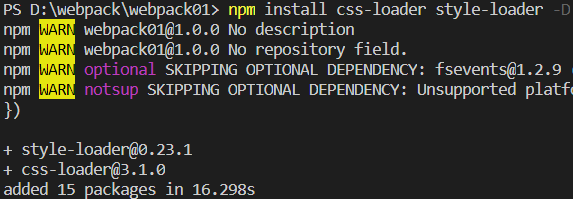
* 1. 在src目录中的index.js文件中引入index.css文件。



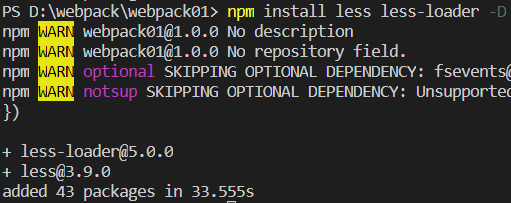
* 1. 此时，执行npm run build会发生以下错误。这个错误表示webpack找不到合适的loader来处理css文件。



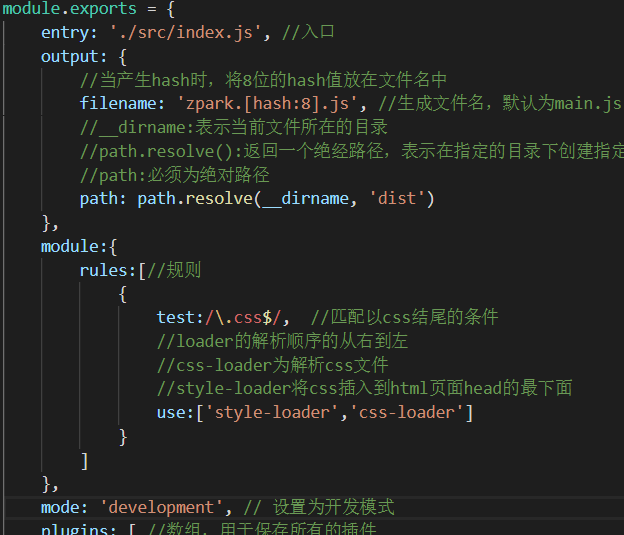
* 1. 所以，我们需要安装与css相关的loader。



* 1. 以后我们还需要使用到less，所以再安装与less相关的loader。



* 1. 在webpack.config.js文件中配置与css相关的loader。

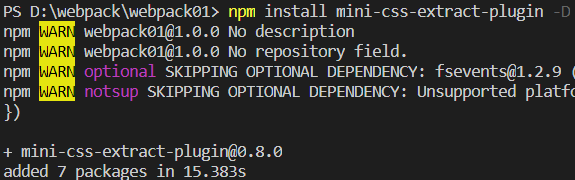


* 1. 测试：执行npm run dev即可看到以下效果。

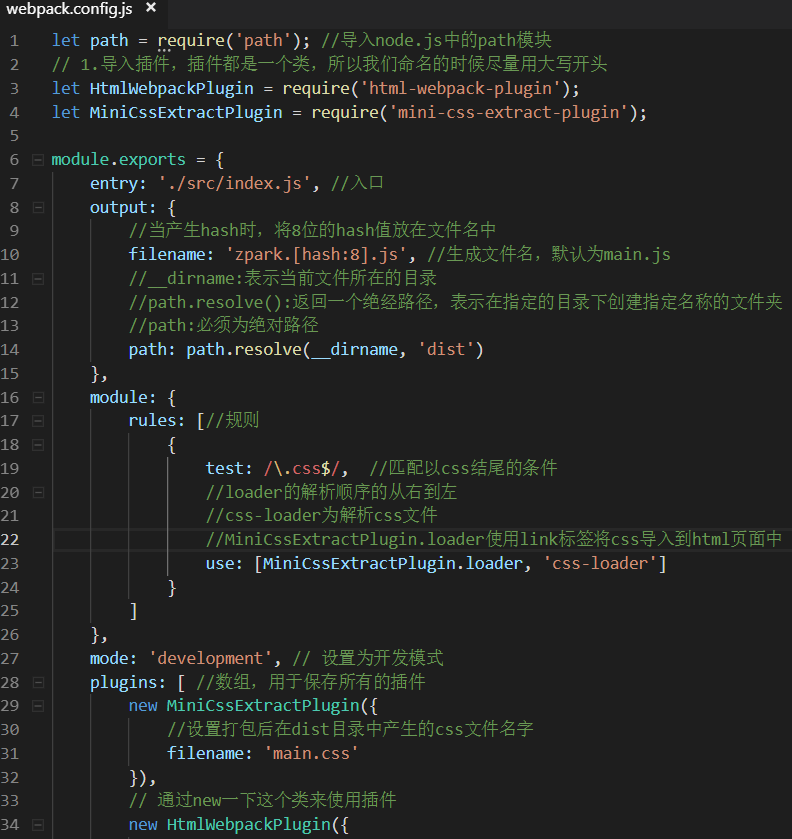


# 使用link标签引入CSS

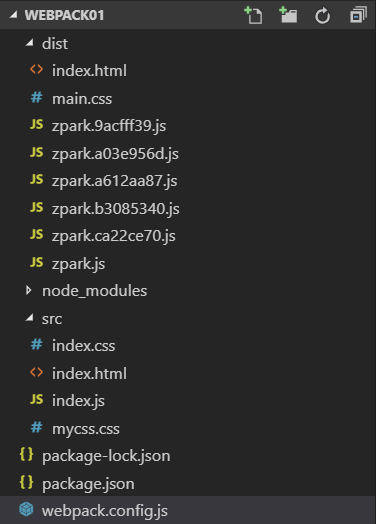
1. 在上一节中，我们成功的将css代码插入到了html页面的head标签中。但如果有很多css代码，这样做会有些不太好。所以下面介绍如何使用link引入css。
2. 安装插件：mini-css-extract-plugin。它可以将css代码打包成一个css文件。



