webpack.config.js

var path = require("path");

var webpack = require("webpack");

var CopyWebpackPlugin = require("copy-webpack-plugin");

module.exports = {

context: \_\_dirname,

entry: {

main: ["./src/main.ts"],

better: ["./src/good.js"]

},

output: {

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

publicPath: "dist",

filename: "[name].js"

//chunkFilename: 'js/jquery.chunk.js'

},

module: {

rules: [

{ test: /\.ts$/, use: ["ts-loader"], exclude: /node\_modules/ },

{ test: /\.js$/, use: ["babel-loader"], exclude: /node\_modules/ },

{ test: /\.css$/, use:["style-loader", "css-loader"], exclude: /node\_modules/ }

],

noParse: /home/

},

resolve: {

modules:["lib/src", "node\_modules"],

alias: {

haohao: path.join(\_\_dirname , '/src/aa/bb')

},

mainFiles: ["you.ts", "index.js"],

extensions: [".d.ts", ".css", ".js", ".ts"]

},

devServer: {

contentBase: path.resolve(\_\_dirname, "dist"),

compress: true,

port: 8018,

host: "localhost",

inline: true

},

optimization: {

removeAvailableModules: true,

removeEmptyChunks: true,

mergeDuplicateChunks: true,

minimize: false,

runtimeChunk: {

name: 'vendorsss'

},

splitChunks: {

cacheGroups: {

default: false,

commons: {

test: /node\_modules|layout/,

name: "vendor111",

chunks: "initial",

minSize: 1

}

}

}

},

plugins: [

new CopyWebpackPlugin([{

from: path.join(\_\_dirname, "src/index.html"),

to: path.join(\_\_dirname, "dist")

}]),

//import \* as $ from 'jquery';

//可以全局使用，导入即可以使用 jquery

new webpack.ProvidePlugin({

$: "jquery",

jQuery: "jquery",

"window.jQuery": "jquery"

})

]

/\*

,

externals: {

jquery: "jQuery"

}

\*/

# }

# 注意：

# 对于 .js 的转换器loader, 建议使用 babel-loader , 因为 babel 可以兼顾 commonjs 的 require 加载 ， 而且可以兼顾 ES6 的import 加载模块指令

# splitChunks 值得关注提取公共模块放到公共文件里

# entry, output, module, resolve, devServer, optimization, plugins, externals

# 搞清楚这些段的定义就可以成为专家了

# WebPack 使用：

# 安装 node.js , 我们将使用 npm install 来安装各种模块

# 安装全局webpack： npm install -g webpack

# npm install -g webpack

# npm install webpack

# 最好也在项目里安装 webpack 因为需要使用到一些配置文件

# 安装 webpack-cli 命令行工具：

# npm install webpack-cli --save-dev

# 创建或者进入到我们的项目目录里，项目的根目录。

# 确保有 package.json 文件：

{

"name": "webpack02",

"version": "1.0.0",

"description": "webpack app",

"main": "hello.js",

"scripts": {

"start": "webpack-dev-server --mode development --open",

"build": "webpack --mode production",

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

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

},

"author": "",

"license": "ISC",

"dependencies": {

"jquery": "^3.3.1",

"typescript": "^2.7.2",

"css-loader": "^0.28.11",

"style-loader": "^0.20.3",

"ts-loader": "^4.1.0",

"webpack": "^4.2.0"

},

"devDependencies": {

}

}

运行： npm run build 等价于： webpack --mode production

# 如果遇到： webpack-dev-server 不能启动， 或者webpack^4.1.1安装打包问题

# 错误信息如下：

The CLI moved into a separate package: webpack-cli.

## 1.在项目目录下要本地安装webpack-cli -D

## npm

npm install webpack-cli -D

# 

# 新建webpack.config.js文件

var path = require('path')

var config = require('../config')

var utils = require('./utils')

var projectRoot = path.resolve(\_\_dirname, '../')

var env = process.env.NODE\_ENV

// check env & config/index.js to decide whether to enable CSS source maps for the

// various preprocessor loaders added to vue-loader at the end of this file

var cssSourceMapDev = (env === 'development' && config.dev.cssSourceMap)

var cssSourceMapProd = (env === 'production' && config.build.productionSourceMap)

var useCssSourceMap = cssSourceMapDev || cssSourceMapProd

// 配置文件的内容需要通过module.exports暴露

module.exports = {

// 配置需要打包的入口文件，值可以是字符串、数组、对象。

// 1. 字符串： entry： './entry'

// 2. 字符串： entry：[ './entry1','entry2'] (多入口)

// 3. 对象： entry： { 'alert/index': path.resolve(pagesDir, `./alert/index/page`)}

// 多入口书写的形式应为object，因为object,的key在webpack里相当于此入口的name,

entry: {

app: './src/main.js'

},

output: {

// 输出文件配置，output 输出有自己的一套规则，常用的参数基本就是这三个

// path: 表示生成文件的根目录 需要一个\*\*绝对路径\*\* path仅仅告诉Webpack结果存储在哪里

path: config.build.assetsRoot,

// publicPath 参数表示的是一个URL 路径（指向生成文件的跟目录），用于生成css/js/图片/字体文件

// 等资源的路径以确保网页能正确地加载到这些资源.

// “publicPath”项则被许多Webpack的插件用于在生产模式下更新内嵌到css、html文件里的url值.

// 例如，在localhost（即本地开发模式）里的css文件中边你可能用“./test.png”这样的url来加载图片，

// 但是在生产模式下“test.png”文件可能会定位到CDN上并且你的Node.js服务器可能是运行在HeroKu上边的。

// 这就意味着在生产环境你必须手动更新所有文件里的url为CDN的路径。

//开发环境：Server和图片都是在localhost（域名）下

//.image {

// background-image: url('./test.png');

//}

// 生产环境：Server部署下HeroKu但是图片在CDN上

//.image {

// background-image: url('https://someCDN/test.png');

//}

![](http://images2015.cnblogs.com/blog/1108527/201703/1108527-20170304195944626-432609161.png)

publicPath: process.env.NODE\_ENV === 'production' ? config.build.assetsPublicPath : config.dev.assetsPublicPath,

// filename 属性表示的是如何命名出来的入口文件，规则是一下三种：

// [name] 指代入口文件的name，也就是上面提到的entry参数的key，因此，我们可以在name里利用/，即可达到控制文件目录结构的效果。

// [hash]，指代本次编译的一个hash版本，值得注意的是，只要是在同一次编译过程中生成的文件，这个[hash].js

//的值就是一样的；在缓存的层面来说，相当于一次全量的替换。

filename: '[name].js'

},

// 用来配置依赖文件的匹配，如依赖文件的别名配置、模块的查找目录、默认查找的

// 文件后缀名

// resolve.root 该选型用来制定模块查找的根路径，必须为\*\*绝对路径\*\*，值可以

// 是路径字符串或者路径数组若是数组，则会依次查找

resolve: {

extensions: ['', '.js', '.vue', '.json'],

fallback: [path.join(\_\_dirname, '../node\_modules')],

// 用来配置依赖文件的别名，值是一个对，该对象的键是别名，值是实际路径

alias: {

'vue$': 'vue/dist/vue.common.js',

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

'assets': path.resolve(\_\_dirname, '../src/assets'),

'components': path.resolve(\_\_dirname, '../src/components')

}

},

resolveLoader: {

fallback: [path.join(\_\_dirname, '../node\_modules')]

},

// 用来进行模块加载相关的配置

module: {

preLoaders: [

{

test: /\.vue$/,

loader: 'eslint',

include: projectRoot,

exclude: /node\_modules/

},

{

test: /\.js$/,

loader: 'eslint',

include: projectRoot,

exclude: /node\_modules/

}

],

loaders: [

// webpack拥有一个类似于插件的机制，名为Loader，通过Loader，webpack能够针对每一种特定的资源做出相应的处理

// 1.test参数用来指示当前配置项针对哪些资源，该值应是一个条件值(condition)。

// 2.exclude参数用来剔除掉需要忽略的资源，该值应是一个条件值(condition)。

// 3.include参数用来表示本loader配置仅针对哪些目录/文件，该值应是一个条件值(condition)。

// 而include参数则用来指示目录；注意同时使用这两者的时候，实际上是and的关系。

// 4.loader/loaders参数，用来指示用哪个或哪些loader来处理目标资源，这俩货

// 表达的其实是一个意思，只是写法不一样，我个人推荐用loader写成一行，多个

// loader间使用!分割，这种形式类似于管道的概念，又或者说是函数式编程。形

// 如loader: 'css?!postcss!less'，可以很明显地看出，目标资源先经less-loader

// 处理过后将结果交给postcss-loader作进一步处理，然后最后再交给css-loader。

{

test: /\.vue$/,

loader: 'vue'

},

{

test: /\.js$/,

loader: 'babel',

include: projectRoot,

exclude: /node\_modules/

},

{

test: /\.json$/,

loader: 'json'

},

{

test: /\.(png|jpe?g|gif|svg)(\?.\*)?$/,

loader: 'url',

query: {

limit: 10000,

name: utils.assetsPath('img/[name].[hash:7].[ext]')

}

},

{

test: /\.(woff2?|eot|ttf|otf)(\?.\*)?$/,

loader: 'url',

query: {

limit: 10000,

name: utils.assetsPath('fonts/[name].[hash:7].[ext]')

}

},

// expose-loader，这个loader的作用是，将指定js模块export的变量声明为全局变量

{

test: require.resolve('jquery'), // 此loader配置项的目标是NPM中的jquery

loader: 'expose?$!expose?jQuery', // 先把jQuery对象声明成为全局变量`jQuery`，再通过管道进一步又声明成为全局变量`$`

},

]

},

eslint: {

formatter: require('eslint-friendly-formatter')

},

vue: {

loaders: utils.cssLoaders({ sourceMap: useCssSourceMap }),

// 解决.vue中文件style的部分一些特性解析，比如scoped

postcss: [

require('autoprefixer')({

browsers: ['last 2 versions']

})

]

}

}

# 例如：

entry: {

"ajax/index": [ 可以带上目录， 输出时也会自动创建目录；结果如下图

path.resolve(\_\_dirname, "code/index.js"),

path.resolve(\_\_dirname, "code/sale.js")

],

"order/myorder": [

path.resolve(\_\_dirname, "code/index.js"),

path.resolve(\_\_dirname, "code/sale.js"),

path.resolve(\_\_dirname, "code/order.ts")

]

},

output: {

path: path.resolve(\_\_dirname, "public"),

publicPath: "assets",

filename: "[name].bundle.js"

},

# 

# 

# webpack typescript – jquery

# 安装 typescript / jquery

# npm install –save-dev @types/jquery

# xxx.ts 文件里引用 jquery

# import { ORDER } from "./ORDER";

# import \* as $ from 'jquery'; -- 正确， 把jquery 的所有输出归于 $

# import $ from 'jquery'; -- 引用出错， 提示： module jquery has no default export 没有默认输出

# 如果单独引用 jquery 文件 webpack.config.js 配置文件，可以设定为外部模块

externals: {

jquery: "jQuery"

},

resolve: {

extensions: [".js", ".ts"]

},

# webpack.config.js loader

var path = require("path");

var webpack = require("webpack");

var HtmlWebPackPlugin = require("html-webpack-plugin");

var CopyWebPackPlugin = require("copy-webpack-plugin");

module.exports = {

devtool: 'inline-source-map',

entry: {

"ajax/index": [

path.resolve(\_\_dirname, "code/index.js"),

path.resolve(\_\_dirname, "code/sale.js")

],

"order/myorder": [

path.resolve(\_\_dirname, "code/index.js"),

path.resolve(\_\_dirname, "code/sale.js"),

path.resolve(\_\_dirname, "code/order.ts")

]

},

output: {

path: path.resolve(\_\_dirname, "public"),

publicPath: "assets",

filename: "[name].bundle.js"

},

module: {

rules: [

{ test: /\.css$/, use: ["style-loader", "css-loader"] },

{ test: /\.ts$/, use: ["ts-loader"], exclude: /node\_modules/ }

]

},

resolve: {

extensions: [".js", ".css", ".ts"]

},

devServer: {

contentBase: \_\_dirname,

compress: true,

port: 9099,

host: "localhost",

//hot: true,

inline: true

},

plugins: [

new HtmlWebPackPlugin({

template: "index.html",

filename: "index.html"

}),

new CopyWebPackPlugin([{

from: path.resolve(\_\_dirname),

to: path.resolve(\_\_dirname, "public"),

toType: File

}])

]

# }

# 配置webpack-dev-server：

module.exports = {

devtool: 'eval-source-map',

entry: {

index: [

path.resolve(\_\_dirname, "code/index.js"),

path.resolve(\_\_dirname, "code/sale.js")

]

},

output: {

path: path.resolve(\_\_dirname, "public"), 实际的JS文件输出的目录

publicPath: "assets", webpack-dev-server 虚拟的文件目录（不需要真实存在）

filename: "index.bundle.js"

},

module: {

rules: [

{ test: /\.css$/, use: ["style-loader", "css-loader"] },

{

test: /\.ts$/,

use: ["ts-loader"],

exclude: /node\_modules/

}

]

},

resolve: {

extensions: [".js", ".css", ".ts"]

},

devServer: {

contentBase: \_\_dirname, contentBase 是

compress: true,

port: 9099,

host: "localhost",

//hot: true, 如果为true 则自动更新会被替换掉

inline: true 必须设置为 true 浏览器才会自动更新

}

# }

<html>

<head>

<meta charset="utf-8" />

<title>WebPack Test</title>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

<script src="assets/index.bundle.js"></script>

</head>

<body>

<h1>this is webpack test</h1>

</body>

# </html>

# 浏览器自动更新的条件：

1. **inline: true,**//注意：不写hot: true

# publicPath: "assets",

# <script src="assets/index.bundle.js"></script> 必须路径要一致， 否则浏览器无法自动更新

# <https://segmentfault.com/a/1190000006964335>

### 在Nodejs API中运行inline模式，并启用热模块替换（没有试过）

这里需要做以下三点:

* 在webpack.config.js的entry选项中添加:webpack/hot/dev-server
* 在webpack.config.js的plugins选项中添加:new webpack.HotModuleReplacementPlugin()
* 在webpack-dev-server的配置中添加：hot:true

var WebpackDevServer = require("webpack-dev-server");

var webpack = require("webpack");

var compiler = webpack({

// configuration

});

var server = new WebpackDevServer(compiler, {

// webpack-dev-server options

contentBase: "/path/to/directory",

// Can also be an array, or: contentBase: "http://localhost/",

hot: true,

// Enable special support for Hot Module Replacement

// Page is no longer updated, but a "webpackHotUpdate" message is send to the content

// Use "webpack/hot/dev-server" as additional module in your entry point

// Note: this does \_not\_ add the `HotModuleReplacementPlugin` like the CLI option does.

