目录

[cli 3](#_Toc534406122)

[输入输出 3](#_Toc534406123)

[entry 3](#_Toc534406124)

[output 3](#_Toc534406125)

[常用配置 3](#_Toc534406126)

[输出统计数据 3](#_Toc534406127)

[基础知识 4](#_Toc534406128)

[基本配置 4](#_Toc534406129)

[命令行方式 4](#_Toc534406130)

[运行函数方式(nodeApi) 4](#_Toc534406131)

[混合方式 5](#_Toc534406132)

[基本配置文件格式 5](#_Toc534406133)

[chunk 6](#_Toc534406134)

[nodeApi 6](#_Toc534406135)

[webpack 6](#_Toc534406136)

[complier实例 6](#_Toc534406137)

[stats 6](#_Toc534406138)

[stats.hasErrors() 6](#_Toc534406139)

[stats.hasWarnings() 7](#_Toc534406140)

[stats.toJson(options) 7](#_Toc534406141)

[stats.toString(options) 7](#_Toc534406142)

[常用配置对象 8](#_Toc534406143)

[入口 8](#_Toc534406144)

[出口 8](#_Toc534406145)

[filename 8](#_Toc534406146)

[path 8](#_Toc534406147)

[publicPath 9](#_Toc534406148)

[chunkFilename和filename 9](#_Toc534406149)

[crossOriginLoading 9](#_Toc534406150)

[sourceMapFilename 10](#_Toc534406151)

[devtoolModuleFilenameTemplate 10](#_Toc534406152)

[模块 10](#_Toc534406153)

[loaders/rules 10](#_Toc534406154)

[loaders 10](#_Toc534406155)

[rules 11](#_Toc534406156)

[Rule 条件 11](#_Toc534406157)

[Rule 结果 12](#_Toc534406158)

[嵌套的 Rule 12](#_Toc534406159)

[rule.use 12](#_Toc534406160)

[test与rule.resource 13](#_Toc534406161)

[rule.query和rule.options 13](#_Toc534406162)

[Rule.exclude 13](#_Toc534406163)

[Rule.include 13](#_Toc534406164)

[rule.loaders 14](#_Toc534406165)

[rule.loader和rule.use 14](#_Toc534406166)

[rule.enforce 14](#_Toc534406167)

[UseEntry 14](#_Toc534406168)

[loader 14](#_Toc534406169)

[babel-loader 14](#_Toc534406170)

[vue-loader 15](#_Toc534406171)

[template 15](#_Toc534406172)

[script 15](#_Toc534406173)

[style 15](#_Toc534406174)

[css-loader 16](#_Toc534406175)

[解析resolve 16](#_Toc534406176)

[alias 16](#_Toc534406177)

[root 17](#_Toc534406178)

[modules 18](#_Toc534406179)

[plugins 18](#_Toc534406180)

[modulesDirectories 18](#_Toc534406181)

[fallback 18](#_Toc534406182)

[extensions 18](#_Toc534406183)

[插件 19](#_Toc534406184)

[ExtractTextPlugin 19](#_Toc534406185)

[loader-options-plugin 21](#_Toc534406186)

[DefinePlugin 22](#_Toc534406187)

[UglifyJsPlugin 22](#_Toc534406188)

[DedupePlugin 23](#_Toc534406189)

[NoErrorsPlugin 23](#_Toc534406190)

[ProvidePlugin 23](#_Toc534406191)

[CommonsChunkPlugin 23](#_Toc534406192)

[HtmlWebpackPlugin 24](#_Toc534406193)

[CopyWebpackPlugin 26](#_Toc534406194)

[ExtractTextPlugin 26](#_Toc534406195)

[配置对象 26](#_Toc534406196)

[context 26](#_Toc534406197)

[webpack-dev-server 26](#_Toc534406198)

[常见情况 27](#_Toc534406199)

[默认搜索 27](#_Toc534406200)