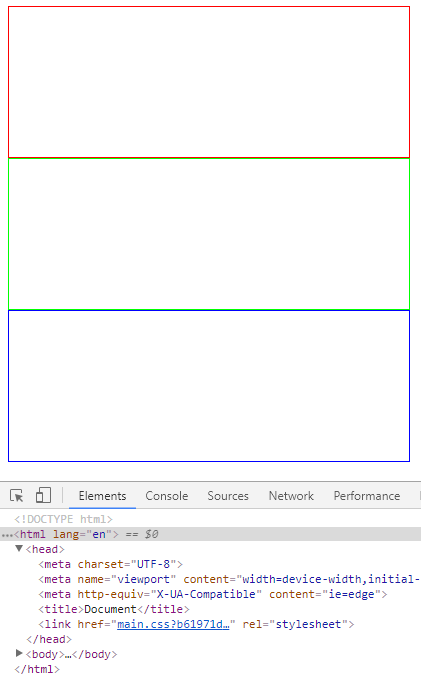
1. 修改webpack.config.js文件中的代码。



1. 执行npm run build可以看到生成的main.css文件。

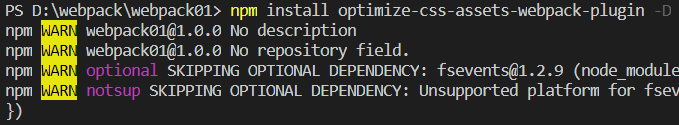


1. 再执行npm run dev。

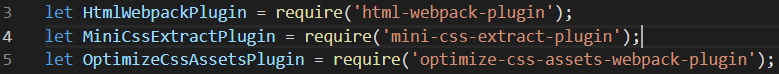


# 压缩CSS

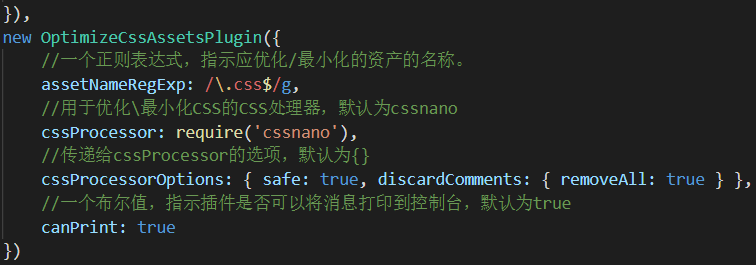
1. 在前面的示例中，当我们将webpack.config.js文件中mode属性设置为production时，webpack在打包时会自动压缩html与js文件中的代码。但webpack不会压缩css文件中的代码，下面就介绍一下压缩css文件的方法。
2. 本示例使用optimize-css-assets-webpack-plugin插件完成CSS代码的压缩。
   1. 安装optimize-css-assets-webpack-plugin插件。



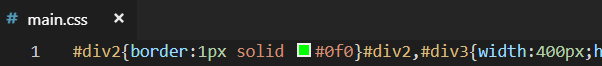
* 1. 在webpack.config.js文件中导入optimize-css-assets-webpack-plugin插件。



* 1. 在webpack.config.js文件配置optimize-css-assets-webpack-plugin插件。

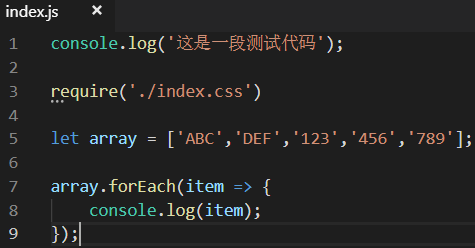


* 1. 此时再使用npm run build打包，便可以看到css文件中的代码已经被压缩了。



# ES6转换为ES5

1. ECMAScript 6(以下简称ES6)是JavaScript语言的下一代标准。因为当前版本的ES6是在2015年发布的，所以又称ECMAScript 2015。也就是说，**ES6就是ES2015**。
2. 为什么需要将ES6转换为ES5，因为IE。现在大部分的浏览器都可以很好的支持ES6语法，但IE却不能。所以如果你的项目需要考虑IE兼容性的问题，就需要本章节的内容，反之可以略过本章。
3. 为了更好的演示效果，最好将webpack.config.js文件中的mode改为development。
4. 在index.js中添加一段ES6代码。

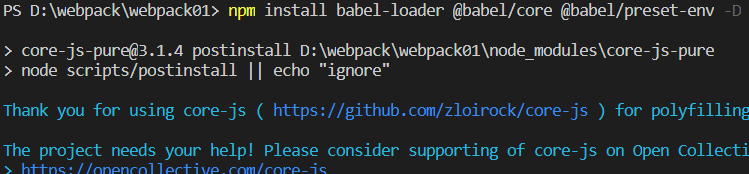


1. 此时如果打包，打包后的js文件中的代码仍是ES6语法的。此时的代码是无法在IE中运行的。





1. 为了解决IE的兼容问题，我们需要安装babel-loader @babel/core @babel/preset-env。



1. 在webpack.config.js文件中配置babel-loader。



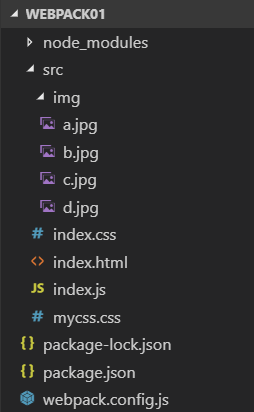
1. 再次打包会看到js文件中的代码已经转换为ES5的代码了。



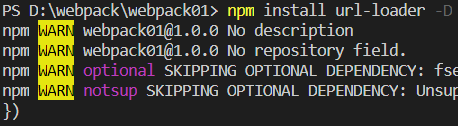
1. 其它与ES6转换ES5语法的示例省略。

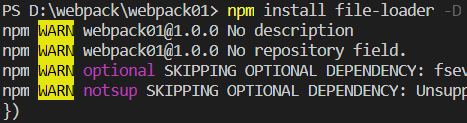
# 处理图片一

1. 在前面的章节中我们看到了在webpack中如何处理html,css与js。但web项目中图片也是不可缺少的元素。下面就看一下在webpack中处理图片的几种方法。
2. 将图片添加到项目的src/img目录下。

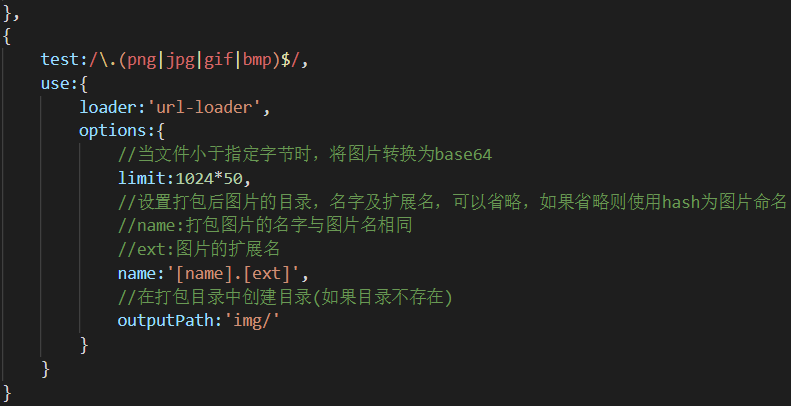


1. 安装url-loader与file-loader，在webpack中处理图片经常会用到两个loader：file-loader与url-loader。这两个loader的关系是url-loader包含了file-loader。
   1. file-loader：可以根据url将指定的图片打包到指定的目录下。
   2. url-loader：会将图片转换为base64。如果页面中图片较多时，可以减少页面发出的http请求，提高页面加载的效率。**设置url-loader时会将字节数小于N的图片转换为base64，如果字节数大于N时url-loader会自动调用file-loader**。

