# 1.Webpack1.x

## 1.1 概述

Webpack是一款用户打包前端模块的工具。主要是用来打包javascript的。同时也能转换、捆绑、打包其他的静态资源，包括css、image、font file、template等。这里就尽量详细的来介绍下一些基本功能的使用。

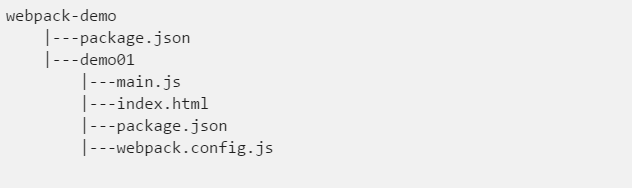
webpack的背景介绍：[webpack的简单介绍](https://github.com/kkcode24/blog/blob/master/2018/2018-04-28-webpack.md)

## 1.2 怎么使用webpack

### 1.2.1 搭建node运行的环境

参照我之前的博客：[nvm环境配置](https://github.com/kkcode24/blog/blob/master/2018/2018-04-08-nvm-npm.md)

### 1.2.2 按照以下目录创建项目



### 1.2.3 webpack的配置文件webpack.config.js

module.exports = {

entry: './main.js',

output: {

filename: 'bundle.js'

}

}

### 1.2.4 demo01下的index.html

<**html**>

<**body**>

<**script** type="text/javascript" src="bundle.js"></**script**>

</**body**>

</**html**>

### 1.2.5 demo01下的 main.js

document.write('<h1>Webpack@1.0.5 and webpack-dev-server@1.0.2</h1>');

### 1.2.6 demo01下的 package.json

{

**"name"**: "webpack-demo01",

**"version"**: "1.0.0",

**"main"**: "main.js",

**"scripts"**: {

**"dev"**: "webpack-dev-server --open",

**"build"**: "webpack -p"

},

**"license"**: "MIT"

}

### 1.2.7 webpack-demo下的package.json

{

**"name"**: "webpack-demo",

**"version"**: "1.0.0",

**"description"**: "webpack-practice",

**"main"**: "index.js",

**"keywords"**: [

"webpack",

"webpack@1",

"webpack@2",

"webpack@3"

],

**"author"**: "yike",

**"license"**: "ISC",

**"repository"**: {

**"type"**: "git",

**"url"**: "git+https://github.com/kkcode24/webpack-demo.git"

},

**"bugs"**: {

**"url"**: "https://github.com/kkcode24/webpack-demo/issues"

},

**"homepage"**: "https://github.com/kkcode24/webpack-demo#readme",

**"devDependencies"**: {

**"open-browser-webpack-plugin"**: "0.0.5",

**"webpack"**: "^1.0.5",

**"webpack-dev-server"**: "^1.0.2"

}

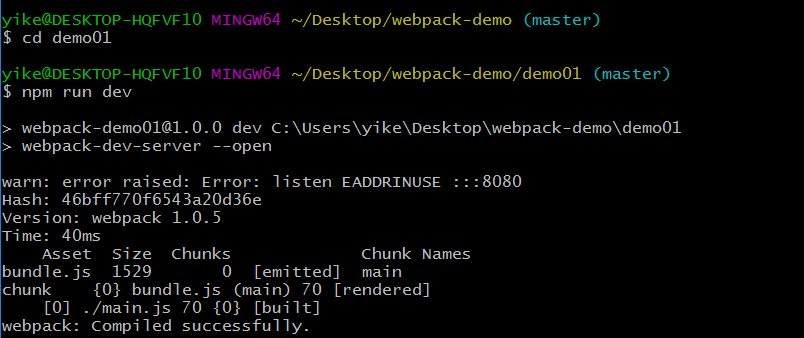
}

### 1.2.8 全局安装webpack&&webpack-dev-server

按照以下步骤完成webpack和webpack-dev-server的全局安装。然后再进入到刚才创建的项目的根目录下，然后执行 npm install 安装开发依赖，也就是open-browser-webpack-plugin插件，即自动打开浏览器插件。



### 1.2.9 运行demo01



如果不能自动打开浏览器，你需要自己打开<http://127.0.0.1:8080>

当浏览器中出现如下图所示情况，说明你已经入门了。



## 1.3 demo01中webpack.config.js的配置项

### 1.3.1 入口文件的路径entry

配置要打包的文件的入口;可以配置多个入口文件，下面会有介绍。

### 1.3.2 输出文件的路径output

配置输出文件的路径，文件名等。

## 1.4创建demo02

1. 复制demo01到demo02

2. 修改webpack配置，修改demo02文件内容

3. 运行webpack，参考1.2.9

说明： 下面的demo都是基于上一个进行修改的。

## 1.5 案例demo02

### 1.5.1 调整webpack配置

**// webpack.config.js**

**var** path = require('path');

**var** buildPath = path.resolve(\_\_dirname,"build");

**const** config = {

entry: path.resolve(\_\_dirname,'main.js'),

output: {

path: buildPath,

filename: "bundle.js"

},

resolve: {

extentions:["","js"] *// 当rewquire的模块找不到时，添加这些后缀。*

}

}

module.exports = config;

### 1.5.2 添加a模块

1. 在demo02的目录下添加js目录。

2. 在js目录下，创建a.js文件，其内容如下：

module.exports = **function**(){

**return** '<h1>我从a模块而来</h1>';

}

### 1.5.3 webpack打包运行

1. 调整index.html,把打包后的bundle.js文件引入

<**html**>

<**head**>

<**meta** charset="utf-8"/>

<**title**></**title**>

</**head**>

<**body**>

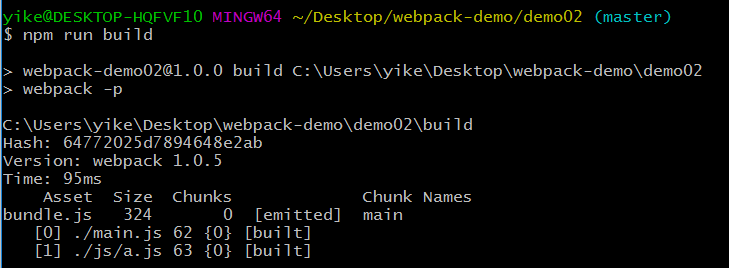
<**script** type="text/javascript" src="./build/bundle.js"></**script**>

