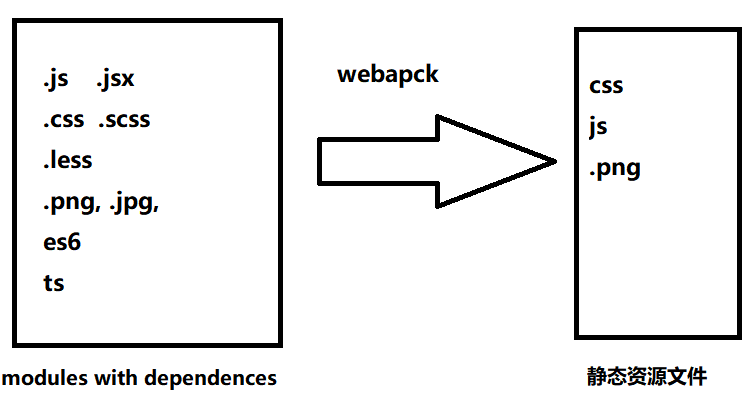
什么是WebPack？

简单地说，Webpack其最核心的功能就是 解决模板之间的依赖，把各个模块按照特定的规则和顺序组织在一起，最终合并成一个JS文件（比如bundle.js）。这个整个过程也常常被称为是模块打包。

WebPack可以看做是模块打包机：它做的事情是，分析你的项目结构，找到

JavaScript模块以及其它的一些浏览器不能直接运行的拓展语言（Sass，

TypeScript等），并将其转换和打包为合适的格式供浏览器使用。



# 为什么要用Webpack

一直以来，在开发Web页面或Web应用程序的时候，都习惯性的将不同资源放置在不同的文件目录之中，比如图片放置在images（或img）下，样式文件放置在styles(或css)中，脚本文件放在js和模板文件放置在pages中。一直以来，发布的时候都会一次性的将所有资源打包发布，不管这些资源用到了还是没用到（事实上很多时候自己都分不清楚哪资源被使用）。用一句话来描述就是：依赖太复杂，太混乱，无法维护和有效跟踪。比如哪个样式文件引用了a.img，哪个样式文件引用了b.img；另外页面到底是引用了a.css呢还是b.css呢？

而Webpack这样的工具却能很多好的解决它们之间的依赖关系，使其打包后的结果能运行在浏览器上。其目前的工作方式主要被分为两种：

将存在依赖关系的模块按照特定规则合并成为单个.js文件，一次性全部加载进页面

在页面初始时加载一个入口模块，其他模块异步加载

**Webpack的主要优势：**

• webpack-dev-server搭建本地环境，进行热更新

• 预处理（Less，Sass，ES6，TypeScript,……）

• 图片添加hash,方便线上CDN缓存

• 自动处理CSS3属性前缀

• 单文件，多文件打包 。。。。

• 有完备的代码分割解决方案

• 模块化开发（import，require） ，支持多种模块标准

• 压缩js代码

• 复制

相比于Parcel、Rollup具有同等功能的工具而言，Webpack还具有其他的优势：

* Webpack支持多种模块标准：这对于一些同时使用多种模块标准的工程非常有用，Webpack会帮我们处理好不同类型模块之间的依赖关系
* Webpack有完备的代码分割解决方案：它可以分割打包后的资源，首屏只加载必要的部分，不太重要的功能放到后面动态地加载
* Webpack可以处理各种类型的资源：除了JavaScript之外，Webpack还可以处理样式、模板、图片等资源。开发者要做的只是将之些资源导入，而无需关注其他. 另外，Webpack还拥有一个强大的社区。这也是其受开发者青眯的原因之一。接下来，我们还是实际一点，动手来撸码。

# Webpack4与Webpack3版本的区别：

1. 增加了mode属性，可以是 development 或 production

"scripts": {

"dev": "webpack --mode development",

"build": "webpack --mode production"

}

通过mode, 你可以轻松设置打包环境。

**如果你将 mode 设置成 development**，你将获得最好的开发阶段体验。这得益于webpack针对开发模式提供的特性：

* 浏览器调试工具
* 注释、开发阶段的详细错误日志和提示
* 快速和优化的增量构建机制

**如果你将mode设置成了 production**, webpack将会专注项目的部署，包括以下特性：

* 开启所有的优化代码
* 更小的bundle大小
* 去除掉只在开发阶段运行的代码
* Scope hoisting和Tree-shaking

1. webpack4在loader,optimize上进行了很多改动。

**module.loaders**改为**module.rules**

**module.loaders.loader**改成**module.loaders.use**

**loader**的”**-loader**”均不可省略

1. 插件和优化

webpack4删除了合并相同的文件CommonsChunkPlugin插件，它使用内置API optimization.splitChunks 和 optimization.runtimeChunk，这意味着webpack会默认为你生成共享的代码块。