[打印错误信息 28](#_Toc534406201)

[路径兼容windows 28](#_Toc534406202)

[Loaders 28](#_Toc534406203)

[省略写法 28](#_Toc534406204)

[-loader 可以省略 28](#_Toc534406205)

[css-loader和style-loader 28](#_Toc534406206)

[权限问题 29](#_Toc534406207)

[Npm 29](#_Toc534406208)

[镜像 29](#_Toc534406209)

# cli

## 输入输出

webpack <entry> [<entry>] -o <output>

最后一个表示output,可以省略-o

### entry

如果传递一个形式为 <name> = <request> 的键值对，则可以创建一个额外的入口起点。它将被映射到配置选项(configuration option)的 entry 属性。

index=./src/index.js(给了个名字)

### output

映射到配置选项 output.path 和 output.filename

## 常用配置

**列出命令行所有可用的配置选项**

webpack --help , webpack -h

**使用配置文件进行构建**

webpack --config example.config.js（默认会找webpack.config.js）

**传入 Webpack 配置文件中使用的环境变量(需要使用运行函数的方式)**

webpack --env=DEVELOPMENT

**以 JSON 格式输出 Webpack 的运行结果(方便可视化)**

webpack --json , webpack -j, webpack -j > stats.json

### 输出统计数据

webpack --profile --json > compilation-stats.json

--profile 统计编译时间

# 基础知识

## 基本配置

### 命令行方式

使用webpack命令，命令行会读取一个叫webpack.config.js（或者用 –config 选项传入的一个配置文件）的文件。

### 运行函数方式(nodeApi)

如果 你用[**Node.js**](http://lib.csdn.net/base/nodejs) API，你需要把配置文件作为一个参数，传给webpack。

const webpack = require("webpack");

webpack({

// [配置对象](/configuration/)

}, (err, [stats](#stats-object)) => {

if (err || stats.hasErrors()) {

// [在这里处理错误](#-error-handling-)

}

// 处理完成

});

编译错误**不**在 err 对象内，而是需要使用 stats.hasErrors() 单独处理，

webpack **不**会并行执行多个配置。每个配置只会在前一个处理结束后才会开始处理。如果你需要 webpack 并行执行它们，你可以使用像 [parallel-webpack](https://www.npmjs.com/package/parallel-webpack) 这样的第三方解决方案。

同时传入回调函数时就会执行 webpack compiler：不传入回调函数到 webpack 执行函数中，就会得到一个 webpack Compiler 实例。你可以通过它手动触发 webpack 执行器，或者是让它执行构建并监听变更。

传入参数也会返回compiler对象，可以传给webpack-dev-server。

* .run(callback)
* .watch(watchOptions, handler)

const compiler = webpack({

// [配置对象](/configuration/)

});

compiler**.run**((err, [stats](#stats-object)) => {

// ...

});

const **watching** = compiler.watch({

<details><summary>/\* [watchOptions](/configuration/watch/#watchoptions) \*/</summary>

aggregateTimeout: 300,

poll: undefined

</details>

}, (err, [stats](#stats-object)) => {

// 在这里打印 watch/build 结果...

console.log(stats);

});

watching.close(() => {

console.log("Watching Ended.");

});

手动将当前的编译阶段废弃，但不停止监听。

watching**.invalidate**(() => {

console.warn("Invalidated.");

});

### 混合方式

### 基本配置文件格式

不需要写纯粹的JSON到配置文件中。**[JavaScript](http://lib.csdn.net/base/javascript" \t "_blank" \o "JavaScript知识库)**随便用。webpack只是一个[**node.js**](http://lib.csdn.net/base/nodejs) 模块

module.exports = {

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

output: {

path: './bin',

filename: 'app.bundle.js',

},

module: {

loaders: [{

test: /\.js$/,

exclude: /node\_modules/,

loader: 'babel-loader',

}]