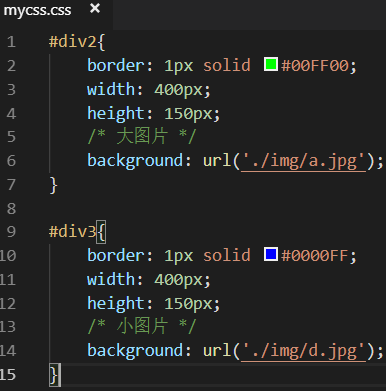




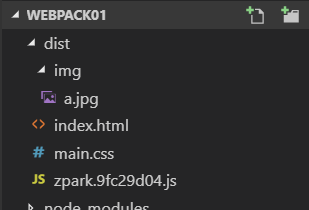
1. 在webpack.config.js文件中配置url-loader(不需要配置file-loader)。



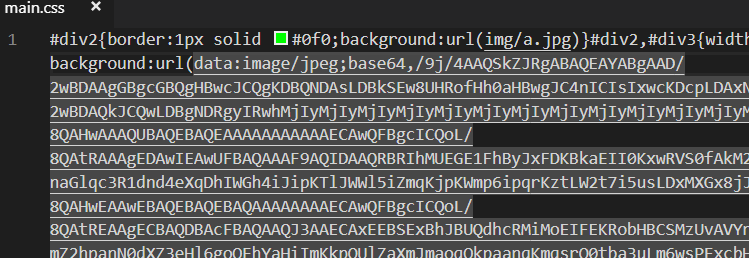
1. 在mycss.css中引入图片。



1. 此时可以打包，可以看到dist/img目录下只有a.jpg而没有d.jpg。因为d.jpg小于指定的字节数，直接在css文件中生成了base64。



1. 在打包后的css文件中可以看到图片对应的base64。



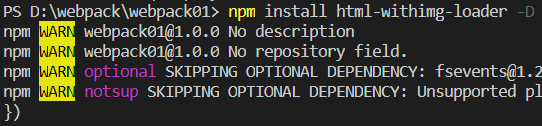
1. webpack处理js文件中的图片仍然会使用url-loader与file-loader，所以在处理js中的图片时不需要再安装新的loader了。
2. 修改index.js文件中的代码。



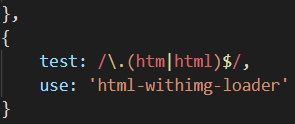
1. 打包运行即可看到效果。

# 处理图片二

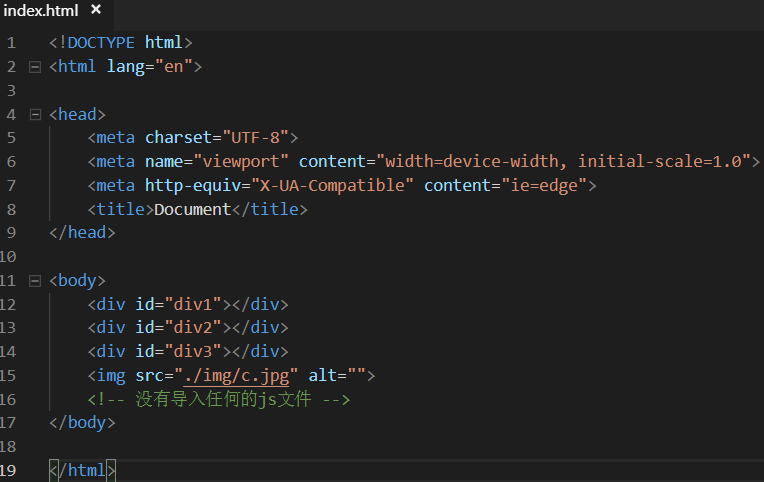
1. 本节介绍webpack如何处理html文件中的图片，但无论是url-loader还是file-loader都不能处理html中的图片，所以我们需要新的loader。
2. 安装html-withimg-loader。



1. 在webpack.config.js文件中配置html-withimg-loader。



1. 在index.html中添加图片。



1. 打包运行即可看到效果。

# 多入口文件（一）

1. 如果使用VUE则不需要本节内容。
2. 利用本节再复习一个创建webpack项目的过程。
   1. 创建目录。
   2. 初始化项目：



* 1. 在项目中创建src/js/目录，并在目录中创建两个index.js与login.js文件。



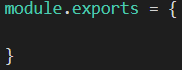
* 1. 安装webpack与webpack-cli。



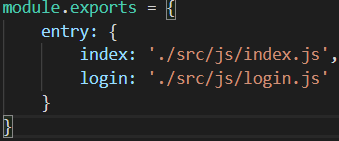
* 1. 创建webpack.config.js文件。
     1. 导入path。