其它插件变化如下:

NoEmitOnErrorsPlugin 废弃，使用optimization.noEmitOnErrors替代，在生产环境中默认开启该插件。

ModuleConcatenationPlugin 废弃，使用optimization.concatenateModules替代，在生产环境默认开启该插件。

NamedModulesPlugin 废弃，使用optimization.namedModules替代，在生产环境默认开启。

uglifyjs-webpack-plugin升级到了v1.0版本, 默认开启缓存和并行功能。

之前是这样：

plugins : [

new webpack.optimize.CommonsChunkPlugin({

name : 'main',

children : true,

minChunks : 2,

})]

此段代码已经在webpack4的高版本里面被移除

换成了如下的配置选项：

optimization: {

        splitChunks: {

            cacheGroups: {

                commons: {

                    name: "commons",

                    chunks: "initial",

                    minChunks: 2

                }

            }

        }

}

1. 升级extract-text-webpack-plugin 插件 yarn add extract-text-webpack-plugin@next -D

# Webpack5展望

已经有不少关于webpack5的计划正在进行中了，包括以下：

* 对WebAssembly的支持更加稳定
* 支持开发者自定义模块类型
* 去除ExtractTextWebpackPlugig插件，支持开箱即用的CSS模块类型
* 支持Html模块类型
* 持久化缓存

# 从零开始构建你自己的开发环境

为了更好的理解Webpack能帮我们做什么，我打算从零开始构建一个属于自己的开发环境。可能在后面的内容中会涉及到很多关键词，比如Webpack、loaders、Babel、sourcemaps、Vue、React、TypeScript，CSS Modules等等。

## 配置版本说明：

**node:** v10.16.0

**Yarn:** 1.16.0

**Webpack:** 4.34.0

**Webpack-cli:** 3.3.4

演示的系统是window ， 以及node 的下载，安装成以后，会自带npm命令，如果想体验yarn 命令 可以通过npm install yarn -g 或者下载安装

环境支持：

当使用[webpack4](https://twitter.com/hashtag/webpack?src=hash&ref_src=twsrc^tfw)时，确保使用 Node.js的版本 >= 8.9.4。因为webpack4使用了很多JS新的语法，它们在新版本的 v8 里经过了优化。

### 安装命令yarn或者npm的选择：

yarn的安装和使用参考：<https://blog.csdn.net/x550392236/article/details/79205812>

### 方式一：yarn命令安装：

yarn global add webpack webpack-cli

yarn add webpack webpack-cli --dev //简写 yarn add webpack webpack-cli -D

--dev 指安装到devDependencies：开发时的依赖包

--save dependencies：运行程序时的依赖包

### 方式二：NPM命令安装：

或者： npm install webpack webpack-cli -g

npm install webpack webpack-cli --save-dev

或者可以指定webpack版本安装：

npm i -D webpack@3

npm i -D 是 npm install --save-dev 的简写，是指安装模块并保存到

注意，在命令终端使用npm i安装依赖关系时，如果带后缀 -D（或--save-dev） 安装的包会记录在"devDependencies"下；如果使用 --save后缀安装的包会记录在"dependencies"下。

## Step01:初始化项目，生成package.json

进入到新创建的项目目录下,执行 yarn init -y 或者npm init 生成package.json

首先在你的本地创建一个项目，比如我这里创建了一个webpack-sample项目：

mkdir webpack-sample && cd webpack-sample

进入到新创建的项目目录下，执行yarn init或者yarn init -y命令来初始化项目，执行完该命令之后，在你的命令终端会看到类似下图这样的命令询问，你可以根据你自己的需要去输入你想要的内容，或者一路Enter键执行下去：

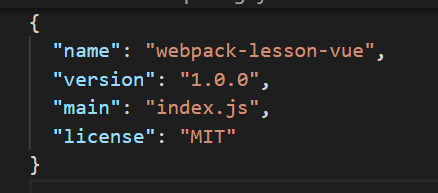
此时你的项目根目录下会增加一些文件和文件夹：

|--webpack-sample/

|----package.json

|----package-lock.json

其中package.json文件里将包含一些项目信息：



而**package-lock.json**文件是当 node\_modules/ 或 package.json 发生变化时自动生成的文件，它的主要功能是 确定当前安装的包的依赖，以便后续重新安装的时候生成相同的依赖，而忽略项目开发过程中有些依赖已经发生的更新。