</**body**>

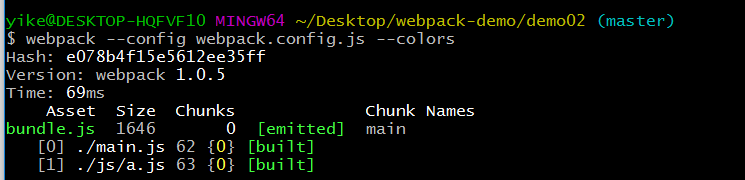
</**html**>

2. 在demo02目录下，运行

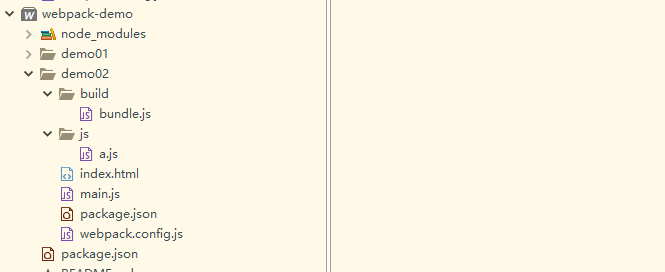
直接打包



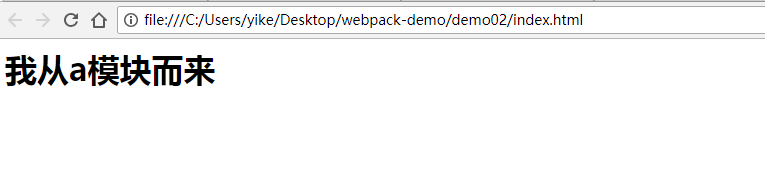
或打包过程彩色显示。



执行完成后，在demo02目录下，生成build目录。



3. 运行index.html，效果如下：



### 1.5.4 webpack的配置项resolve

额外的处理方案resolve,配置文件后缀名(extensions)，除了js，还有jsx、coffee等等。

alias配置项，可以为常用模块配置改属性，可以节省编译的搜索时间。例如：

resolve:{

extensions:['.js','.jsx'],

alias:{

'react':path.join(nodeModulesPath,'react/react.js')

}

}

## 1.6 案例 demo03内置插件

### 1.6.1修改webpack.config.js文件

新增内容

...

**var** webpack = require('webpack');

...

**const** config = {

...

plugins: [

*//压缩打包的文件*

**new** webpack.optimize.UglifyJsPlugin({

compress: **true**,

debug: **true**

})

]

}

...

### 1.6.2 webpack打包运行

在浏览器中运行的情况与demo02并无二样，但是打包成的bundle.js文件经过了压缩。

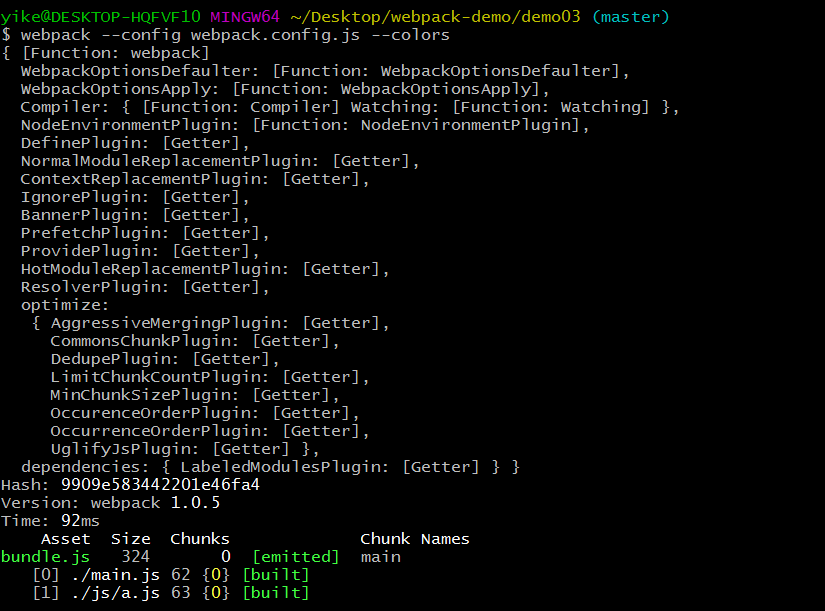
### 1.6.3 webpack的配置项plugins

顾名思义，就是配置要使用的插件。plugin是比loader功能更强大的插件，能使用更多的wepack api。从上面的webpack.config.js中不难发现，有些插件已经内置挂载到webpack对象下。那么webpack下都有哪些内置的插件呢？

### 1.6.4 webpack的内置插件

1. 我们去打印看看这个webpack对象

在webpack.config.js中console出webpack对象，然后打包运行，可以在git窗口中看到webpack对象，如下所示：



可以看到在webpack对象下挂载了多个插件，如：

banner-plugin

prefetch-plugin

hot-module-replacement-plugin

resolver-plugin

还有挂在webpack的optimize属性上的插件,如:

limit-chunk-count-plugin

min-chunk-size-plugin

ugligy-js-plugin

以上只列举了一部分,如果你有兴趣可以一一去探究.

