Webapck

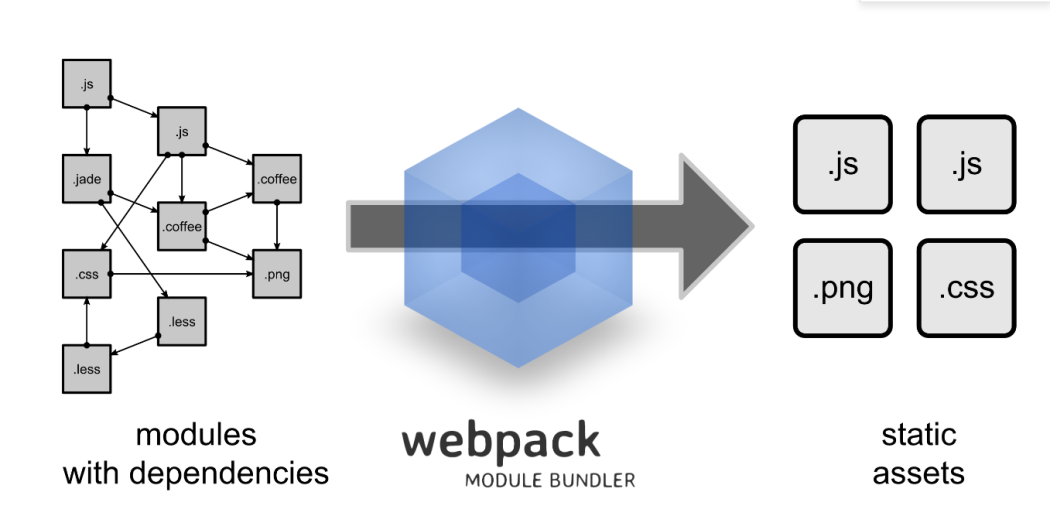
## Webapck的基本介绍

官网地址<http://webpack.github.io/docs/>

### 1.1.大致的打包过程

通过官网的 ‘what is Webpack’我们可以看到它将文件看作是一个个的模块，他们直接的箭头可以看成他们之间的依赖关系。

这样相互依赖的模块群通过webpack的打包处理将它们打包成浏览器可以直接运行额js，css或者png文件



### 1.2.目标：

* 切分代码树，将代码树切分到不同的代码块里面，按需加载依赖（和前端的懒加载相似）
* 保持初始化加载的时间更短
* 任何静态资源都可以被视为一个模块，在项目中被引用
* 可以整合第三方的类库，并且将其视为模块在项目中引用
* 在打包的过程中任何一份都可以自定义
* 非常适合大型项目

### 1.3和其他打包工具的不同

**1.代码分割**

webpack依赖树有两种类型的依赖关系:同步和异步。异步依赖性作为分割点,形成一个新块。块树优化后,每个块的文件发出。

**2.Loaders （**[**装载机**](http://webpack.github.io/docs/loaders.html)**）**

webpack只能自己处理JavaScript，但是加载器用于将其他资源转换为JavaScript。通过这样做，每个资源组成一个模块。

**3.Clever parsing（聪明解析）**

webpack有一个聪明的解析器，可以处理几乎每一个第三方库。它甚至允许在依赖关系中的表达式，如：require("./templates/" + name + ".jade")。它处理最常见的模块样式：[CommonJs](http://webpack.github.io/docs/commonjs.html)和[AMD](http://webpack.github.io/docs/amd.html)。

**4.Plugin System（**[**插件系统**](http://webpack.github.io/docs/plugins.html)**）**

webpack具有丰富的插件系统。大多数内部功能都是基于这个插件系统。这允许您根据需要自定义webpack，并将普通的插件作为开放源代码分发。

## Webpack的安装和命令行

### 1.安装

npm install webpack -g 全局安装webpack

webpack -help 查看是否安装，以及安装的版本

npm init 建立一个文件后

npm install webpack --save-dev 在局部安装webpack

（注意执行npm install webpack --save-dev命令时由npm init生成的package.json的name不能叫webpack，否则会报错）

### 2.命令行

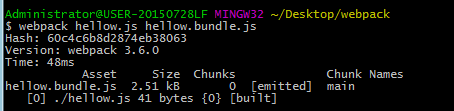
#### 1.编辑打包

##### 1.1基本打包

我们在目录中新建hellow.Js,对齐进行打包，打包后的文件叫hellow.bundle.js

webpack hellow.js hellow.bundle.js

**结果为：**



执行完命令后出现列表，

Hash；

Verison：webapck的版本号；

Time：打包花费的时间

|  |  |  |  |
| --- | --- | --- | --- |
| Asset（打包后的文件） | Size（文件的大小） | Chunks（打包的分块） | Chunk Names（这次打包的块名称） |
| hellow.bundle.js | 2.51KB | 0 | Main |

Hellow.js:

1. function say(str){
2. console.log(str);
3. }

生成的hellow.bundle.js的解析：

1. //打包过程中会生成webpack所需要的一些函数
2. (function(modules) { // webpackBootstrap
3. 。。。。。。。。
4. })
5. /\*\*\*\*\*\*/ ([
6. // 对依赖的模块（就是我们hellow.js中的代码）进行编号
7. /\* 0 \*/
8. /\*\*\*/ (function(module, exports) {
9. function say(str){
10. console.log(str);
11. }
12. /\*\*\*/ })
13. /\*\*\*\*\*\*/ ]);

打包—案例1

##### 1.2 A引用B.js，打包A

**World.js:**

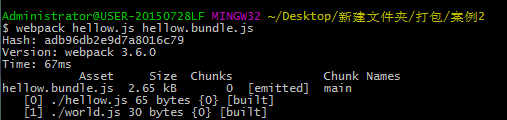
1. function add(){
2. return {}
3. }

Hellow.js

1. require('./world.js');
2. function say(str){
3. console.log(str);
4. }

执行 webpack hellow.js hellow.bundle.js 命令

结果：



可以看到【0】为hellow.js [1]为world.js

生成hellow.bundle.js

1. //打包过程中引入webpack的依赖
2. /\*\*\*\*\*\*/ (function(modules) { // webpackBootstrap
3. })
4. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
5. /\*\*\*\*\*\*/ ([
6. //对依赖的模块进行编号
7. /\* 0 \*/
8. /\*\*\*/ (function(module, exports, \_\_webpack\_require\_\_) {
9. //提示在0模块中引用了1模块
10. //hellow.js的内容
11. \_\_webpack\_require\_\_(1);
12. function say(str){
13. console.log(str);
14. }
15. /\*\*\*/ }),
16. /\* 1 \*/
17. /\*\*\*/ (function(module, exports) {
18. //world.js的内容
19. function add(){
20. return {}
21. }
22. /\*\*\*/ })
23. /\*\*\*\*\*\*/ ]);

打包—案例2

##### 1.3引入css文件，并且展示在HTML中

这是我们需要安装两个loader

然后用命令行加上一个css-loader（识别css文件，否则无法识别css文件）与style-loader（应用css样式，否则HTML无法应用）

npm install css-loader style-loader --save-dev （在根目录下安装，将loader安装在依赖中）

案例：

Hellow.js

注意：style-loader!css-loader顺序不能变

1. //对引入的文件使用css-loader使得webpack能够识别css文件，在使用style-loader使得HTML能够应用css
2. require('style-loader!css-loader!./style.css');
3. function say(str){
4. console.log(str);
5. }
6. say(‘hello world’)

