# Webpack1

**webpack支持的三种模块化方式: AMD,COMMON,ES6**

**webpack官网很多文档都在** ”**configuration”**

**在webpack中，任何资源都视为一个模块**

**loader的处理方式是从右到左**

**webpack-2：**

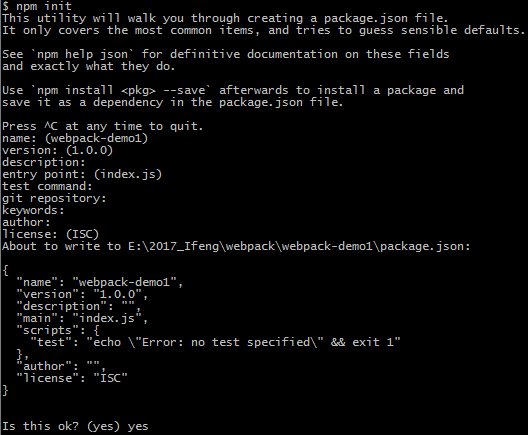
**https://qiutc.me/post/webpack-2-%E7%9A%84%E6%96%B0%E7%89%B9%E6%80%A7-What-s-new-in-webpack-2.html**

### 一、初始webpack

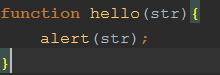
1. **初识webpack**

webpack官网地址：http://webpack.github.io/docs/what-is-webpack.html

1. 安装全局 => npm install webpack –g；
2. 进入文件根目录,例如test；
3. 初始化npm => npm init (回车创建package.json)；



1. 在文件夹安装webpack => npm install webpack - -save-dev;安装完之后有文件夹node\_modules;
2. 在根目录test下新建hello.js,

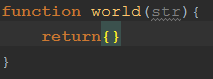


-- 使用webpack打包文件hello.js; 命令行 => webpack hello.js hello.bundle.js(打包以后文件的名称);

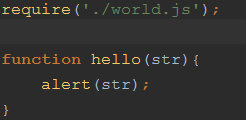
-- **打包完成会给一些信息**：哈希；版本；时间；Asset(打包生成的文件)；size(文件大小)；chunks(这次打包的分块)；chunk Names (这次打包的块名称)

-- 打包完以后的文件hello.bundle.js，上面有一大段代码是在打包的过程中，生成的它自己所需要的一些内置函数；比如webpack\_require; 在后面能看见我们的代码，以及给这个代码块的编号；

1. 新建一个文件，熟悉下webpack的工作原理；



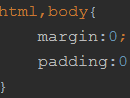
-- 新建一个world.js，在hello.js文件中 rquire () 引用进来;

  
-- 再运行一下 webpack hello.js hello.bundle.js

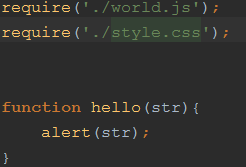
-- 打包完成，会有两个模块(0,1)；再看一下文件名内容，world已经被打包进来，hello.js里面的require() 已经变成了webpack 内置的 webpack\_require;这是webpack的一种工作方式；寻找它的依赖然后把它们打包在一起；

1. 试一下打包css文件

-- 新建一个style.css



-- 在hello.js文件中 rquire 进来;



打包；会报错 You may need an appropriate loader to handle this file type.（你需要一个适当的loader来处理这种文件！）

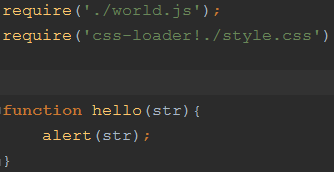
-- 这说明webpack 天生是不支持这种类型.css的，要处理这种类型需要依赖loader;

-- 在项目目录下安装这个css-loader;配合css-loaer 还需要 style-loader;

npm install css-loader style-loader –-save-dev;

再运行一下webpack hello.js hello.bundle.js 还是报错；文件中require()的css并没有指定相应的loder;

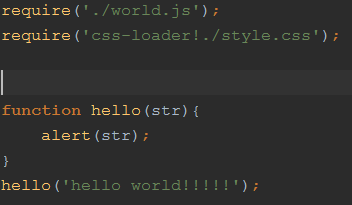
-- require('css-loader!./style.css');这种语法，在引用这个文件之前，必须先经过css-loader的处理;



在运行一下命令;css文件已被打包进来；也多了一些其他的自定义的内置函数；

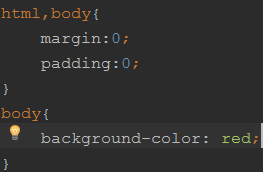
1. 为了更清晰的展示效果，在建一个html文件；引入打包后的文件；

——在hello中，添加执行函数代码 hello('hello world');



打包；然后在浏览器打开html文件，hello函数已经执行；

——再修改一下css 比如给body加背景颜色；



打包；刷新页面；并没有效果；因为我们要是css生效，还需要借助一个loader:style-loader; require('style-loader!css-loader!./style.css');



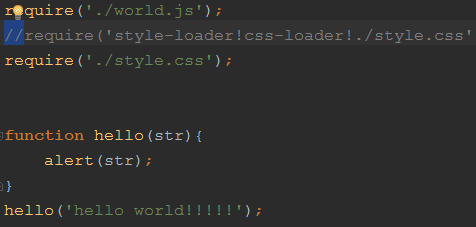
打包；刷新页面；发现css生效；

F12看下代码，会发现我们写的css，被新建了一个style标签插在了head里面；这是通过style-loader做到的；

**style-loader：把css-loader处理完的文件，新建一个style标签放在head里面；**

**css-loader： 是的webpack可以处理.css文件；**

1. 如果不把loader放在require里，可以通过命令行工具；



Webpack hello.js hello.bundle.js –module-bind ‘css=style-loader!css-loader’;

模块绑定,如果是css, 用style-loader,style-loader之前用css-loader处理;

1. –watch监听；文件改变就会打包；

Webpack hello.js hello.bundle.js –module-bind ‘css=style-loader!css-loader’ --watch

1. --progress 可以看到打包过程；有个百分比读条
2. –display-modules 可以看到打包的模块 会把引用的所有的模块会列出来
3. –display-reasons 会有打包模块的原因

### 二、配置webpack 以及配置参数功能

1. **配置webpack 以及配置参数功能**
2. 新建目录mkdir webpack-demo;
3. 用npm初始化目录 npm init;
4. 安装webpack： npm install webpack –save-dev;
5. 建一些文件夹供项目使用；

mkdir src => 项目源文件   
mkdir dist =>静态资源

1. 给项目webpack-demo新建一个初始化页面index.html;引入打包后的文件（假设是bundle.js）
2. 在src目录下新建文件夹script（脚本）,style（样式）；
3. 给项目建立一个配置文件；webpack.config.js (如果我们直接执行命令webpack ,他会在项目根目录下找这个文件作为默认配置，当然可以通过--config option 来指定其他的配置文件)；官网 => API => configuration
4. 使用webpack --help 查看命令
5. webpack.config.js 内容：

module.exports={ //commonJs模块化的输出

entry:’打包的入口’,

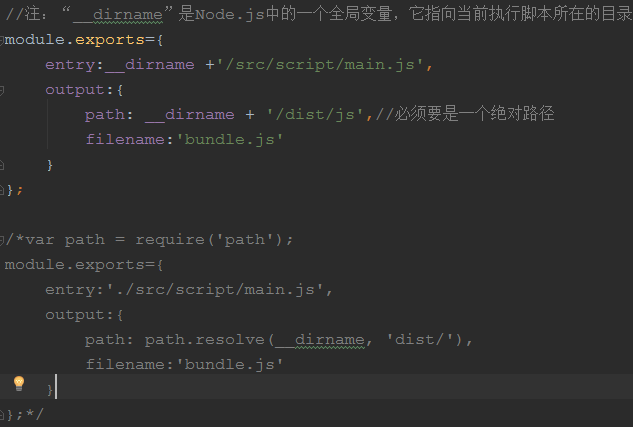
output:{

path:’ 打包以后放在什么地方’,

filename:’打包以后的文件名’