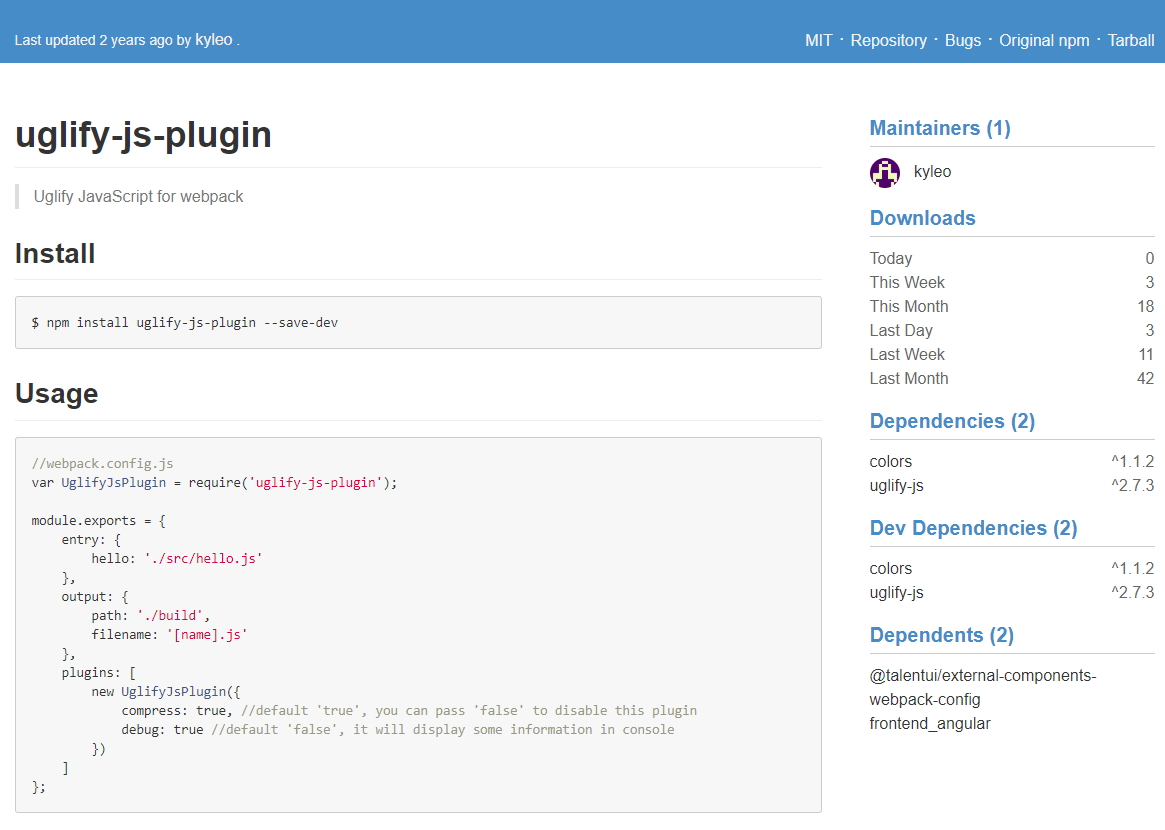
2. 如何使用webpack内置的这些插件

细心的同学，可能已经注意到我把webpack对象下的以驼峰命名的插件，拆成了小写，并且单词之间使用中划线进行分割。

这属于一个单独的插件，既然可以挂载到webpack下，当然也可以拆解出来，单独引入使用。具体使用方法可以查看其发布的包的地址，如ugligy-js-plugin的包的镜像地址：

<https://npm.taobao.org/package/uglify-js-plugin>

打开来看，你会发现关于这个包的详情介绍：



这里为什么会有安装这一步骤，是因为随着webpack的升级，webpack把内置的一些可用的插件放弃了一部分，发布成了一个单独的包，因此如果想在使用，就需要进行安装并引入才可以。关于插件的用法，具体参考package的地址中的Usage（用法）。

注意：如果有的插件在包的网站中搜索不到，则很无奈，你只能去百度or 谷歌or去官网探索一番了。

### 1.6.5 尝试下BannerPlugin插件

1.了解到其作用是为每个 chunk 文件头部添加 banner。

/\*\*

\* 插件选项，该插件的版本是2.2.0

\* webpack1.x下面的版本不用细究，因为你还会升级webpack，并不只想使用

\* webpack1.x的版本

\*/

{

banner: string, *// 其值为字符串，将作为注释存在*

raw: **boolean**, *// 如果值为 true，将直出，不会被作为注释*

entryOnly: **boolean**, *// 如果值为 true，将只在入口 chunks 文件中添加*

test: string | RegExp | Array,

include: string | RegExp | Array,

exclude: string | RegExp | Array,

}

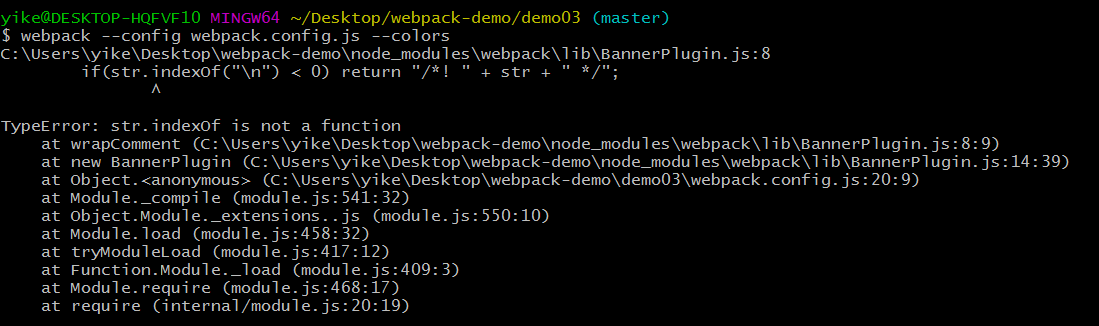
2.在webpack.config.js中的插件配置项中，配置BannerPlugin插件，我们也不使用上面那么复杂的配置。

new webpack.BannerPlugin({

banner: "作者：kkcode"

})

3. 打包试运行



并不理想，报错了。

4. 解决bug，打开BannerPlugin.js文件

构造函数如下：

function BannerPlugin(banner, options) {

if(!options) options = {};

this.banner = options.raw ? banner : wrapComment(banner);

this.entryOnly = options.entryOnly;

}

发现其构造函数需要的参数是两个，banner和options。并且当options.raw 不存在是调用wrapComment()函数，同时把banner传过去，报错是

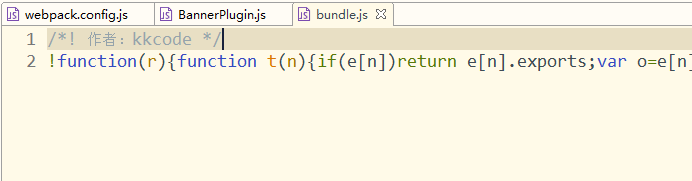
TypeError: str.indexOf is not a function

而我们在配置该插件时，传入的是一个对象。因此把配置调整为：

new webpack.BannerPlugin(banner: "作者：kkcode")

5. 再次打包运行

Bingo，打开bundle.js文件，发现头部已经多了一行作者注释信息。



其他插件就不一一尝试了，如果你有兴趣，Do it yourself

### 1.6.6 webpack内置plugin满足不了需求

当你的项目需求越来阅读时，你的webpack.config.js文件配置也就越来越复杂。有可能webpack内置的插件还满足不了，你对打包的理想期望。那么这就是接下来讲的demo04

## 1.7 案例demo04外部插件

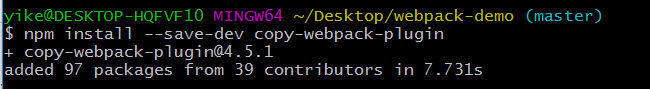
有时候，我们不希望webpack处理我们的文件，我们只需要把某些目录下的文件原样输出到build目录。copy文件需要通过插件"copy-webpack-plugin"来完成。

插件地址： <https://npm.taobao.org/package/copy-webpack-plugin>

### 1.7.1 安装copy-webpack-plugin

在项目的根目录下进行安装，以下所有的类似安装都是在本项目的根目录下进行的。

npm install --save-dev transfer-webpack-plugin



### 1.7.2 配置webpack.config.js

**var** CopyWebpackPlugin = require('copy-webpack-plugin');

*//其他节点省略*

plugins: [

*//把指定文件夹下的文件复制到指定的目录*

**new** CopyWebpackPlugin([

{from: 'copy', to: 'paste'}

], {debug: 'info'})