// Set this as true if you want to access dev server from arbitrary url.

// This is handy if you are using a html5 router.

historyApiFallback: false,

// Set this if you want to enable gzip compression for assets

compress: true,

// Set this if you want webpack-dev-server to delegate a single path to an arbitrary server.

// Use "\*\*" to proxy all paths to the specified server.

// This is useful if you want to get rid of 'http://localhost:8080/' in script[src],

// and has many other use cases (see https://github.com/webpack/webpack-dev-server/pull/127 ).

proxy: {

"\*\*": "http://localhost:9090"

},

setup: function(app) {

// Here you can access the Express app object and add your own custom middleware to it.

// For example, to define custom handlers for some paths:

// app.get('/some/path', function(req, res) {

// res.json({ custom: 'response' });

// });

},

// pass [static options](http://expressjs.com/en/4x/api.html#express.static) to inner express server

staticOptions: {

},

// webpack-dev-middleware options

quiet: false,

noInfo: false,

lazy: true,

filename: "bundle.js",

watchOptions: {

aggregateTimeout: 300,

poll: 1000

},

// It's a required option.

publicPath: "/assets/",

headers: { "X-Custom-Header": "yes" },

stats: { colors: true }

});

server.listen(8080, "localhost", function() {});

// server.close();

 a. webpack-dev-server简介：

* 是一个小型node.js express服务器
* 新建一个开发服务器，可以serve我们pack以后的代码，并且当代码更新的时候自动刷新浏览器
* 启动webpack-dev-server后，你在目标文件夹中是看不到编译后的文件的，实时编译后的文件都保存到了内存当中。
* 两种自动刷新方式：
  + iframe mode  
    在网页中嵌入了一个 iframe ，将我们自己的应用注入到这个 iframe 当中去，因此每次你修改的文件后，都是这个 iframe 进行了 reload  
    命令行：webpack-dev-server，无需--inline  
    浏览器访问：<http://localhost:8080/webpack-dev-server/index.html>
  + inline mode  
    命令行：webpack-dev-server --inline  
    浏览器访问：<http://localhost:8080>

 b. 安装webpack-dev-server  
npm install webpack-dev-server --save-dev  
可在项目根目录下安装（不加-g，在项目根目录出现node\_modules文件夹，内含webpack-dev-server及其依赖包），也可全局安装（加-g 必须sudo）

 c. 在webpack.config.js中添加配置

var webpack=require('webpack');

module.exports = {

……

devServer: {

historyApiFallback: true,

inline: true,//注意：不写hot: true，否则浏览器无法自动更新；也不要写colors:true，progress:true等，webpack2.x已不支持这些

},

plugins:[

……

new webpack.HotModuleReplacementPlugin()

]

……

};

 d. 在package.json里配置运行的命令（npm支持自定义一些命令）

"scripts":

{

　　"start": "webpack-dev-server --inline"//注意：不写--hot，否则浏览器无法自动更新

},

# webpack-dev-server --mode development --open

# i ｢wds｣: Project is running at http://localhost:9099/

# i ｢wds｣: webpack output is served from /public

# i ｢wds｣: Content not from webpack is served from D:\Projects\VS2017\WebPack02

# i ｢wdm｣: wait until bundle finished: /public/sale.bundle.js

# i ｢wdm｣: wait until bundle finished: /public/index.bundle.js

# i ｢wdm｣: Hash: 0bc5970e55550ed9ab1d

# Version: webpack 4.2.0

# webpack 插件

# 先安装，再引用，再到plugins 实例化

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

npm install --save-dev copy-webpack-plugin

var HtmlWebPackPlugin = require("html-webpack-plugin");

var CopyWebPackPlugin = require("copy-webpack-plugin");

plugins: [

new HtmlWebPackPlugin({

template: "index.html",

filename: "index.html"

}),

new CopyWebPackPlugin([{

from: path.resolve(\_\_dirname),

to: path.resolve(\_\_dirname, "public")

}])

]

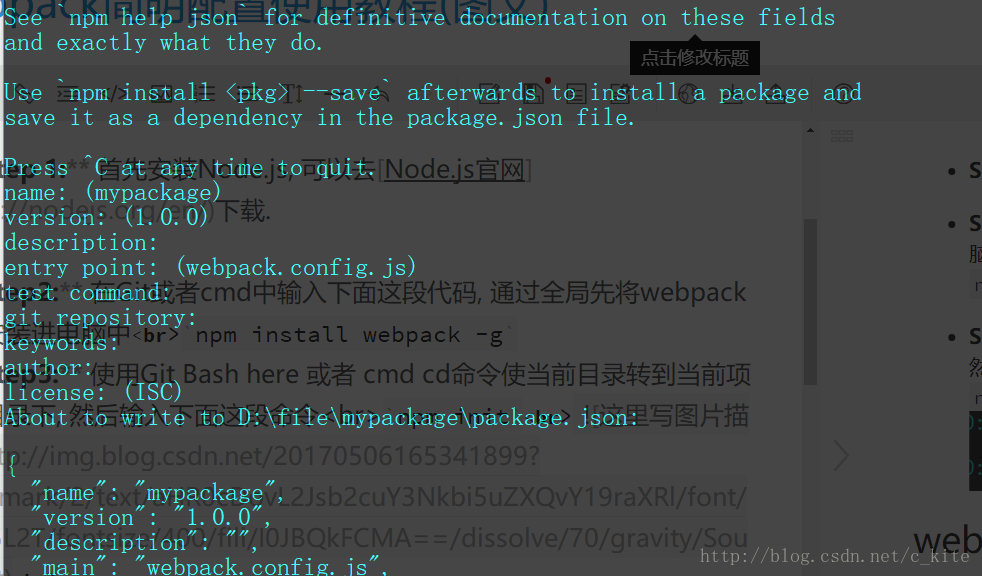
# webpack 命令

# <https://blog.csdn.net/c_kite/article/details/71279853>

# webpack超详细配置, 使用教程(图文)

## 流程

### webpack安装

* **Step 1:** 首先安装Node.js, 可以去[Node.js官网](https://nodejs.org/en/)下载.
* **Step2:** 在Git或者cmd中输入下面这段代码, 通过全局先将webpack指令安装进电脑中  
  npm install webpack -g
* **Step3:** 使用Git Bash here 或者 cmd cd命令使当前目录转到当前项目的目录下, 然后输入下面这段命令  
  npm init  
  
* 接下来会弹出一些信息, 就是一些项目名和一些参数的描述, 可以全部按回车使用默认信息, 完成之后项目文件夹下会生成一个**package.json**的文件  
    
  这样webpack就安装完成了.

# [webpack学习笔记——--save-dev和--save](http://www.cnblogs.com/yueliangcl/p/6625228.html)

**--save-dev 是你开发时候依赖的东西，--save 是你发布之后还依赖的东西。**

**dependencies是运行时(发布后)依赖，devDependencies是开发时的依赖**

比如，你写 ES6 代码，如果你想编译成 ES5 发布那么 babel 就是devDependencies。  
如果你用了 jQuery，由于发布之后还是依赖jQuery，所以是dependencies。

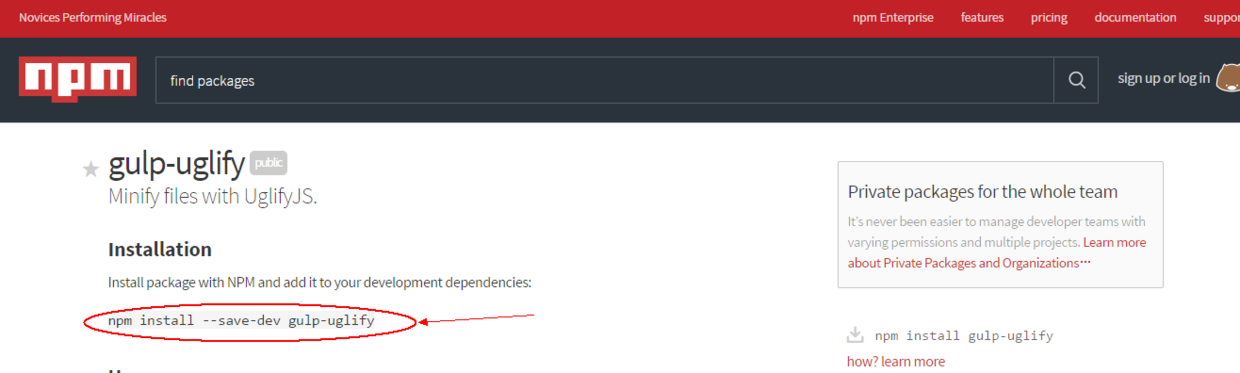
如果上线之后项目还依赖的模块，直接放 dependencies 里就行了，devDependencies 里面就不用放了，开发的时候会自动去调用 dependencies 中的模块的，不需要重复了。  
比如，我做项目用到了 vue vuex n-zepto ，这三个模块都是上线之后还需要依赖的，那么我直接放到 dependencies 里，devDependencies 里不需要放。开发的时候功能不会受到任何影响。

--save-dev 是你开发时候依赖的东西，--save 是你发布之后还依赖的东西。

比如，你写 ES6 代码，如果你想编译成 ES5 发布那么 babel 就是devDependencies。  
如果你用了 jQuery，由于发布之后还是依赖jQuery，所以是dependencies。

但是在 npm 里面除了二进制的依赖，似乎也不用区分是不是dev。  
因为使用npm就是自己编译的意思，而不使用npm直接拿编译后的版本的，这些依赖项也看不到。

npm install 在安装 npm 包时，有两种命令参数可以把它们的信息写入 package.json 文件，一个是npm install --save另一个是 npm install --save-dev，他们表面上的区别是--save 会把依赖包名称添加到 package.json 文件 dependencies 键下，--save-dev 则添加到 package.json 文件 devDependencies 键下，譬如：  
{  
"name": "yo",  
"version": "0.0.0",  
"dependencies": {},  
"devDependencies": {  
"grunt": "~0.4.1",  
"grunt-contrib-copy": "~0.4.1",  
"grunt-contrib-concat": "~0.3.0",  
"grunt-contrib-uglify": "~0.2.0",  
"grunt-contrib-compass": "~0.7.0",  
"grunt-contrib-jshint": "~0.7.0",  
"grunt-contrib-cssmin": "~0.7.0",  
}  
}  
不过这只是它们的表面区别。它们真正的区别是，devDependencies 下列出的模块，是我们开发时用的，比如 我们安装 js的压缩包gulp-uglify 时，

  
我们采用的是 “npm install --save-dev gulp-uglify ” （见上图）命令安装，因为我们在发布后用不到它，而只是在我们开发才用到它。dependencies 下的模块，则是我们发布后还需要依赖的模块，譬如像jQuery库或者Angular框架类似的，我们在开发完后后肯定还要依赖它们，否则就运行不了。  
另外需要补充的是：  
正常使用npm install时，会下载dependencies和devDependencies中的模块，当使用npm install --production或者注明NODE\_ENV变量值为production时，只会下载dependencies中的模块。