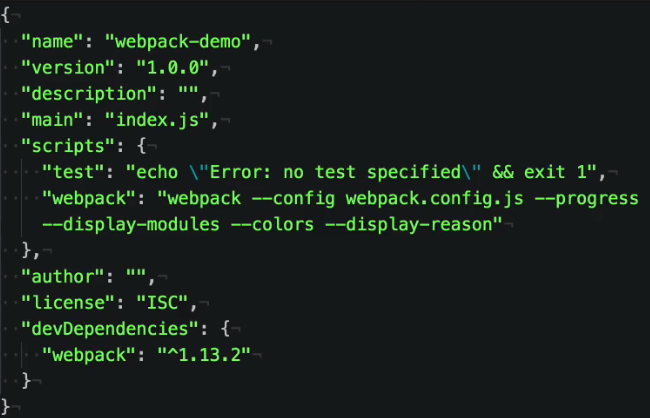
}

}

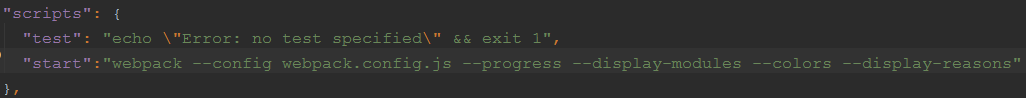


这是一个最简单的webpack 的配置；运行webpack ，文件被打包好了；

—\*—参数：参数可以写在npm的脚本里 package.json => scripts





npm start;

\*\*\***注：**package.json中的脚本部分已经默认在命令前添加了node\_modules/.bin路径，所以无论是全局还是局部安装的Webpack，你都不需要写前面那指明详细的路径了。

\*\*\* npm的start是一个特殊的脚本名称，它的特殊性表现在，在命令行中使用npm start就可以执行相关命令，如果对应的此脚本名称不是start，想要在命令行中运行时，需要这样用npm run {script name}如npm run build

1. 参数：

--entry的三种方式:

1）.string

2) .array entry: ["./entry1", "./entry2"]

3) .object 多页面应用

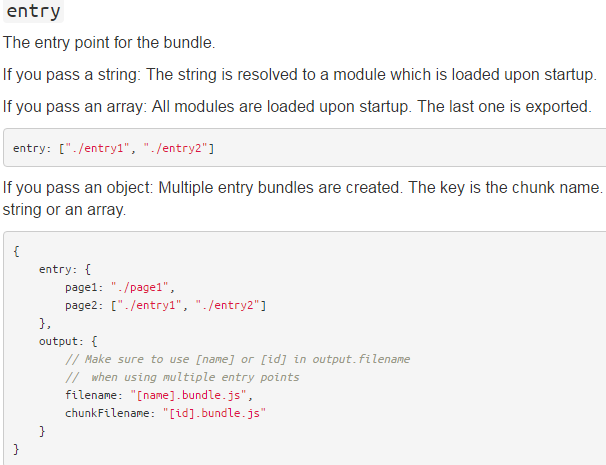
entry: {

page1: "./page1",

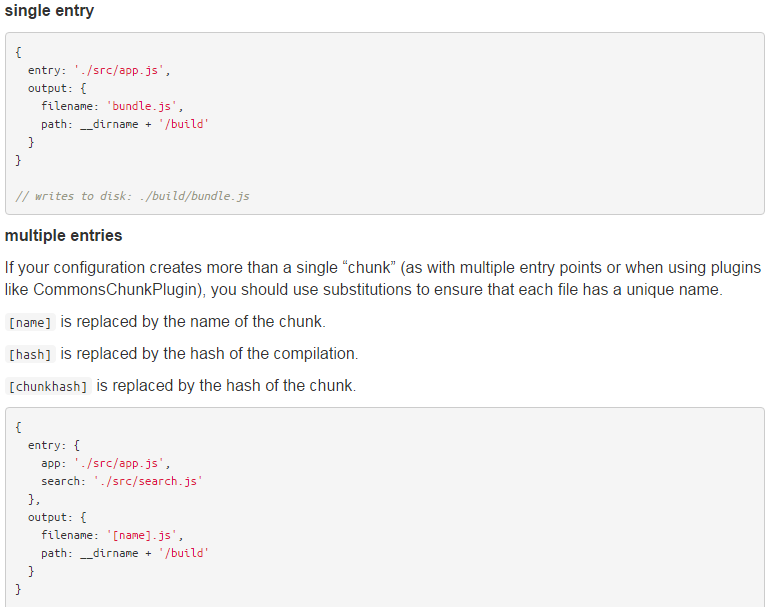
page2: ["./entry1", "./entry2"]

}

=>



--output：多个entry的时候使用占位符；



[name] is replaced by the name of the chunk.

[hash] is replaced by the hash of the compilation.本次打包的hash

[chunkhash] is replaced by the hash of the chunk.文件的版本号，md5值，保证文件的唯一性；文件修改才会改变；

{

entry: {

app: './src/app.js',

search: './src/search.js'

},

output: {

filename: '[name].js',

path: \_\_dirname + '/build'

}

}

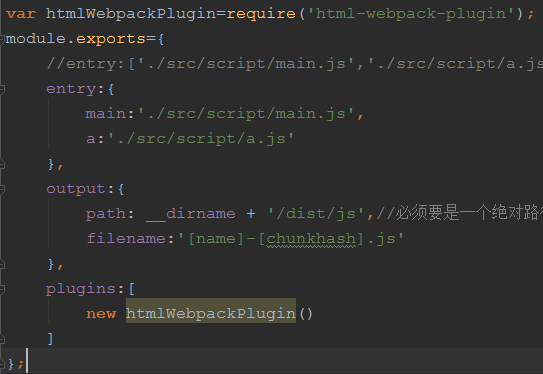
### 三、自动化生成项目中的html页面

**三、自动化生成项目中的html页面**

**1**. 现在项目下已经有了初始页面，页面引用的是bundle.js,但是我们打包后的文件名是不确定的(带hash等)，有什么办法能自动改变路径？借助插件;

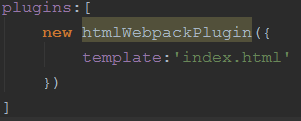
**2.** 安装npm install html-webpack-plugin –save-dev;

**3.** 配置文件中引用

官网=>using Plugins  


初始化插件，运行 npm run webpack ;发现插件生效;生成了html文件，并且对两个js进行了引用；但是这个跟我们根目录下的html文件没有什么关系，或者说怎样以根目录下的html为模板来生成这个html?

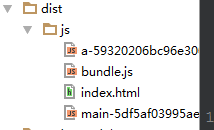
**4**. 给插件传参；传一个模板；



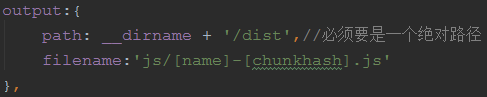
路径：为什么能确定这个index.html是模板html => 上下文概念；在配置里有一个**context属性（默认根目录）**

在运行npm run webpack ；发现生成的html具有模板的内容，还引入了打包生成的文件；试着改变模板；运行npm run webpack；生成的html也改变了。

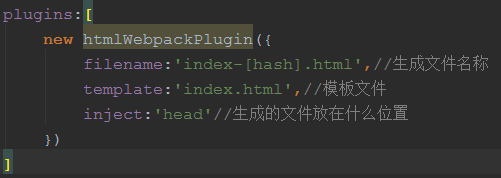
**5**. 现在有一个问题？生成的所有文件都在同一个文件夹，不符合开发模式



因为我们output 的path 就是指向同一个文件夹；修改path



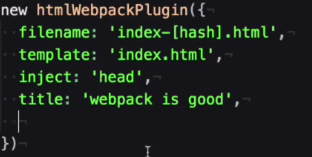
**6**.html-webpack-plugin的其他参数



7.怎样在配置文件传参，然后在模板引用；

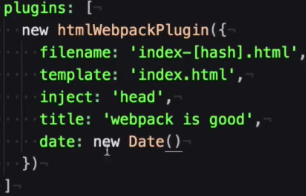
使用**js模板语言=> ejs语法 <%= htmlWebpackPlugin.options.title %>**





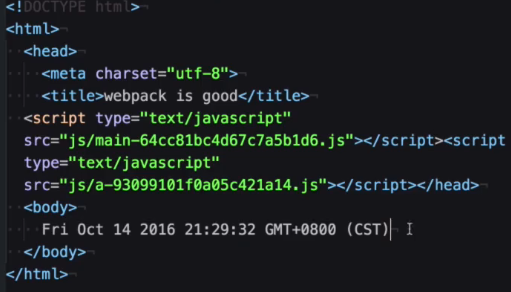
Npm run webpack;