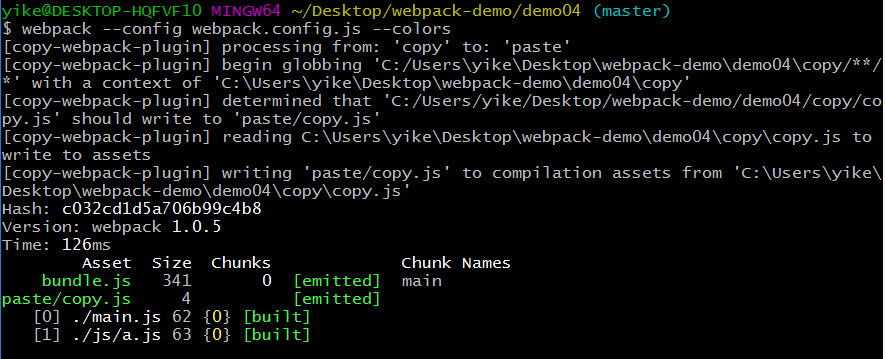
]

### 1.7.3 其他文件调整

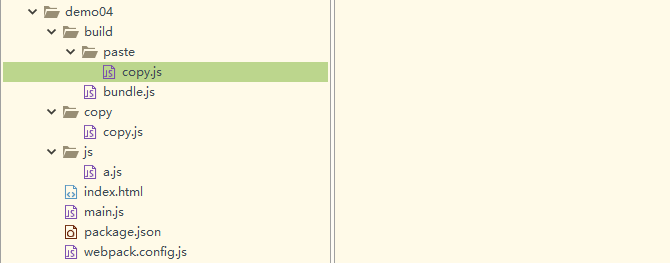
在demo04目录下，增加copy目录，copy目录下有个copy.js文件，文件内容，你随意写。

删除之前编译的build目录。等会看copy效果会更加明显。

### 1.7.4 webpack打包运行



### 1.7.5 最终效果



### 1.7.6 总结使用外部插件的流程

1. 按照需求在网上找到你所需要的插件

2. 在项目中安装

3. 配置webpack.config.js文件，即引入插件，使用插件

4. 调整运行，看效果。

## 1.8 案例demo05-babel-loader

现在比较流行js模块化开发，js模块化方案包括：

1. ES6 模块

**import** MyModule from './MyModule.js';

1. CommonJS

**var** MyModule = require('./MyModule.js');

1. AMD

define(['./MyModule.js'], **function** (MyModule) {});

上面已经演示了打包js模块。ES6的模块需要配置babel-loader来先把处理一下js文件。下面展示下打包ES模块。

babel-loader的包地址：<https://npm.taobao.org/package/babel-loader>

### 1.8.1 安装babel-loader

从包的安装地址，我们知道webpack 1.x | babel-loader <= 6.x ，因此安装babel-loader的版本必须小于 6。

npm install --save-dev babel-loader@5.0

### 1.8.2 配置webpack.config.js

// 新增内容

...

**var nodeModulesPath = path.resolve(\_\_dirname, 'node\_modules');**

...

**const** config = {

...

plugins: [],

module: {

loaders: [

{

test: /\.js$/,

loader: 'babel-loader?optional=runtime&stage=0',

exclude: [nodeModulesPath]

}

]

},

}

...

### 1.8.3 js目录下a.js内容调整

*// es6箭头函数*

**var** sum = (num1, num2) => num1 + num2;

module.exports = {

sum:sum

}

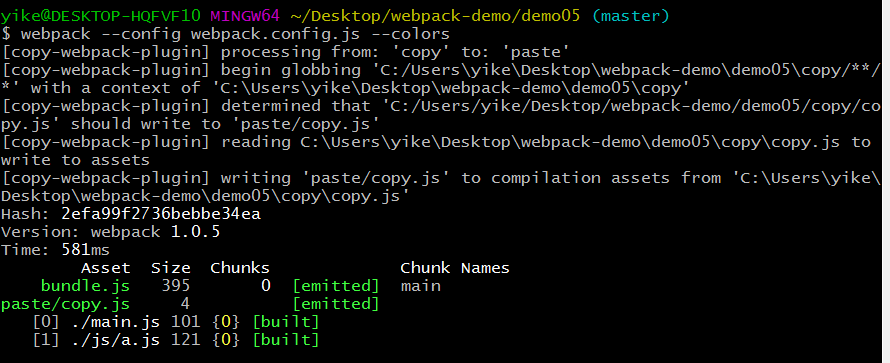
### 1.8.4 main.js内容调整

*// 引入a模块*

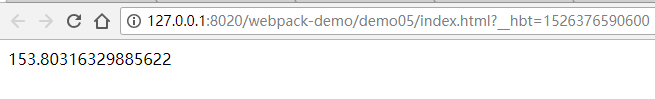
**var** a = require('./js/a');

document.write(a.sum(Math.random() \* 100,100));

### 1.8.5 webpack打包运行

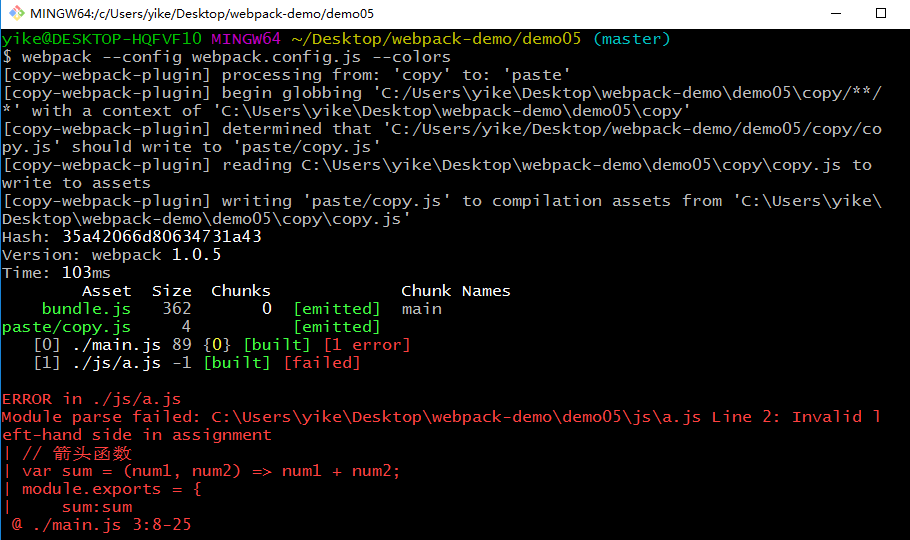


### 1.8.6 最终运行效果



### 1.8.7 测试不使用loader进行处理的情况

把webpack.config.js中配置的loader给删除掉，再次webpack打包，打包失败。



错误很清楚：

ERROR **in** ./js/a.js

Module parse failed

模块解析失败，不能解析箭头函数

## 1.9 案例demo06打包静态资源

css/sass/less images等都是需要打包的静态资源。本章节只说css类样式。

style-loader会把css文件嵌入到html的style标签里，css-loader会把css按字符串导出，这两个基本都是组合使用的。打包完成的文件，引用执行后，会发现css的内容都插入到了head里的一个style标签里。