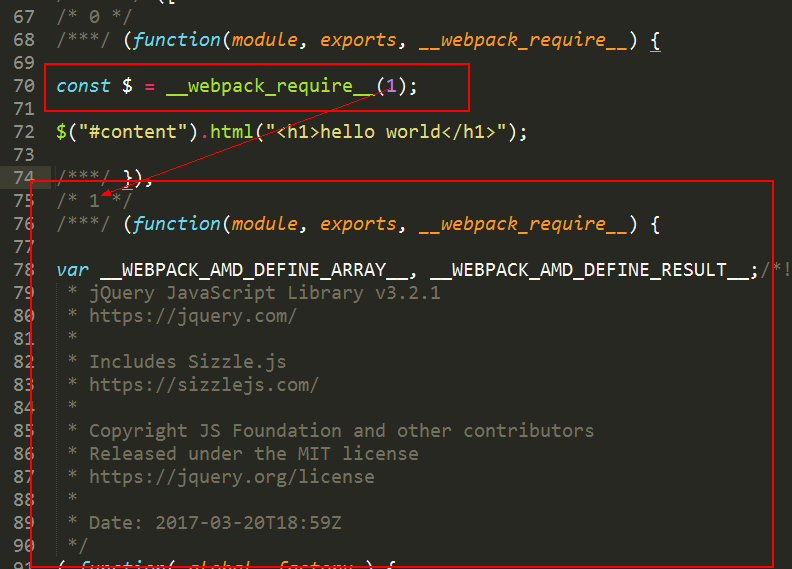
[**http://www.jb51.net/article/129670.htm**](http://www.jb51.net/article/129670.htm)

**深入浅出webpack之externals的使用**

*转载*  2017-12-04   作者：前端小豪   [[http://www.jb51.net/skin/2018/images/text-message.png](http://www.jb51.net/article/129670.htm#comments)我要评论](http://www.jb51.net/article/129670.htm#comments)

这篇文章主要介绍了深入浅出webpack之externals的使用，小编觉得挺不错的，现在分享给大家，也给大家做个参考。一起跟随小编过来看看吧

我们通常在做项目时可能会把第三方库打包到bundle中，比如下面这张图

[](http://files.jb51.net/file_images/article/201712/2017120409262418.png)

如果不想把第三方库打包到bundle中，这就有了externals。官方的使用externals比较简单

**externals**

官网文档解释的很清楚，就是webpack可以不处理应用的某些依赖库，使用externals配置后，依旧可以在代码中通过CMD、AMD或者window/global全局的方式访问。

只需三步——

1.在HTML中引入第三方库的cdn

2.在webpack中配置externals

[?](http://www.jb51.net/article/129670.htm)

|  |  |
| --- | --- |
| 1  2  3 | externals: {   jquery: "jQuery",  } |

3.在js中引用

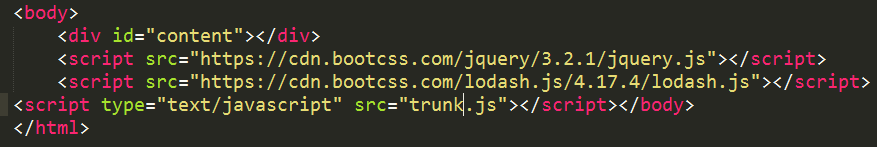
[?](http://www.jb51.net/article/129670.htm)

|  |  |
| --- | --- |
| 1  2 | const $ = require("jquery");  $("#content").html("<h1>hello world</h1>"); |

好，现在我们可以随心所欲的使用jquery插件并保证不会打包到bundle中。external是怎么办到的呢？下面我们通过bundle的源码来分析下原理。

[](http://files.jb51.net/file_images/article/201712/2017120409262419.png)

这里的/\* 0 \*/和\_\_webpack\_require\_\_分别指打包前js对应的模块函数，这里就不细说了。这里可以看到module.exports = jQuery，就是说我们externals中的key指的是require的东西，value指的就是它，就是说“当require的参数是jquery的时候，使用jQuery这个全局变量引用它”。这种最简洁的externals配置方式为默认的global模式，就是在window上挂一个全局变量，然后直接可以使用这个变量。具体的流程是这样，我们在源码中使用require('jquery')后，可以直接把jquery加到externals中，得到一个打包的trunk.js，但是在引入这个trunkjs之前，肯定要先引入jquery这个库文件，这个库文件会创建一个全局变量jQuery，而咱们的trunkjs中externals的jquery是global模式，所以实际上trunkjs引入jquery的时候，就会从全局变量中引用，即module.export = jQuery

[](http://files.jb51.net/file_images/article/201712/2017120409262420.png)

当然，既然是通过这种externals方式，其实我们可以不用require引入，直接使用全局变量也是可以的。

[?](http://www.jb51.net/article/129670.htm)

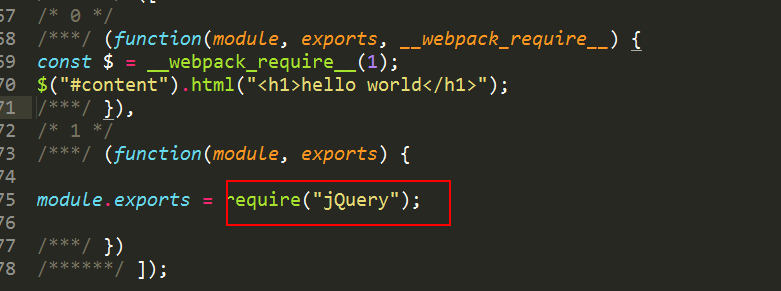
|  |  |
| --- | --- |
| 1 | jQuery("#content").html("<h1>hello world</h1>"); |

大家如果注意到我刚说过的global模式的话，没错，你也许已经猜到了，我可以任意的使用不同的输出方式。如果打包文件我想运行到node环境下，我得使用commonjs规范，所以你要这么写。

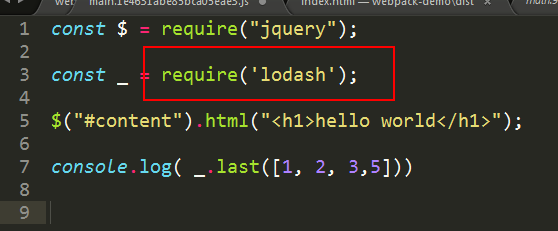
[?](http://www.jb51.net/article/129670.htm)

|  |  |
| --- | --- |
| 1  2  3 | externals: {   jquery: "commonjs2 jQuery",  } |

打包后会是这样子。

[](http://files.jb51.net/file_images/article/201712/2017120409262421.png)

然后我的项目中还用到了lodash，也想把它从bundle中移除，之前我的代码是这样子，引的是npm包

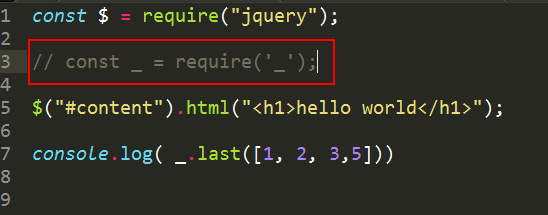


[](http://files.jb51.net/file_images/article/201712/2017120409262423.png)

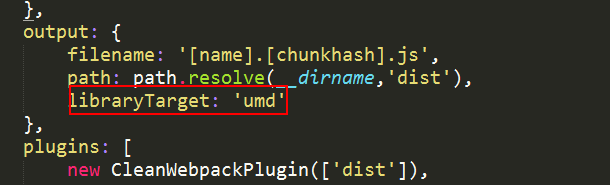
现在我们的externals配置如下

[?](http://www.jb51.net/article/129670.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5 | externals: {    jquery: "jQuery",    \_: "lodash"    } |



我们必须要去掉这个const ，否则的话会报一个错误 lodash is not defined。为什么会这样呢？因为我们的lodash输出是global格式的，我在这里先卖一个关子，我们先统一一下输出格式，加一个libraryTarget字段

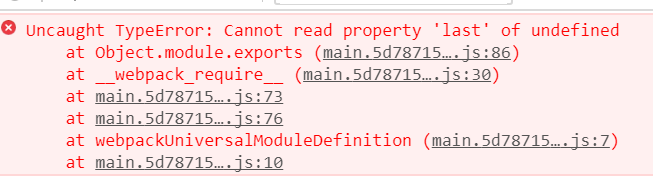


这个东西是干嘛用的呢？

他是我们输出文件的模块化规范，想想我们上面配置的commonjs jquery是运行在node下，总之记住一句话——我们最长使用的模块化方案是commonjs2和umd，前者是为node环境，后者是为浏览器环境。一共有这几种规范：

"var" - Export by setting a variable: var Library = xxx (default)

"this" - Export by setting a property of this: this["Library"] = xxx  
"commonjs" - Export by setting a property of exports: exports["Library"] = xxx  
"commonjs2" - Export by setting module.exports: module.exports = xxx  
"amd" - Export to AMD (optionally named - set the name via the library option)  
"umd" - Export to AMD, CommonJS2 or as property in root

*[](http://files.jb51.net/file_images/article/201712/2017120409262526.png)*

然后报这个错误，也就是说我们的模块没有正确的输出，回到我们的externals，它更多的是指定当你引用一个包的时候，这个包(lodash)应该遵循哪种模块化方式(common,root,amd等等)引入，这意思就是说，打包的时候不需要关心他到底怎么输出。

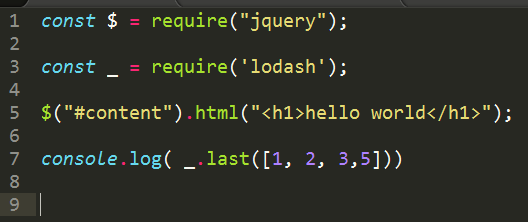
[?](http://www.jb51.net/article/129670.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | externals: {    jquery: "jQuery",    lodash: {     commonjs: 'lodash',     commonjs2: 'lodash',     amd: 'lodash',     root: '\_'    }   }, |

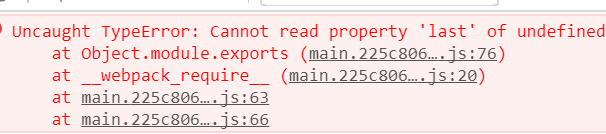
ok，记得要将之前的覆盖掉，替换成下面的require，因为在externals中我们规范的commmonjs规范为lodash

http://files.jb51.net/file_images/article/201712/2017120409262527.png

也就是说，这就是我们最初的代码，即没有用过externals时候的代码，看，也就是说我们只需要配置externals和libraryTarget就可以，其他的业务逻辑代码不需要改变。包括我们的项目中还用了echarts，这个通通不用改变！！！！！



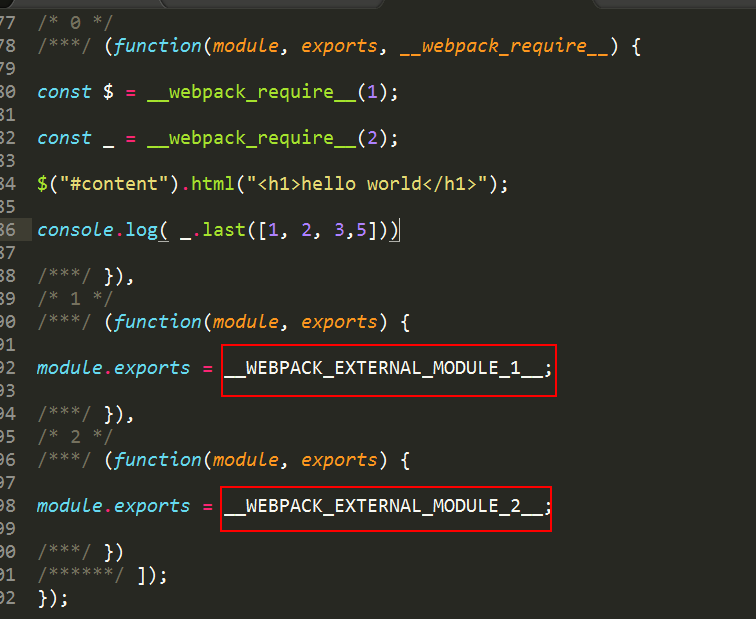
也就是说最终的代码是externals配合libraryTarget一起使用，如果去掉umd的话，会报这个错误



相应的源码是这样子

[](http://files.jb51.net/file_images/article/201712/2017120409262530.png)

就是说我不知道通过那种方式输出，所以我应该告诉webpack，我通过umd方式输出,即将你的 lodash 暴露为所有的模块定义下都可运行的方式。它将在 CommonJS, AMD 环境下运行，或将模块导出到 global 下的变量.加上umd的源码如下

[](http://files.jb51.net/file_images/article/201712/2017120409262531.png)

看到了吧，我通过require('lodash')引入模块，输出走的是commonjs*规范,贴下最终的配置*

[?](http://www.jb51.net/article/129670.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | entry: {    main: './src/index.js'   },   externals: {    jquery: "jQuery",    lodash: {     commonjs: 'lodash',     commonjs2: 'lodash',     amd: 'lodash',     root: '\_'    }   },   output: {    filename: '[name].[chunkhash].js',    path: path.resolve(\_\_dirname,'dist'),    libraryTarget: 'umd'   }, |

 以上就是本文的全部内容，希望对大家的学习有所帮助，也希望大家多多支持脚本之家。

### <https://segmentfault.com/a/1190000012113011>

### 不同环境设置externals方式

1. 如果你的代码想运行在Node环境中s，那么你需要在external中添加前缀commonjs2或者commonjs

externals:{

react:'commonjs2 react',

jquery:'commonjs2 jquery'