是否任何一个变量都能传到页面呢？





Npm run webpack

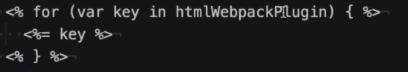


这样我们可以很轻松的配置我们的模板，得到我们想要的html;

我们究竟能够从htmlWebpackPlugin上取到哪些信息呢？

对htmlWebpackPlugin进行一个遍历：

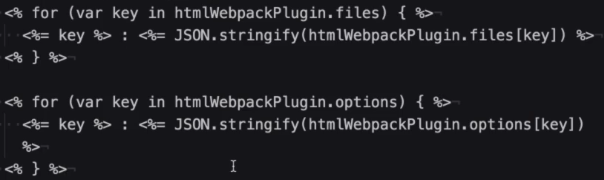
1）.先遍历key



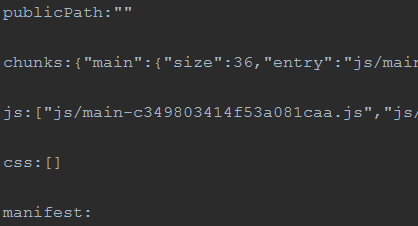
打包后得key值：

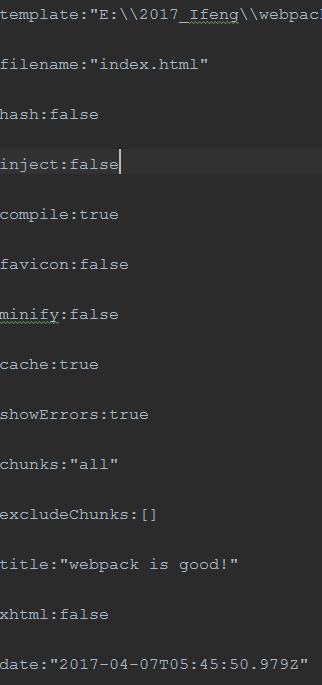


2).分别遍历key值



打包；



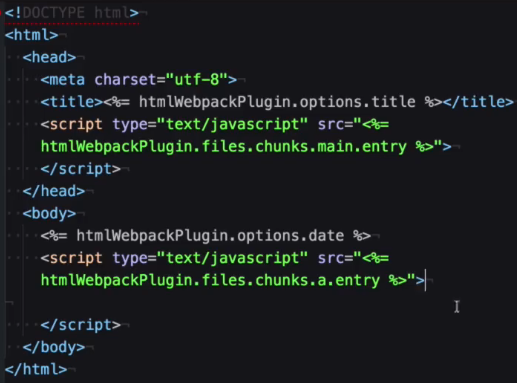


打开<https://www.npmjs.com/package/html-webpack-plugin> ；

可以看见参数的详细解释；

通过这些可以控制输出的html的内容；

--比如我们想把一部分js放在head里，一部分放在body里；只通过配置是做不到的，要改变模板；因为模板可以直接引用打包后的信息；



在配置中 将**inject:false ;**

打包；

-- 再比如我们项目要上线，上线后地址跟我们本地的相对路径肯定不一样，借助新属性：pubicPath (上线后地址)

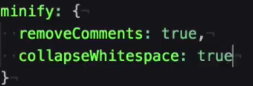


打包后就是线上地址



--项目上线html压缩，minify

<https://github.com/kangax/html-minifier#options-quick-reference>

删除注释；删除空格；

打包；

**8. 之前我们提到的都是单页面应用，现在我们看下多页面应用：**

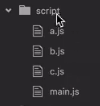
1). entry可以传一个对象，可以用它还处理多页面应用；

2). 多页面需要多个html, plugins是一个数组，可以继续调用一次插件htmlWebpackPlugin

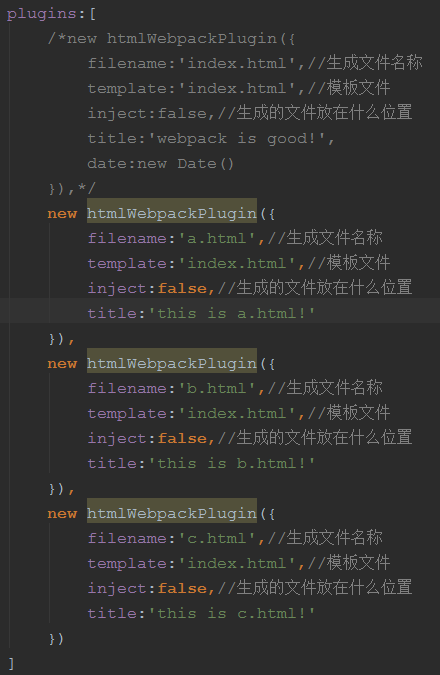
--添加多页面入口：



--在scripts文件夹下增加a.js b.js c.js;让entry跟script文件夹下是一一对应的；



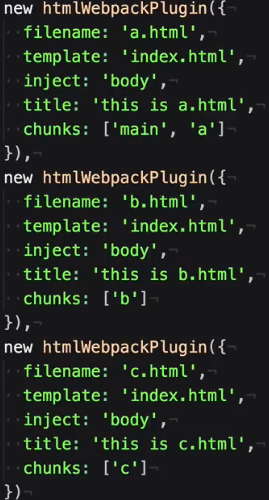
--增加chunk（页面插件）的调用；



运行；这时发现三个页面的内容除了title不一样 ，其他内容都一样；引用相同的js(因为模板是这么设置的);怎样让三个页面引用各自的js呢？

* htmlWebpackPlugin提供了一个参数chunks

**chunks**: Allows you to add only some chunks (e.g. only the unit-test chunk)，允许添加一些对自己有用的chunk



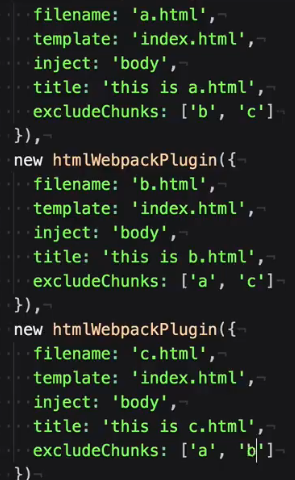
将模板html指定引入的js去掉;

运行；

这时每个页面都引入自己需要的js ,生成的html文件和我们的入口文件的chunk一一对应上了；

* htmlWebpackPlugin提供了还提供了另外一个参数**excludeChunks** **排除法**

excludeChunks: Allows you to skip some chunks (e.g. don't add the unit-test chunk) 除了那些chunk被排除之外；其他的chunk都被引入进来；



这样可以自由的匹配每个页面需要的chunk和要排除的chunk

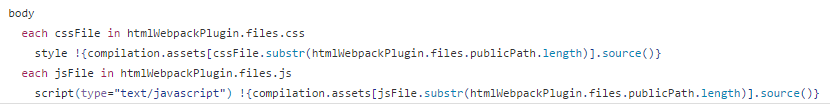
3).如果我们想提高性能，比如把一些初始化的脚本直接嵌入到页面；减少http请求；提高脚本的运行速度；开始webpack并没有这个功能，后来有人在githup上不断地提这个需求：

<https://github.com/jantimon/html-webpack-plugin>









Compilation：webpack自身暴露出来的 ，给我们使用的一个引用；

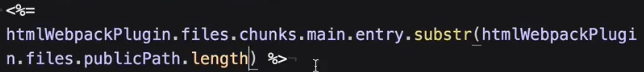
Assets：打包以后生成的文件的对象；**通过文件名的路径进这个对象中，可以拿到这个文件的索引**；

source()：拿到js文件的内容；

上次演示过在模板中取chunk文件路径（带publicPath）：



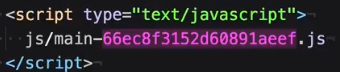
但是 compilation.assets的key文件名并不是带publicPath，所以要把entry里的publicPath过滤掉；