我们对package.json文件只做一个修改，删除"main": "index.js"入口，并添加"private":true选项，以便确保安装包是私有的，这样可以防止意外发布你的代码

## Step02：安装Webpack和初始配置Webpack

在这一步，先来安装Webpack。执行下面的命令安装Webpack配置所需要的包：

yarn add webpack webpack-cli webpack-dev-server --dev

此时打开package.json文件，你会发现在文件中有一个新增项 devDependencies：

{

 // 其他项信息在这省略，详细请查看该文件

"devDependencies": {

"webpack": "^4.35.0",

"webpack-cli": "^3.3.5",

 "webpack-dev-server": "^3.7.2"

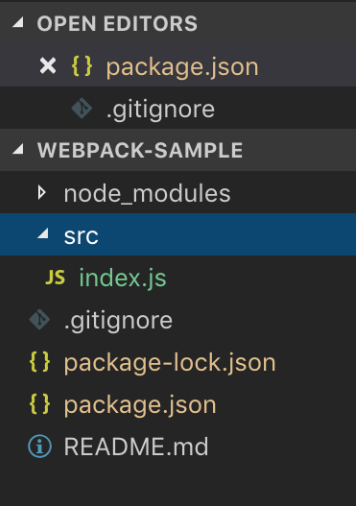
 }

}

为了验证Webpack是否能正常工作，这个时候我们需要创建一些新的文件。在webpack-sample根目录下创建/src目录，并且在该目录下创建一个index.js文件：

mkdir src && cd src && touch index.js

执行完上面的命令，你会发现你的项目目录结构变成下图这样：



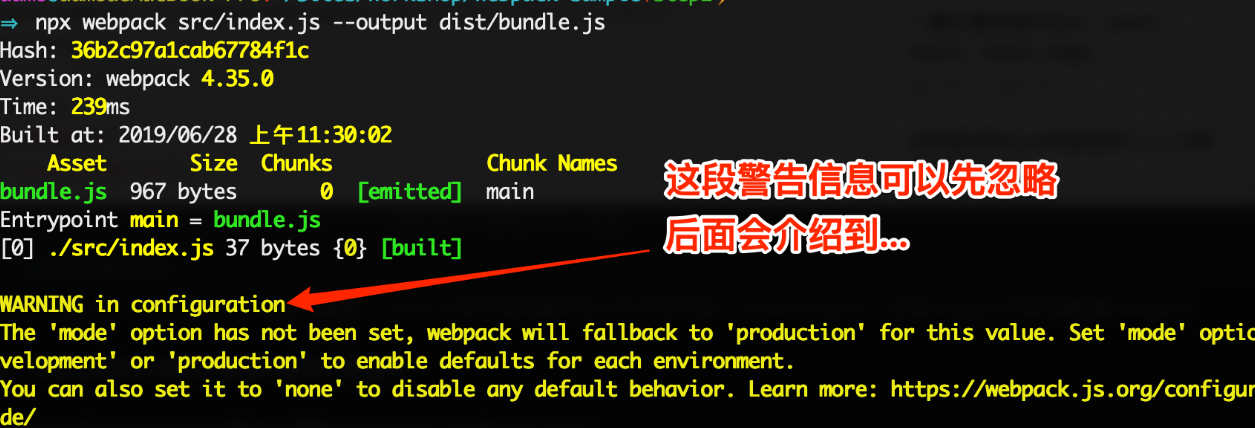
我们在新创建的/src/index.js文件下添加一行最简单的JavaScript代码：

console.log("Hello, Webpack!(^\_^)~~")