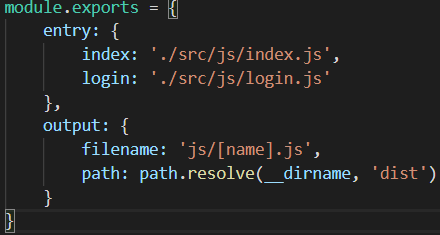
* + 1. 创建module.exports。



* + 1. 配置entry入口。因为有多个入口，所以entry这次是个对象。



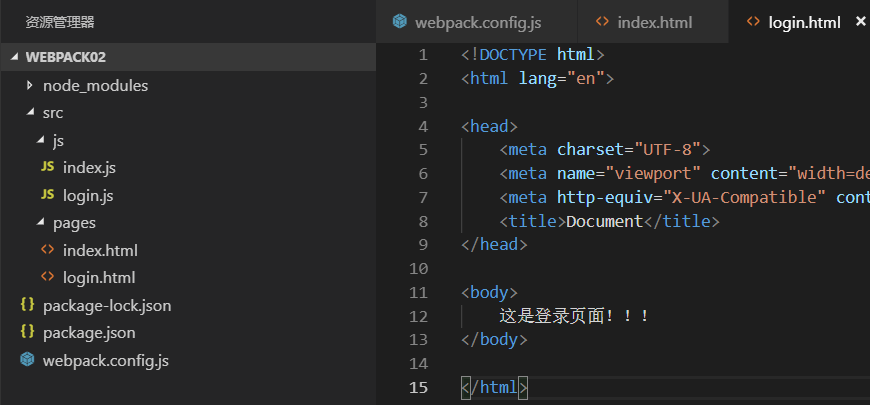
* + 1. 配置output出口。



* + 1. 安装html-webpack-plugin插件。



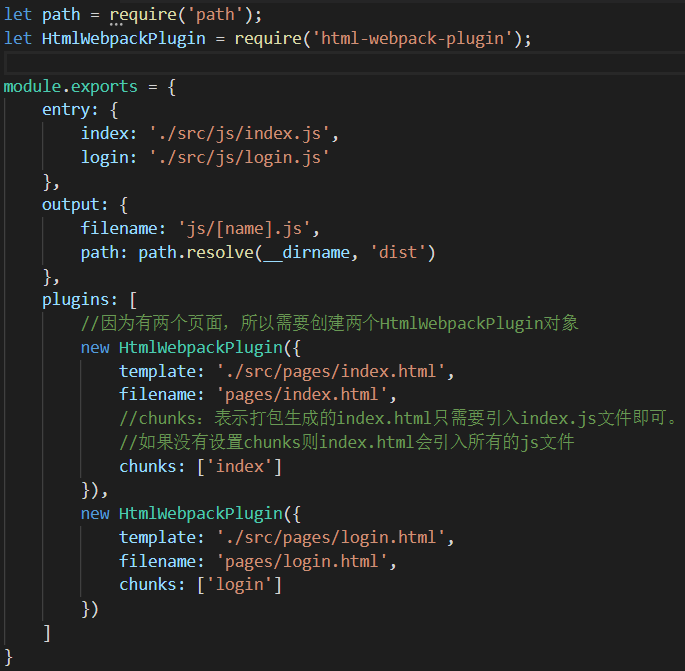
* + 1. 在src目录中创建pages目录，在目录中创建index.html、login.html。



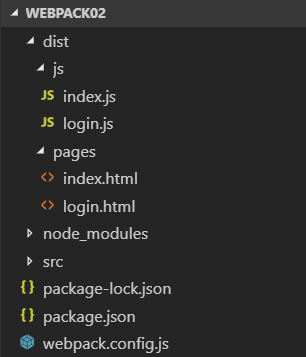
* + 1. 在webpack.config.js文件中导入html-webpack-plugin插件。



* + 1. 在webpack.config.js文件中配置html-webpack-plugin插件。

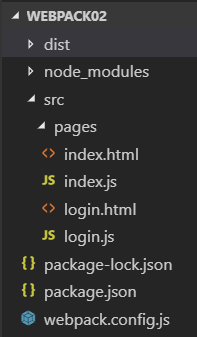


* + 1. 运行npx webpack即可打包。

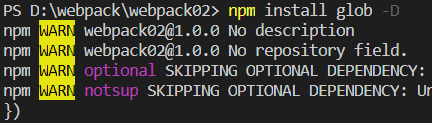


# 多入口文件（二）

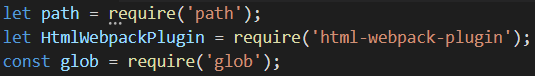
1. 在上一节中我们完成了webpack多入口文件，但上一节的示例中出现了一个小问题。当项目中有多个入口页面时，我们需要配置多个entry与html-webpack-plugin。页面越多配置的越多，很不方便。
2. 本节将介绍通过glob完成自动配置。
3. 本节中的代码需要将js文件移动到pages目录中。



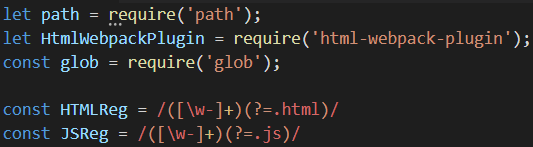
1. glob可以在webpack中更方便的解析路径。
   1. 安装glob。



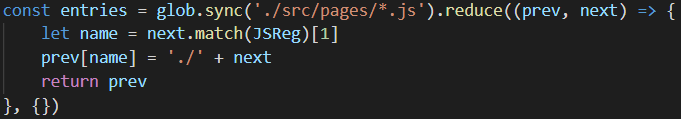
* 1. 在webpack.config.js文件中导入glob。



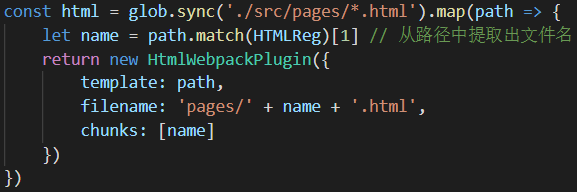
* 1. 创建两个正则表达式(前瞻)用于获得指定扩展名的文件名字(不包含扩展名)。



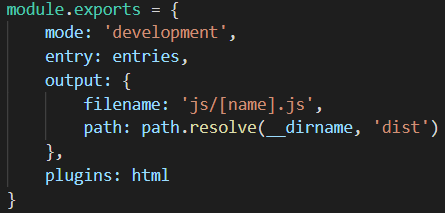
* 1. 动态获得指定目录中所有js文件的信息，并生成入口。



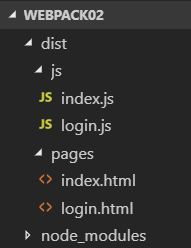
* + 1. glob.sync()：返回指定目录中指定类型的文件，返回一个数组。
    2. glob.sync().map()：对数组中的所有元素调用指定的回调函数。回调函数的返回值是累积的结果，并在下一次调用回调函数时作为参数提供服务。
  1. 动态获得指定目录中所有的html文件，并为每个html文件动态创建html-webpack-plugin对象。



* + 1. glob.sync().map()：为数组中的每个元素上调用定义的回调函数，并返回包含结果的数组。
  1. 修改module.exports。

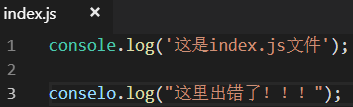


* 1. 执行npx webpack进行打包，可以看到编译出的文件。

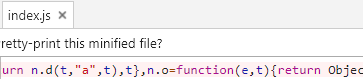


# source-map

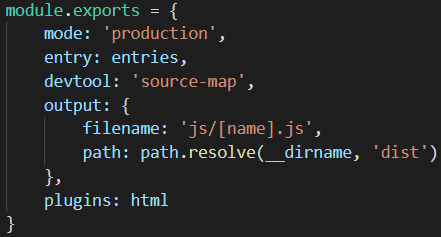
1. source-map的作用：当打包时压缩了js代码后，如果js代码出现错误，我们很在在压缩后的js代码中找到错误在哪。此时，可以使用source-map对源码进行映射，从而更简单的找到错误发生的位置。
2. 直接在上例中进行修改即可，注意，需要将mode改为production。
3. 为了演示source-map，先在index.js文件中添加一行错误的js代码。



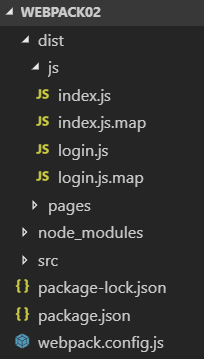
1. 此时运行npx webpack打包，并运行页面时在浏览器中可以看到错误，但无法准确的找到错误的位置。



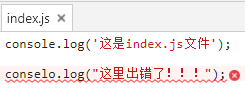
1. 此时，我们可以在webpack.config.js文件中配置devtool，来设置source-map。