style.css

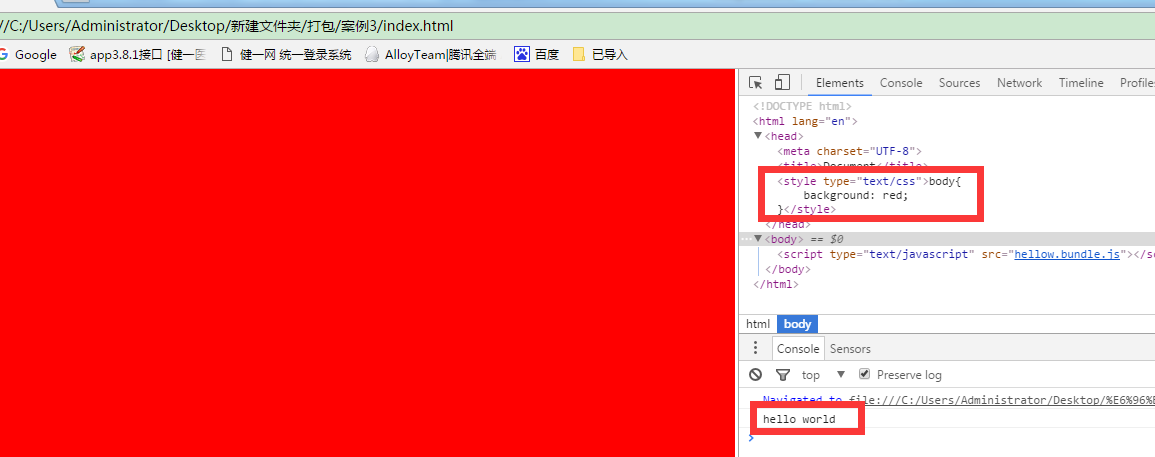
1. body{
2. background: red;
3. }

这是执行打包命令webpack hellow.js hellow.bundle.js

HTML：

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Document</title>
6. </head>
7. <body>
8. //引入 打包后的hellow.bundle.js
9. <script type="text/javascript" src='hellow.bundle.js'></script>
10. </body>
11. </html>

结果：



可以看到html因为style-loader的关系在header标签中新增了<style>标签，css起作用，且hellow.js的函数输出‘hellow world’

1. 打包—案例3

##### 1.4简单引入css文件和实时更新

看到1.3中每一次引用css我们都要先引用css-loader，再引用style-loader

1. require('style-loader!css-loader!./style.css');

这很麻烦

我们可以直接用命令行实现，不用在代码中写

webpack hellow.js hellow.bundle.js --module-bind ‘css=style-loader！css-loader！’

实时跟新：

在命令行的最后加上

--watch

就会实时更新打包了

##### 1.5常用的命令

**能看到打包过程的：**

--progress

**看打包的模块（列出所有引用模块）：**

--display-modules

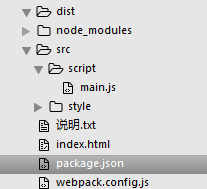
**为何打包这个模块的原因：**

--display reasons

## 三．webpack基本配置

### 1.关于webpack.config.js

首先我们先建立这样的目录



文件说明：

src 放置源文件的目录

dist 放置打包后文件的目录

index.html 初始化页面

webpack.config.js webapck的配置文件

#### 1.1 webpack.config.js的说明

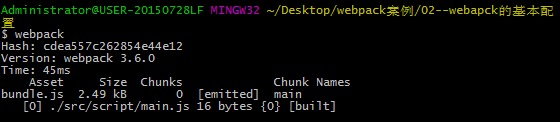
webapck的配置文件， (具体的配置在文档 API->configuration，<http://webpack.github.io/docs/configuration.html> )为何建立webpack.config.js ，是因为在命令行中执行webpack会默认寻找目录下的webpack.config.js，以webpack.config.js的配置去运行。

当然也可以webpack --config hcd.js 执行hcd.js中的配置

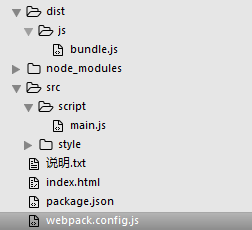
**webpack.config.js配置：**

1. //模块化输出
2. module.exports = {
3. //入口文件,一般使用绝对路径，\_\_dirname为webpack.config.js所在的文件夹
4. entry:\_\_dirname+'/src/script/main.js',
5. // 打包后的文件
6. output:{
7. //打包后文件在./dist/js的文件夹中
8. path:\_\_dirname+'/dist/js',
9. //打包后文件的名称
10. filename:'bundle.js'
11. }
12. }

**再在命令行中执行**  webpack **命令：**



**结果目录为：**



Webpack的基本配置---案例1

### 2.通过package.json简便的执行命令

首先目录和前面是一样的，webapck.config.js的配置也是一样的。

只不过可以通过在package.jsond的script中设置便捷的执行命令的方式。

1. {
2. "name": "webpack-hcd",
3. "version": "1.0.0",
4. "description": "",
5. "main": "index.js",
6. "scripts": {
7. "test": "echo \"Error: no test specified\" && exit 1",
8. "webpack":"webpack --config webpack.config.js --progress --display-modules --colors"
9. },
10. "author": "",
11. "license": "ISC",
12. "devDependencies": {
13. "webpack": "^3.6.0"
14. }
15. }

**解释：**

"webpack":"webpack --config webpack.config.js --progress --display-modules --colors"

webpack --config webpack.config.js 执行webpack.config.js

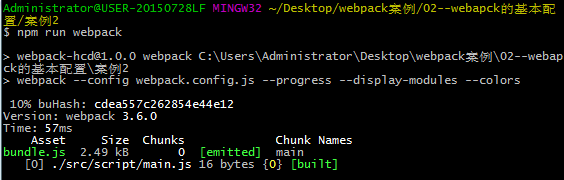
--progress 查看过程

--display-modules 打包了哪些模块

--colors 彩色的

**执行命令行：**

npm run webpack



Webpack的基本配置---案例1

### 3.entry

在上面的例子中，我们已经知道，entry是入口文件的意思，webpack关于入口文件有3种方式

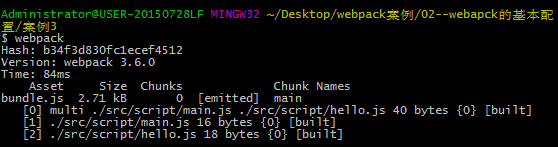
API：<http://webpack.github.io/docs/configuration.html#entry>

#### 1.引入单个文件

1. entry: "./src/js/main.js",

#### 2.引入数组

1. entry:['./src/js/main.js','./src/js/hello.js']



可以看到一共打包了【0】【1】【2】3个模块，【0】是multi，【1】，【2】才是我们想要打包的内容，这是为何呢，我们来看一下打包后的文件bundle.js



可以看到其实【0】模块是用来引用我们的【1】【2】模块的，将【1】【2】打包到一起去，这样我们就可以使用了

Webpack的基本配置---案例3

#### 3．entry对象

这种传递entry对象的方式多应用于多页面， chunk：path（模块：路径）

page1页面使用”./page1“打包后的文件，page2页面使用的是["./entry1", "./entry2"]打包后的文件

1. entry: {
2. page1: "./page1",
3. page2: ["./entry1", "./entry2"]
4. },

但是如果不更改output将会导致两个文件打包后输出的文件名字一样，造成了重复覆盖，所以要更改output，可以查看： [1.output.filename](#_1.output.filename)

Webpack的基本配置---案例5

### 4.output

API：<http://webpack.github.io/docs/configuration.html#output>

影响编译输出的选项。output选项告诉Webpack如何将编译的文件写入磁盘。请注意，虽然可以有多个entry点，但output只指定一个配置。

#### 1.output.filename

指定磁盘上每个输出文件的名称。你不能在这里指定绝对路径！该output.path选项确定磁盘上写入文件的位置。filename仅用于命名单个文件。

**单个入口文件：**

1. {
2. entry: './src/app.js',
3. output: {
4. filename: 'bundle.js',
5. path: \_\_dirname + '/build'
6. }
7. }

**多个引入文件：**

如果您的配置创建多个单个“块”（如多个入口点或使用像CommonsChunkPlugin这样的插件），则应使用替换来确保每个文件具有唯一的名称。

[name] 被块的名称所取代。（就是entry对象的chunk）

[hash] 被编译的哈希替换。（在编译的时候可以看到）

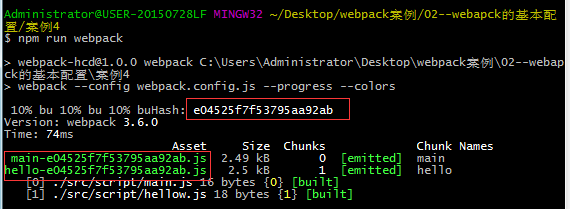
[chunkhash] 被块的哈希替代。（块的哈希值如同MD5一样，只要代码不一样，哈希就不同）