}

1. 如果需要requirejs等符合AMD规范的环境中加载，那就要添加amd

externals:{

react:'amd React',

jquery:'amd jQuery'

}

1. 如果要在浏览器中运行，那么不用添加什么前缀，默认设置就是global。

externals:{

react:'React',

jquery:'jQuery'

}

也可以这样

externals:["React","jQuery"]

这种方式配置下，就是配置你所引用你的库暴露出的全局变量。上面两种模式下或者说，如果你想运行代码在浏览器中，你所引用的包，必须暴露出一个全局变量。如果没有，这种方式不适合在浏览器下使用，可以尝试dll的方式。

这里你可以看出，不同模式下，value是不一样的。2，3模式下，是要引入去全局变量，1模式是要加载包名。那如果这个包的包名和在浏览器下引入的全局变量一致，上面就可以写成一样了，比如moment。

### externals和libraryTarget的关系

* libraryTarget配置如何暴露 library。如果不设置library,那这个library就不暴露。就相当于一个自执行函数
* externals是决定的是以哪种模式去加载所引入的额外的包
* libraryTarget决定了你的library运行在哪个环境，哪个环境也就决定了你哪种模式去加载所引入的额外的包。也就是说，externals应该和libraryTarget保持一致。library运行在浏览器中的，你设置externals的模式为commonjs，那代码肯定就运行不了了。
* 如果是应用程序开发，一般是运行在浏览器环境libraryTarget可以不设置，externals默认的模式是global，也就是以全局变量的模式加载所引入外部的库。

## Webpack详细入门教程之Source Maps调试

**生成Source Maps（使调试更容易）**

开发总是离不开调试，方便的调试能极大的提高开发效率，不过有时候通过打包后的文件，你是不容易找到出错了的地方，对应的你写的代码的位置的，Source Maps就是来帮我们解决这个问题的。

通过简单的配置，webpack就可以在打包时为我们生成的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：



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，较快的打包速度的后果就是对打包后的文件的的执行有一定影响。

对小到中型的项目中，eval-source-map是一个很好的选项，再次强调你只应该开发阶段使用它，我们继续对上文新建的webpack.config.js，进行如下配置:

module.exports = {

devtool: 'eval-source-map',

entry : \_\_dirname + '/demo/js/main.js',//入口文件

output : {//输出文件

filename : 'index.js',//输出文件名

path : \_\_dirname + '/out'//输出文件路径

},

}

**注：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时间成本时使用。**

# [webpack学习笔记——publicPath路径问题](http://www.cnblogs.com/yueliangcl/p/6679427.html)

output: {

filename: "[name].js",

path:path.resolve(\_\_dirname,"build")

}

如果没有指定pubicPath,则引入路径如下

<body>

<script src="b.js"></script>

</body>

如果有指定publicPath

output: {

filename: "[name].js",

path:path.resolve(\_\_dirname,"build"),

publicPath:"/assets/"

}

则引入如下

<body>

<script src="assets/b.js"></script>

</body>

webpack-dev-server环境下，path、publicPath、区别与联系

path：指定编译目录而已（/build/js/），不能用于html中的js引用。

publicPath：虚拟目录，自动指向path编译目录（/assets/ => /build/js/）。**html中引用js文件时，必须引用此虚拟路径**（但实际上引用的是内存中的文件，既不是/build/js/也不是/assets/）。

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

发布至生产环境：

1.webpack进行编译（当然是编译到/build/js/）

2.把编译目录（/build/js/）下的文件，全部复制到/assets/目录下（注意：不是去修改index.html中引用bundle.js的路径）  
  
  
之前因为publicPath的路径错误原因，导致webpack-dev-server不能自动刷新

# <https://blog.csdn.net/itkingone/article/details/70331783>

# [webpack开发和生产两个环境的配置详解](https://blog.csdn.net/itKingOne/article/details/70331783)

标签： [webpack](http://so.csdn.net/so/search/s.do?q=webpack&t=blog) [配置](http://so.csdn.net/so/search/s.do?q=配置&t=blog)

2017年04月21日 16:35:20 19900人阅读 [评论](javascript:void(0);)(3) [收藏](javascript:void(0);) [举报](javascript:void(0);)

https://csdnimg.cn/release/phoenix/images/category_icon.jpg分类：

vueJs（14） https://csdnimg.cn/release/phoenix/images/arrow_triangle%20_down.jpgjavascript（9） https://csdnimg.cn/release/phoenix/images/arrow_triangle%20_down.jpgwebpack（16） https://csdnimg.cn/release/phoenix/images/arrow_triangle%20_down.jpg

目录[(?)[+]](https://blog.csdn.net/itkingone/article/details/70331783)

## 目录

* [webpack.base.conf.js](http://www.cnblogs.com/iSAM2016/p/6492951.html#base)
* [webpack.dev.conf.js](http://www.cnblogs.com/iSAM2016/p/6492951.html#dev)
* [webpack.prod.conf.js](http://www.cnblogs.com/iSAM2016/p/6492951.html#prod)

一开始在接触webpack 的时候，简直痛不欲生，现在回头看，做个注释，  
当然参考了很多文章。这是一个关于vue 开发的webpack 架构会列举出来  
[webpack 系列教程](https://segmentfault.com/a/1190000006863968)  
[Webpack——令人困惑的地方](http://blog.csdn.net/a1104258464/article/details/51914450)  
[Express结合Webpack的全栈自动刷新](https://segmentfault.com/a/1190000004505747)  
[Webpack傻瓜式指南（一）](https://zhuanlan.zhihu.com/p/20367175?columnSlug=FrontendMagazine)  
[Webpack资源总结](https://segmentfault.com/a/1190000005995267)

启动指令

"scripts": {

"dev": "node build/dev-server.js", //

"build": "node build/build.js",// 打包

"lint": "eslint --ext .js,.vue src"

},

## webpack.base.conf.js

webpack基本配置

var path = require('path')

var config = require('../config')

var utils = require('./utils')

var projectRoot = path.resolve(\_\_dirname, '../')

var env = process.env.NODE\_ENV

// check env & config/index.js to decide whether to enable CSS source maps for the

// various preprocessor loaders added to vue-loader at the end of this file

var cssSourceMapDev = (env === 'development' && config.dev.cssSourceMap)

var cssSourceMapProd = (env === 'production' && config.build.productionSourceMap)

var useCssSourceMap = cssSourceMapDev || cssSourceMapProd

// 配置文件的内容需要通过module.exports暴露

module.exports = {

// 配置需要打包的入口文件，值可以是字符串、数组、对象。

// 1. 字符串： entry： './entry'

// 2. 字符串： entry：[ './entry1','entry2'] (多入口)

// 3. 对象： entry： {alert/index': path.resolve(pagesDir, `./alert/index/page`)}

// 多入口书写的形式应为object，因为object,的key在webpack里相当于此入口的name,

entry: {

app: './src/main.js'

},

output: {

// 输出文件配置，output 输出有自己的一套规则，常用的参数基本就是这三个

// path: 表示生成文件的根目录 需要一个\*\*绝对路径\*\* path仅仅告诉Webpack结果存储在哪里

path: config.build.assetsRoot,

// publicPath 参数表示的是一个URL 路径（指向生成文件的跟目录），用于生成css/js/图片/字体文件

// 等资源的路径以确保网页能正确地加载到这些资源.

// “publicPath”项则被许多Webpack的插件用于在生产模式下更新内嵌到css、html文件里的url值.

// 例如，在localhost（即本地开发模式）里的css文件中边你可能用“./test.png”这样的url来加载图片，

// 但是在生产模式下“test.png”文件可能会定位到CDN上并且你的Node.js服务器可能是运行在HeroKu上边的。

// 这就意味着在生产环境你必须手动更新所有文件里的url为CDN的路径。

//开发环境：Server和图片都是在localhost（域名）下

//.image {

// background-image: url('./test.png');

//}

// 生产环境：Server部署下HeroKu但是图片在CDN上

//.image {

// background-image: url('https://someCDN/test.png');

//}

![](http://images2015.cnblogs.com/blog/1108527/201703/1108527-20170304195944626-432609161.png)

publicPath: process.env.NODE\_ENV === 'production' ? config.build.assetsPublicPath : config.dev.assetsPublicPath,

// filename 属性表示的是如何命名出来的入口文件，规则是一下三种：

// [name] 指代入口文件的name，也就是上面提到的entry参数的key，因此，我们可以在name里利用/，即可达到控制文件目录结构的效果。

// [hash]，指代本次编译的一个hash版本，值得注意的是，只要是在同一次编译过程中生成的文件，这个[hash].js

//的值就是一样的；在缓存的层面来说，相当于一次全量的替换。

filename: '[name].js'

},

// 用来配置依赖文件的匹配，如依赖文件的别名配置、模块的查找目录、默认查找的

// 文件后缀名

// resolve.root 该选型用来制定模块查找的根路径，必须为\*\*绝对路径\*\*，值可以

// 是路径字符串或者路径数组若是数组，则会依次查找

resolve: {

extensions: ['', '.js', '.vue', '.json'],

fallback: [path.join(\_\_dirname, '../node\_modules')],

// 用来配置依赖文件的别名，值是一个对，该对象的键是别名，值是实际路径

alias: {

'vue$': 'vue/dist/vue.common.js',

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

'assets': path.resolve(\_\_dirname, '../src/assets'),

'components': path.resolve(\_\_dirname, '../src/components')

}

},

resolveLoader: {

fallback: [path.join(\_\_dirname, '../node\_modules')]

},

// 用来进行模块加载相关的配置

module: {

preLoaders: [

{

test: /\.vue$/,

loader: 'eslint',

include: projectRoot,

exclude: /node\_modules/

},

{

test: /\.js$/,

loader: 'eslint',

include: projectRoot,

exclude: /node\_modules/

}

],

loaders: [

// webpack拥有一个类似于插件的机制，名为Loader，通过Loader，webpack能够针对每一种特定的资源做出相应的处理

// 1.test参数用来指示当前配置项针对哪些资源，该值应是一个条件值(condition)。

// 2.exclude参数用来剔除掉需要忽略的资源，该值应是一个条件值(condition)。

// 3.include参数用来表示本loader配置仅针对哪些目录/文件，该值应是一个条件值(condition)。

// 而include参数则用来指示目录；注意同时使用这两者的时候，实际上是and的关系。

// 4.loader/loaders参数，用来指示用哪个或哪些loader来处理目标资源，这俩货

// 表达的其实是一个意思，只是写法不一样，我个人推荐用loader写成一行，多个

// loader间使用!分割，这种形式类似于管道的概念，又或者说是函数式编程。形

// 如loader: 'css?!postcss!less'，可以很明显地看出，目标资源先经less-loader

// 处理过后将结果交给postcss-loader作进一步处理，然后最后再交给css-loader。

{

test: /\.vue$/,

loader: 'vue'

},

{

test: /\.js$/,

loader: 'babel',

include: projectRoot,

exclude: /node\_modules/

},

{

test: /\.json$/,

loader: 'json'

},

{

test: /\.(png|jpe?g|gif|svg)(\?.\*)?$/,

loader: 'url',

query: {

limit: 10000,

name: utils.assetsPath('img/[name].[hash:7].[ext]')

}

},

{

test: /\.(woff2?|eot|ttf|otf)(\?.\*)?$/,

loader: 'url',

query: {

limit: 10000,

name: utils.assetsPath('fonts/[name].[hash:7].[ext]')

}

},

// expose-loader，这个loader的作用是，将指定js模块export的变量声明为全局变量

{

test: require.resolve('jquery'), // 此loader配置项的目标是NPM中的jquery

loader: 'expose?$!expose?jQuery', // 先把jQuery对象声明成为全局变量`jQuery`，再通过管道进一步又声明成为全局变量`$`

},

]

},

eslint: {

formatter: require('eslint-friendly-formatter')

},

vue: {

loaders: utils.cssLoaders({ sourceMap: useCssSourceMap }),

// 解决.vue中文件style的部分一些特性解析，比如scoped

postcss: [

require('autoprefixer')({

browsers: ['last 2 versions']

})

]

}

}

## webpack.dev.conf.js

var config = require('../config')

var webpack = require('webpack')

var merge = require('webpack-merge')

var utils = require('./utils')

var baseWebpackConfig = require('./webpack.base.conf')

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

// add hot-reload related code to entry chunks

Object.keys(baseWebpackConfig.entry).forEach(function (name) {

baseWebpackConfig.entry[name] = ['./build/dev-client'].concat(baseWebpackConfig.entry[name])

})

module.exports = merge(baseWebpackConfig, {

module: {

loaders: utils.styleLoaders({ sourceMap: config.dev.cssSourceMap })