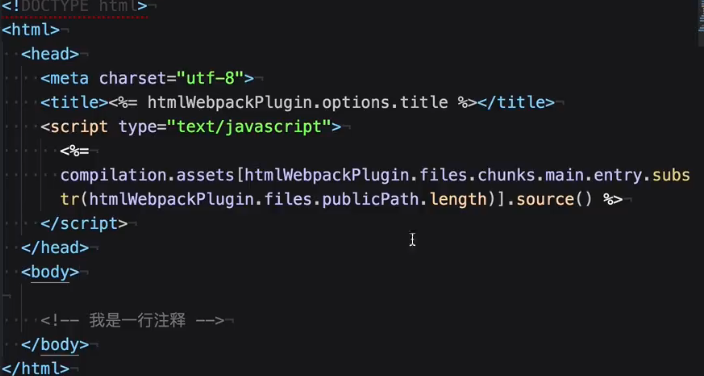
使用拿到publicPath的长度；

运行；

这时得到啦不带publicPath的路径

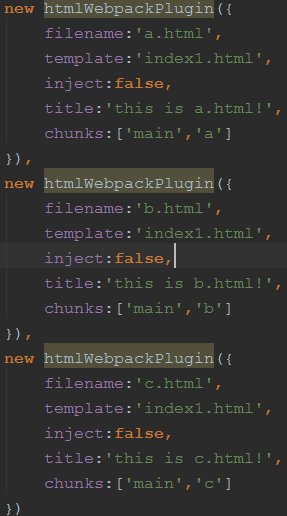
；

然后使用官方给的方法：



运行；

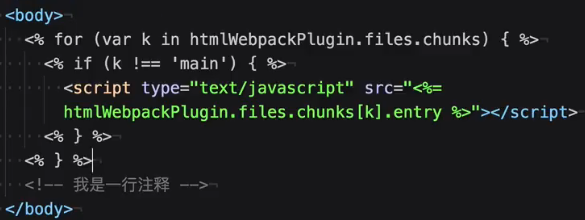
(如果报错)检查下：出错，entry未定义，原因是每个都需要引入main.js文件



运行；

Main里面的内容已经被插入到script标签来啦；但是chunk中mian的 src也被插入进来；修改模板和参数：



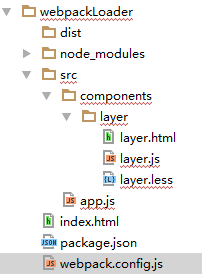


运行；ok!

### 四、处理项目中的资源文件：es6、css、图片压缩、图片转成base64的编码形式

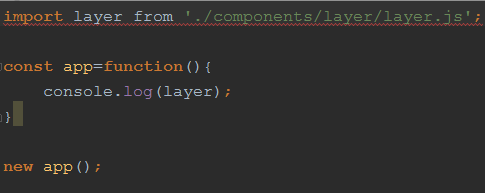
**四．处理项目中的资源文件：es6、css、图片压缩、图片转成base64的编码形式**

新建目录



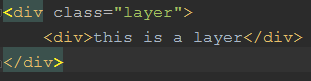
Coponents : 放需要的应用组件，创建layer组件；

--App.js ： 应用的入口

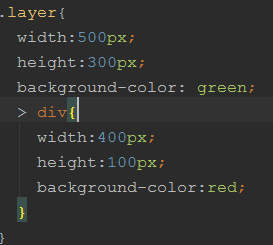


--创建layer组件：

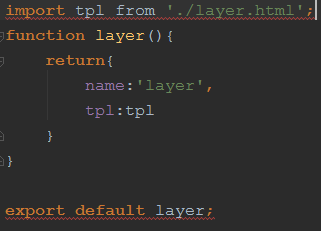
Layer.html



Layer.less



Layer.js



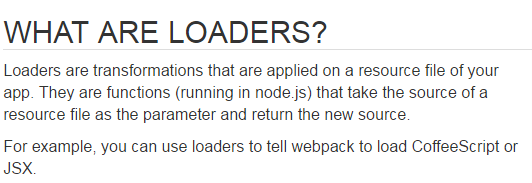
--修改配置文件：



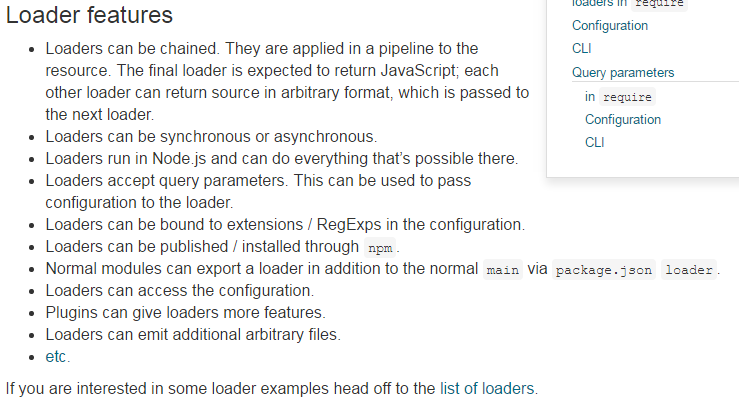
\*记得安装插件html-webpack-plugins;

**loader**

官网 => Using Loaders



处理资源文件；接受一个资源文件作为参数，处理完返回一个资源文件；比如 CoffeeScript or JSX.这两种原本webpack不支持的格式；



特性：

--串联

--同步和异步

--nodejs环境下运行

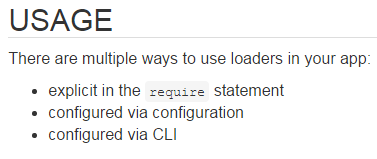
--loader可以接受参数

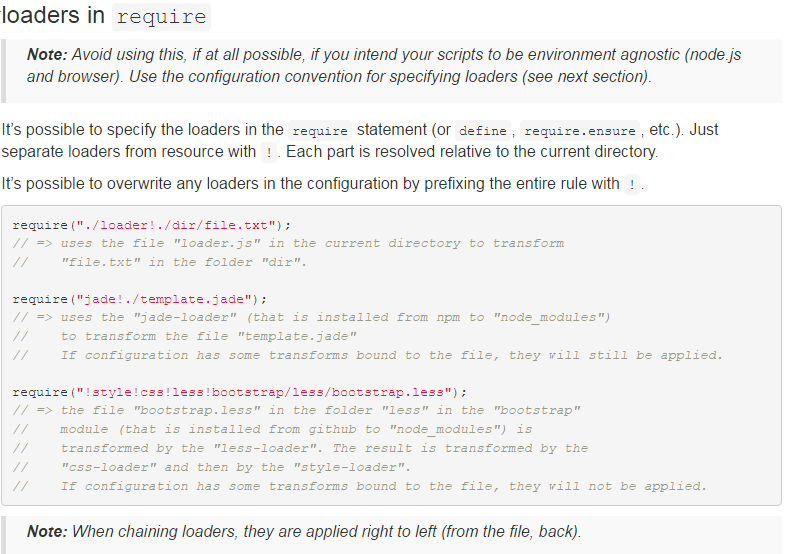
--可以通过正则指定文件后缀名，某一个后缀名文件用哪个loader

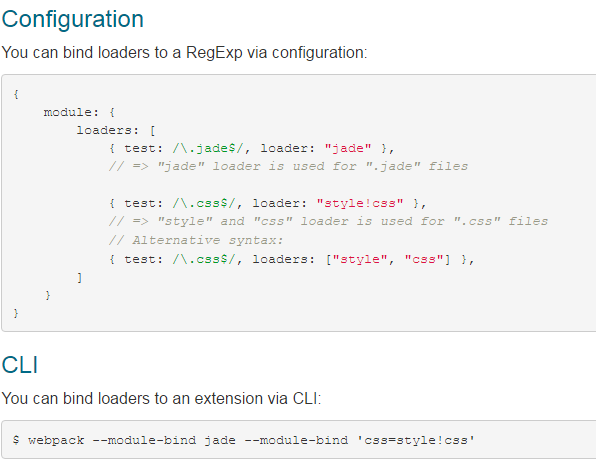
--可以通过npm安装

--可以获取webpack的配置，插件可以给loader更多的特性，loader可以生产额外的文件

**使用loader:**





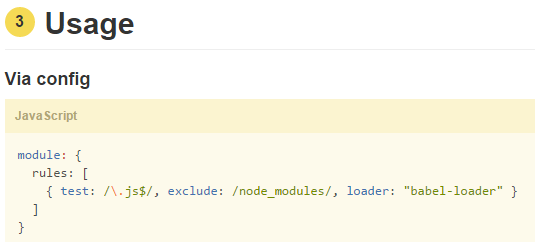


打开我们的配置文件：=>添加loader

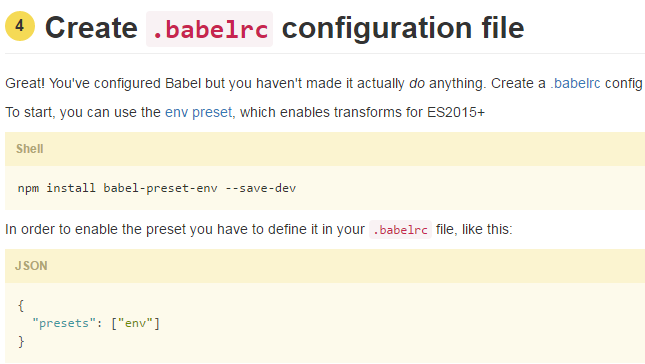
**1.将es6转es5 使用babel**： <http://babeljs.io/docs/setup/#installation>