1. {
2. entry: {
3. app: './src/app.js',
4. search: './src/search.js'
5. },
6. output: {
7. filename: '[name].js',
8. path: \_\_dirname + '/build'
9. }
10. }

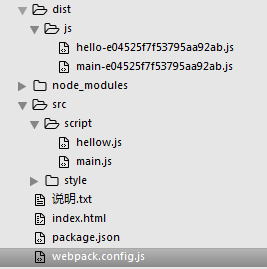
**例如：**

1. //模块化输出
2. module.exports = {
3. //入口文件,这里采用entry对象的方式，分别将main.js和hellow.js打包
4. entry:{
5. main:'./src/script/main.js',
6. hello:'./src/script/hellow.js'
7. },
8. // 打包后的文件
9. output:{
10. //打包后文件在./dist/js的文件夹中
11. path:\_\_dirname+'/dist/js',
12. //打包后文件的名称为entry的chunk名称-编译的哈希值
13. filename:'[name]-[hash].js'
14. }
15. }

编译过程：



可以看到打包成了两个文件 （因为编译的哈希一样所以后面的是一样的）



当然可以采用【name】-【chunkhash】这样产生的文件名就完全不一样了，【chunkhash】就如同是MD5一样，是完全不同的，哪怕是同一个js只要代码发生变化【chunkhash】就会不同，这是非常有用的，我们一般只上线我们更改的js文件。

Webpack的基本配置---案例4

#### 2. output.path

输出目录为绝对路径（必需）。

### 5.问题

在上面的讲解中我们可以看到，可以通过编译打包不同的js文件，但是如果通过【name】-【chunkhash】的方式产生的js文件的名字是不一定的，导致我们html在引入js文件时会非常的麻烦，所以我们需要引入插件html-webpack-plugin，看下面的简介。

## 四．自动生成项目中的HTML文件

### 1.HTML插件的使用

**安装插件**

npm install html-webpack-plugin --save-dev

配置webpack.config.js

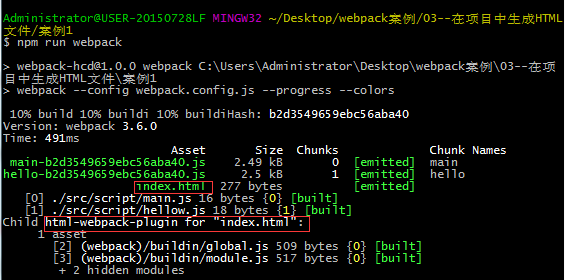
安装完成后需要在webpack.config.js中引用并且初始化插件，具体的可以查看API <http://webpack.github.io/docs/using-plugins.html>

博客<http://www.cnblogs.com/wonyun/p/6030090.html>

Npm插件详解：<https://www.npmjs.com/package/html-webpack-plugin>

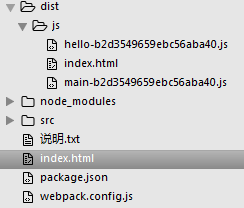
1. //引用webpack.config.js插件
2. var htmlWebpackPlugin = require('html-webpack-plugin');
3. //模块化输出
4. module.exports = {
5. //入口文件,这里采用entry对象的方式，分别将main.js和hellow.js打包
6. entry:{
7. main:'./src/script/main.js',
8. hello:'./src/script/hellow.js'
9. },
10. // 打包后的文件
11. output:{
12. //打包后文件在./dist/js的文件夹中
13. path:\_\_dirname+'/dist/js',
14. //打包后文件的名称为entry的chunk名称-编译的哈希值
15. filename:'[name]-[hash].js'
16. },
17. //插件数组
18. plugins:[
19. //初始化插件
20. new htmlWebpackPlugin()
21. ]
22. }

编译打包

******

会自动在output.path的路径下生成index.html

**结果：**

****

dist中的index.html是自动生成的，已经自动引入了打包后的js文件，内容为



在项目中生成HTML文件---案例1

### 2.以模板生成HTML

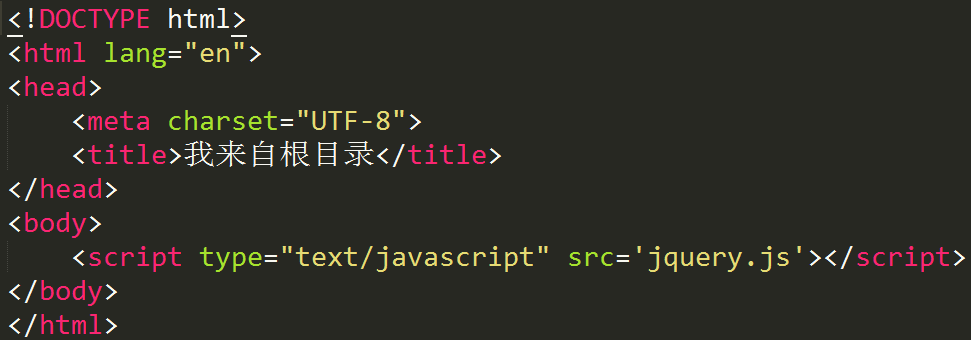
在4.1的例子中会自动生成index.html，但是在大的项目中我们的index.html需要引入许多的js文件，所以我们可以在引入插件初始化时插入模板参数，那么就会自动生成以模板为基本的HTML文件，并且自动引入打包后的js文件了。

**例如：**

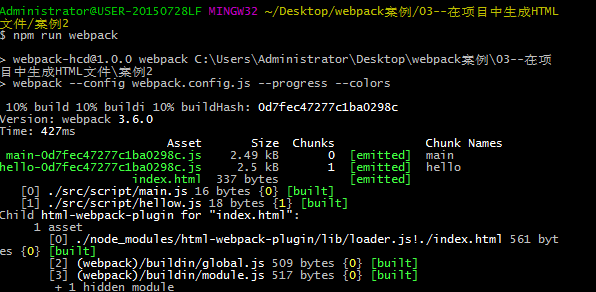
**wbepack.config.js的配置:**

1. var htmlWebpackPlugin = require('html-webpack-plugin');
2. module.exports = {
3. //执行上下文，默认为webpack.config.js所在的文件夹
4. // context：
5. entry:{
6. main:'./src/script/main.js',
7. hello:'./src/script/hellow.js'
8. },
9. output:{
10. path:\_\_dirname+'/dist/js',
11. filename:'[name]-[hash].js'
12. },
13. //插件数组
14. plugins:[
15. //初始化插件,传递模板参数
16. new htmlWebpackPlugin({
17. //模板为同级目录下的index.html，为何不用写路径，是因为默认上下文问webpack.config.js所在的文件夹
18. template:'index.html'
19. })
20. ]
21. }

**我们的根目录下的index.html**

****

**编辑运行：**

****

**结果：**

**会根据output.path生成index.html文件，**

****

在项目中生成HTML文件---案例2

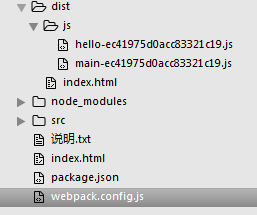
### 3.改变生成HTML的位置

在上面的例子中，生成的HTML和js是放在一起的，会非常的不方便，所以我们可以改变output.path

1. output:{
2. path:\_\_dirname+'/dist',
3. //注意写法
4. filename:'js/[name]-[hash].js'
5. }

**运行后结果为：**

这样一来自动生成的index.html就会在dist目录下，而打包的js文件将在dist/.js目录下

****

在项目中生成HTML文件---案例3

### 4.初始化HTML插件的详解

Npm插件详解：<https://www.npmjs.com/package/html-webpack-plugin>

#### **1.简介html插件的options参数**

具体可以看详解的configuration部分

Webpack.congif.js:

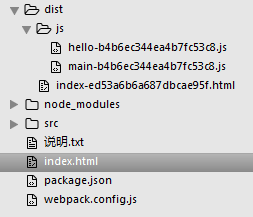
1. //插件数组
2. plugins:[
3. //初始化插件,传递模板参数
4. new htmlWebpackPlugin({
5. //模板为同级目录下的index.html，为何不用写路径，是因为默认上下文问webpack.config.js所在的文件夹
6. template:'index.html',
7. //自动生成HTML文件的名字
8. filename:'index-[hash].html',
9. //引入打包后的js的script标签所在的位置,这里表示放在head标签中
10. inject:'head',
11. //可以向模板传递参数，然后应用于自动生成的html,(模板需要获取参数)
12. title:'我来自参数',
13. //任何的参数都是可以传递的
14. date:new Date()
15. })
16. ]

**Index.html(模板)**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <!-- 这里利用ejs的语法，获取htmlWebpackPlugin(名字来自于webpack.config.js的require命名)插件中参数的title属性值 -->
6. <title><%= htmlWebpackPlugin.options.title%></title>
7. </head>
8. <body>
9. <script type="text/javascript" src='jquery.js'></script>
10. <h2>
11. 现在的时间是：<%= htmlWebpackPlugin.options.date%>
12. </h2>
13. </body>
14. </html>

**结果为：**

**目录：**

****

**自动生成的html：**

可以看到连注释都一起生成了

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <!-- 这里利用ejs的语法，获取htmlWebpackPlugin(名字来自于webpack.config.js的require命名)插件中参数的title属性值 -->
6. <title>我来自参数</title>
7. <script type="text/javascript" src="js/main-b4b6ec344ea4b7fc53c8.js"></script><script type="text/javascript" src="js/hello-b4b6ec344ea4b7fc53c8.js"></script></head>
8. <body>
9. <script type="text/javascript" src='jquery.js'></script>
10. <h2>
11. 现在的时间是：Fri Oct 13 2017 10:03:45 GMT+0800 (中国标准时间)
12. </h2>
13. </body>
14. </html>

在项目中生成HTML文件---案例4

#### 2．简介html插件的files属性

files属性是描述自动生成的HTML的一些特征的，是内部自动生成的，我们不用写的，但是我们可以利用

1. "htmlWebpackPlugin": {
2. //files其实就是指自动编译的HTML文件
3. "files": {
4. //引用的css文件
5. "css": [ "main.css" ],
6. //当前引用的打包后的js数组
7. "js": [ "assets/head\_bundle.js", "assets/main\_bundle.js"],
8. //webpack.config.js中用于打包的两个chaunk
9. "chunks": {
10. "head": {
11. //打包后js文件名
12. "entry": "assets/head\_bundle.js",
13. //打包后css文件名
14. "css": [ "main.css" ]
15. },
16. "main": {
17. "entry": "assets/main\_bundle.js",
18. "css": []
19. },
20. }
21. }
22. }

**例子：**

在4.1中我们知道我们可以利用html插件参数的enject属性来设置引入打包后js文件的位置，但是如果引入的是多个打包js文件，我们想一个放在head中一个放在body中，只利用options的属性是办不到的，我们可以利用插件的files下的属性来做

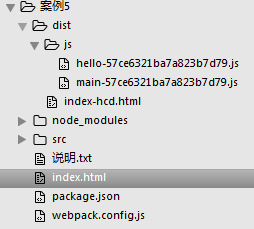
**Webpack.config.js:**

1. var htmlWebpackPlugin = require('html-webpack-plugin');
2. module.exports = {
3. //执行上下文，默认为webpack.config.js所在的文件夹
4. // context：
5. entry:{
6. main:'./src/script/main.js',
7. hello:'./src/script/hellow.js'
8. },
9. output:{
10. path:\_\_dirname+'/dist',
11. filename:'js/[name]-[hash].js'
12. },
13. //插件数组
14. plugins:[
15. //初始化插件,传递模板参数
16. new htmlWebpackPlugin({
17. //模板为同级目录下的index.html，为何不用写路径，是因为默认上下文问webpack.config.js所在的文件夹
18. template:'index.html',
19. //自动生成HTML文件的名字
20. filename:'index-hcd.html',
21. //引入打包后的js的script标签所在的位置,false表示不自动引入打包后的js
22. inject:false,
23. //可以向模板传递参数，然后应用于自动生成的html,(模板需要获取参数)
24. title:'我来自参数'
25. })
26. ]
27. }

**Index.html模板为：**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <!-- 这里利用ejs的语法，获取htmlWebpackPlugin(名字来自于webpack.config.js的require命名)插件中参数的title属性值 -->
6. <title><%= htmlWebpackPlugin.options.title%></title>
7. <!-- 将main这个chunk打包成的js文件放在head中 -->
8. <script type="text/javascript" src='<%=htmlWebpackPlugin.files.chunks.main.entry%>'></script>
9. </head>
10. <body>
11. <script type="text/javascript" src='jquery.js'></script>
12. <!-- 将main这个chunk打包成的js文件放在body中 -->
13. <script type="text/javascript" src='<%=htmlWebpackPlugin.files.chunks.hello.entry%>'></script>
14. </body>
15. </html>

**结果为：**



自动生成的HTML：



在项目中生成HTML文件---案例5

#### 3.实战上线时配置的属性

##### 1.利用oupt的新属性publicPath，设置绝对地址

<http://webpack.github.io/docs/configuration.html#output-publicpath>

##### 2.利用html插件的minify属性压缩代码

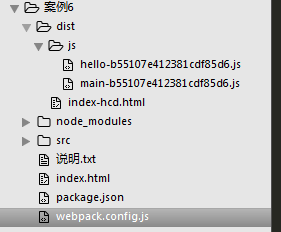
<https://www.npmjs.com/package/html-webpack-plugin>

例如：

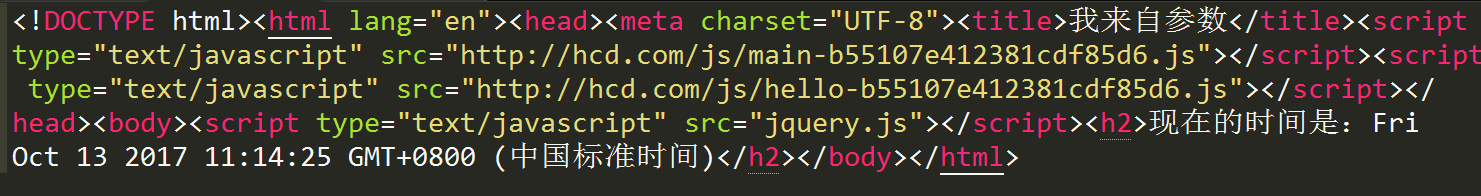
**Wenpack.config.js**

1. var htmlWebpackPlugin = require('html-webpack-plugin');
2. module.exports = {
3. entry:{
4. main:'./src/script/main.js',
5. hello:'./src/script/hellow.js'
6. },
7. output:{
8. path:\_\_dirname+'/dist',
9. filename:'js/[name]-[hash].js',
10. //线上的地址,所有生成的文件将换为以此为开头的绝对路径
11. publicPath:'http://hcd.com/'
12. },
13. //插件数组
14. plugins:[
15. //初始化插件,传递模板参数
16. new htmlWebpackPlugin({
17. //模板为同级目录下的index.html，为何不用写路径，是因为默认上下文问webpack.config.js所在的文件夹
18. template:'index.html',
19. //自动生成HTML文件的名字
20. filename:'index-hcd.html',
21. //引入打包后的js的script标签所在的位置,这里表示放在head标签中
22. inject:'head',
23. //可以向模板传递参数，然后应用于自动生成的html,(模板需要获取参数)
24. title:'我来自参数',
25. //任何的参数都是可以传递的
26. date:new Date(),
27. //上线时减小html代码的规格,压缩代码
28. minify:{
29. //删除html的注释
30. removeComments:true,
31. //删除空格
32. collapseWhitespace:true
33. }
34. })
35. ]
36. }