### 1.9.1 安装css-loader和style-loader

npm install css-loader --save-dev

npm install style-loader --save-dev

### 1.9.2 配置webpack.config.js

module: {

loaders: [

...其他规则省略

{

test:/\.css$/,

loader:'style!css',

exclude: [nodeModulesPath]

}

]

},

### 1.9.3 在demo06下新建css目录和文件

**// style.css**

**html**,**body** {

**color**: **red**;

**background**: #ccc;

}

### 1.9.4 在main.js中引入style.css

*// 引入a模块*

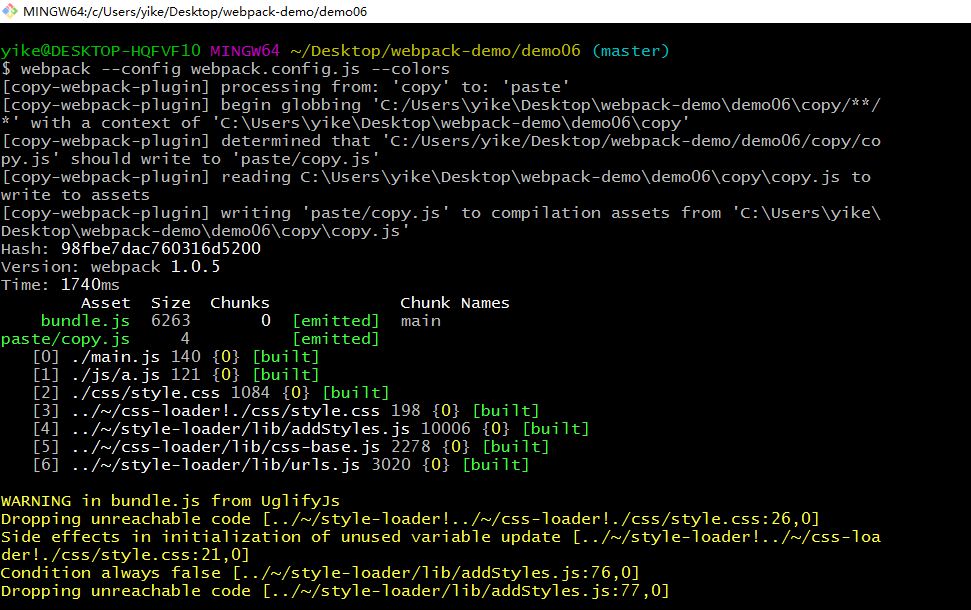
**var** a = require('./js/a');

*// 引入css模块*

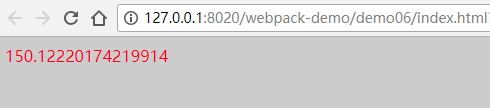
require('./css/style.css');

document.write(a.sum(Math.random() \* 100,100));

### 1.9.5 webpack打包运行



警告先不用管。浏览器中运行效果如下图所示：



F12 看源码，可以发现head中通过style标签的形式把css引入进来了，效果如下图所示：



## 2.0 案例demo07尝试sass或less

按照demo06的步骤来尝试使用less

### 2.0.1 安装less-loader

因为less-loader开发依赖less，所以需要把两个都安装上。因为webpack的1.x，所以我们也使用低版本的less-loader和低版本的less，以保证其能够正常运行。

npm install --save-dev less-loader@2.2.3 less@2.2.0

### 2.0.2 配置webpack.config.js

module: {

loaders: [

...其他规则省略

{

test:/\.less$/,

loader:'style!css!less',

exclude: [nodeModulesPath]

}

]

},

### 2.0.3 在demo07下新建less目录和文件

**// style.less**

**@nice-blue: #5B83AD;**

**html**,**body** {

**background-color**: @nice-blue;

}

### 2.0.4 在main.js中引入style.css

*// 引入a模块*

**var** a = require('./js/a');

*// 引入css模块*

require('./css/style.css');

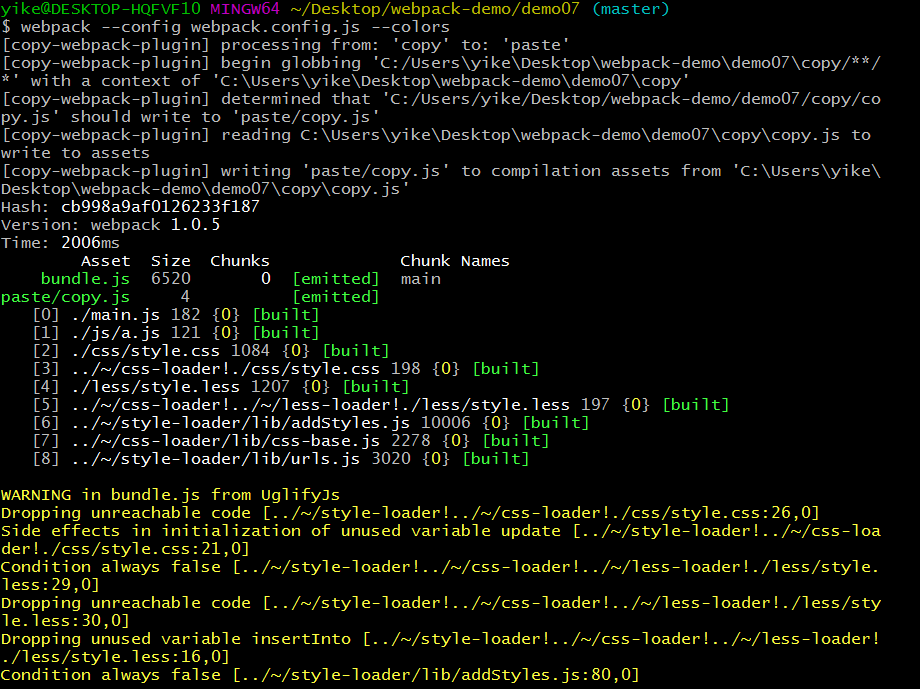
*// 引入less模块*

require('./less/style.less');

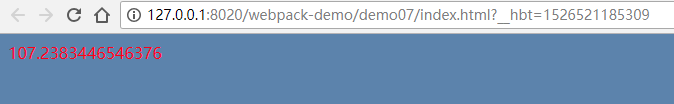
*// 在style.css 之后引入，样式覆盖，因此body的背景色应该是亮蓝色。*

document.write(a.sum(Math.random() \* 100,100));