安装



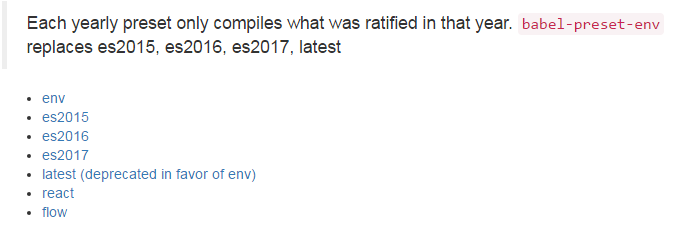
还需要一个配置：**preset**：例如：



先看一下插件：



这个插件有什么用呢，我们知道es6每年都在不断的更新、修订等，



每年的修订都会有一个当年的知识的列表；我们点开es2015看一下：



这都是es6的一个特性；

**我们可以通过指定 Presets来告诉babel-loader，让babel-loader知道，来转换某一些特性为我们的js, 如果需要它转换成所有的特性，可以指定为 latest (包括2015.2016.2017)；怎样指定插件？** 涉及到参数问题，

**给loader指定参数：**

* webpack官网给了两种方法：

一种是在loader后面跟个问号？ 后面跟指定参数；

另外一种在配置文件中，加一个query属性，指定参数

=>

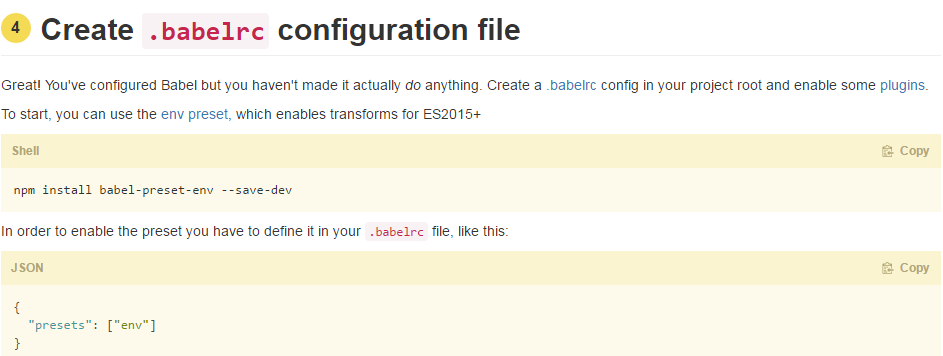




但是**首先要安装preset这个插件；**



* Babel官网还给了一种配置方式：



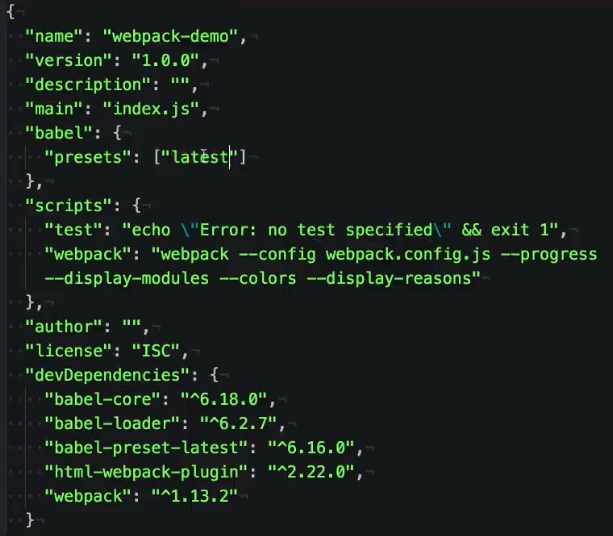
在项目的根目录下建.babelrc这个文件，内容是

{

"presets": ["env"]

}

* 还有一种方式是在我们package.json文件这种配置：

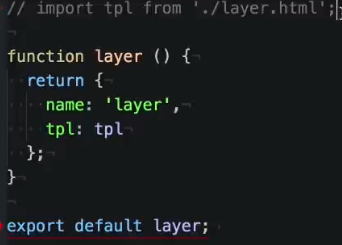


配置好了这些东西 运行一下试试;

发现报错：

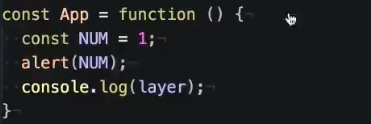


先注释掉：



再运行:检查生成的html 和js ;

在浏览器看下生成的html,以及html的运行情况；修改一下试试：



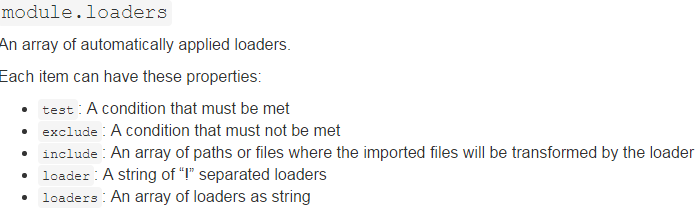
为了更清晰的知道loader应用的哪个参数，我们配置用以下这种方式：



刚刚打包的过程中，发现打包速度非常慢，因为babel loader 处理非常耗时：

怎样改善？

Webpack官网=> => module.loaders



Exclude:排除范围



排除这个文件，不让它处理，减少打包时间；

其实发现，现在的打包速度还是很慢；

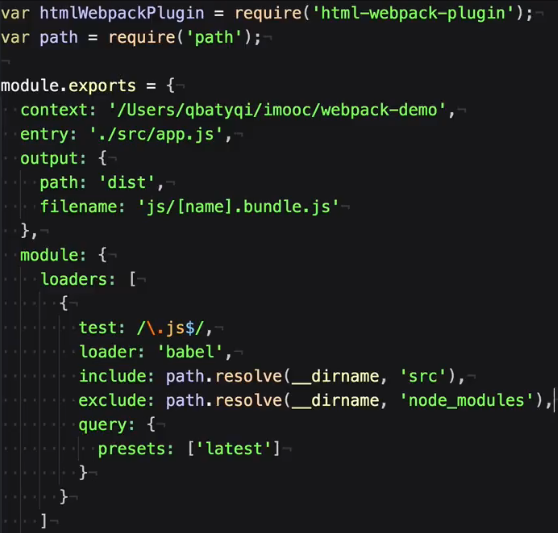
我们还可以指定打包范围：



再试一下；

发现报错，因为路径必须是一个绝对路径：





**2.处理项目中的css预处理语言;**

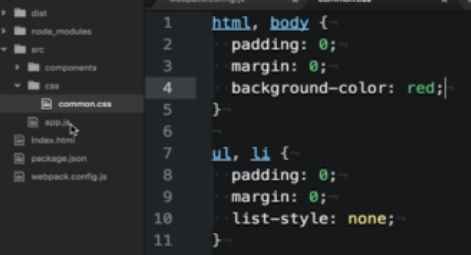
**Css-loader style-loader**

**先安装这两个loader**



1). 先学习怎么处理css文件：

-- 在src下新建css文件夹（用来放项目中公用的css），新建common.css样式文件；



* 在入口文件对common.css进行引用 => 用import或require;