**结果：**



生成的html代码：

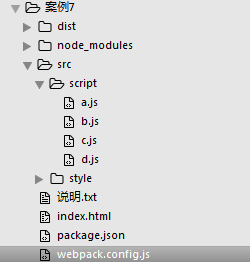


在项目中生成HTML文件---案例6

### 5.多页面应用

上面的例子多数为单页面的应用，这里讲解的是生成多个HTML，而不同的html引入不同的打包后的js

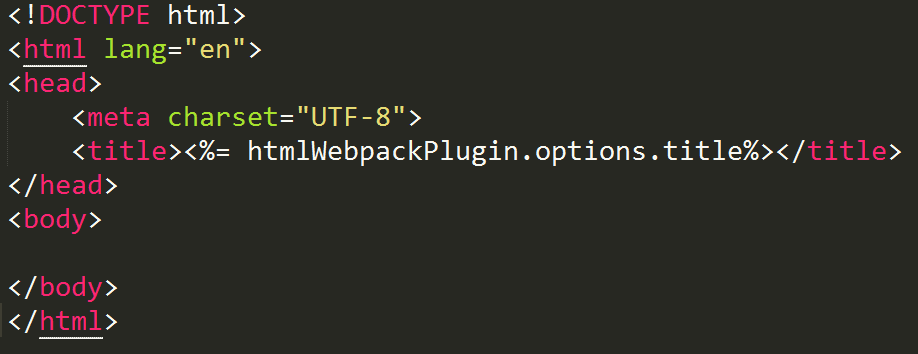
**目录：**



**Webpack.config.js：**

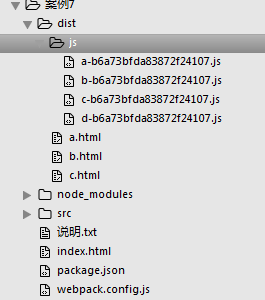
1. var htmlWebpackPlugin = require('html-webpack-plugin');
2. module.exports = {
3. entry:{
4. a:'./src/script/a.js',
5. b:'./src/script/b.js',
6. c:'./src/script/c.js',
7. d:'./src/script/d.js'
8. },
9. output:{
10. path:\_\_dirname+'/dist',
11. filename:'js/[name]-[hash].js',
12. publicPath:'http://hcd.com/'
13. },
14. plugins:[
15. //自动生成多少个HTML页面就new几次插件
16. new htmlWebpackPlugin({
17. template:'index.html',
18. filename:'a.html',
19. title:'this is A',
20. //在该页面加载chunk为‘a’打包生成的js
21. chunks:['a']
22. }),
23. new htmlWebpackPlugin({
24. template:'index.html',
25. filename:'b.html',
26. title:'this is B',
27. //加载除了‘a’chunk以外所有的打包生成的js
28. excludeChunks:['a']
29. }),
30. new htmlWebpackPlugin({
31. template:'index.html',
32. filename:'c.html',
33. title:'this is C',
34. chunks:['c','d']
35. })
36. ]
37. }

Index.html模板：



**结果：**

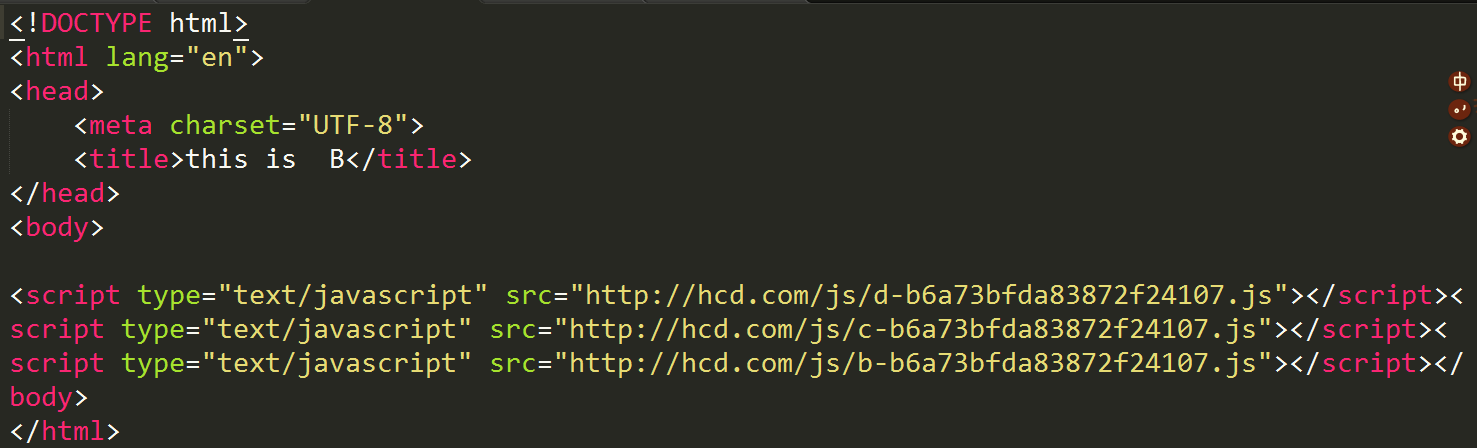
**目录**



**a.html：**



**b.html：**



**c.html：**



在项目中生成HTML文件---案例7

### 6.将js内嵌入HTML

上面的我们都是用的http去请求js，有的时候我们需要在HTML插入js文件

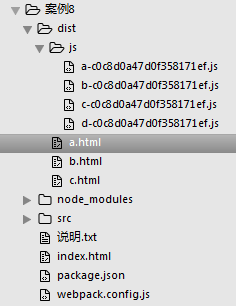
**模板index.html**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title><%= htmlWebpackPlugin.options.title%></title>
6. <script type="text/javascript">
7. //获取打包后chunk d的文件地址，因为我们采用的是publicPath,所以绝对地址是publicPath
8. 绝对地址:<%=
9. htmlWebpackPlugin.files.chunks.d.entry
10. %>
11. //获取去除publicPath绝对地址的d的打包js地址
12. 去除绝对地址:<%=
13. htmlWebpackPlugin.files.chunks.d.entry.substr(htmlWebpackPlugin.files.publicPath.length)
14. %>
15. //获取d打包后的代码
16. js代码:<%=
17. compilation.assets[htmlWebpackPlugin.files.chunks.d.entry.substr(htmlWebpackPlugin.files.publicPath.length)].source()
18. %>
19. </script>
20. </head>
21. <body>
22. <!-- 因为每一个自动生成的html都是要引入两个js的,除了内嵌的,还有http请求的一个,（注意先要在webpack.config.js中间inject设置为false，否则会直接建两个js文件引入） -->
23. <% for(var key in htmlWebpackPlugin.files.chunks){ %>
24. <% if( key !== 'd'){ %>
25. <script type="text/javascript" src="<%=htmlWebpackPlugin.files.chunks[key].entry%>"></script>
26. <% } %>
27. <%}%>
28. </body>
29. </html>

**Webpack.config.js**

1. var htmlWebpackPlugin = require('html-webpack-plugin');
2. module.exports = {
3. entry:{
4. a:'./src/script/a.js',
5. b:'./src/script/b.js',
6. c:'./src/script/c.js',
7. d:'./src/script/d.js'
8. },
9. output:{
10. path:\_\_dirname+'/dist',
11. filename:'js/[name]-[hash].js',
12. publicPath:'http://hcd.com/'
13. },
14. plugins:[
15. //自动生成多少个HTML页面就new几次插件
16. new htmlWebpackPlugin({
17. template:'index.html',
18. filename:'a.html',
19. title:'this is A',
20. //避免自动加载js
21. inject:false,
22. //在该页面加载chunk为‘a’’b’打包生成的js
23. chunks:['a','d']
24. }),
25. new htmlWebpackPlugin({
26. template:'index.html',
27. filename:'b.html',
28. title:'this is B',
29. inject:false,
30. //加载除了‘a’chunk以外所有的打包生成的js
31. excludeChunks:['a','c']
32. }),
33. new htmlWebpackPlugin({
34. template:'index.html',
35. filename:'c.html',
36. title:'this is C',
37. inject:false,
38. chunks:['c','d']
39. })
40. ]
41. }