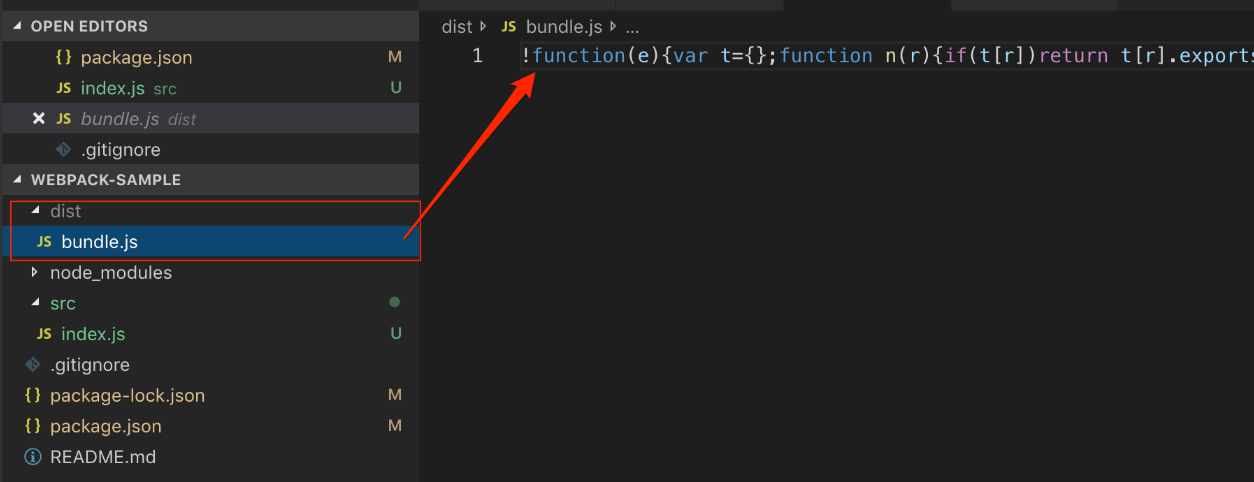
保存之后回到命令终端，第一次执行有关于Webpack相关的命令：

webpack src/index.js --output dist/bundle.js

执行完上面的命令后，如果看到下图这样的结果，那么要恭喜你，Webpack的安装已经成功，你可以在你的命令终端执行有关于Webpack相关的命令：

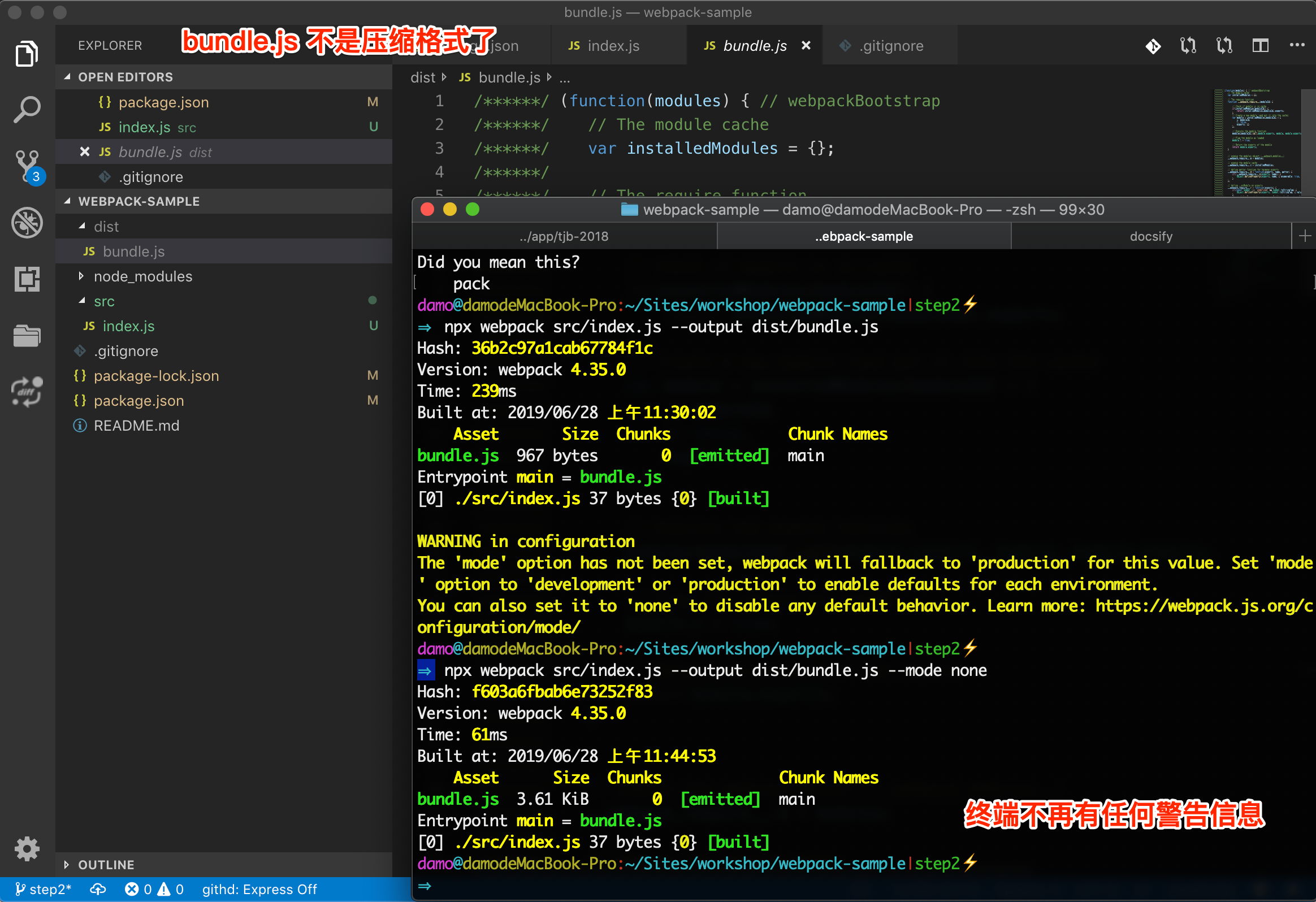


回到项目中，会发现项目根目下自动创建了一个/dist目录，而且该目录下包含了一个bundle.js文件：



执行完上面的命令之后，可以看到有相关的警告信息。那是因为[Webpack4增加了mode属性](https://webpack.docschina.org/concepts/" \l "%E6%A8%A1%E5%BC%8F-mode-" \t "https://www.w3cplus.com/javascript/_blank)，用来表示不同的环境。mode模式具有development，production 和 none三个值，其默认值是production 。也就是说，在执行上面的命令的时候，我们可以带上相应的mode属性的值，比如说，设置none来禁用任何默认行为

npx webpack src/index.js --output dist/bundle.js --mode none



执行到这里，只知道我们可以运行Webpack相关命令。并不知道/src/index.js的代码是否打包到/dist/bundle.js中。为此，我们可以在/dist目录下创建一个index.html

cd dist && touch index.html

并且将生成出来的bundle.js引入到新创建的index.html：



在浏览器中打开/dist/index.html，或者在命令行中执行：

⇒ yarn global add http-server 或者 npm i -g http-server