},

// eval-source-map is faster for development

devtool: '#eval-source-map',

plugins: [

// DefinePlugin 是webpack 的内置插件，该插件可以在打包时候替换制定的变量

//

new webpack.DefinePlugin({

'process.env': config.dev.env

}),

// https://github.com/glenjamin/webpack-hot-middleware#installation--usage

new webpack.optimize.OccurrenceOrderPlugin(),

new webpack.HotModuleReplacementPlugin(),

new webpack.NoErrorsPlugin(),

// https://github.com/ampedandwired/html-webpack-plugin

new HtmlWebpackPlugin({

filename: 'index.html',

template: 'index.html',

inject: true

}),

// 可以自动加载当前模块依赖的其他模块并已制定别名注入到当前的模块中，引入jq

// 在网上看到的文章，救了我的命 ProvidePlugin + expose-loader 引入jq

//

// 如果你把jQuery看做是一个普通的js模块来加载（要用到jQuery的模块统统先require

// 后再使用），那么，当你加载老式jQuery插件时，往往会提示找不到jQuery实例

// 有时候是提示找不到$），这是为啥呢？

// 要解释这个问题，就必须先稍微解释一下jQuery插件的机制：jQuery插件是通过

// jQuery提供的jQuery.fn.extend(object)和jQuery.extend(object)这俩方法，来

// 把插件本身实现的方法挂载到jQuery（也即$）这个对象上的。传统引用jQuery及

// 其插件的方式是先用<script>加载jQuery本身，然后再用同样的方法来加载其插件；

// jQuery会把jQuery对象设置为全局变量（当然也包括了$），既然是全局变量，那么

// 插件们很容易就能找到jQuery对象并挂载自身的方法了。

//

// 而webpack作为一个遵从模块化原则的构建工具，自然是要把各模块的上下文环境给

// 分隔开以减少相互间的影响；而jQuery也早已适配了AMD/CMD等加载方式，换句话说，

// 我们在require jQuery的时候，实际上并不会把jQuery对象设置为全局变量。说到

// 这里，问题也很明显了，jQuery插件们找不到jQuery对象了，因为在它们各自的上下

// 文环境里，既没有局部变量jQuery（因为没有适配AMD/CMD，所以就没有相应的requi

// re语句了），也没有全局变量jQuery。

//

// A: ProvidePlugin的机制是：当webpack加载到某个js模块里，出现了未定义且名称符合

// （字符串完全匹配）配置中key的变量时，会自动require配置中value所指定的js模块

// expose-loader，这个loader的作用是，将指定js模块export的变量声明为全局变量。

//

// B:externals 调用jq

// externals是webpack配置中的一项，用来将某个全局变量“伪装”成某个js模块的exports，

// 如下面这个配置：

// externals: {'jquery': 'window.jQuery',},

// 那么，当某个js模块显式地调用var $ = require('jquery')的时候，就会把window,

// jQuery返回给它,与上述ProvidePlugin + expose-loader的方案相反，此方案是先用

// <script>加载的jQuery满足老式jQuery插件的需要，再通过externals将其转换成符合

// 模块化要求的exports。

new webpack.ProvidePlugin({

$: "jquery",

jQuery: "jquery",

"window.jQuery": "jquery",

'window.$': 'jquery',

})

]

})

## webpack.prod.conf.js

var path = require('path')

var config = require('../config')

var utils = require('./utils')

var webpack = require('webpack')

var merge = require('webpack-merge')

var baseWebpackConfig = require('./webpack.base.conf')

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

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

var env = config.build.env

var webpackConfig = merge(baseWebpackConfig, {

module: {

loaders: utils.styleLoaders({ sourceMap: config.build.productionSourceMap, extract: true })

},

devtool: config.build.productionSourceMap ? '#source-map' : false,

output: {

path: config.build.assetsRoot,

filename: utils.assetsPath('js/[name].[chunkhash].js'),

chunkFilename: utils.assetsPath('js/[id].[chunkhash].js')

},

vue: {

loaders: utils.cssLoaders({

sourceMap: config.build.productionSourceMap,

extract: true

})

},

// webpack插件位置，有固定的用法

// 1. 利用Plugin的初始方法并传入Plugin预设的参数进行初始化，生成一个实例。

// 2. 将此实例插入到webpack配置文件中的plugins参数（数组类型）里即可。

//

// 1.

plugins: [

// http://vuejs.github.io/vue-loader/en/workflow/production.html

new webpack.DefinePlugin({

'process.env': env

}),

new webpack.optimize.UglifyJsPlugin({

compress: {

warnings: false

}

}),

new webpack.optimize.OccurrenceOrderPlugin(),

// extract css into its own file

new ExtractTextPlugin(utils.assetsPath('css/[name].[contenthash].css')),

// generate dist index.html with correct asset hash for caching.

// you can customize output by editing /index.html

// see https://github.com/ampedandwired/html-webpack-plugin

new HtmlWebpackPlugin({

// filename 生成网页的HTML名字，可以使用/来控制文件文件的目录结构，最

// 终生成的路径是基于webpac配置的output.path的

filename: config.build.index,

template: 'index.html',

inject: true,

// inject，指示把加载js文件用的<script>插入到哪里，默认是插到<body>

// 的末端，如果设置为'head'，则把<script>插入到<head>里。

minify: {

removeComments: true,

collapseWhitespace: true,

removeAttributeQuotes: true

// more options:

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

},

// necessary to consistently work with multiple chunks via CommonsChunkPlugin

chunksSortMode: 'dependency'

}),

// 如果文件是多入口的文件，可能存在，重复代码，把公共代码提取出来，又不会重复下载公共代码了

// （多个页面间会共享此文件的缓存）

// CommonsChunkPlugin的初始化常用参数有解析？

// name: 这个给公共代码的chunk唯一的标识

// filename，如何命名打包后生产的js文件，也是可以用上[name]、[hash]、[chunkhash]

// minChunks，公共代码的判断标准：某个js模块被多少个chunk加载了才算是公共代码

new webpack.optimize.CommonsChunkPlugin({

name: 'vendor',

minChunks: function (module, count) {

// any required modules inside node\_modules are extracted to vendor

return (

module.resource &&

/\.js$/.test(module.resource) &&

module.resource.indexOf(

path.join(\_\_dirname, '../node\_modules')

) === 0

)

}

}),

// extract webpack runtime and module manifest to its own file in order to

// prevent vendor hash from being updated whenever app bundle is updated

new webpack.optimize.CommonsChunkPlugin({

name: 'manifest',

chunks: ['vendor']

})

]

})

if (config.build.productionGzip) {

var CompressionWebpackPlugin = require('compression-webpack-plugin')

webpackConfig.plugins.push(

new CompressionWebpackPlugin({

asset: '[path].gz[query]',

algorithm: 'gzip',

test: new RegExp(

'\\.(' +

config.build.productionGzipExtensions.join('|') +

')$'

),

threshold: 10240,

minRatio: 0.8

})

)

}

module.exports = webpackConfig

<https://blog.csdn.net/liangklfang/article/details/54944012>

# webpack-dev-server简单使用

原创 2017年02月09日 09:56:02

建议在[github阅读](https://github.com/liangklfangl/webpack-dev-server)

### 1.npwebpack-dev-server配置

#### 1.1 ContentBase

webpack-dev-server会使用当前的路径作为请求的资源路径，但是你可以通过指定content base来修改这个默认行为:

$ webpack-dev-server --content-base build/

* 1

这样webpack-dev-server就会使用build目录下的文件来处理网络请求。他会监听资源文件，当他们改变的时候会自动编译。这些改变的bundle将会从内存中直接拿出来进而处理网络请求(所谓的改变的bundle指的就是,相对你在publicPath中指定的路径的资源)，而不会被写出到我们的output路径下面。如果一个已经存在的bundle具有相同的URL，那么我们也会使用内存中的资源来替换他!比如我们有一个如下的配置:

module.exports = {

entry: {

app: ["./app/main.js"]

},

output: {

path: path.resolve(\_\_dirname, "build"),

publicPath: "/assets/",

filename: "bundle.js"

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

那么我们要访问编译后的资源可以通过localhost:8080/assets/bundle.js来访问。如果我们在build目录下有一个文件，那么我们可以使用下面的方式来访问js资源:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

<script src="assets/bundle.js"></script>

</body>

</html>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

此时你会看到控制台输出如下内容:

主要关注下面两句输出:

Webpack result is served from /assets/

Content is served from /users/…./build(因为我们设置了contentBase)

注意：我们此时通过http://localhost:8080/index.html来访问build下的index.html;同时，我们的会发现在build目录下并没有生成我们的bundle.js文件本身。也就是说，此时我们的静态文件全部从build目录来获取，我们生成的静态文件bundle.js也在build目录之下，不过要访问它必须要加上publicPath路径才可以！

#### 1.2 自动刷新

##### 1.2.1 iframe mode:

我们的页面被嵌套在一个iframe中，当资源改变的时候会重新加载。只需要在路径中加入webpack-dev-server就可以了,不需要其他的任何处理:

http://localhost:8080/webpack-dev-server/index.html

* 1

从而在页面中就会产生如下的一个iframe标签并注入css/js/DOM:

这个iframe页面会请求 live.bundle.js ,其中里面会新建一个 Iframe ，你的应用就被注入到了这个 Iframe 当中。同时 live.bundle.js 中含有 socket.io 的 client 代码，这样它就能和 webpack-dev-server 建立的 http server 进行 websocket 通讯了，并根据返回的信息完成相应的动作。(总之，因为我们的http://localhost:8080/webpack-dev-server/index.html访问的时候加载了live.bundle.js，其具有websocket的client代码，所以当websocket-dev-server服务端代码发生变化的时候会通知到这个页面，这个页面只是需要重新刷新iframe中的页面就可以了)

该模式有如下作用:

No configuration change needed.（不需要修改配置文件）

Nice information bar on top of your app.(在app上面有information bar)

URL changes in the app are not reflected in the browser’s URL bar.(在app里面的URL改变不会反应到浏览器的地址栏中)

##### 1.2.2 inline mode

一个小的webpack-dev-server的客户端入口被添加到文件中，用于自动刷新页面。其中在cli中输入的是:

webpack-dev-server --inline --content-base ./build

* 1

此时在页面中输出的内容中看不到插入任何的js代码:

但是在控制台中可以清楚的知道页面的重新编译等信息:

该模式有如下作用:

Config option or command line flag needed.(webpack配置或者命令行配置)

Status information in the console and (briefly) in the browser’s console log.(状态信息在浏览器的console.log中)

URL changes in the app are reflected in the browser’s URL bar(URL的改变会反应到浏览器的地址栏中).

每一个模式都是支持Hot Module Replacement的，在HMR模式下，每一个文件都会被通知内容已经改变而不是重新加载整个页面。因此，在HMR执行的时候可以加载更新的模块，从而把他们注册到运行的应用里面。

##### 1.2.3 如何在nodejs中开启inline mode:

在webpack-dev-server配置中没有inline:true去开启inline模式，因为webpack-dev-server模块无法访问webpack的配置。因此，用户必须添加webpack-dev-server的客户端入口文件到webpack的配置中，具体方式如下:

方式1：To do this, simply add the following to all entry points: webpack-dev-server/client?http://«path»:«port»/,也就是在entry中添加一个内容:

entry: {

app: [

'webpack-dev-server/client?http://localhost:8080/',

"./app/main.js"

]

}

* 1
* 2
* 3
* 4
* 5
* 6

方式2：通过下面的代码来完成:

var config = require("./webpack.config.js");

config.entry.app.unshift("webpack-dev-server/client?http://localhost:8080/");

var compiler = webpack(config);

var server = new WebpackDevServer(compiler, {...});

server.listen(8080);

* 1
* 2
* 3
* 4
* 5

或者也可以在HTML中加入下面的文件来完成:

<script src="http://localhost:8080/webpack-dev-server.js"></script>

* 1

#### 1.3 Hot Module Replacement

为我们的webpack-dev-server开启HMR模式只需要在命令行中添加–hot，他会将HotModuleReplacementPlugin这个插件添加到webpack的配置中去，所以开启HotModuleReplacementPlugin最简单的方式就是使用inline模式。

##### 1.3.1 inline model in ClI

你只需要在命令行中添加–inline –hot就可以自动实现。这时候webpack-dev-server就会自动添加webpack/hot/dev-server入口文件到你的配置中去，这时候你只是需要访问下面的路径就可以了http://«host»:«port»/«path»。在控制台中你可以看到如下的内容：

其中以[HMR]开头的部分来自于webpack/hot/dev-server模块，而以[WDS]开头的部分来自于webpack-dev-server的客户端。下面的部分来自于webpack-dev-server/client/index.js内容，其中的log都是以[WDS]开头的:

function reloadApp() {

if(hot) {

log("info", "[WDS] App hot update...");

window.postMessage("webpackHotUpdate" + currentHash, "\*");

} else {

log("info", "[WDS] App updated. Reloading...");

window.location.reload();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

而在我们的webpack/hot/dev-server中的log都是以[HMR]开头的(他是来自于webpack本身的一个plugin):

if(!updatedModules) {

console.warn("[HMR] Cannot find update. Need to do a full reload!");

console.warn("[HMR] (Probably because of restarting the webpack-dev-server)");

window.location.reload();

return;

}

* 1
* 2
* 3
* 4
* 5
* 6

注意：我们必须指定正确的output.publicPath,否则热更新的chunks不会被加载！

##### 1.3.2 Hot Module Replacement with node.js API

此时需要修改三处配置文件：

第一：添加一个webpack的入口点，也就是webpack/hot/dev-server

第二：添加一个new webpack.HotModuleReplacementPlugin()到webpack的配置中

第三：添加hot:true到webpack-dev-server配置中，从而在服务端启动HMR(可以在cli中使用webpack-dev-server –hot)

if(options.inline) {

var devClient = [require.resolve("../client/") + "?" + protocol + "://" + (options.public || (options.host + ":" + options.port))];

//将webpack-dev-server的客户端入口添加到的bundle中，从而达到自动刷新

if(options.hot)

devClient.push("webpack/hot/dev-server");

//这里是webpack-dev-server中对hot配置的处理

[].concat(wpOpt).forEach(function(wpOpt) {

if(typeof wpOpt.entry === "object" && !Array.isArray(wpOpt.entry)) {

Object.keys(wpOpt.entry).forEach(function(key) {

wpOpt.entry[key] = devClient.concat(wpOpt.entry[key]);

});

} else {

wpOpt.entry = devClient.concat(wpOpt.entry);

}

});

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

满足上面三个条件的nodejs使用方式如下:

var config = require("./webpack.config.js");

config.entry.app.unshift("webpack-dev-server/client?http://localhost:8080/", "webpack/hot/dev-server");

//条件一(添加了webpack-dev-server的客户端和HMR的服务端)

var compiler = webpack(config);

var server = new webpackDevServer(compiler, {

hot: true //条件二(--hot配置，webpack-dev-server会自动添加HotModuleReplacementPlugin),条件三

...