**结果：**



在项目中生成HTML文件---案例8

## 五．处理项目中的资源文件

### 1.ES6语法的转译

Webpack的API：

Webpack使用loader

<http://webpack.github.io/docs/using-loaders.html>

webpack的API中使用module-loader（包括include，exclude等）

<http://webpack.github.io/docs/configuration.html#module-loaders>

转译工具babel

Babel中在webpack中的使用

<http://babeljs.cn/docs/setup/#webpack>

node的path的用法

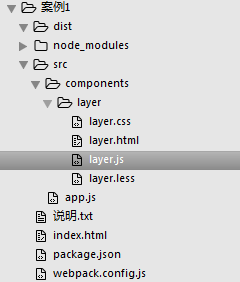
<http://nodejs.cn/api/path.html#path_path_resolve_paths>

安装：

Babel安装：npm install --save –dev babel-loader babel-core

因为我们用的是es2015 npm install bable-preset-es2015

**文档结构：**



Webpack.config.js

1. var htmlWebpackPlugin = require('html-webpack-plugin');
2. // 因为要用绝对路径,所以我们使用node.js自带的path
3. var path = require('path');
4. module.exports = {
5. entry:'./src/app.js',
6. output:{
7. path:\_\_dirname+'/dist',
8. filename:'js/[name]-bundle.js'
9. },
10. // 通过配置将加载器绑定到RegExp(正则表达式)
11. //ES6的转译插件babel
12. module:{
13. loaders:[
14. {
15. //检测所有的js文件
16. test: /\.js$/,
17. //使用babel-loader
18. loader: "babel-loader",
19. //不用使用(使用绝对路径)的数组
20. //path.resolve可以得出绝对路径
21. exclude:[
22. path.resolve(\_\_dirname,'node\_modules/'),
23. path.resolve(\_\_dirname,'dist/')
24. ],
25. // 只是src下的js使用(使用绝对路径)
26. include:path.resolve(\_\_dirname,'src/'), //因为es6是不断更新的,babel转译es6是有不同版本的,这里采用‘es2015’版,也可以在loader?presets='es2015'表示
27. query:{
28. presets:['es2015']
29. }
30. }
31. ]
32. },
33. plugins:[
34. new htmlWebpackPlugin({
35. template:'index.html',
36. filename:'index.html',
37. title:'this is C',
38. inject:'body'
39. })
40. ]
41. }

处理项目中的资源文件---案例1

### 2.处理项目中的css

安装css-loader和style-loader

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

对css编译内容进行再次封装

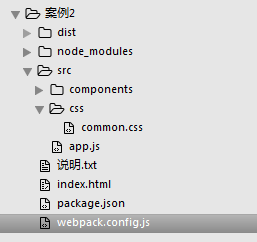
npm install postcss-loader –save-dev

<https://www.npmjs.com/package/postcss-loader>

自动补充浏览器兼容前缀（postcss-loader的插件）

npm install autoprefixer --save-dev

**目录结构：**



**common.js**



**App.js**



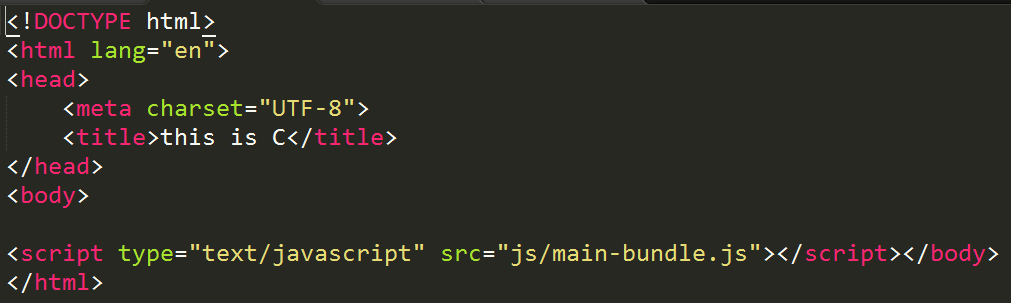
**webpack.config.js**

1. var htmlWebpackPlugin = require('html-webpack-plugin');
2. // 因为要用绝对路径,所以我们使用node.js自带的path
3. var path = require('path');
4. module.exports = {
5. //上下文环境(默认为当前路径，当然设置为绝对路径更好)
6. context:\_\_dirname,
7. entry:'./src/app.js',
8. output:{
9. path:\_\_dirname+'/dist',
10. filename:'js/[name]-bundle.js'
11. },
12. // 通过配置将加载器绑定到RegExp(正则表达式)
13. //ES6的转译插件babel
14. module:{
15. loaders:[
16. {
17. //检测所有的js文件
18. test: /\.js$/,
19. //使用babel-loader
20. loader: "babel-loader",
21. //不用使用(使用绝对路径)的数组
22. //path.resolve可以得出绝对路径
23. exclude:[path.resolve(\_\_dirname,'node\_modules/'),
24. path.resolve(\_\_dirname,'dist/')],
25. // 只是src下的js使用(使用绝对路径)
26. include:path.resolve(\_\_dirname,'src/'),
27. //因为es6是不断更新的,babel转译es6是有不同版本的,这里采用‘es2015’版,也可以在loader?presets='es2015'表示
28. query:{
29. presets:['es2015']
30. }
31. },
32. {
33. //检测所有的css文件
34. test:/\.css$/,
35. // css-loader是的在js中可以引用css文件，再用！将style-loader(是的css应用在HTML)串联起来
36. //post-loader将编译的css进行封装
37. //loader是从右向左读的，所以先进行postcss-loader再css-loader再style-loader
38. // loader:'style-loader!css-loader!postcss-loader'
40. //或者不写loader，写成loaders也是可以的
41. //从下往上读
42. loaders:[
43. 'style-loader',
44. 'css-loader',
45. {
46. loader:'postcss-loader',
47. //对postcss-loader进行设置,引用它的autoprefixer插件,对浏览器最近5个版本进行兼容
48. options: {
49. ident: 'postcss',
50. plugins: [
51. require('autoprefixer')({ browsers:['last 5 versions']
52. })
53. ]
54. }
55. }
57. ]
58. }
59. ]
60. },
61. plugins:[
62. new htmlWebpackPlugin({
63. template:'index.html',
64. filename:'index.html',
65. title:'this is C',
66. inject:'body'
67. })
68. ]
69. }

**结果：**

**生成的index.html：**

虽然看不到引入的js，其实已经因为style-loader自动生成<style></style>了



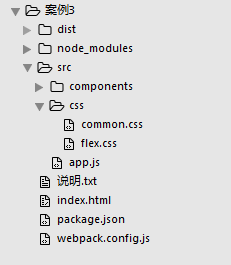
可以看到postcss-loader的autoprefixer插件使得flex-div兼容浏览器了



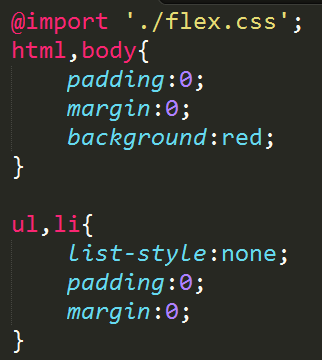
处理项目中的资源文件---案例2

### [3.处理css中@import](mailto:3.处理css中@import)进来的css文件

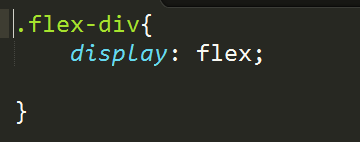
当我们的css文件中引入另外的css文件时



**Common.js**



**flex.css**



当我们还按照 案例2 中的方法去给flex加兼容性前缀时结果为：



可以看到flex的前没有加上

**解决方法：**

对webpack.config.js的loader中的css-loader进行参数配置

**Webpack.config.js**

1. loaders:[
2. 'style-loader',
3. //利用css的参数解决@import css文件的问题
4. // importLoaders=1表示import进来的css使用一次css-loader(import进来2个文件就是2)
5. 'css-loader?importLoaders=1',
6. {
7. loader:'postcss-loader',
8. //对postcss-loader进行设置,引用它的autoprefixer插件,对浏览器最近5个版本进行兼容
9. options: {
10. ident: 'postcss',
11. plugins: [
12. require('autoprefixer')({
13. browsers:['last 5 versions']
14. })
15. ]
16. }
17. }
18. ]