⇒ http-server dist

http-server是一个启动服务器的npm包，执行上面的命令之后，就可以在浏览器中访问http://127.0.0.1:8080/（访问的/dist/index.html），在浏览器的console.log控制台中，可以看到src/index.js的脚本输出的值：

或者通过vscode的插件，安装一个 live Server 也可以直接在vscode里直接起服务，

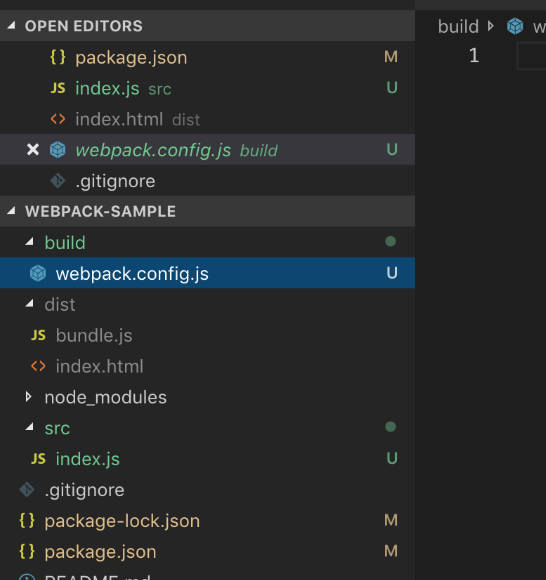
<https://jingyan.baidu.com/article/9989c746e671e6f648ecfea7.html>

不过，当你需要构建的东西越复杂，需要的标志就会越多。在某种程度上说，就会变得难以控制。这个时候我们就需要一个文件来管理这些配置。接下来我们需要创建一个webpack.config.js这样的一个文件，用来配置Webpack要做的事情。注意，这个文件是一个node.js文件，所以你可以在任何节点文件中执行任何你能够执行的操作。你也可以写成json文件，但是node文件更强大一些。

首先创建Webpack的配置文件，在webpack-sample根目录下创建一个/build目录，然后在该目录下添加一个名为webpack.config.js文件：

mkdir build && cd build && touch webpack.config.js

执行完上面的命令之后，你会发现你的项目文件目录结构变成下面这样了：



这个时候，新创建的webpack.config.js文件里面是一片空白，它就是Webpack的配置文件，将会导出一个对象的JavaScript文件。我们需要在这个文件中添加一些配置：

var webpack = require('webpack');

var path = require('path');

var DIST\_PATH = path.resolve(\_\_dirname, '../dist');