### 2.0.5 webpack打包运行



警告先不用管。浏览器中运行效果如下图所示：



附：sass的使用同less使用步骤一样。

## 2.1案例demo08打包图片

可以通过url-loader把较小的图片转换成base64的字符串内嵌在生成的文件里。

### 2.1.1 安装url-loader

npm install --save-dev url-loader@0.6.0

### 2.1.2 配置webpack.config.js

module: {

loaders: [

...其他规则省略

{

test:/\.jpg$/,

loader:'url-loader',

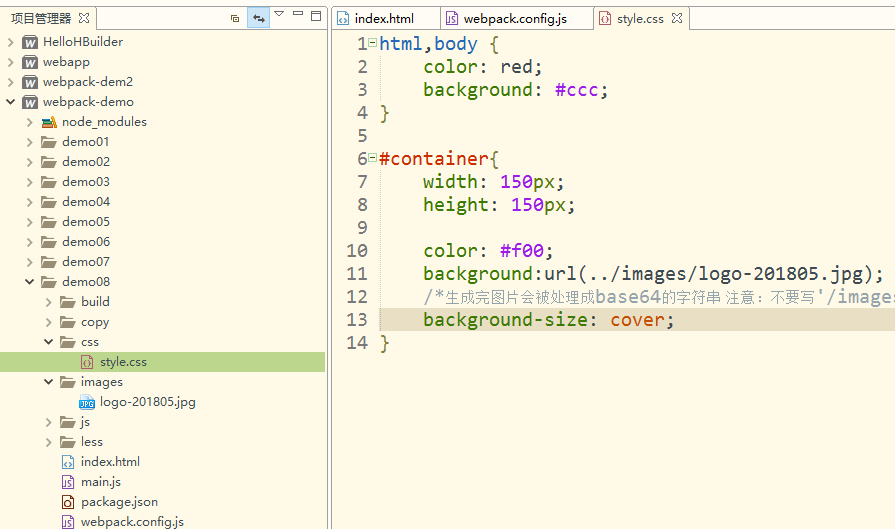
exclude: [nodeModulesPath]

}

]

},

### 2.1.3 在demo08下新建images目录和图片

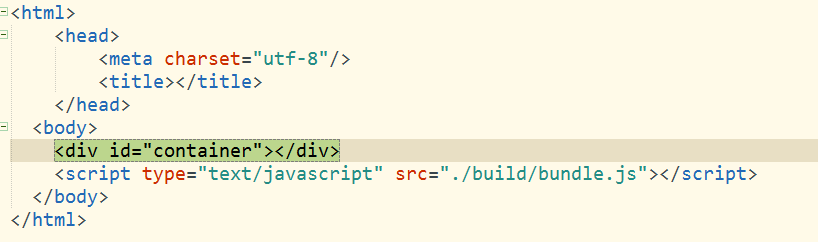


### 2.1.4 如上图所示在style.css 中引入图片

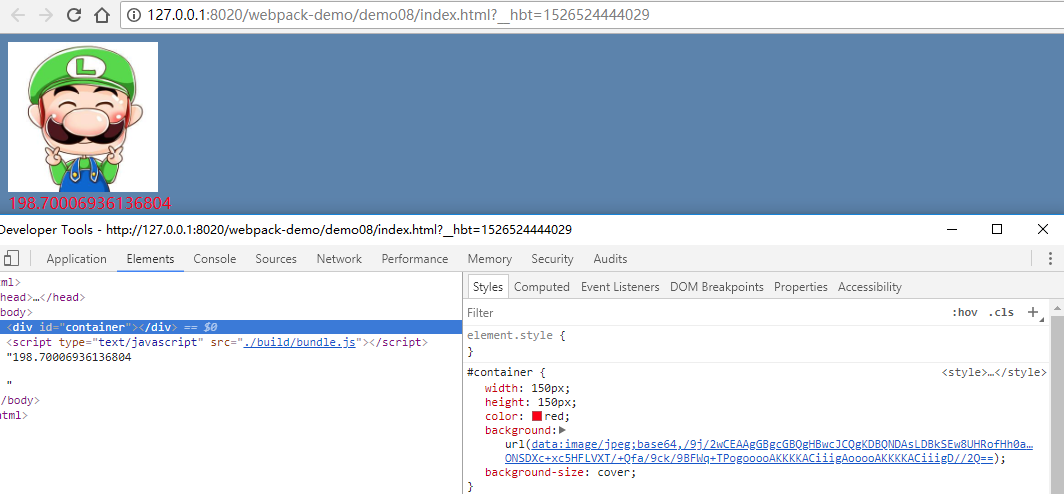
注意，不要在引入图片的时候加入引号，因为这会导致url-loader不会处理你的图片。

### 2.1.5 webpack打包运行

打包成功后，在index.html中加入如下一行代码：



然后在浏览器中运行，效果如下：



### 2.1.6 iconfont打包

Iconfont 打包与图片类似，只需要调整下webpack.config.js文件的配置

module: {

loaders: [

...其他规则省略

{

test:/\.(jpg|png|jpeg|woff|svg|ttf|eot)$/,

loader:'url-loader',

exclude: [nodeModulesPath]

}

]

},

css文件内容：

@**font-face** {**font-family**: 'iconfont';

**src**: **url**('fonts/iconfont.eot'); */\* IE9\*/*

**src**: **url**('fonts/iconfont.eot?#iefix') **format**('embedded-opentype'), */\* IE6-IE8 \*/*

**url**('fonts/iconfont.woff') **format**('woff'), */\* chrome、firefox \*/*

**url**('fonts/iconfont.ttf') **format**('truetype'), */\* chrome、firefox、opera、Safari, Android, iOS 4.2+\*/*

**url**('fonts/iconfont.svg#iconfont') **format**('svg'); */\* iOS 4.1- \*/*

}

执行打包后会把字体文件都转换成base64字符串内容到文件里。这里有个头疼的问题，就是每个浏览器支持的字体格式不一样，由于把全部格式的字体打包进去，造成不必要的资源浪费。这里不在演示，自己来。