1. 此时，再打包时会发现webpack多生成了两个文件。



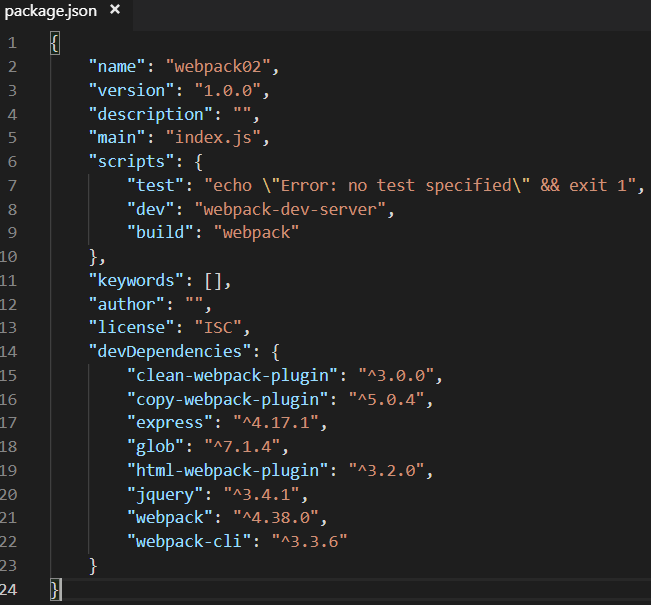
1. 运行index.html页面时，可以看到错误发生的位置。



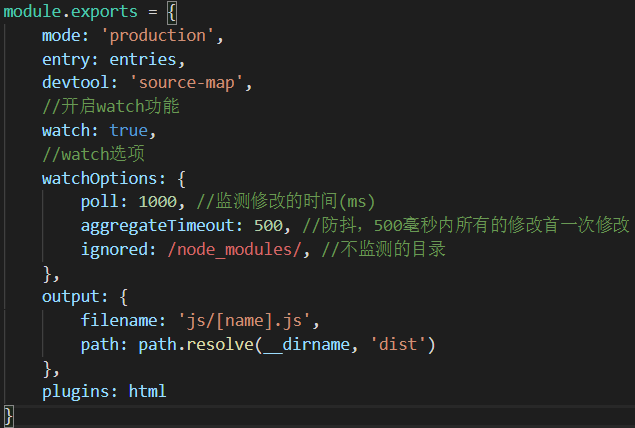
1. 这就是source-map的功能。
2. devtool中除了source-map还可以出现其它选项(这里不再演示)：
   1. eval: 使用eval将模块包裹，在末尾加入sourceURL，速度快
   2. source-map: 产生一个.map的文件，文件里面保留了打包后的文件与原始文件之间的映射关系
   3. cheap: 与source-map类似，区别在与map文件中没有列信息
   4. module: 会定位到用loader转换之前的文件里
   5. inline: 产生.map文件，但这个文件会以base64的格式编码，在模块末尾以dataURL的方式引入

# webpack中watch的用法

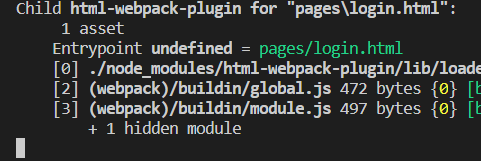
1. watch的作用：在之前所有的示例中，我们每次修改代码以后都需要重新打包，才能看到效果。这对于开发来说有些麻烦，而watch为我们提供了自动打包功能(有点像springboot中的热布署)。
2. watch是自带插件，不需要另行安装。
3. 为了演示watch更方便，建议在package.json文件中配置dev与build。



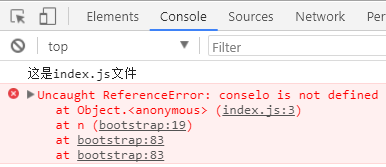
1. 在webpack.config.js文件中配置watch信息。



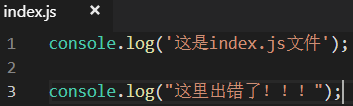
1. 此时，运行npm run build命令时，会发现npm run build与之前不太一样，它并没有结束，而是在等待。



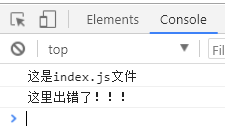
1. 此时，运行页面仍然会看到上一节的错误。



1. 将上一节中演示用的错误改正。

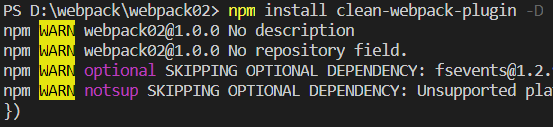


1. 在修改错误时，会看到控制台中出现了变化。此时刷新页面可直接看到效果，而不再需要重新打包。

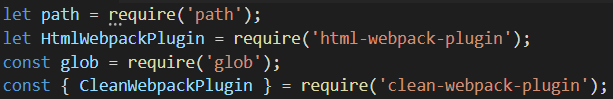


# webpack常用插件

1. 本节中将介绍3个webpack中常用的插件，可以让开发webpack变得更简单。
2. 插件1：clean-webpack-plugin。在之前示例中我们经常需要手动删除dist目录中的文件，以便获得更好的演示效果。而clean-webpack-plugin插件可以自动删除指定目录中的内容。
   1. 安装clean-webpack-plugin插件：



* 1. 在webpack.config.js文件中声明clean-webpack-plugin插件。注意：这是新版本中的写法。



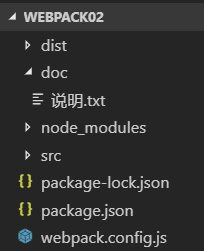
* 1. 在webpack.config.js文件中配置clean-webpack-plugin插件。注意：因为使用了自动

打包多入口文件，所以需要对plugins进行一些修改(其实就是将两个数组连接成一个数组)。

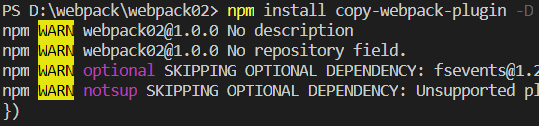


* 1. 此时，可以将dist目录中的文件改名，再运行npm run build命令，可以看到改名的文件消失了，说明clean-webpack-plugin插件删除了dist目录中的内容。

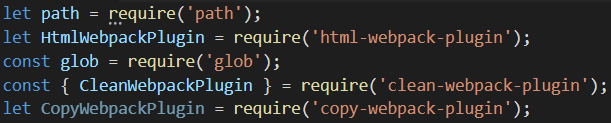
1. 插件2：copy-webpack-plugin插件。它的功能很简单，就是将单个文件或目录在打包时拷贝到构建目录中(dist目录)。
   1. 创建doc目录，并创建名为:说明.txt文件。



* 1. 安装copy-webpack-plugin插件。



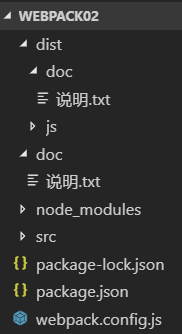
* 1. 在webpack.config.js文件中声明copy-webpack-plugin插件。