}

}

## chunk

是 webpack 特定的术语被用在内部来管理 building 过程。bundle 由 chunk 组成，其中有几种类型（例如，入口 chunk(entry chunk) 和子 chunk(child chunk)）。通常 chunk 会直接对应所输出的 bundle，但是有一些配置并不会产生一对一的关系。

一个bundle可能对应多个chunk。

# nodeApi

使用nodeapi时，stats 配置选项不会在 webpack() 调用中生效，需要通过编译过程的stats对象输出统计信息。

## webpack

webpack({}, compiler(err, stats))，或者返回一个compiler实例。

可以传入数组，进行MultiCompiler。（串行处理）

### complier实例

.run(callback)

callback也是传入err,stats

.watch(watchOptions, handler)

err 对象只会包含 webpack 相关的问题，比如配置错误等。编译错误需要使用 stats.hasErrors() 单独处理

## stats

对于 webpack-dev-server，这个属性要放在 devServer 对象里。

### stats.hasErrors()

可以用来检查编译期是否有错误，返回 true 或 false。

要打印error需要toJson().errors;

### stats.hasWarnings()

可以用来检查编译期是否有警告，返回 true 或 false。

### stats.toJson(options)

以 JSON 对象形式返回编译信息。options 可以是一个字符串（预设值）或是颗粒化控制的对象：

stats.toJson("minimal"); // [更多选项如: "verbose" 等](/configuration/stats).

stats.toJson({

assets: false,

hash: true

});

### stats.toString(options)

以格式化的字符串形式返回描述编译信息（类似 [CLI](http://www.css88.com/doc/webpack2/api/cli) 的输出）。

配置对象与 [stats.toJson(options)](http://www.css88.com/doc/webpack2/api/node/#stats-tojson-options-) 一致，除了额外增加的一个选项：

stats.toString({

// ...

// 增加控制台颜色开关

colors: true

});

下面是 stats.toString() 用法的示例：

const webpack = require("webpack");

webpack({

// [配置对象](/configuration/)

}, (err, stats) => {

if (err) {

console.error(err);

return;

}

console.log(stats.toString({

chunks: false, // 使构建过程更静默无输出

colors: true // 在控制台展示颜色

}));

});

# 常用配置对象

# 入口

可以是字符串，数组，对象。

简单规则：每个 HTML 页面都有一个入口起点。单页应用(SPA)：一个入口起点，多页应用(MPA)：多个入口起点。

entry: {

home: "./home.js",

about: "./about.js",

contact: "./contact.js"

}

值可以是一个数组，数组中所有模块都会启动，但最后一个会被导出，比如：

entry:["./entry1","./engtry2"]

这种方式可以用来引入一些前置文件：比如polyfill。

相对目录是当前工作目录(调用命令的目录)，即使配置文件是在子目录里。（所以一般用\_\_dirname确定文件所在目录）。

# 出口

必须有filename

## filename

默认值是 "[name].js"。

可以用多个chunk属性

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

## path

绝对路径！！！

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

## publicPath

按需加载(on-demand-load)或加载外部资源(external resources)（如图片、文件等）可能用到。常用下面红色那两种方式。

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

publicPath: "https://cdn.example.com/assets/"

示例

publicPath: "https://cdn.example.com/assets/", // CDN（总是 HTTPS 协议）

publicPath: "//cdn.example.com/assets/", // CDN (协议相同)

publicPath: "/assets/", // 相对于服务(server-relative)

publicPath: "assets/", // 相对于 HTML 页面

publicPath: "../assets/", // 相对于 HTML 页面

publicPath: "", // 相对于 HTML 页面（目录相同）

## chunkFilename和filename

可以理解成前面是后面的一部分，单文件只有一个chunk，所以不需要chunkFilename

默认使用 [id].js 或从 [output.filename](http://www.css88.com/doc/webpack2/configuration/output/#output-filename) 中推断出的值（[name] 会被预先替换为 [id] 或 [id].）