// 声明/dist的路径 为成绝对路径

module.exports = {

// 入口JS路径

// 指示Webpack应该使用哪个模块，来作为构建其内部依赖图的开始

entry: path.resolve(\_\_dirname,'../src/index.js'),

支持单文件，多文件打包

 // entry: './src/index.js',   //方式一

 // entry: ['./src/index.js','./src/main.js'], //方法二

//entry: {

// index:'./src/index.js',

// main:'./src/main.js'

// }

// 编译输出的JS入路径

// 告诉Webpack在哪里输出它所创建的bundle，以及如何命名这些文件

output: {

path: DIST\_PATH, // 创建的bundle生成到哪里

filename: 'bundle.js', // 创建的bundle的名称

},

// 模块解析

module: { },

// 插件

plugins: [ ],

// 开发服务器

devServer: { }

}

Webpack配置是标准的 [Node.js CommonJS模块](https://nodejs.org/docs/latest/api/modules.html" \t "https://www.w3cplus.com/javascript/_blank)，它通过require来引入其他模块，通过module.exports导出模块，由Webpack根据对象定义属性进行解析。

上面很简单，到目前为止只通过[entry设置了入口起点](https://webpack.docschina.org/concepts/entry-points/" \t "https://www.w3cplus.com/javascript/_blank)，然后通过[output配置了打包文件输出的目的地和方式](https://webpack.docschina.org/concepts/output/" \t "https://www.w3cplus.com/javascript/_blank)。你可能也发现了，在配置文件中还有module、plugins和devServer没有添加任何东西。不需要太急，后面会一步一步带着大家把这里的内容补全的，而且随着配置的东西越来越多，整个webpack.config.js也会更变越复杂。

完成webpack.config.js的基础配置之后，回到package.json文件，并在"scripts"下添加"build": "webpack --config ./build/webpack.config.js"：

// package.json {

// ...

"scripts": {

"build": "webpack --config ./build/webpack.config.js",

  "test": "echo \"Error: no test specified\" && exit 1"

},

}

这样做的，为让我们直接在命令终端执行相关的命令就可以实现相应的功能。比如上面配置的build，在命令终端执行：

⇒ npm run build

面的命令执行的效果前面提到的webpack src/index.js --output dist/bundle.js --mode none等同。同样有警告信息，主要是mode的配置没有添加。在上面的配置中添加:

{

"scripts": {

"build": "webpack --config ./build/webpack.config.js --mode production",

"test": "echo \"Error: no test specified\" && exit 1"

}, }

再次执行npm run build，不会再有警告信息。你可以试着修改/src/index.js的代码：

alert(`Hello, Webpack! Let's Go`);

重新编译之后，打开/dist/index.html你会发现浏览器会弹出alert()框：

为了开发方便，不可能通过http-server来启用服务。我们可以把这部分事件放到开发服务器中来做，对应的就是devServer，所以我们接着在webpack.config.js中添加devServer相关的配置：

### webpack-dev-server

webpack-dev-server是一个用来快速搭建本地运行环境的工具。webpack-dev-server是一个小型的Node.js Express服务器,它使用webpack-dev-middleware来服务于webpack的包,除此自外，它还有一个通过Sock.js来连接到服务器的微型运行时.

命令简单webpack-dev-server或配置命令脚本快捷运行,模拟服务器运行情况，进行上线前调试。

安装： yarn add webpack-dev-server --dev

#### **[webpack-dev-server和webpack-dev-middleware的区别](https://www.cnblogs.com/wangpenghui522/p/6826182.html)**

##### webpack-dev-server

webpack-dev-server实际上相当于启用了一个express的Http服务器+调用webpack-dev-middleware。它的作用主要是用来伺服资源文件。这个Http服务器和client使用了websocket通讯协议，原始文件作出改动后，webpack-dev-server会用webpack实时的编译，再用webpack-dev-middleware将webpack编译后文件会输出到内存中。适合纯前端项目，很难编写后端服务，进行整合。

##### webpack-dev-middleware

 webpack-dev-middleware输出的文件存在于内存中。你定义了 webpack.config，webpack 就能据此梳理出entry和output模块的关系脉络，而 webpack-dev-middleware 就在此基础上形成一个文件映射系统，每当应用程序请求一个文件，它匹配到了就把内存中缓存的对应结果以文件的格式返回给你，反之则进入到下一个中间件。