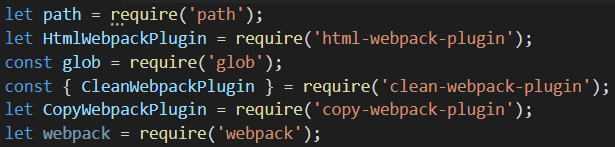
* 1. 在webpack.config.js文件中配置copy-webpack-plugin插件。



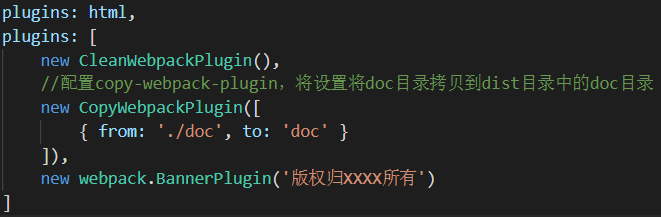
* 1. 此时运行npm run build命令，可以在dist目录中看到doc目录以及目录中的文件。



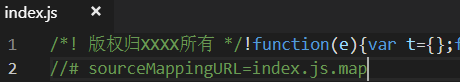
1. 插件3：bannerPlugin插件。此插件是webpack自带插件，所以不需要另行安装。bannerPlugin插件的功能很简单，就是为每个生成的js文件添加版权信息。
   1. 在webpack.config.js文件中声明webpack。因为bannerPlugin是webpack的插件，所以需要声明webpack而不是bannerPlugin。



* 1. 在webpack.config.js文件中配置webpack.BannerPlugin插件。

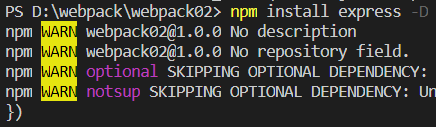


* 1. 此时运行npm run build打包，在任意js文件的前面都可以看到相应的版权信息。

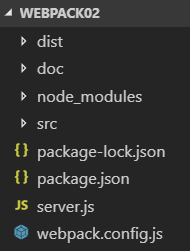


# webpack跨域

1. 在web开发中经常会遇到跨域的问题(如：使用第三方接口)，所以本节将介绍一个webpack中跨域的几种方法。
2. 在演示跨域操作前，我们先使用express创建一个端口为web应用。
   1. 安装express。



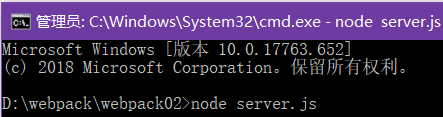
* 1. 创建名为server.js的文件。



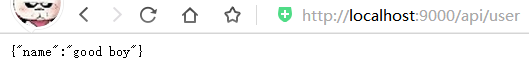
* 1. 在server.js文件中进行配置。



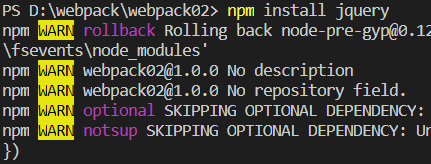
* 1. 在控制台中运行node server.js命令启动web应用(最好打开一个新的控制台，因为运行后进程只能手动关闭)。



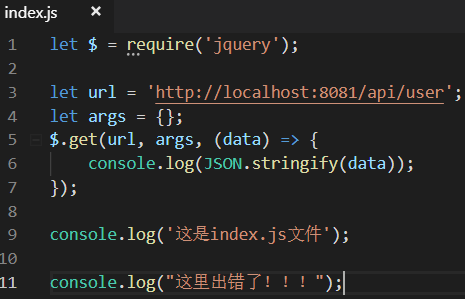
* 1. 在浏览器地址栏中输入：<http://localhost:9000/api/user>，可以看到json结果。



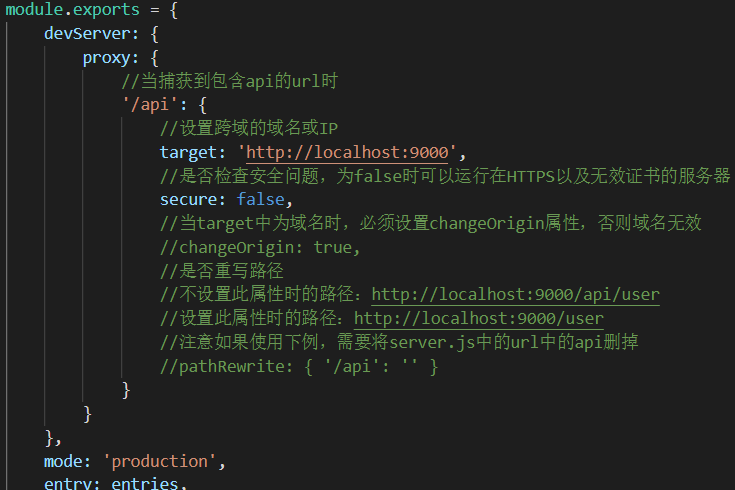
1. 跨域方法1：使用proxy代理实现。
   1. 为了方便演示方法1，先安装jquery。



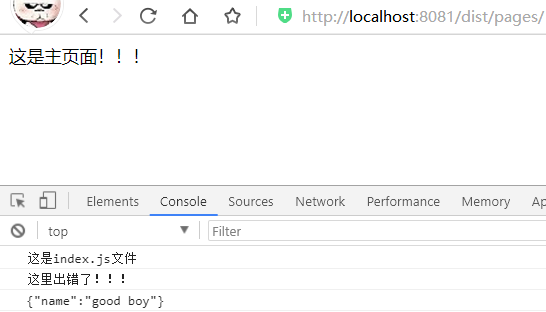
* 1. 在index.js中导入jquery，并使用jquery发出get请求。



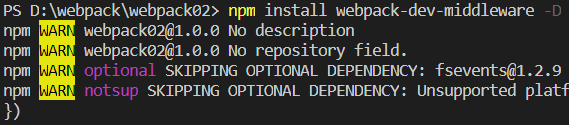
* 1. 在webpack.config.js文件中配置devServer。



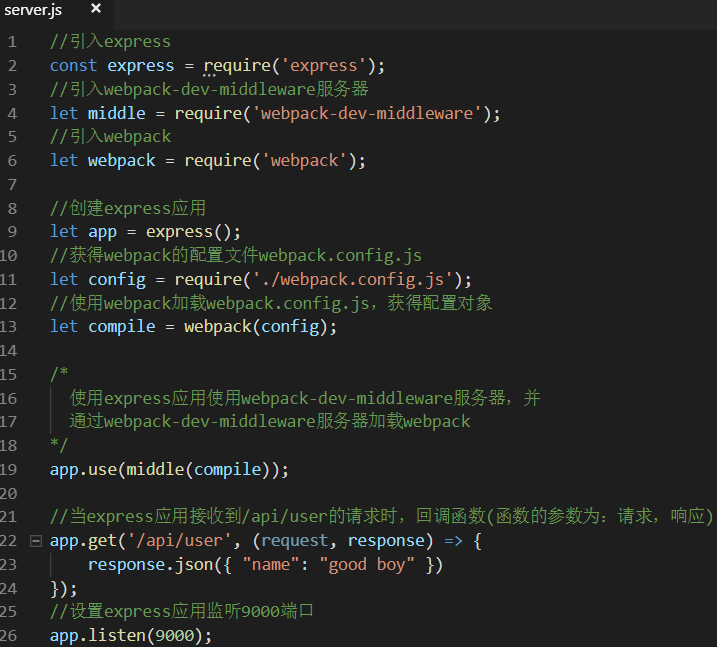
* 1. 此时，先使用npm run build打包，再使用npm run dev运行，可以看到以下效果。