[id] 被 chunk 的 id 替换。

[name] 被 chunk 的 name 替换（或者，在 chunk 没有 name 时使用 id 替换）。

[hash] 被 compilation 生命周期的 hash 替换。

[chunkhash] 被 chunk 的 hash 替换。

## crossOriginLoading

只用于 [target](http://www.css88.com/doc/webpack2/configuration/target) 是 web，使用了通过 script 标签的 JSONP 来按需加载 chunk。

启用 [cross-origin 属性](https://developer.mozilla.org/en/docs/Web/HTML/Element/script#attr-crossorigin) 加载 chunk。以下是可接收的值……

crossOriginLoading: false - 禁用跨域加载（默认）

crossOriginLoading: "anonymous" - **不带凭据(credential)**启用跨域加载

crossOriginLoading: "use-credentials" - **带凭据(credential)**启用跨域加载 **with credentials**

## sourceMapFilename

javascript 的sourceMap文件的文件名。它会被放在 output.path 目录下。

[file] 会被javascript的文件名替换掉。

[id] 会被 chunk id替换掉。

[hash] 会被 编译hash替换掉。

Default: “[file].map”

## devtoolModuleFilenameTemplate

用于调试时资源映射。默认只是js文件名（可能重名），可以把路径放进字符串里。

devtoolModuleFilenameTemplate: info =>

path.resolve(info.absoluteResourcePath).replace(/\\/g, '/'),

# 模块

在v2中，旧的 **loader 配置被更强大的 rules 系统取代**，后者允许配置 loader 以及其他更多项。为了兼容旧版，module.loaders 语法被保留，旧的属性名依然可以被解析。新的命名约定更易于理解并且是升级配置使用 module.rules 的好理由。

## loaders/rules

### loaders

每一个都可以有以下几个属性。   
\* test :一个必须满足的条件   
\* exclude : 一个排除的条件   
\* include : 要用Loader转换的导入文件的路径数组。   
\* loader : 一个用“！”隔开 loader的字符串。  
\* loaders : 一个loader字符串的数组。

**这5个在v2里面都有变化！！！**

### rules

rules数组里面的每一个项被称为rule，但没有rule的写法。

module: {

**- loaders: [**

**+ rules: [**

{

test: /\.css$/,

**- loaders: [**

**+ use: [**

{

loader: "style-loader"

},

{

loader: "css-loader",

**- query: {**

**+ options: {**

modules: true

}

}

]

},

{

test: /\.jsx$/,

**loader: "babel-loader", // Do not use "use" here,可以使用use: [ { loader } ],也可以用user:['babel-loader']**

options: {

// ...

}

}

]

}

### Rule 条件

条件有两种输入值：

resource：请求文件的绝对路径。它已经根据 resolve 规则解析。

issuer: 被请求资源(requested the resource)的模块文件的绝对路径。是导入时的位置。

例如: 从 app.js 导入 "./style.css"，resource 是 /path/to/style.css. issuer 是 /path/to/app.js。

在规则中，属性 test, include, exclude 和 resource 对 **resource** 匹配（在resource中），并且属性 issuer 对 **issuer** 匹配。

当使用多个条件时，所有条件都匹配。

### Rule 结果

规则结果只在规则条件匹配时使用。

规则有两种输入值：

应用的 loader：应用在 resource 上的 loader 数组。

Parser 选项：用于为模块创建解析器的选项对象。

这些属性会影响 loader：**loader**, **options**, **use**。

也兼容这些属性：**query**, **loaders**。

**enforce** 属性会影响 loader 种类。不论是普通的，前置的，后置的 loader。

**parser** 属性会影响 parser 选项。

**上面几个只有loader和options在use中，其他都在rule中**

### 嵌套的 Rule

可以使用属性 rules 和 oneOf 指定嵌套规则。

这些规则用于在规则条件(rule condition)匹配时进行取值。

### rule.use

应用于模块的 [UseEntries](http://www.css88.com/doc/webpack2/configuration/module/#useentry) 列表。每个入口(entry)指定使用一个 loader。