* 配置css的loader



定义好了loader,开始打包：看看我们设置的样式有没有生效

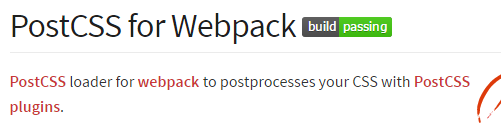
接下来看下css-loader其他的情况，项目中会用到现在浏览器都支持的不太好的属性，比如flex;



我们需要手动添加前缀，

**也可以使用loader: postcss-loader 先安装**；

打开npm官网,搜下postcss-loader



它是一个后处理器，结合它的插件可以做很多事情；比如autoprefixer，

先安装

装好之后怎么使用：

在css的loader中加上刚刚安装的postcss-loader，loader的处理方式是从右到左：

官网给的例子：



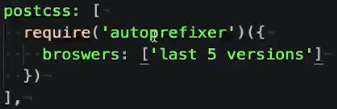


Autoprefixer插件如何使用？

看官网：



这是webpack1给的方案；



最近的5个浏览器的版本加前缀；

运行发现会报错：

换一种方法：



这样就ok了！

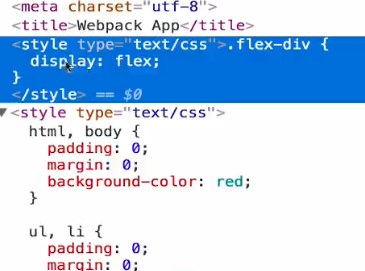
Postcss有很多的插件；

点击进入看一下；

* 在css中我们会用 @import ‘’ 引入一个文件



运行一下看看webpack怎么处理的；



发现flex.css虽然插入到了页面中，但是并没有加前缀，也就是经过了css-loader和style-loader,但是并没有经过postcss-loader;怎么办？

使用css-loader的一个参数，之前介绍过loader传参的方式，用query或者？；



在css-loader之后,来指定,几个数量的loader来处理@import进来的资源；

再运行；

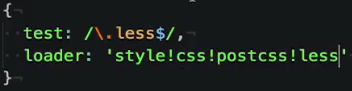
**2). 处理.Less文件**；

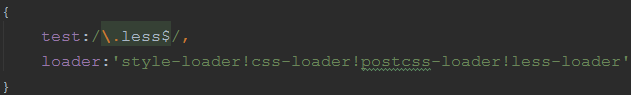
先安装 less-loader ,没安装less的要安装less

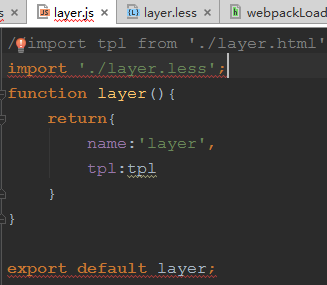
新增匹配规则：

Add PostCSS Loader to webpack.config.js.

Put it after css-loader and style-loader. But before sass-loader, if you use it.

如果报错 ，写全：





打包；

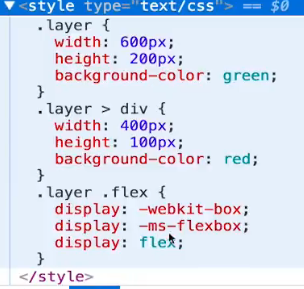
看一下打包生成的页面；发现layer.less的内容已经处理到页面；

修改layer.less



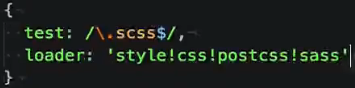
打包；

发现生效：



* 在Less中也使用@import ‘’ : 运行；发现我们没加less已经帮我们处理了 @import ‘’;
* 如果我们项目中要使用sass，安装sass-loader





同less

### 五、处理项目中的组件模板文件

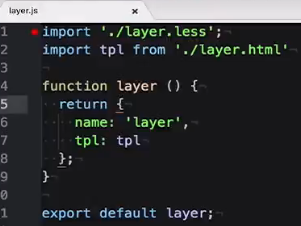
**五、处理项目中的组件模板文件**

**例如：layer.html**

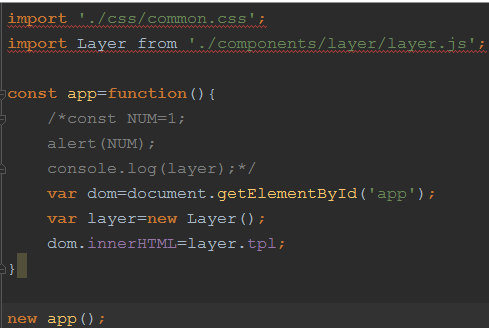
**官网=》****=》**

**1）.使webpack，让他把模板文件当字符串处理**

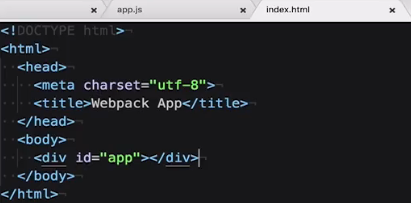
在layer.js里面把模板引入进来



在app.js里面操作：



模板文件添加div

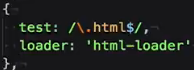


运行有一个错误；需要安装一个loader来处理他；

安装：



添加匹配规则：

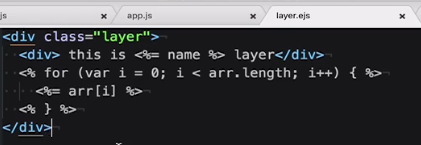


再运行；修改一下看看生成页面变化；

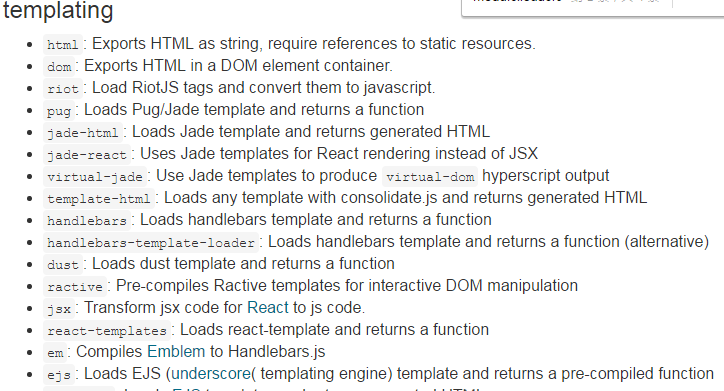
**2）.使webpack，让他把模板文件当已经编译好的模板的处理函数**

* 有时候我们的模板会用到一些语法，比如ejs语法：

建一个layer.ejs 模板



看官网ejs说明：



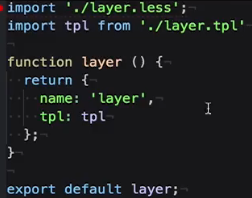
载入 ejs模板，并返回一个已经编译好函数；

安装 ejs-loader 

规则：

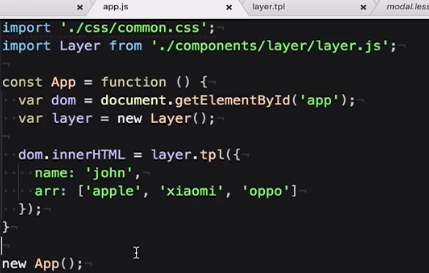
可以**把layer.ejs改成layer.tpl**



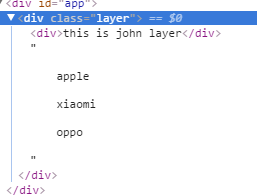


这个时候的tpl已经是一个已经编译好函数；

可以用这个函数传入模板中要用到的数据：



再打包一下，发现模板被更新，内容也已经展示出来。



**现在比较流行的loader**

**已经被集成到babel；无论vue2.0，还是react, 都可以通过babel这个工具支持jsx的语法**

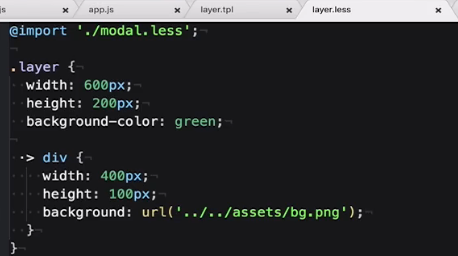
### 六、处理图片及其他文件

**六、处理图片及其他文件**

**1.图片文件：**

**在src下新建文件夹assets，放入一张图片：**

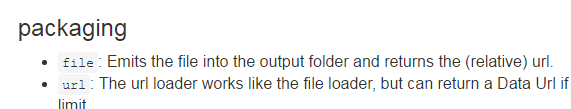
1）.css背景中的图片



打包发现报错，并且打包过程中并没有图片：

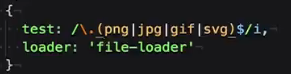
**安装图片处理loader**

**Webpack官网=>****=>**



**第一种：file-loader;**





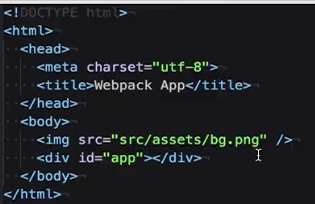
打包；打开html,ok!