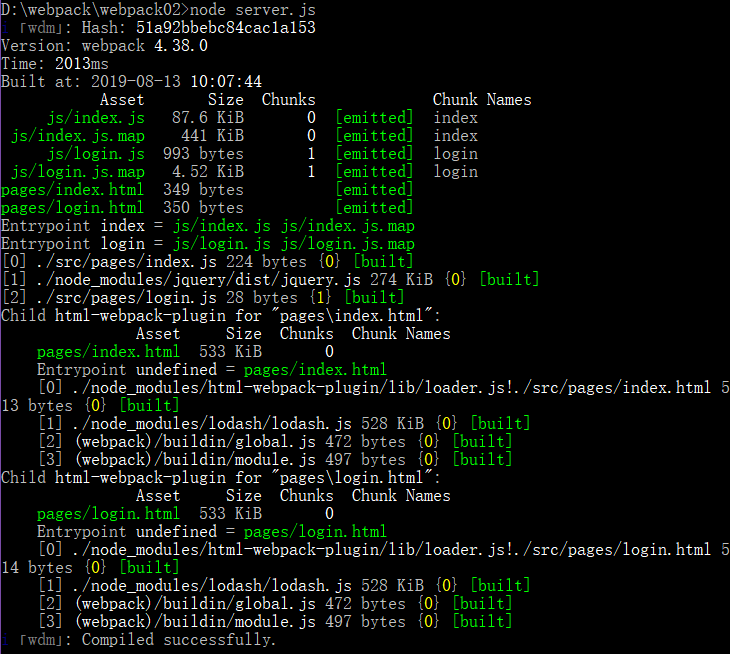
1. 跨域方法2：使用webpack-dev-middleware服务器。使用这种方法需要进行跨域的两个服务器必须都是自己的。其实就是使用webpack-dev-middleware服务器(假设端口为9000的服务器)将webpack(端口为8081的服务器)一起加载并运行。
   1. 首先，安装webpack-dev-middleware服务器。



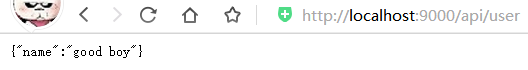
* 1. 修改server.js文件。



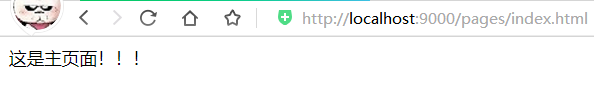
* 1. 将webpack.config.js文件中的devServer注释。
  2. 此时，再控制台中运行node server.js，会发现与之前运行时有所不同(因为在启动express服务器时，还会加载webpack)。



* 1. 在浏览器地址栏中输入:http://localhost:9000/api/user。



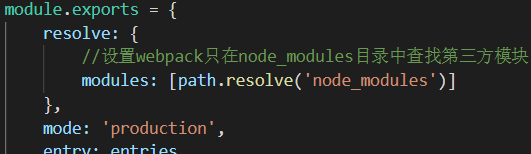
* 1. 在浏览器地址栏中输入:http://localhost:9000/pages/index.html。



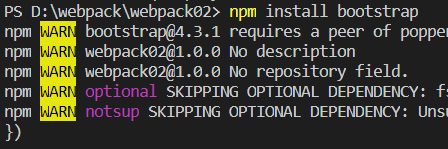
* 1. 这时我们会发现两个服务在一个服务器中，因为都在一个服务器中所以就不再存在跨域的问题了。

# webpack中的resolve

1. resolve是webpack中比较重要的内容，主要用于设置webpack如何解析模块。
2. 下面介绍resolve常用的一些属性：
   1. resolve. modules：设置webpack在哪些目录中查找第三方模块。值是一个数组，可以同时设置多个目录。



* 1. 以了演示其它属性，先安装bootstrap。



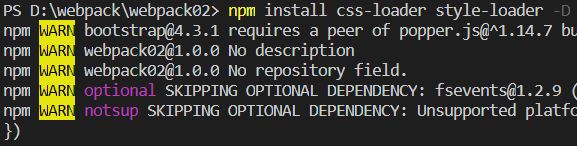
* 1. 在index.js文件中导入bootstrap。

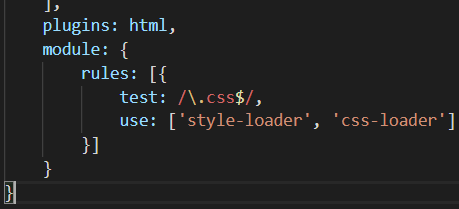


* 1. 在index.html页面中使用bootstrap样式。

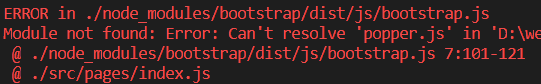


* 1. 安装与配置css-loader，style-loader。





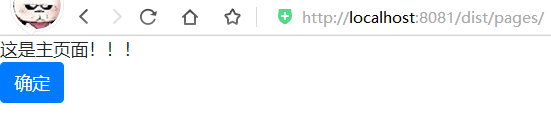
* 1. 此时运行npm run dev命令，会发现webpack报错。



* 1. 发生错误的原因是在index.js文件中使用import ‘bootstrap’时，import语句导入的并不是bootstrap.css文件，而是默认导入的是bootstrap.js文件。所以才会发生这个错误。
  2. resolve. alias属性：设置别名。可以使用alias属性设置bootstrap应该加载哪个文件。



* 1. 此时再运行，可以在页面中看到相应的效果。



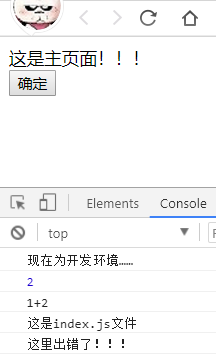
* 1. resolve的其它属性不再进行介绍了。

# 使用DefinePlugin设置全局常量

1. 在webpack中可以使用DefinePlugin设置一些全局常量。这些常量经常用于设置开发与生产环境的参数(如：是否开启日志等)。
2. 因为DefinePlugin设置的全局常量，所以在所有的js文件中均可以使用这些常量。
3. DefinePlugin是webpack自带的组件，所以不需要重新安装。
4. 在webpack.config.js文件中配置DefinePlugin。