});

server.listen(8080);

### 2.webpack-dev-server启动proxy代理

#### 2.1 代理配置

webpack-dev-server使用[http-proxy-middleware](https://github.com/chimurai/http-proxy-middleware)去把请求代理到一个外部的服务器，配置的样例如下：

proxy: {

'/api': {

target: 'https://other-server.example.com',

secure: false

}

}

// In webpack.config.js

{

devServer: {

proxy: {

'/api': {

target: 'https://other-server.example.com',

secure: false

}

}

}

}

// Multiple entry

proxy: [

{

context: ['/api-v1/\*\*', '/api-v2/\*\*'],

target: 'https://other-server.example.com',

secure: false

}

]

* 5

这种代理在很多情况下是很重要的，比如你可以把一些静态文件通过本地的服务器加载，而一些API请求全部通过一个远程的服务器来完成。还有一个情景就是在两个独立的服务器之间进行请求分割，如一个服务器负责授权而另外一个服务应用本身。

#### 2.2 绕开代理

通过一个函数的返回值可以视情况的绕开一个代理。这个函数可以查看http请求和响应以及一些代理的选项。它必须返回要么是false要么是一个URL的path，这个path将会用于处理请求而不是使用原来代理的方式完成。下面的例子的配置将会忽略来自于浏览器的HTTP请求，他和historyApiFallback配置类似。浏览器请求可以像往常一样接收到HTML文件，但是API请求将会被代理到另外的服务器：

proxy: {

'/some/path': {

target: 'https://other-server.example.com',

secure: false,

bypass: function(req, res, proxyOptions) {

if (req.headers.accept.indexOf('html') !== -1) {

console.log('Skipping proxy for browser request.');

return '/index.html';

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

#### 2.3 代理请求中重写URL

对于代理的请求可以通过提供一个函数来重写，这个函数可以查看或者改变http请求。下面的例子就会重写HTTP请求，其主要作用就是移除URL前面的/api部分。

proxy: {

'/api': {

target: 'https://other-server.example.com',

pathRewrite: {'^/api' : ''}

}

}

* 1
* 2
* 3
* 4
* 5
* 6

其中pathRewrite配置来自于http-proxy-middleware。

#### 2.4 代理本地虚拟主机

http-proxy-middleware会预解析本地hostname成为localhost,你可以使用下面的配置来修改这种默认行为：

var server = new webpackDevServer(compiler, {

quiet: false,

stats: { colors: true },

proxy: {

"/api": {

"target": {

"host": "action-js.dev",

"protocol": 'http:',

"port": 80

},

ignorePath: true,

changeOrigin: true,

secure: false

}

}

});

server.listen(8080);

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

### 3.webpack-dev-server CLI

webpack-dev-server命令行的使用如下:

$ webpack-dev-server <entry>

* 1

所有的webpack cli配置在webpack-dev-server cli中都是存在的有效的，除了output的默认参数。   
–content-base

var wpOpt = require("webpack/bin/convert-argv")(optimist, argv, {

outputFilename: "/bundle.js"

});

if(options.inline) {

var devClient = [require.resolve("../client/") + "?" + protocol + "://" + (options.public || (options.host + ":" + options.port))];

if(options.hot)

devClient.push("webpack/hot/dev-server");

//添加webpack/hot/dev-server入口

[].concat(wpOpt).forEach(function(wpOpt) {

if(typeof wpOpt.entry === "object" && !Array.isArray(wpOpt.entry)) {

Object.keys(wpOpt.entry).forEach(function(key) {

wpOpt.entry[key] = devClient.concat(wpOpt.entry[key]);

});

} else {

wpOpt.entry = devClient.concat(wpOpt.entry);

}

});

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

–hot: adds the HotModuleReplacementPlugin and switch the server to hot mode. Note: make sure you don’t add HotModuleReplacementPlugin twice.

–hot –inline also adds the webpack/hot/dev-server entry.

–public: overrides the host and port used in –inline mode for the client (useful for a VM or Docker).

–lazy: no watching, compiles on request (cannot be combined with –hot).

–https: serves webpack-dev-server over HTTPS Protocol. Includes a self-signed certificate that is used when serving the requests.

–cert, –cacert, –key: Paths the certificate files.

–open: opens the url in default browser (for webpack-dev-server versions > 2.0).

–history-api-fallback: enables support for history API fallback.

–client-log-level: controls the console log messages shown in the browser. Use error, warning, info or none.

### 4.Additional configuration options

#### 4.1 webpack-dev-server配置

当使用cli的时候，可以把webpack-dev-server的配置放在一个单独的文件中，其中key是devServer。在cli中传入的参数将会覆盖我们的配置文件的内容。如下例：

module.exports = {

// ...

devServer: {

hot: true

}

}

* 1
* 2
* 3
* 4
* 5
* 6

var WebpackDevServer = require("webpack-dev-server");

var webpack = require("webpack");

var fs = require("fs");

var compiler = webpack({

// configuration

});

var server = new WebpackDevServer(compiler, {

// webpack-dev-server options

contentBase: "/path/to/directory",

// Can also be an array, or: contentBase: "http://localhost/",

hot: true,

// Enable special support for Hot Module Replacement

// Page is no longer updated, but a "webpackHotUpdate" message is sent to the content

// Use "webpack/hot/dev-server" as additional module in your entry point

// Note: this does \_not\_ add the `HotModuleReplacementPlugin` like the CLI option does.

historyApiFallback: false,

// Set this as true if you want to access dev server from arbitrary url.

// This is handy if you are using a html5 router.

compress: true,

// Set this if you want to enable gzip compression for assets

proxy: {

"\*\*": "http://localhost:9090"

},

// Set this if you want webpack-dev-server to delegate a single path to an arbitrary server.

// Use "\*\*" to proxy all paths to the specified server.

// This is useful if you want to get rid of 'http://localhost:8080/' in script[src],

// and has many other use cases (see https://github.com/webpack/webpack-dev-server/pull/127 ).

setup: function(app) {

// Here you can access the Express app object and add your own custom middleware to it.

// For example, to define custom handlers for some paths:

// app.get('/some/path', function(req, res) {

// res.json({ custom: 'response' });

// });

},

// pass [static options](http://expressjs.com/en/4x/api.html#express.static) to inner express server

staticOptions: {

},

clientLogLevel: "info",

// Control the console log messages shown in the browser when using inline mode. Can be `error`, `warning`, `info` or `none`.

// webpack-dev-middleware options

quiet: false,

noInfo: false,

lazy: true,

filename: "bundle.js",

watchOptions: {

aggregateTimeout: 300,

poll: 1000

},

// It's a required option.

publicPath: "/assets/",

headers: { "X-Custom-Header": "yes" },

stats: { colors: true },

https: {

cert: fs.readFileSync("path-to-cert-file.pem"),

key: fs.readFileSync("path-to-key-file.pem"),

cacert: fs.readFileSync("path-to-cacert-file.pem")

}

});

server.listen(8080, "localhost", function() {});

// server.close();

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71

其中的配置可以查看[webpack-dev-server](http://webpack.github.io/docs/webpack-dev-middleware.html)。注意：我们的webpack配置没有传入到我们的WebpackDevServer中，因此，webpack中的devServer配置并非用于这个场景。而且，在webpackDevServer中是没有inline模式的，因此如下的js必须手动插入到页面中:

<script src="http://localhost:8080/webpack-dev-server.js"><\/script>

* 1

#### 4.2 historyApiFallback选项

当你使用HTML5的history API的时候，当404出现的时候你可能希望使用index.html来作为请求的资源，这时候你可以使用这个配置historyApiFallback:true。然而，如果你修改了output.publicPath，你就需要指定重定向的URL，你可以使用historyApiFallback.index选项。

// output.publicPath: '/foo-app/'

historyApiFallback: {

index: '/foo-app/'

}

* 1
* 2
* 3
* 4

使用rewrite选项你可以重新设置静态资源

historyApiFallback: {

rewrites: [

// shows views/landing.html as the landing page

{ from: /^\/$/, to: '/views/landing.html' },

// shows views/subpage.html for all routes starting with /subpage

{ from: /^\/subpage/, to: '/views/subpage.html' },

// shows views/404.html on all other pages

{ from: /./, to: '/views/404.html' },

],

},

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

### 5.组合一个已经存在的服务器

你可能想要在生产环境中运行一个后置服务器，而我们的webpack-dev-server是不应该作为一个后置服务器的，他的主要工作就是处理静态文件的请求。

你可以运行两个服务器：webpack-dev-server和后置服务器

这时候，一方面：你需要让webpack产生的资源去请求我们的webpack-dev-server，即使是后置服务器中的HTML请求。另一方面：你需要让你的后置服务器产生HTML页面，而这个页面包括script标签，其指向我们的webpack-dev-server中的资源；除了这两点，你需要把webpack-dev-server和webpack-dev-server runtime连接起来以便当重新编译后可以触发加载。

让webpack请求我们的webpack-dev-server，你需要在output.publicPath选项中提供一个完整的URL；为了让webpack-dev-server和他的runtime链接起来，我们可以开启–inline模式。webpack-dev-server cli会自动添加一个入口点，其可以建立Websocket连接(你也可以使用iframe模式，如果你为webpack-dev-server指定了–content-base，这个contentBase指向后置服务器。如果你需要为你的后置服务器添加一个websocket连接，那么你就需要使用iframe模式)。

当你使用的是inline模式的时候，你只需要在web浏览器中打开你的后置服务器的URL即可(iframe模式的时候，你需要为URL添加/webpack-dev-server/前缀)。下面是一个例子:

webpack-dev-server on port 8080.

backend server on port 9090.

generate HTML pages with

版权声明：webpack专栏文章已经整理成册，欢迎访问http://gitbook.cn/gitchat/column/59f57e2549cd43306135e255预定 https://blog.csdn.net/liangklfang/article/details/54944012

<https://www.cnblogs.com/y896926473/articles/6011711.html>

## 1. webpack简介

**webpack 是一个模块打包工具**。它使得模块相互依赖并且可构建等价于这些模块的静态资源。相比于已经存在的模块打包器（module bundler），webpack的开发动机是实现代码分包（[Code Splitting](http://webpack.github.io/docs/code-splitting.html) ）和通过模块化完成代码的无缝集成。webpack可以根据项目需求合并代码，并且支持按需加载。

webpack入门，可以参看：petehunt的[Webpack howto](http://www.h-simon.com/?p=9)

**webpack的实现目标是：**

* 拆分依赖树（dependency tree）为多个按需加载的chunk
* 保证快速首屏加载
* 每种静态资源都可成为模块
* 能够将第三方库视作一个模块来处理
* 能够定制模块打包器的几乎任何部分
* 适合大型项目

## 2. webpack基本使用

安装webpack之前，请确认node已经安装完毕，且npm包管理器可用。

### 2.1 全局安装webpack

|  |  |
| --- | --- |
| 1  2 | npm  install  webpack -g |

### 2.2 全局安装 webpack-dev-server

|  |  |
| --- | --- |
| 1  2 | npm install webpack-dev-server -g |

### 2.3 项目搭建

在项目的根目录下运行：

|  |  |
| --- | --- |
| 1  2 | npm install webpack |

在项目根目录下新建webpack.config.js。我们通过这个文件来处理控制webpack，给出我们想要的输出。

**webpack.config.js的简单的配置如下：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | module.exports = {      context: \_\_dirname + '/src',      entry: './index/index.js',      output: {          path: \_\_dirname + '/build',          filename: 'bundle.js'      }  }; |

### 2.4 开发调试

项目根目录运行：

|  |  |
| --- | --- |
| 1  2 | webpack-dev-server --progress --colors |

此时，访问： <http://localhost:8080/index.bundle.js> 即可访问到编译之后的js了。

### 2.5 产品发布

通过webpack打包发布，运行：

|  |  |
| --- | --- |
| 1  2 | webpack -p |

按照我们上面的webpack.config.js文件的配置，打包成功之后会生成一个build文件夹，里面会包含打包好的js文件，集成了所有的依赖库和业务逻辑代码，我们只需将此build文件夹发布到线上即可。