因为是内存型文件系统，所以重建速度非常快，很适合于开发阶段用作静态资源服务器；因为 webpack 可以把任何一种资源都当作是模块来处理，因此能向客户端反馈各种格式的资源，所以可以替代HTTP 服务器。事实上，大多数 webpack 用户用过的 webpack-dev-server 就是一个 express＋webpack-dev-middleware 的实现。二者的区别仅在于 webpack-dev-server 是封装好的，除了 webpack.config 和命令行参数之外，很难去做定制型开发。而 webpack-dev-middleware 是中间件，可以编写自己的后端服务然后把它整合进来，相对而言比较灵活自由。

##### webpack-hot-middleware：

是一个结合webpack-dev-middleware使用的middleware，它可以实现浏览器的无刷新更新（hot reload），这也是webpack文档里常说的HMR（Hot Module Replacement）。HMR和热加载的区别是：热加载是刷新整个页面。

// webpack.config.js

// 开发服务器

devServer: {

hot: true, // 热更新，无需手动刷新

contentBase: DIST\_PATH,  //热启动文件所指向的文件路径

// host: '0.0.0.0', // host地址

port: 8080, // 服务器端口

historyApiFallback: true, // 该选项的作用所用404都连接到index.html

proxy: {

"/api": "http://localhost:3000"

// 代理到后端的服务地址，会拦截所有以api开头的请求地址

 }

}

有关于devServer更详细的配置参数描述

devServer:{

port:7788,//控制端口

open:true //是否自动打开默认浏览器

headers :在所有响应中添加首部内容

historyApiFallback: 使用 [HTML5 History API](https://developer.mozilla.org/en-US/docs/Web/API/History) 时，任意的 404 响应都可能需要被替代为 index.html

host:’0.0.0.0’ //指定使用一个 host。默认是 localhost。如果你希望服务器外部可访问，指定如下

hot:true //启用 webpack 的模块热替换特性

https：true //默认情况下，dev-server 通过 HTTP 提供服务。也可以选择带有 HTTPS 的 HTTP/2 提供服务

Info: true //是否要输出一些打包信息

noInfo: true //启用 noInfo 后，诸如「那些显示的 webpack 包(bundle)信息」的消息将被隐藏

Proxy：//这样启用代理 ，[https://www.webpackjs.com/configuration/dev-server/#devserver-proxy](https://www.webpackjs.com/configuration/dev-server/" \l "devserver-proxy)

progress ：将运行进度输出到控制台

publicPath ： //此路径下的打包文件可在浏览器中访问，默认 publicPath是 "/"

//[https://www.webpackjs.com/configuration/dev-server/#devserver-publicpath-](https://www.webpackjs.com/configuration/dev-server/" \l "devserver-publicpath-)

useLocalIp: true //是否在打包的时候用自己的IP

watchContentBase：true //告诉服务器监视devserver.contentbase选项提供的文件。文件更改将触发整页重新加载

}

和build类似，需要在package.json的scripts中添加相关的命令：

// package.json

"scripts": {

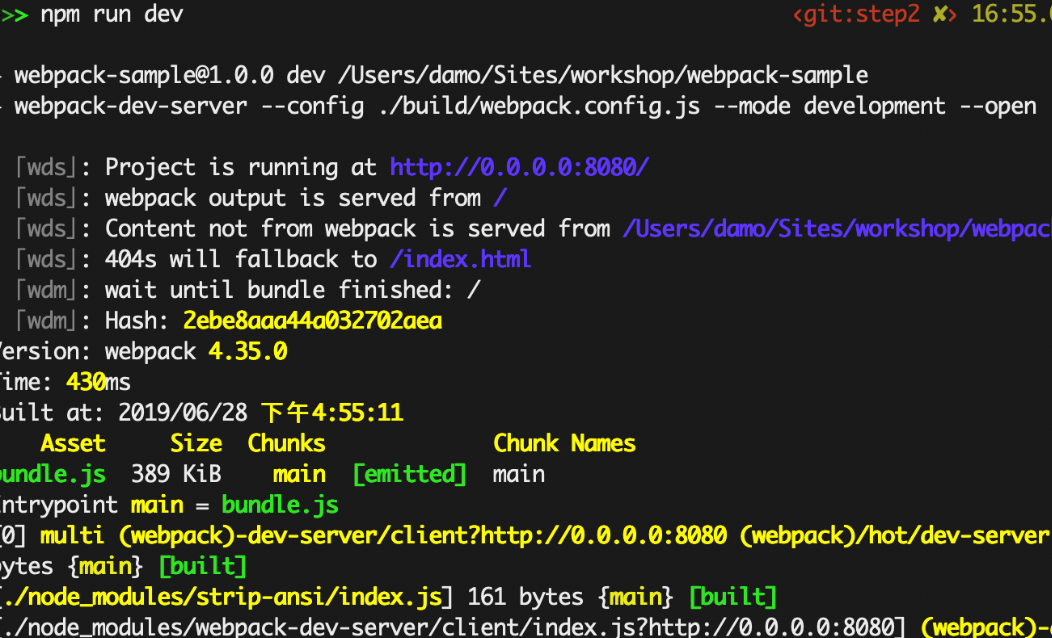
"build": "webpack --config ./build/webpack.config.js --mode production",