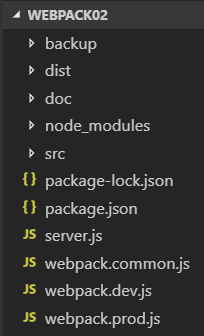


1. 直接打包运行，在浏览器中可以看到效果(注意MESSAGE的值不是1+1，而是2)。

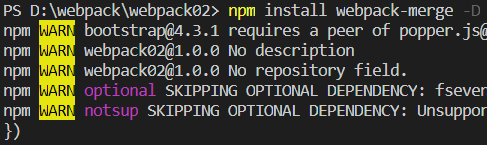


# 区分不同环境

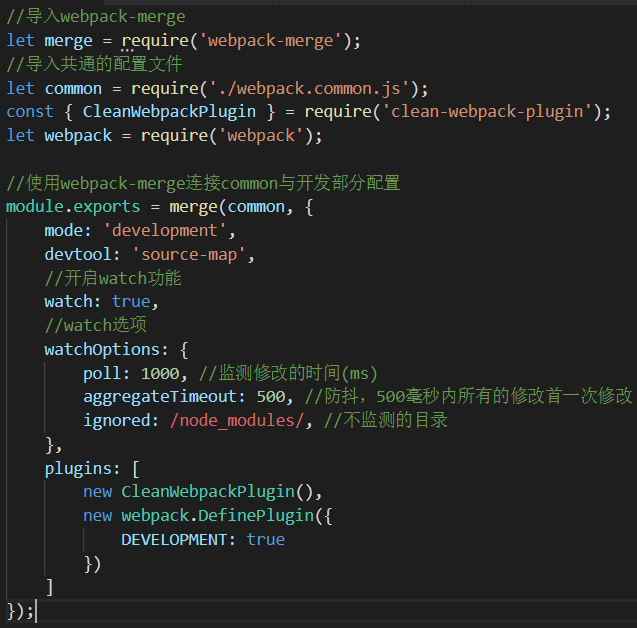
1. 在上一节中我们通过设置全局常量的方式区分开发与生主环境，这种方法会产生大量的if代码。不利于代码的维护。
2. 在工作中我们会将webpack.config.js文件分为三个文件(三个文件的名字可能不同)：
   1. webpack.common.js：用于保存开发与生产环境中共同的部分。
   2. webpack.dev.js：用于保存开发环境特有的部分。
   3. webpack.prod.js：用于保存生产环境特有的部分。
3. 示例：
   1. 首先创建webpack.dev.js与webpack.prod.js文件，并将webpack.config.js文件改名为webpack.common.js文件。



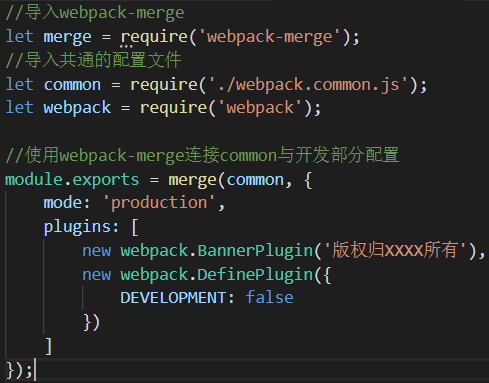
* 1. 安装webpack-merge，webpack-merge可以合并多个文件。



* 1. 配置webpack.dev.js文件(将webpack.common.js中与开发相关的配置剪切到webpack.dev.js文件中)



* 1. 配置webpack.prod.js文件。



* 1. webpack.common.js文件。

let path = require('path');

let HtmlWebpackPlugin = require('html-webpack-plugin');

const glob = require('glob');

let CopyWebpackPlugin = require('copy-webpack-plugin');

let webpack = require('webpack');

const HTMLReg = /([\w-]+)(?=.html)/

const JSReg = /([\w-]+)(?=.js)/

const html = glob.sync('./src/pages/\*.html').map(path => {

let name = path.match(HTMLReg)[1] // 从路径中提取出文件名

return new HtmlWebpackPlugin({

template: path,

filename: 'pages/' + name + '.html',

chunks: [name]

})

})

const entries = glob.sync('./src/pages/\*.js').reduce((prev, next) => {

let name = next.match(JSReg)[1]

prev[name] = './' + next

return prev

}, {})

module.exports = {

resolve: {

//设置webpack只在node\_modules目录中查找第三方模块

modules: [path.resolve('node\_modules')],

//为bootstrap/dist/css/bootstrap.css设置别名

alias: {

bootstrap: 'bootstrap/dist/css/bootstrap.css'

}

},

entry: entries,

output: {

filename: 'js/[name].js',

path: path.resolve(\_\_dirname, 'dist')

},

plugins: [

//配置copy-webpack-plugin，将设置将doc目录拷贝到dist目录中的doc目录

new CopyWebpackPlugin([

{ from: './doc', to: 'doc' }

]),

new webpack.DefinePlugin({

VERSION: JSON.stringify('1.0'),

MESSAGE: '1+1',

RESULT: JSON.stringify('1+2')

})

].concat(html),

module: {

rules: [{

test: /\.css$/,

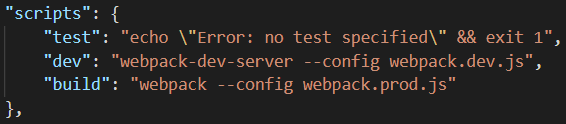
use: ['style-loader', 'css-loader']

}]

}

};

* 1. 在控制台中运行npm run build -- --config webpack.prod.js与npm run build -- --config webpack.dev.js命令(需要两对横线)，可以通过打包后的index.js文件中的代码是否压缩就可以直接看到开发与生产环境的不同。
  2. 当然，这种方式运行比较麻烦，建议修改package.json中的script。



# webpack优化(1)

1. 本节将介绍noParse、exclude的使用方法。
2. 为了更好的演示，本节重新创建webpack项目。
   1. 初始化项目。
   2. 安装：webpack,webpack-cli。
   3. 安装：html-webpack-plugin。
   4. 安装：@babel/core。
   5. 安装：babel-loader。
   6. 安装：@babel/preset-env
   7. 安装：jquery
   8. 创建src目录。
   9. 在src目录创建index目录。
   10. 在index目录中创建index.js与index.html。
   11. 创建webpack.config.js文件。



* 1. 此时如果在index.js中引用了jquery，**那么webpack在打包时会在jquery中查找jquery是否引用了其它的资源。但jquery通常不会引入其它的资源，这时需要设置webpack在打包时不需要在jquery中查找引用**。



* 1. 此时，可以在webpack.config.js文件中使用noParse设置webpack打包时不需要在jquery中再查找引用。



* 1. 还可以在loader中使用exclude，设置loader在解析时不需要解析哪些目录中的文件。