至此一个完整流程的webpack运行流程梳理完毕，当然webpack还有很多功能来实现我们开发中所遇到的各种变态需求，在第三章中我抽出了一些项目中常用的功能。

## 3. 常用功能

### 3.1 设置入口

配置那些js需要处理，entry有三种写法，每个入口称为一个chunk。

* **字符串：**  
  entry: "./index/index.js" ：配置模块会被解析为模块，并在启动时加载。chunk名为默认为main， 具体打包文件名视output配置而定。

* **数组**  
  entry: ['./src/mod1.js', [...,] './src/index.js'] ：所有的模块会在启动时 **按照配置顺序** 加载，合并到最后一个模块会被导出。chunk名默认为main
* **对象**  
  entry: {index: '...', login : [...] }：如果传入Object，则会生成多个入口打包文件， key是chunk名，value可以是字符串，也可是数组。

**例如**

|  |  |
| --- | --- |
| 1  2  3  4  5 | entry: {      index: './index/index.js',      login: ['./mod/mod1.js', './index/login.js']  } |

### 3.2 配置输出目录

设置入口配置的文件的输出规则，通过output对象实现，常用设置：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | output: {      path: \_\_dirname + '/build',      filename: '[name]-[id].js',      publicPath: '/asstes/'  } |

**其中：**

* output.path ：指定输出文件路径，通常设置为\_\_dirname + ‘/build’,
* output.filename: 输出文件名称，有下面列出的四种可选的变量。 filename项的配置可以是这几种的任意一种或多种的组合。 如 output.filename = ‘[name]-[id].js’, 则输出就是 index-1.js、 login-2.js。
  + [id], chunk的id
  + [name] ,chunk名
  + [hash], 编译哈希值
  + [chunkhash] , chunk的hash值
* output.publicPath：设置为想要的资源访问路径。访问时，则需要通过类似http://localhost:8080/asstes/index-1.js来访问资源，如果没有设置，则默认从站点根目录加载。

### 3.3 设置loader

loader是webpack中比较重要的部分，她是处理各类资源的执行者。它们是一系列的函数（运行在node.js中），将资源中的代码作为参数，然后返回新的代码。你可以用loader告诉webpack可以加载哪些文件，或者不加载哪些文件。

**Loader的特点**

* 可以链式执行。它们在一个管道中被提交，只需要保证最后的loader返回JavaScript即可，其他loader可以返回任意方便下一个loader处理的内容。
* 可以异步or同步执行
* 运行在Node.js中，可以做几乎任何事儿
* 可以接收query参数，用于向loader传递参数
* 配置中可与正则/扩展结合使用
* 可以在npm中发布并使用
* 除了main,其他模块可以导出成loader
* 可以通过配置调入
* 和插件（plugins）配合可获得更多功能
* 可生成其他格式文件

**安装loader**

|  |  |
| --- | --- |
| 1  2 | npm install xxx-loader --save |

或者

|  |  |
| --- | --- |
| 1  2 | npm install xxx-loader --save-dev |

其中，XXX为webpack支持的loader名，常用的有：html、css、jsx、coffee、jade、less、sass、style等。  
你可以通过[webpack loader 列表](http://webpack.github.io/docs/list-of-loaders.html) 查看所有支持的loader。当然你可以自己根据需求创建并发布loader。

**配置loader**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | modules: {      loaders: [          {              test: /\.js$/, //匹配希望处理文件的路径              exclude: /node\_modules/, // 匹配不希望处理文件的路径              loaders: 'xxx-loader?a=x&b=y'  //此处xxx-loader 可以简写成xxx , ？后以query方式传递给loader参数          },          ...      ]  } |

**多loader调用示例:**

在js中，如果要直接解析某个文件，你可以采用：

|  |  |
| --- | --- |
| 1  2 | require('jade!./index.jade') |

如果要解析css，并内联之，需要使用到分隔符 !

|  |  |
| --- | --- |
| 1  2 | require(!style!css!./style.css) |

同理，如果要解析less， 转换成css之后，再内联之，写法如：

|  |  |
| --- | --- |
| 1  2 | require('!style!css!less!./style.less!') ; // 此语句的含义是，先调用less-loader解析style.less文件，输出结果会被css-loader处理, 然后再被style-loader处理 |

同理，在webpack.config.jsp配置文件中，只需要制定处理的loader序列：  
即：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ...  loaders: [      {              test: /\.less$/,              loader: "style!css!less"          }  ]  ... |

本章会介绍比较常用的loader的配置方法。

#### 3.3.1 解析并抽取css

在webpack中css默认方案是，将css编译并通过内联的方式在html页面中插入<style>样式标签。当然这远远不能满足我们的要求，webpack提供css-loader模块用于编译css文件，并且提供了插件extract-text-webpack-plugin将css从js代码中抽出并合并。你可以访问[此处](https://github.com/webpack/extract-text-webpack-plugin),查看文档和例子。  
这样你可以在模块中，尽情使用 require(style.css), webpack会帮你做解析，合并entry中定义js及其依赖中所用到的所有css，然后生成一个指定的css文件。

**配置如下：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | <br />var ExtractTextPlugin = require('extract-text-webpack-plugin');    ...  module: {      loaders: [          {              test: /\.css$/,              loader: ExtractTextPlugin.extract(                  "style-loader",                  "css-loader?sourceMap"              )          }, {              test: /\.less$/,              loader: ExtractTextPlugin.extract("style-loader", "css-loader!less-loader")          }      ]    },  plugins: [      new ExtractTextPlugin("css/[name].css?[hash]-[chunkhash]-[contenthash]-[name]", {          disable: false,          allChunks: true      })  ]  ... |

#### 3.3.2 处理图片、字体等文件

在css中或者js逻辑中，都会涉及到require图片的情况，webpack可以内联图片地址到打包js中并且通过require()返回图片路径。当然，不只是图片，还有css中用到的iconfont，特殊情况用到的flash等，都可以相似处理。这里，我们需要用到url-loader 或 file-loader。

* file-loader: 将匹配到的文件复制到输出文件夹，并根据output.publicPath的设置返回文件路径
* url-loader: 类似file-loader ,但是它可以返回一个DataUrl (base 64)如果文件小于设置的限制值limit。

同样，这之前，你需要实现配置相关loader。

安装url-loader 和 file-loader:

|  |  |
| --- | --- |
| 1  2 | npm install url-loader file-loader --save |

配置：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | module:{      loaders:[          {              test: /\.(png|jpg)$/,              loader: 'url-loader?limit=8192' //  <= 8kb的图片base64内联          },              {                  test: /\.woff(\?v=\d+\.\d+\.\d+)?$/,                  loader: 'url?limit=10000&minetype=application/font-woff'              },              {                  test: /\.woff2(\?v=\d+\.\d+\.\d+)?$/,                  loader: 'url?limit=10&minetype=application/font-woff'              },              {                  test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/,                  loader: 'url?limit=10&minetype=application/octet-stream'              },              {                  test: /\.eot(\?v=\d+\.\d+\.\d+)?$/,                  loader: 'file'              },              {                  test: /\.svg(\?v=\d+\.\d+\.\d+)?$/,                  loader: 'url?limit=10&minetype=image/svg+xml'              }      ]  } |

通过向url-loader传递参数，如果图片小于8kb，则base64内联，大于8kb，则通过output.publishPath配置的前缀将图片路径写入代码，并提取图片到输出目录。

#### 3.3.3 解析JSX

在React项目中，需要解析JSX和相关JavaScript文件，需要下载loader：

|  |  |
| --- | --- |
| 1  2 | npm install react-hot-loader jsx-loader --save |

同样，配置loader:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | ...  loaders: [         {             test: /\.js$/,             exclude: /node\_modules/,             loader: 'react-hot!jsx-loader?harmony'         }     ]     ... |

#### 3.3.4解析VUE

和React项目类似，如果要解析VUE框架编写的.vue文件，需要下载loader:

|  |  |
| --- | --- |
| 1  2 | npm install vue-loader --save |

配置loader:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | loaders: [    {        test: /\.vue$/,        loader: 'vue-loader'    }  ] |

但需要注意的是，如果你的代码中用到了如jade，less等其他语法，可能需要提前下载相应loader到本地。vue-loader的介绍可以查看： [vue-loader](https://www.npmjs.com/package/vue-loader)

#### 3.3.5 解析ES6语法

babel可以让我们在编写代码的时候，使用更高级的ECMAScript6的语法。然后我们编写的JS文件可以被编译成可被低版本浏览器处理的常规代码。

**使用方法**  
安装loader:

|  |  |
| --- | --- |
| 1  2 | npm install babel-loader --save |

配置loader:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | loaders: [    {        test: /\.js$/,        exclude: /node\_modules/,        loader: 'babel'    }  ] |

**例如:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | const a = 1;    console.log(a);    if (true) {      let a = 3;      console.log(a);  }    console.log(a); |

解析为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var a = 1;    console.log(a);    if (true) {      var \_a = 3;      console.log(\_a);  }    console.log(a); |

### 3.4 其他

####3.4.1 借助web\_modules引用外部库  
有些时候，我们用到的第三方库并没有采用CommonJS或AMD规范，也没有提交到npm。这样的话，我们无法通过npm来下载，并通过require()来引用这些库。  
webpack给我们提供了一个很好的实现方式。我们可以在项目根目录下，创建一个叫做web\_modules的文件夹，然后将需要用到的第三方库存放在此处。那么之后，不需要做任何设置，可以在我们的逻辑代码中使用require(  
‘xx-lib.js’)并且使用了。

文件组织如下：

此时，我们就可以在业务逻辑中，大胆地使用web\_modules中配置的库了，打包的时候，webpack会自动将web\_modules中被用到的库封装。

\*\*例如： \*\*

|  |  |
| --- | --- |
| 1  2  3  4 | var director = require('director')      var Router = director.Router();      ... |

#### 3.4.2 去除多个文件中的频繁依赖

当我们经常使用React、jQuery等外部第三方库的时候，通常在每个业务逻辑JS中都会遇到这些库。  
如我们需要在各个文件中都是有jQuery的$对象，因此我们需要在每个用到jQuery的JS文件的头部通过require('jquery')来依赖jQuery。 这样做非常繁琐且重复，因此webpack提供了我们一种比较高效的方法，我们可以通过在配置文件中配置使用到的变量名，那么webpack会自动分析，并且在编译时帮我们完成这些依赖的引入。

webpack.config.js中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var webpack = require('webpack');    ...  plugins: [     new webpack.ProvidePlugin({         'Moment': 'moment',         "$": "jquery",         "jQuery": "jquery",         "window.jQuery": "jquery",         "React": "react"     })  ]  ... |

这样，我们在JS中，就不需要引入jQuery等常用模块了，直接使用配置的这些变量，webpack就会自动引入配置的库。

#### 3.4.3 开发环境与发布环境配置

某些情况，我们需要在页面中输出开发调试内容，但是又不想让这些调试内容在发布的时候泄露出去，那么我们可以采用魔力变量(magic globals)来处理。

**配置文件：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | var webpack = require('webpack');    var definePlugin = new webpack.DefinePlugin({      \_\_DEV\_\_: JSON.stringify(JSON.parse(process.env.BUILD\_DEV || 'false')),      \_\_PRERELEASE\_\_: JSON.stringify(JSON.parse(process.env.BUILD\_PRERELEASE || 'true'))  });    ...    plugins: [      definePlugin  ]  ... |

**业务逻辑代码中写入**  
按照下面的代码写入，我们就可以在我们自己设定的环境下进行更具针对性的调试。比如我们希望在开发环境下可以AJAX可以调试本地mock数据，然后在发布的时候，可以正常访问服务端数据。那么通过此种方式可以完全实现。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | if (\_\_DEV\_\_) {      console.warn('Extra logging');      //开发环境需要进行的处理      //...  }    if (\_\_PRERELEASE\_\_) {      console.log('prerelease');      //预发环境需要进行的处理     //...  } |

**设置环境命令**

要告诉webpack我们希望当前是什么环境，只需要在命令中写入 BUILD\_DEV=1 webpck 那么webpack通过配置，就会将所有我们引用到的\_\_DEV\_\_变量设置为true。

我们可以在package.json中事先定义好命令：

|  |  |
| --- | --- |
| 1  2  3  4  5 | "scripts": {      "dev": "BUILD\_DEV=1 webpack-dev-server --progress --colors",      "build": "BUILD\_PRERELEASE=1 webpack -p"    } |

那么就可以避免输入冗长的命令了：

开发时输入：

|  |  |
| --- | --- |
| 1  2 | npm run dev |

发布时输入:

|  |  |
| --- | --- |
| 1  2 | npm run build |

#### 3.4.5 合并公共代码

项目中，对于一些常用的组件，站点公用模块经常需要与其他逻辑分开，然后合并到同一个文件，以便于长时间的缓存。要实现这一功能，配置参照:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <br />var webpack            = require('webpack');  var CommonsChunkPlugin = webpack.optimize.CommonsChunkPlugin;    ...  entry: {     a: './index/a.js',     b: './idnex/b.js',     c: './index/c.js',     d: './index/d.js'  },  ...  plugins: [     new CommonsChunkPlugin('part1.js', ['a', 'b']),     new CommonsChunkPlugin('common.js', ['part1', 'c'])  ]  ... |