## 2.2 案例demo09-html模板文件

我以打包handlebars的模块为例，来演示下打包模块的过程。

### 2.2.1安装handlebars和handlebars-loader

npm install --save-dev handlebars-loader@1.1.1 handlebars@4.0.0

### 2.2.2 配置webpack.config.js

module: {

loaders: [

...其他规则省略

{

test: /\.handlebars$/,

loader: "handlebars-loader",

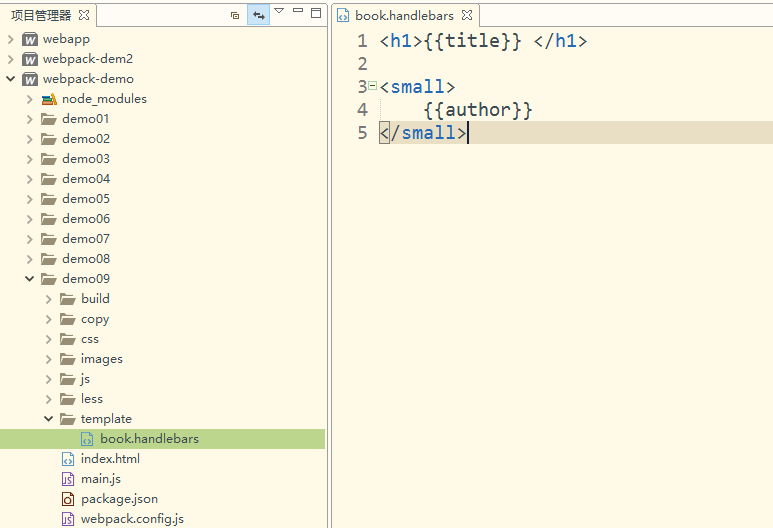
exclude: [nodeModulesPath]

}

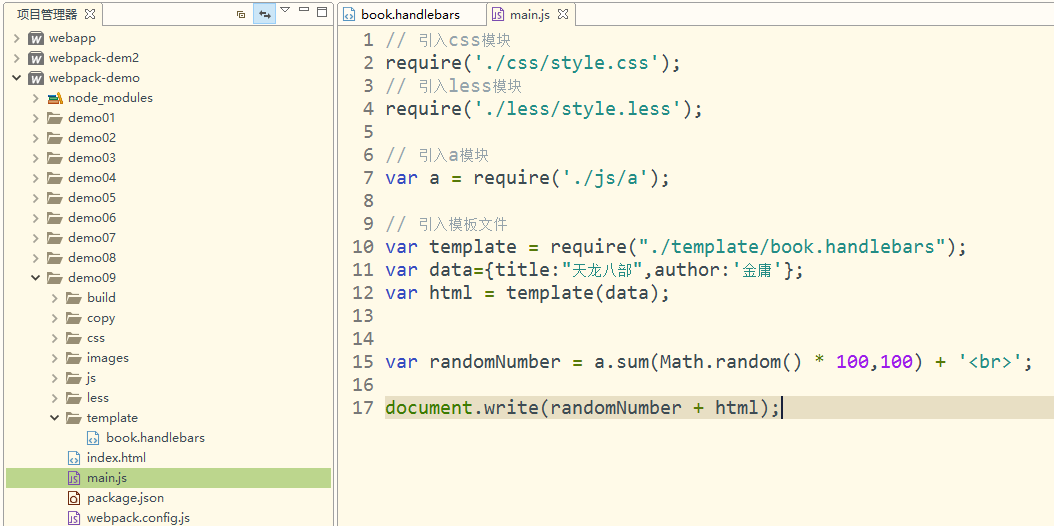
]

},

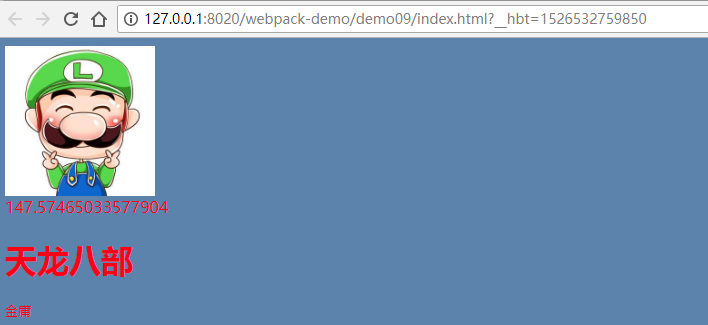
### 2.2.3 在demo09目录下新建template目录和文件



### 2.2.4 在main.js中引入模板文件并处理



### 2.2.5 webpack打包运行



## 2.3 案例demo10多入口

### 2.3.1配置webpack.config.js

entry: {

m1:path.resolve(\_\_dirname, 'main.js'),

m2:path.resolve(\_\_dirname, 'main1.js')

},

output: {

path: buildPath,

filename:"[name].js" // 注意这里使用了name变量

}

### 2.3.2 新建另一个入口文件main1.js

把之前的main.js中的一部分拆开放到main1.js中。

// 引入模板文件

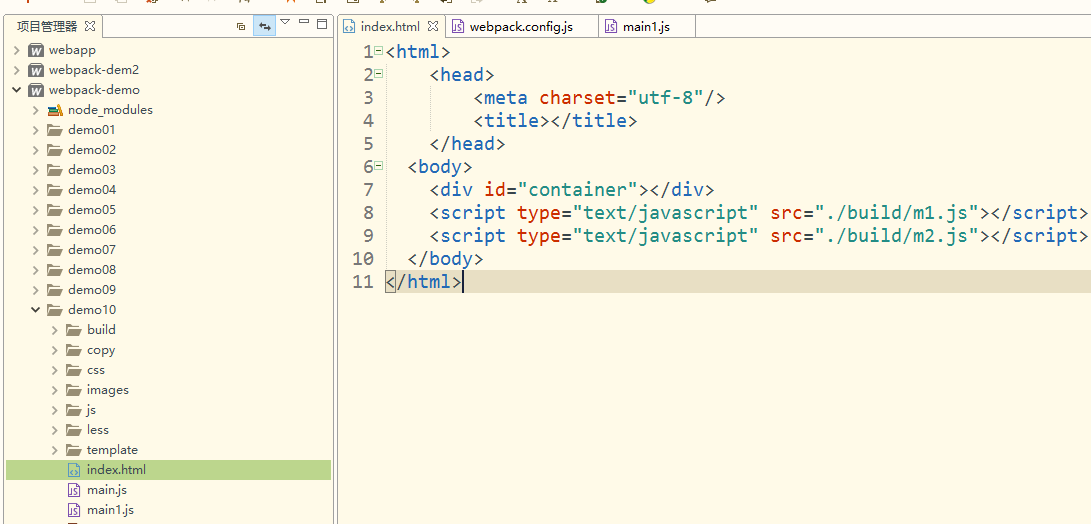
var template = require("./template/book.handlebars");

var data={title:"天龙八部",author:'金庸'};