结果：



处理项目中的资源文件---案例3

### 4.使用less和sass

安装less

npm install --save-dev less-loader less

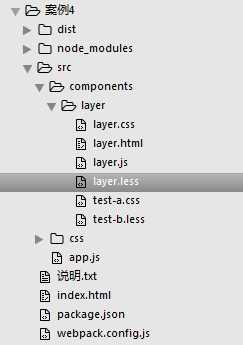
API：<https://www.npmjs.com/package/less-loader>

安装sass

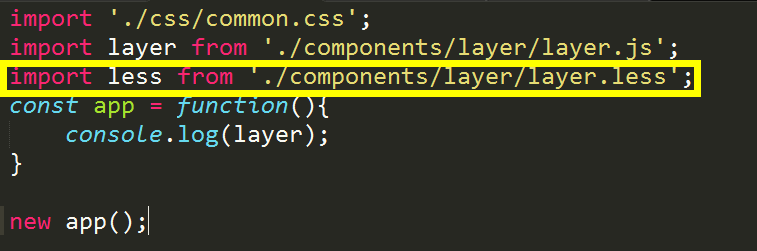
npm install sass-loader node-sass  --save-dev

API: <https://www.npmjs.com/package/sass-loader>

目录：

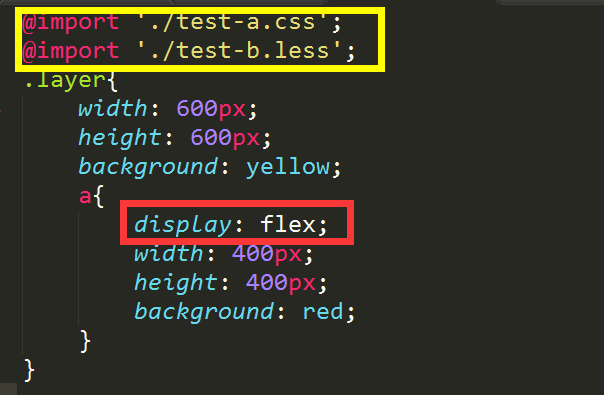


App.js引入less



layer.less中引入test-a.css和test-b.less

并且有一个需要自动设置浏览器兼容的

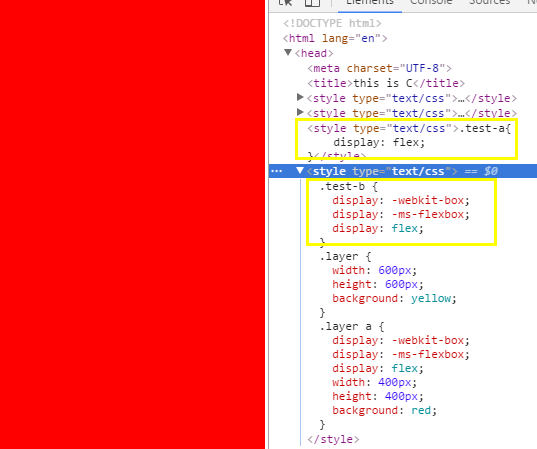


Webpack.config.js

在原来的基础上加上处理less的loader

1. {
2. test: /\.less$/,
3. //postcss-loader需要在style-loader，css-loader之前处理，less-loader，sass-loader之后处理
4. loaders:[
5. 'style-loader',
6. //利用less@import less文件的时候不需要设置css-loader的参数
7. //@import css文件的时候需要设置css-loader的参数
8. 'css-loader',
9. {
10. loader:'postcss-loader',
11. //对postcss-loader进行设置,引用它的autoprefixer插件,对浏览器最近5个版本进行兼容
12. options: {
13. ident: 'postcss',
14. plugins: [
15. require('autoprefixer')({
16. browsers:['last 5 versions']
17. })
18. ]
19. }
20. },
21. // 引用less-loader
22. 'less-loader'
23. ]
24. }

结果：



可以看到一共有4个style标签，第1,2为引入的css，与less无关，第3个为less中引入css文件而没有设置css-loader参数，可以看到前缀没有生效，第4个为less文件和less中引入的less都在一个style中展示了。

处理项目中的资源文件---案例4

（sass和less是一样的，只不过 test:‘ /\.scss$/,’使用的是sass-loader）

### 5.处理模板文件

API：<http://webpack.github.io/docs/list-of-loaders.html#templating>

html模板插件

npm install html-loader –save-dev

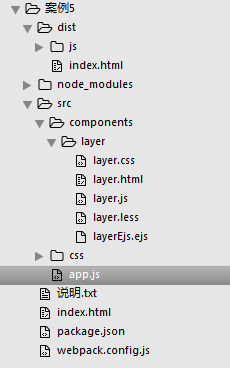
jade模板

npm install jade-loader –save-dev

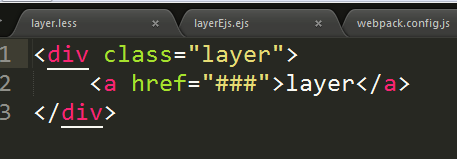
ejs模板

npm install ejs-loader –save-dev

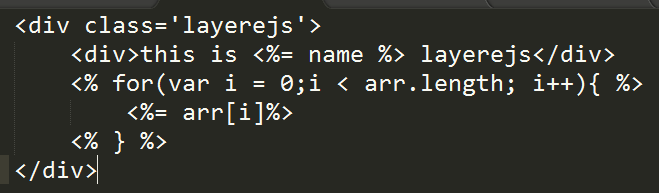
目录结构：



Layer.html



layerEjs.ejs



**Layer.js**

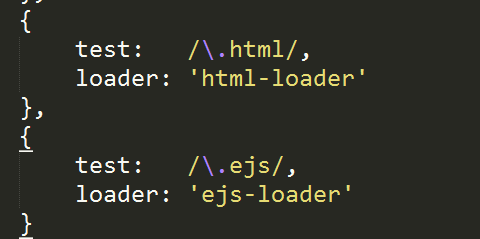
1. //引入html模板 其实是字符串
2. import tpl from './layer.html';
3. //引入ejs模板 其实是函数(用时可能会需要传入参数)
4. import ejs from './layerEjs.ejs';
5. function layer(){
6. return{
7. name:'layer',
8. tpl:tpl,
9. ejs:ejs
10. }
11. }
12. Export default layer

**App.js**

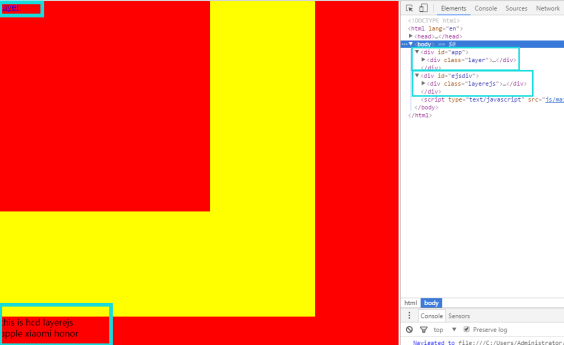
1. import './css/common.css';
2. import layer from './components/layer/layer.js';
3. import less from './components/layer/layer.less';
4. const app = function(){
5. var dom = document.getElementById('app');
6. var newLayer = layer();
7. //模板其实就是newLayer.tpl属性，将模板当做字符串
8. dom.innerHTML = newLayer.tpl;
9. var ejsdom = document.getElementById('ejsdiv');
10. //向ejs模板中传递参数
11. ejsdom.innerHTML = newLayer.ejs({
12. name:'hcd',
13. arr:['apple','xiaomi','honor']
14. })
15. }
16. new app();

**webnpack.config.js**

在原来的基础上加上



**结果：**



可以看到都在页面上显示出来了

处理项目中的资源文件---案例5

### 6.处理图片及其他资源

安装file-loader

npm install file-loader –save-dev

<https://www.npmjs.com/package/file-loader>

安装url-loaer

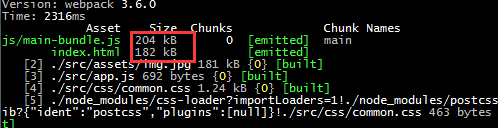
npm install url-loader –save-dev

<https://www.npmjs.com/package/url-loader>

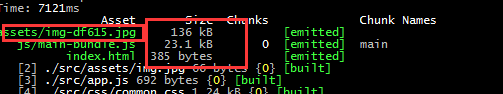
不同之处：

Url-loader（内部含有file-loader）有一个参数limit参数，设置文件的大小（1000表示1K），当文件的大小<limit时，会将文件转化为base64的编码，当>limit时再交给file-loader处理