##4 使用devtool调试

可以通过在配置中加入devtool项，选择预设调试工具来提高代码调试质量和效率：

* eval – 每个模块采用eval和 //@ sourceURL 来执行
* source-map – sourceMap是发散的，和output.sourceMapFilename协调使用
* hidden-source-map – 和source-map类似，但是不会添加一个打包文件的尾部添加引用注释
* inline-source-map – SourceMap以DataUrl的方式插入打包文件的尾部
* eval-source-map – 每个模块以eval方式执行并且SourceMap以DataUrl的方式添加进eval
* cheap-source-map – 去除column-mappings的SourceMap， 来自于loader中的内容不会被使用。
* cheap-module-source-map – 去除column-mappings的SourceMap, 来自于loader中的SourceMaps被简化为单个mapping文件

**各种模式的对比：**

| **devtool** | **构建速度** | **再次构建速度** | **支持发布版** | **质量** |
| --- | --- | --- | --- | --- |
| eval | +++ | +++ | no | 生成代码 |
| cheap-eval-source-map | + | ++ | no | 转换代码(lines only) |
| cheap-source-map | + | o | yes | 转换代码(lines only) |
| cheap-module-eval-source-map | o | ++ | no | 源代码 (lines only) |
| cheap-module-source-map | o | – | yes | 源代码(lines only) |
| eval-source-map | — | + | no | 源代码 |
| source-map | — | — | yes | 源代码 |

## 5. 一个常用的配置

为了方便大家摘取，和补全文章中用于示例的代码片段，特将配置文件整理如下，作参考：

**配置文件:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105 | var webpack            = require('webpack');  var CommonsChunkPlugin = webpack.optimize.CommonsChunkPlugin;  var ExtractTextPlugin  = require('extract-text-webpack-plugin');      //自定义"魔力"变量  var definePlugin = new webpack.DefinePlugin({      \_\_DEV\_\_: JSON.stringify(JSON.parse(process.env.BUILD\_DEV || 'false')),      \_\_PRERELEASE\_\_: JSON.stringify(JSON.parse(process.env.BUILD\_PRERELEASE || 'false'))  });      module.exports = {      //上下文      context: \_\_dirname + '/src',      //配置入口      entry: {          a: './view/index/index.js',          b: './view/index/b.js',          vender: ['./view/index/c.js', './view/index/d.js']      },      //配置输出      output: {          path: \_\_dirname + '/build/',          filename: '[name].js?[hash]',          publicPath: '/assets/',          sourceMapFilename: '[file].map'      },      devtool: '#source-map',      //模块      module: {          loaders: [              {                  //处理javascript                  test: /\.js$/,                  exclude: /node\_modules/,                  loader: 'babel'              }, {                  test: /\.css$/,                  loader: ExtractTextPlugin.extract(                      "style-loader",                      "css-loader?sourceMap"                  )              }, {                  test: /\.less$/,                  loader: ExtractTextPlugin.extract(                      "style-loader",                      "css-loader!less-loader"                  )              }, {                  test: /\.(png|jpg)$/,                  loader: 'url-loader?limit=1024'              }, {                  //处理vue                  test: /\.vue$/,                  loader: 'vue-loader'              },              {                  test: /\.woff(\?v=\d+\.\d+\.\d+)?$/,                  loader: 'url?limit=10000&minetype=application/font-woff'              },              {                  test: /\.woff2(\?v=\d+\.\d+\.\d+)?$/,                  loader: 'url?limit=10&minetype=application/font-woff'              },              {                  test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/,                  loader: 'url?limit=10&minetype=application/octet-stream'              },              {                  test: /\.eot(\?v=\d+\.\d+\.\d+)?$/,                  loader: 'file'              },              {                  test: /\.svg(\?v=\d+\.\d+\.\d+)?$/,                  loader: 'url?limit=10&minetype=image/svg+xml'              }          ]        },      plugins: [          //公用模块          new CommonsChunkPlugin('common.js', ['a', 'b']),          //设置抽出css文件名          new ExtractTextPlugin("css/[name].css?[hash]-[chunkhash]-[contenthash]-[name]", {              disable: false,              allChunks: true          }),          //定义全局变量          definePlugin,          //设置此处，则在JS中不用类似require('./base')引入基础模块， 只要直接使用Base变量即可          //此处通常可用做，对常用组件，库的提前设置          new webpack.ProvidePlugin({              Moment: 'moment', //直接从node\_modules中获取              Base: '../../base/index.js' //从文件中获取          })      ],      //添加了此项，则表明从外部引入，内部不会打包合并进去      externals: {          jquery: 'window.jQuery',          react: 'window.React',          //...      }  }; |

<https://segmentfault.com/q/1010000008503911>

# [webpack中的resolve和resolveLoader分别是作什么的](https://segmentfault.com/q/1010000008503911)

最好可以配合图中代码解释一下



按照 webpack 官方的说法，resolve配置用来影响webpack模块解析规则。解析规则也可以称之为检索，索引规则。配置索引规则能够缩短webpack的解析时间，提升打包速度。

如图：  
先说几个问题问题。

1. resolve.root 用来配置搜索路径集合。root配置必须是绝对路径，不能存在./app/modules之类的相对路径。
2. resolve.modulesDirectory 是指需要向上搜索的目录名称（即如果当前目录找不到，找上级目录），一般只会是node\_modules之类的。其他自定义的资源一般不需要向上搜索，可以配置alias

root和modulesDirectory 在webpack@2.x中被舍弃了，使用了新的规则resolve.modules，这是前两者的结合体

resolveLoader相当于是针对webpack Loader 的单独 resolve 配置，做用和resolve一样，但只作用于webpack loader

我的完整 webpack.config.js

var path = require("path");

var webpack = require("webpack");

var HtmlWebPackPlugin = require("html-webpack-plugin");

var CopyWebPackPlugin = require("copy-webpack-plugin");

module.exports = {

devtool: 'inline-source-map',

entry: {

"ajax/index": [

path.resolve(\_\_dirname, "code/index.js"),

path.resolve(\_\_dirname, "code/sale.js")

],

"order/myorder": [

path.resolve(\_\_dirname, "code/index.js"),

path.resolve(\_\_dirname, "code/sale.js"),

path.resolve(\_\_dirname, "code/order.ts")

]

},

output: {

path: path.resolve(\_\_dirname, "public"),

publicPath: "assets",

filename: "[name].bundle.js"

},

module: {

rules: [

{ test: /\.css$/, use: ["style-loader", "css-loader"] },

{ test: /\.ts$/, use: ["ts-loader"], exclude: /node\_modules/ }

]

},

resolve: {

extensions: [".js", ".css", ".ts"]

},

devServer: {

contentBase: \_\_dirname,

compress: true,

port: 9099,

host: "localhost",

//hot: true,

inline: true

},

plugins: [

new HtmlWebPackPlugin({

template: "index.html",

filename: "index.html"

}),

new CopyWebPackPlugin([{

from: path.resolve(\_\_dirname),

to: path.resolve(\_\_dirname, "public"),

toType: File

}])

]

}

<https://www.cnblogs.com/mengfangui/p/7560475.html>

**2、配置文件：webpack.config.js**

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

var webpack = require('webpack');

module.exports = {

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

entry: {

//通用模块

'common': ['./src/page/common/index.js'],

'login': ['./src/page/login/index.js'],

'index': ['./src/page/index/index.js']

},

output: {

path: \_\_dirname + '/dist',

filename: 'js/[name].js'

},

//将外部变量或者模块加载进来

externals: {

'jquery': 'window.jQuery'

},

module: {

loaders: [{

test: /\.js$/,

//以下目录不处理

exclude: /node\_modules/,

//只处理以下目录

include: /src/,

loader: "babel-loader",

//配置的目标运行环境（environment）自动启用需要的 babel 插件

query: {

presets: ['latest']

}

},

//css 处理这一块

{

test: /\.css$/,

use: [

'style-loader',

{

loader: 'css-loader',

options: {

//支持@important引入css

importLoaders: 1

}

},

{

loader: 'postcss-loader',

options: {

plugins: function() {

return [

//一定要写在require("autoprefixer")前面，否则require("autoprefixer")无效

require('postcss-import')(),

require("autoprefixer")({

"browsers": ["Android >= 4.1", "iOS >= 7.0", "ie >= 8"]

})

]

}

}

}

]

},

//less 处理这一块

{

test: /\.less$/,

use: ['style-loader',

{

loader: 'css-loader',

options: {

//支持@important引入css

importLoaders: 1

}

},

{

loader: 'postcss-loader',

options: {

plugins: function() {

return [

//一定要写在require("autoprefixer")前面，否则require("autoprefixer")无效

require('postcss-import')(),

require("autoprefixer")({

"browsers": ["Android >= 4.1", "iOS >= 7.0", "ie >= 8"]

})

]

}

}

},

'less-loader'

]

},

//处理html模板

{

test: /\.html$/,

use: {

loader: 'html-loader'

}

},

//处理图片

{

test: /\.(png|jpg|gif|svg)$/i,

loaders: [

//小于8k的图片编译为base64，大于10k的图片使用file-loader

'url-loader?limit=8192&name:img/[name]-[hash:5].[ext]',

//图片压缩

'image-webpack-loader'

]

}

]

},

plugins: [

new htmlWebpackPlugin({

template: 'index.html',

filename: 'index.html'

}),

// 独立通用模块到js/base.js

new webpack.optimize.CommonsChunkPlugin({

//公共块的块名称

name: 'common',

//生成的文件名

filename: 'js/common.js'

})

]

}

# <https://segmentfault.com/a/1190000005184341>

# [webpack配置jquery以及jquery插件](https://segmentfault.com/a/1190000005184341)

* [node.js](https://segmentfault.com/t/node.js/blogs)
* [vue.js](https://segmentfault.com/t/vue.js/blogs)
* [webpack](https://segmentfault.com/t/webpack/blogs)

[**alwaysVe**](https://segmentfault.com/u/ve) 2016年05月23日发布

* **2.4k** 次浏览

项目目录大概结构：

project

|- app

| |- vendor

| | |- plugins

| | | |- plugin-1.js

| | | |- plugin-2.js

| | | |- ...

| | |

| | |- jquery.js

| |

| |- jquery-with-plugins.js

| |- main.js

|

|- js

| |- bundle.js

|

|- webpack.config.js

|- package.json

...

关键文件内容：

// app/jquery-with-plugins.js

require('vendor/jquery');

req = require.context('vendor/plugins', true, /\.js$/);

req.keys().forEach(function (plugin) {

req(plugin);

});

module.exports = jQuery;

// app/main.js

var $ = require('jquery');

$(function () {

$('.selector-1').use\_plugin\_1();

$('.selector-2').use\_plugin\_2();

});

// webpack.config.js

var path = require('path');

module.exports = {

context: path.join(\_\_dirname, 'app'),

entry: './main',

output: {

path: path.join(\_\_dirname, 'js'),

filename: 'bundle.js'

},

module: {

loaders: [

{

test: /\.js$/,

include: [

path.join(\_\_dirname, 'app/vendor')

],

loader: 'script'

}

]

},

resolve: {

alias: {

'jquery$': path.join(\_\_dirname, 'app/jquery\_with\_plugins'),

'vendor': path.join(\_\_dirname, 'app/vendor')

}

},

extensions: ['', '.js']

};

// package.json

{

"name": "temp",

"version": "0.0.0",

"private": true,

"scripts": {

"build": "webpack --progress --colors"

},

"devDependencies": {

"node-libs-browser": "^0.5.0",

"script-loader": "^0.6.1",

"webpack": "^1.9.4"

}

// app/jquery-with-plugins.js

require('vendor/jquery');

req = require.context('vendor/plugins', true, /\.js$/);

req.keys().forEach(function (plugin) {

req(plugin);

});

module.exports = jQuery;

// app/main.js

var $ = require('jquery');

$(function () {

$('.selector-1').use\_plugin\_1();

$('.selector-2').use\_plugin\_2();

});

// webpack.config.js

var path = require('path');

module.exports = {

context: path.join(\_\_dirname, 'app'),

entry: './main',

output: {

path: path.join(\_\_dirname, 'js'),

filename: 'bundle.js'

},

module: {

loaders: [

{

test: /\.js$/,

include: [

path.join(\_\_dirname, 'app/vendor')

],

loader: 'script'

}

]

},

resolve: {

alias: {

'jquery$': path.join(\_\_dirname, 'app/jquery\_with\_plugins'),

'vendor': path.join(\_\_dirname, 'app/vendor')

}

},

extensions: ['', '.js']

};

// package.json

{

"name": "temp",

"version": "0.0.0",

"private": true,

"scripts": {

"build": "webpack --progress --colors"

},

"devDependencies": {

"node-libs-browser": "^0.5.0",

"script-loader": "^0.6.1",

"webpack": "^1.9.4"

}

}