打包过程中生成的路径

2）.根目录下模板图片url

-- 如果引用的是绝对路径的图片没有问题；

--如果引用的是项目上相对路径的图片：

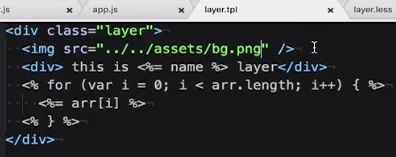


图片路径也会被替换；

3）.组件模板比如layer中加url

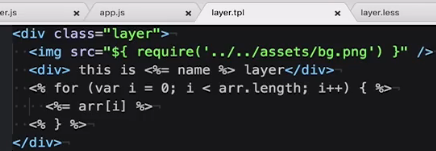
-- 如果引用的是绝对路径的图片没有问题；

--如果引用的是项目上相对路径的图片：（会碰到一些问题）



运行；发现路径没有被替换；怎么解决？

* 尽可能使用绝对路径
* 或者

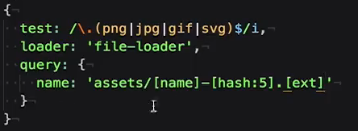


如果我们想改变图片输出地址：给loader加参数

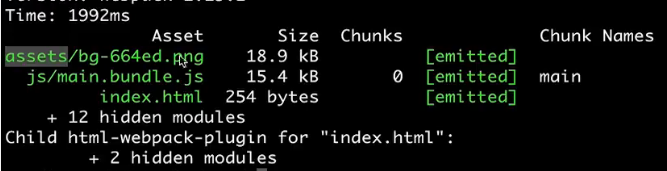
点进去：



Name中也有占位符：



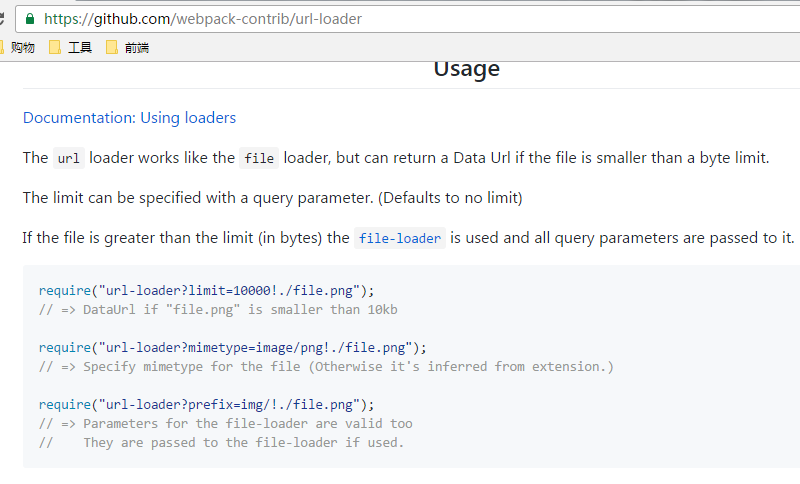
打包



发现图片已经被我们打包到assets文件夹下，文件名也变了；

**图片处理的另外一种url-loader:**





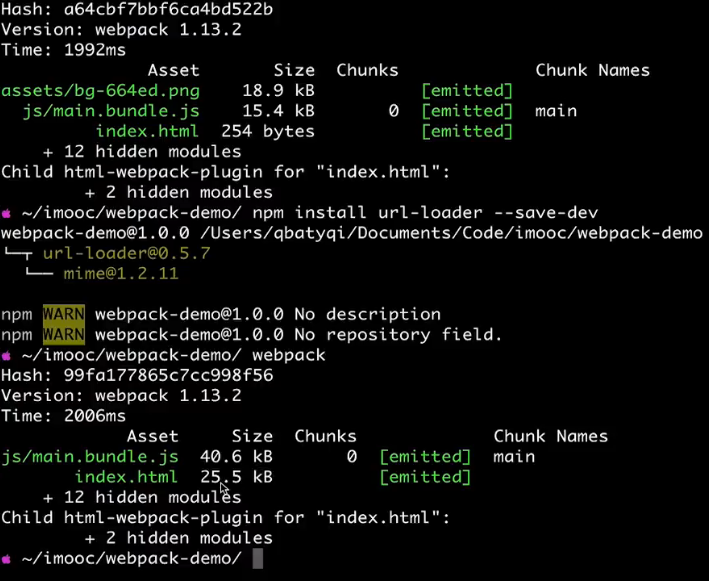
**url-loader跟 file-loader相似，但是url-loader可以处理文件和图片；**

**当你的图片或者文件大于指定的limit, url-loader会丢给file-loader处理；**

**当你的图片或者文件小于指定的limit，会把图片和文件处理成base64的编码；不再是url,试一下：**



打包



发现图片没有再被打包；但是js，html文件变大,看下页面：

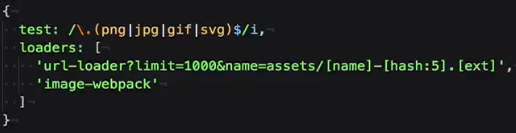


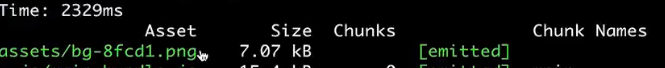
变成了一长串的编码；所以文件才会变大；（从某种程度上增加了代码量）

* 压缩图片



$ npm install image-webpack-loader --save-dev





图片变小；

Image-webpack-loader还有一些更复杂的参数：

https://github.com/tcoopman/image-webpack-loader

报错：换一种方法：

* [img](https://github.com/thetalecrafter/img-loader): Load and compress images with imagemin.



### Webpack的强大功能

### 生成Source Maps（使调试更容易）

开发总是离不开调试，如果可以更加方便的调试当然就能提高开发效率，不过打包后的文件有时候你是不容易找到出错了的地方对应的源代码的位置的，Source Maps就是来帮我们解决这个问题的。  
通过简单的配置后，Webpack在打包时可以为我们生成的source maps，这为我们提供了一种对应编译文件和源文件的方法，使得编译后的代码可读性更高，也更容易调试。

在webpack的配置文件中配置source maps，需要配置devtool，它有以下四种不同的配置选项，各具优缺点，描述如下：

| **devtool选项** | **配置结果** |
| --- | --- |
| source-map | 在一个单独的文件中产生一个完整且功能完全的文件。这个文件具有最好的source map，但是它会减慢打包文件的构建速度； |
| cheap-module-source-map | 在一个单独的文件中生成一个不带列映射的map，不带列映射提高项目构建速度，但是也使得浏览器开发者工具只能对应到具体的行，不能对应到具体的列（符号），会对调试造成不便； |
| eval-source-map | 使用eval打包源文件模块，在同一个文件中生成干净的完整的source map。这个选项可以在不影响构建速度的前提下生成完整的sourcemap，但是对打包后输出的JS文件的执行具有性能和安全的隐患。不过在开发阶段这是一个非常好的选项，但是在生产阶段一定不要用这个选项； |
| cheap-module-eval-source-map | 这是在打包文件时最快的生成source map的方法，生成的Source Map 会和打包后的JavaScript文件同行显示，没有列映射，和eval-source-map选项具有相似的缺点； |