用base不会打包图片，但是会增加文件的体积

******

用file-loader会打包图片，不会增加图片的体积

******

用http请求图片可以起到一定的缓存作用，当有图片重复时，不会再次加载了

用base64呢，代码每次都会出现，增加代码的体积，影响HTML和js的大小

//安装 img-loader

（需要配合url-loader或者file-loader一起使用）

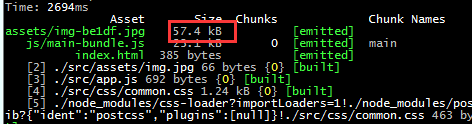
npm install img-loader –save-dev

<https://www.npmjs.com/package/img-loader>

(img-loader是可以根据不同的图片类型进行不同的设置，可以在官网上看)

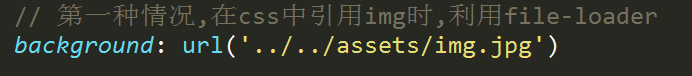
压缩打包的图片

原来的图片是132k的，压缩后只有57.4k

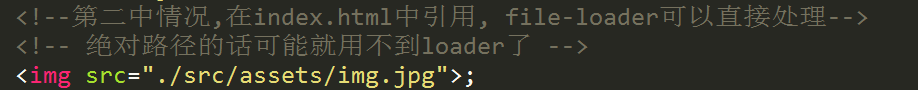
******

我们引入图片一般分为3种方式

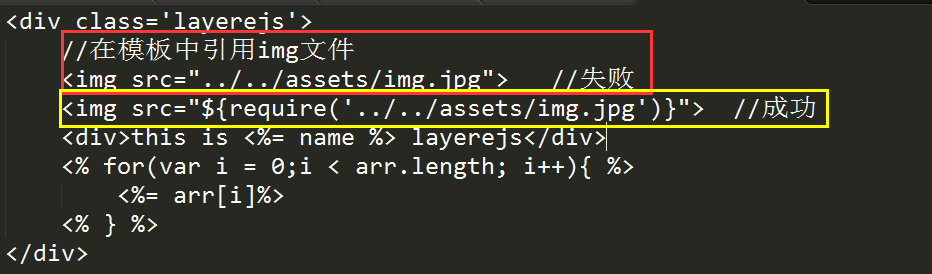
* + - 1. 在css中引入（layer.less）



2.在入口的index.html文件中引入



3.在HTML模板文件中引入



Webpack.config.js

**1.file-loader**

1. //file-loader
2. {
3. test: /\.(png|gif|jpe?g|svg)$/i,
4. loader: 'file-loader',
5. // 设置编译后图片的地址和名称
6. // 放在outpath.path:\_\_dirname+'/dist'下的assets文件夹下
7. // [文件名]-[5位的hash].[文件后缀]
8. options: {
9. name: 'assets/[name]-[hash:5].[ext]'
10. }
11. }

**2.url-loader**

1. // url-loader
2. {
3. test: /\.(png|gif|jpe?g|svg)$/i,
4. loader: 'url-loader',
5. //设置参数
6. //limit表示img大于140k的用http请求，小于140k的用base64
7. // 设置编译后图片的地址和名称
8. // 放在outpath.path:\_\_dirname+'/dist'下的assets文件夹下
9. // [文件名]-[5位的hash].[文件后缀]
10. options: {
11. limit:1400000,
12. name: 'assets/[name]-[hash:5].[ext]'
13. }
14. }

**3.img-loader**

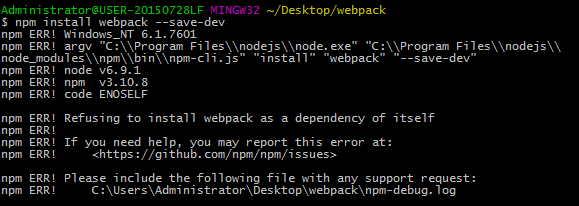
1. //img-loader(图片压缩需要和url-loader或者file-loader一起使用)
2. {
3. test: /\.(png|gif|jpe?g|svg)$/i,
4. loaders:[
5. 'url-loader?limit=1400&name=assets/[name]-[hash:5].[ext]',
6. 'img-loader'
7. ]
8. }

处理项目中的资源文件---案例6

## 附录 1.注意事项

### 1.安装webpack时

npm init生成的package.json的name不能叫webpack，注意执行npm install webpack --save-dev命令时会报错



## 附录 2.网站和地址

### 1.webpack官网

<http://webpack.github.io/docs/>

<http://www.runoob.com/w3cnote/webpack-tutorial.html>

### 2. webpack的配置

(具体的配置在文档 API->configuration)

<http://webpack.github.io/docs/configuration.html>

3.

Webpack.config.js引用插件

<http://webpack.github.io/docs/using-plugins.html>

4.

Output

<http://webpack.github.io/docs/configuration.html#output>

5.entry

<http://webpack.github.io/docs/configuration.html#entry>

### 3.HTML插件

博客<http://www.cnblogs.com/wonyun/p/6030090.html>

API <http://webpack.github.io/docs/using-plugins.html>

Npm插件详解：<https://www.npmjs.com/package/html-webpack-plugin>

### **4.webpack使用loader**

Webpack的API：

Webpack使用loader

<http://webpack.github.io/docs/using-loaders.html>

webpack的API中使用module-loader（包括include，exclude等）

<http://webpack.github.io/docs/configuration.html#module-loaders>

### 5.babel在webpack中的使用

转译工具babel

Babel中在webpack中的使用

<http://babeljs.cn/docs/setup/#webpack>

### 6.自动补充css的浏览器兼容前缀

//对css编译内容进行再次封装

npm install postcss-loader –save-dev

<https://www.npmjs.com/package/postcss-loader>

//自动补充浏览器兼容前缀（postcss-loader的插件）

npm install autoprefixer --save-dev

### 7.处理模板文件

API：<http://webpack.github.io/docs/list-of-loaders.html#templating>

### 8.处理图片及其他文件

安装file-loader

npm install file-loader –save-dev

<https://www.npmjs.com/package/file-loader>

安装url-loaer

npm install url-loader –save-dev

<https://www.npmjs.com/package/url-loader>

//安装 img-loader（需要配合url-loader或者file-loader一起使用）

npm install img-loader –save-dev

<https://www.npmjs.com/package/img-loader>

## 附录 3.NPM命令

npm install webpack -g 全局安装webpack

webpack -help 查看是否安装，以及安装的版本

npm init 建立一个文件后

npm install webpack --save-dev 在局部安装webpack

//新建hellow.Js,对其进行打包，打包后的文件叫hellow.bundle.js

webpack hellow.js hellow.bundle.js

//安装css-loader（识别css文件）与style-loader（应用css样式）

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

//对css编译内容进行再次封装

npm install postcss-loader –save-dev

<https://www.npmjs.com/package/postcss-loader>

//自动补充浏览器兼容前缀（postcss-loader的插件）

npm install autoprefixer --save-dev

//引入css文件直接打包

webpack hellow.js hellow.bundle.js --module-bind ‘css=style-loader！css-loader！’

//实时监听

--watch

//执行配置文件webpack.config.js

Webpack

//执行package.json中script的简便命令

npm run webpack

//安装HTML插件

npm install html-webpack-plugin --save-dev

//Babel安装：

npm install --save –dev babel-loader babel-core

//因为我们babel用的是es2015版本 所以在安装

npm install bable-preset-es2015

//html模板插件

npm install html-loader –save-dev

//jade模板

npm install jade-loader –save-dev

//ejs模板

npm install ejs-loader –save-dev

//安装file-loader

npm install file-loader –save-dev

<https://www.npmjs.com/package/file-loader>

//安装url-loaer

npm install url-loader –save-dev

<https://www.npmjs.com/package/url-loader>

//安装 img-loader（需要配合url-loader或者file-loader一起使用）

npm install img-loader –save-dev

<https://www.npmjs.com/package/img-loader>