"dev": "webpack-dev-server --config ./build/webpack.config.js ",

"test": "echo \"Error: no test specified\" && exit 1"

},

保存所有文件，在命令行中执行npm run dev就可以启动服务器：



你可以验证一下，修改/src/index.js：

document.addEventListener('DOMContentLoaded', () => {

const h1Ele = document.createElement('h1')；

document.body.append(h1Ele);

h1Ele.innerText = 'Hello Webpack (^\_^)'

h1Ele.style.color = '#f46';

})

保存该文件之后，浏览器会立刻刷新，你将看到修改之后的变化：

## 多文件打包，glob配置动态入口

安装：yarn add glob --dev

const glob = require("glob");

const htmls = glob.sync('src/components/\*\*/\*.html');

//扫描出入口页面模板的路径, 如src/components/index/index.html, 存放在 htmls 对象里

//入口文件

var SRC\_PATH = path.resolve(\_\_dirname,'../src');

var newEntries = {};

// var files = glob.sync(path.join(SRC\_PATH,'js/\*.js')); // 方式一

var files = glob.sync(SRC\_PATH+'/js/\*.js');  //方式二

files.forEach(function(file,index){

//    var substr =  file.split('/').pop().split('.')[0];

   var substr = file.match(/src\/js\/(\S\*)\.js/)[1];

   newEntries[substr] = file;

})

多文件的时候，需要修改output输出文件里的 动态文件配置为[name]

**// [\s]---表示，只要出现空白就匹配；**

**// [\S]---表示，非空白就匹配；**

### hash、chunkhash、contenthash区别

hash一般是结合CDN缓存来使用，通过webpack构建之后，生成对应文件名自动带上对应的MD5值。如果文件内容改变的话，那么对应文件哈希值也会改变，对应的HTML引用的URL地址也会改变，触发CDN服务器从源服务器上拉取对应数据，进而更新本地缓存。但是在实际使用的时候，这几种hash计算还是有一定区别。

[hash:8] 可以用来控制输出多少位的hash

#### Hash

**Hash：** hash是跟整个项目的构建相关，只要项目里有文件更改，整个项目构建的hash值都会更改，并且全部文件都共用相同的hash值

filename: 'bundle.[name].[hash].js',

#### chunkhash

**chunkhash：** 采用hash计算的话，每一次构建后生成的哈希值都不一样，即使文件内容压根没有改变。这样子是没办法实现缓存效果，我们需要换另一种哈希值计算方式，即chunkhash。

chunkhash和hash不一样，它根据不同的入口文件(Entry)进行依赖文件解析、构建对应的chunk，生成对应的哈希值。我们在生产环境里把一些公共库和程序入口文件区分开，单独打包构建，接着我们采用chunkhash的方式生成哈希值，那么只要我们不改动公共库的代码，就可以保证其哈希值不会受影响。

filename: 'bundle.[name].[chunkhash].js',

#### contenthash

**contenthash：** 在chunkhash的例子，我们可以看到由于index.css被index.js引用了，所以共用相同的chunkhash值。但是这样子有个问题，如果index.js更改了代码，css文件就算内容没有任何改变，由于是该模块发生了改变，导致css文件会重复构建。  
这个时候，我们可以使用extra-text-webpack-plugin里的contenthash值，保证即使css文件所处的模块里就算其他文件内容改变，只要css文件内容不变，那么不会重复构建。

plugins:[

**new** extractTextPlugin('../css/bundle.[name].[contenthash].css')

 ]

## Step03: 打包之前删除某个文件夹

### rimraf插件

安装：yarn add rimraf --dev

在package.json的script直接使用，比如：

"build": "rimraf dist && webpack --config ./build/webpack.config.js --mode production",

或者：

 "build": "rm -rf dist && webpack --config ./build/webpack.config.js --mode production"

### node常用命令

touch 新建文件

rm -f 删除文件

mkdir 新建文件夹

rm -rf 删除文件夹