正如上表所述，上述选项由上到下打包速度越来越快，不过同时也具有越来越多的负面作用，较快的构建速度的后果就是对打包后的文件的的执行有一定影响。

在学习阶段以及在小到中性的项目上，eval-source-map是一个很好的选项，不过记得只在开发阶段使用它，继续上面的例子，进行如下配置

module.exports = {

devtool: 'eval-source-map',//配置生成Source Maps，选择合适的选项

entry: \_\_dirname + "/app/main.js",

output: {

path: \_\_dirname + "/public",

filename: "bundle.js"

}

}

cheap-module-eval-source-map方法构建速度更快，但是不利于调试，推荐在大型项目考虑da时间成本是使用。

### 使用webpack构建本地服务器

想不想让你的浏览器监测你**代码的修改**，并自动刷新修改后的结果，其实Webpack提供一个可选的本地开发服务器，这个本地服务器基于node.js构建，可以实现你想要的这些功能，不过它是一个单独的组件，在webpack中进行配置之前需要单独安装它作为项目依赖

npm install webpack-dev-server –g

在package.json里配置：

"dev": "webpack-dev-server --port 8080 --colors --progress --hot --inline"

这里**使用localhost:8080访问**

或者直接命令行；

webpack-dev-server --port 8080 --colors --progress --hot –inline

如果要**使用IP**访问：加参数：

webpack-dev-server --inline --port 8080 --host 172.17.20.20

devserver作为webpack配置选项中的一项，具有以下配置选项

| **devserver配置选项** | **功能描述** |
| --- | --- |
| contentBase | 默认webpack-dev-server会为根文件夹提供本地服务器，如果想为另外一个目录下的文件提供本地服务器，应该在这里设置其所在目录（本例设置到“public"目录） |
| port | 设置默认监听端口，如果省略，默认为”8080“ |
| inline | 设置为true，当源文件改变时会自动刷新页面 |
| colors | 设置为true，使终端输出的文件为彩色的 |
| historyApiFallback | 在开发单页应用时非常有用，它依赖于HTML5 history API，如果设置为true，所有的跳转将指向index.html |

#### http://www.jianshu.com/p/42e11515c10f#

#### Hot Module Replacement

Hot Module Replacement（HMR）也是webpack里很有用的一个插件，它允许你在修改组件代码后，自动刷新实时预览修改后的效果。  
在webpack中实现HMR也很简单，只需要做两项配置

1. 在webpack配置文件中添加HMR插件；
2. 在Webpack Dev Server中添加“hot”参数；

不过配置完这些后，JS模块其实还是不能自动热加载的，还需要在你的JS模块中执行一个Webpack提供的API才能实现热加载，虽然这个API不难使用，但是如果是React模块，使用我们已经熟悉的Babel可以更方便的实现功能热加载。

整理下我们的思路，具体实现方法如下

* Babel和webpack是独立的工具
* 二者可以一起工作
* 二者都可以通过插件拓展功能
* HMR是一个webpack插件，它让你能浏览器中实时观察模块修改后的效果，但是如果你想让它工作，需要对模块进行额外的配额；
* Babel有一个叫做react-transform-hrm的插件，可以在不对React模块进行额外的配置的前提下让HMR正常工作；

更新我们的例子来实际看看如何配置

//webpack中的配置

var webpack = require('webpack');

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

module.exports = {

devtool: 'eval-source-map',

entry: \_\_dirname + "/app/main.js",

output: {

path: \_\_dirname + "/build",

filename: "bundle.js"

},

module: {

loaders: [

{ test: /\.json$/, loader: "json" },

{ test: /\.js$/, exclude: /node\_modules/, loader: 'babel' },

{ test: /\.css$/, loader: 'style!css?modules!postcss' }

]

},

postcss: [

require('autoprefixer')

],

plugins: [

new HtmlWebpackPlugin({

template: \_\_dirname + "/app/index.tmpl.html"

}),

new webpack.HotModuleReplacementPlugin()//热加载插件

],

devServer: {

colors: true,

historyApiFallback: true,

inline: true,

hot: true

}

}

安装react-transform-hmr

npm install --save-dev babel-plugin-react-transform react-transform-hmr

配置Babel

{

"presets": ["react", "es2015"],

"env": {

"development": {

"plugins": [["react-transform", {

"transforms": [{

"transform": "react-transform-hmr",

"imports": ["react"],

"locals": ["module"]

}]

}]]

}

}

}

现在当你使用React时，可以热加载模块了

### 产品阶段的构建

目前为止，我们已经使用webpack构建了一个完整的开发环境。但是在产品阶段，可能还需要对打包的文件进行额外的处理，比如说优化，压缩，缓存以及分离CSS和JS。

对于复杂的项目来说，需要复杂的配置，这时候分解配置文件为多个小的文件可以使得事情井井有条，以上面的例子来说，我们创建一个“webpack.production.config.js”的文件，在里面加上基本的配置,它和原始的webpack.config.js很像，如下

var webpack = require('webpack');

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

module.exports = {

entry: \_\_dirname + "/app/main.js",

output: {

path: \_\_dirname + "/build",

filename: "bundle.js"

},

module: {

loaders: [

{

test: /\.json$/,

loader: "json"

},

{

test: /\.js$/,

exclude: /node\_modules/,

loader: 'babel'

},

{

test: /\.css$/,

loader: 'style!css?modules!postcss'

}

]

},

postcss: [

require('autoprefixer')

],

plugins: [

new HtmlWebpackPlugin({

template: \_\_dirname + "/app/index.tmpl.html"

}),

],

}

//package.json

{

"name": "webpack-sample-project",

"version": "1.0.0",

"description": "Sample webpack project",

"scripts": {

"start": "webpack-dev-server --progress",

"build": "NODE\_ENV=production webpack --config ./webpack.production.config.js --progress"

},

"author": "Cássio Zen",

"license": "ISC",

"devDependencies": {...},

"dependencies": {...}

}

#### 优化插件

webpack提供了一些在发布阶段非常有用的优化插件，它们大多来自于webpack社区，可以通过npm安装，通过以下插件可以完成产品发布阶段所需的功能

* OccurenceOrderPlugin :为组件分配ID，通过这个插件webpack可以分析和优先考虑使用最多的模块，并为它们分配最小的ID
* UglifyJsPlugin：压缩JS代码；
* ExtractTextPlugin：分离CSS和JS文件

我们继续用例子来看看如何添加它们，OccurenceOrder 和 UglifyJS plugins 都是内置插件，你需要做的只是安装它们

npm install --save-dev extract-text-webpack-plugin

在配置文件的plugins后引用它们

var webpack = require('webpack');

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

var ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {

entry: \_\_dirname + "/app/main.js",

output: {

path: \_\_dirname + "/build",

filename: "bundle.js"

},

module: {

loaders: [

{

test: /\.json$/,

loader: "json"

},

{

test: /\.js$/,

exclude: /node\_modules/,

loader: 'babel'

},

{

test: /\.css$/,

loader: ExtractTextPlugin.extract('style', 'css?modules!postcss')

}

]

},

postcss: [

require('autoprefixer')

],

plugins: [

new HtmlWebpackPlugin({

template: \_\_dirname + "/app/index.tmpl.html"

}),

new webpack.optimize.OccurenceOrderPlugin(),

new webpack.optimize.UglifyJsPlugin(),

new ExtractTextPlugin("style.css")

]

}

#### 缓存

缓存无处不在，使用缓存的最好方法是保证你的文件名和文件内容是匹配的（内容改变，名称相应改变）

webpack可以把一个哈希值添加到打包的文件名中，使用方法如下,添加特殊的字符串混合体（[name], [id] and [hash]）到输出文件名前

var webpack = require('webpack');

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

var ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {

entry: \_\_dirname + "/app/main.js",

output: {

path: \_\_dirname + "/build",

filename: "[name]-[hash].js"

},

module: {

loaders: [

{

test: /\.json$/,

loader: "json"

},

{

test: /\.js$/,

exclude: /node\_modules/,

loader: 'babel'

},

{

test: /\.css$/,

loader: ExtractTextPlugin.extract('style', 'css?modules!postcss')

}

]

},

postcss: [

require('autoprefixer')

],

plugins: [

new HtmlWebpackPlugin({

template: \_\_dirname + "/app/index.tmpl.html"

}),

new webpack.optimize.OccurenceOrderPlugin(),

new webpack.optimize.UglifyJsPlugin(),

new ExtractTextPlugin("[name]-[hash].css")

]

}

现在用户会有合理的缓存了。