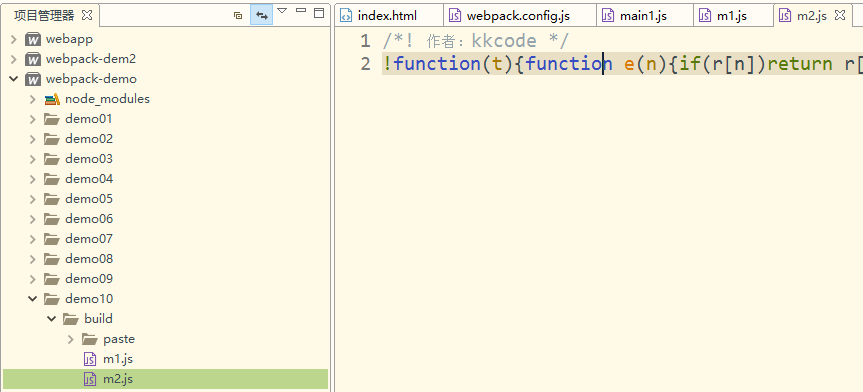
var html = template(data);

document.getElementById('container').innerHTML = html;

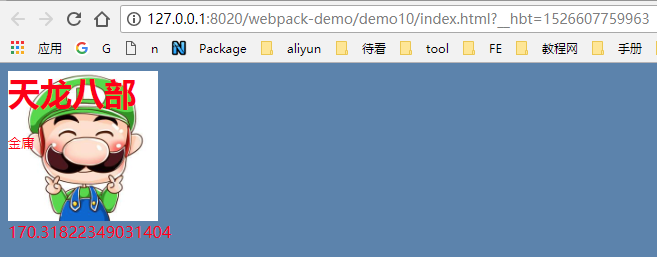
### 2.3.3 调整index.html



### 2.3.4 webpack打包运行



生成了m1.js和m2.js,最终浏览器运行效果如下图：



## 2.4 案例demo11区分开发及生产环境

在webpack.config.js使用process.env.NODE\_ENV进行判断

### 2.4.1 在demo11目录下的package.json文件配置

在package.json里面的script设置环境变量，注意mac与windows的设置方式不一样。

"scripts": {

"publish-mac": "export NODE\_ENV=prod&&webpack -p --progress --colors",

"publish-win": "set NODE\_ENV=prod&&webpack -p --progress --colors",

"dev-mac": "export NODE\_ENV=dev&&webpack-dev-server",

"dev-win": "set NODE\_ENV=dev&&webpack-dev-server"

}

### 2.4.2 配置webpack.config.js

**var** NODE\_ENV = process.env.NODE\_ENV;

**var** path = require('path');

**var** webpack = require('webpack');

**var** buildPath = path.resolve(\_\_dirname, "build");

**var** nodeModulesPath = path.resolve(\_\_dirname, 'node\_modules');

**var** CopyWebpackPlugin = require('copy-webpack-plugin');

**var** config = {

entry: {

m1:path.resolve(\_\_dirname, 'main.js'),

m2:path.resolve(\_\_dirname, 'main1.js')

},

output: {

path: buildPath,

filename:"[name].js" *// 注意这里使用了name变量*

},

resolve: {

extentions: ["", ".js"] *// 当rewquire的模块找不到时，添加这些后缀。*

},

devtool: 'eval',

plugins: [

**new** webpack.BannerPlugin("作者：kkcode"),

*// 把copy目录下的文件复制到build/paste目录下*

**new** CopyWebpackPlugin([{

from: 'copy',

to: 'paste'

}], {

debug: 'info'

})

],

module: {

loaders: [{

test: /\.js$/, *//注意是正则表达式，不要加引号*

loader: 'babel-loader?optional=runtime&stage=0', *//babel模块相关的功能请自查，这里不做介绍*

exclude: [nodeModulesPath]

},

{

test: /\.css$/,

loader: 'style!css',

exclude: [nodeModulesPath]

},

{

test: /\.less$/,

loader: 'style!css!less', *//less配置:style!css!less 执行顺序:左<--右*

exclude: [nodeModulesPath]

},

{

test: /\.jpg$/,

loader: 'url-loader',

exclude: [nodeModulesPath]

},

{

test: /\.handlebars$/,

loader: "handlebars-loader",

exclude: [nodeModulesPath]

}

]

},

}

**if**(NODE\_ENV === "prod"){*//判断是生产环境执行生产配置*

**delete** config.devtool;

config.plugins = [

*//压缩打包的文件*

**new** webpack.optimize.UglifyJsPlugin({

compress: {

*//supresses warnings, usually from module minification*

warnings: **false**

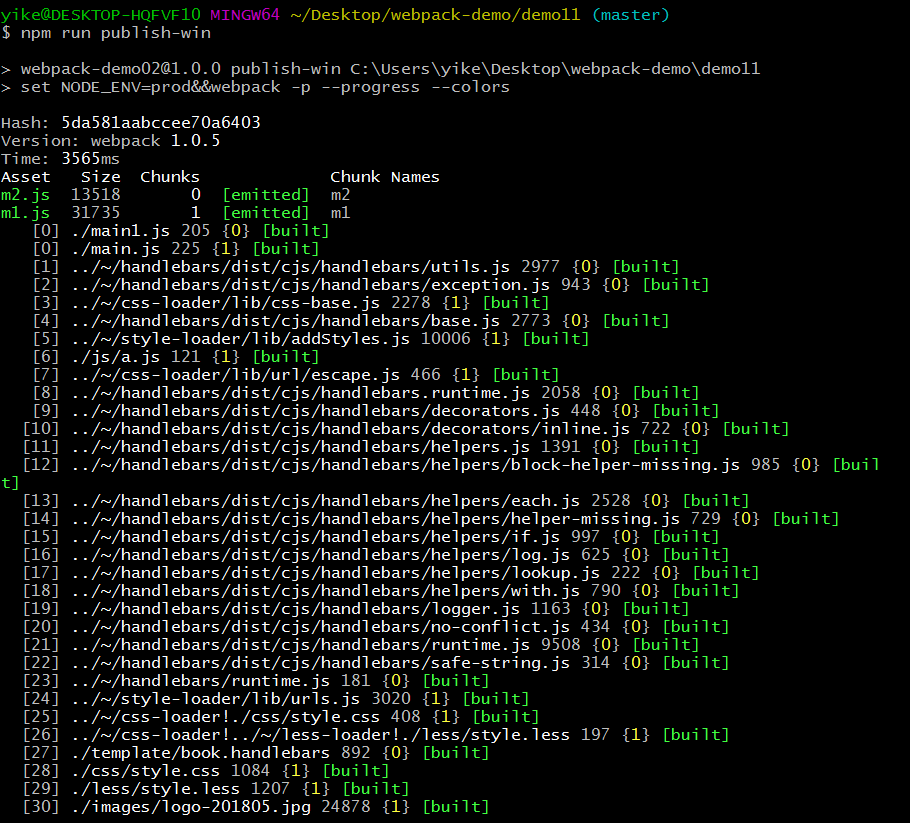
}

})];

}

module.exports = config;

### 2.4.3生产环境尝试运行



### 2.4.4 最终生成的文件