传递字符串（如：use: [ "style-loader" ]）是 loader 属性的简写方式（如：use: [ { loader: "style-loader "} ]）

module: {

- loaders: {

+ rules: [{

test: /\.less$/,

- loader: "style-loader!css-loader!less-loader"

+ use: [

+ "style-loader",

+ "css-loader",

+ "less-loader"

+ ]

}]

}

### test与rule.resource

一个rule中

v1

test: /\.jsx$/

v2

{resource:{test : /\.jsx$/}}

### rule.query和rule.options

属性为字符串或对象。值可以传递到 loader 中，将其理解为 loader 选项。

v1

{query:}

v2

{option:{query:}}

### Rule.exclude

Rule.exclude 是 Rule.resource.exclude 的简写。

### Rule.include

Rule.include 是 Rule.resource.include 的简写。

### rule.loaders

Rule.loaders 是 Rule.use 的别名。

### rule.loader和rule.use

Rule.loader 是 Rule.use: [ { loader } ] 的简写。

### rule.enforce

设置前置或者后置loader，所以preLoader和postLoader被取消了

module: {

- preLoaders: [

+ rules: [

{

test: /\.js$/,

+ enforce: "pre",

loader: "eslint-loader"

}

]

}

### UseEntry

use数组里面的每一项对应一个UseEntry

{

loader: "css-loader",//必须。属性是字符串。它使用 loader 解析选项（resolveLoader），相对于配置中的 context 来解析。

options: {//可选

modules: true

}

}

# loader

## babel-loader

见babel文档

## vue-loader

将vue分成三个语法块（template语言块,script,style）。

vue-loader会解析这个文件中的每个语言块，然后传输到其它的loaders，最终输出到module.exports是vue组件的配置对象的CommonJS模块。

### template

默认语言：html

每个\*.vue最多包含一个块

块中的内容作为字符串提取出来

### script

{ ComponentA }是{ ComponentA: ComponentA }的简写。vue将会自动把键转换为component-a，是以我们可以在中引入组件。(所以可以引入后直接在template中使用component-a)

### style

vue-loader通过指定语言块的lang属性支持css预编译、**html编译模板**等语言格式。如在组件的style块中使用sass。lang="sass",值lang="stylus",

当一个style标签带有scoped属性，它的css只应用于当前组件的元素。<style scoped>（编译时会添加属性混淆 如.example[\_v-f3f3eg9]），

同时包含局部和全局样式

<style>/\* global styles \*/</style>

<style scoped>/\* local styles \*/</style>

子组件的根节点会受到父组件和本组件的局部css样式影响

在包含迭代组件的组件中小心使用子孙选择器。一条关于.a .b的css规则，如果在类名为a的标签中使用了子组件，那么子组件中的类名为b的标签也会应用这条规则。

打开CSS Module模式，class对象会作为$style的属性注入到组件中，进而在中进行动态的类绑定，如$style.red，也可以在js中用, this.$style.red

可以包含多个

<style module="a">/\* identifiers injected as $a \*/</style>

<style module="b">/\* identifiers injected as $b \*/</style>

## css-loader

理论上顺序应该是：less-loader css-loader style-loader。但实际不按这个顺序写好像也没问题。

webpack默认是不会单独抽离css文件的。会在js脚本里通过style标签引入(需要借助style-loader)

## url-loader

升级版的file-loader，一般用来处理图片，可以把图片转成base64，打包进文件里，减少网络请求数量。

options: {

limit: 4000,

name: 'static3/media/[name].[hash:8].[ext]',

fallback: 'responsive-loader',

quality: 85

mimetype: 'image/png' //转换后缀

}

# 解析resolve

值是一个对象

## alias

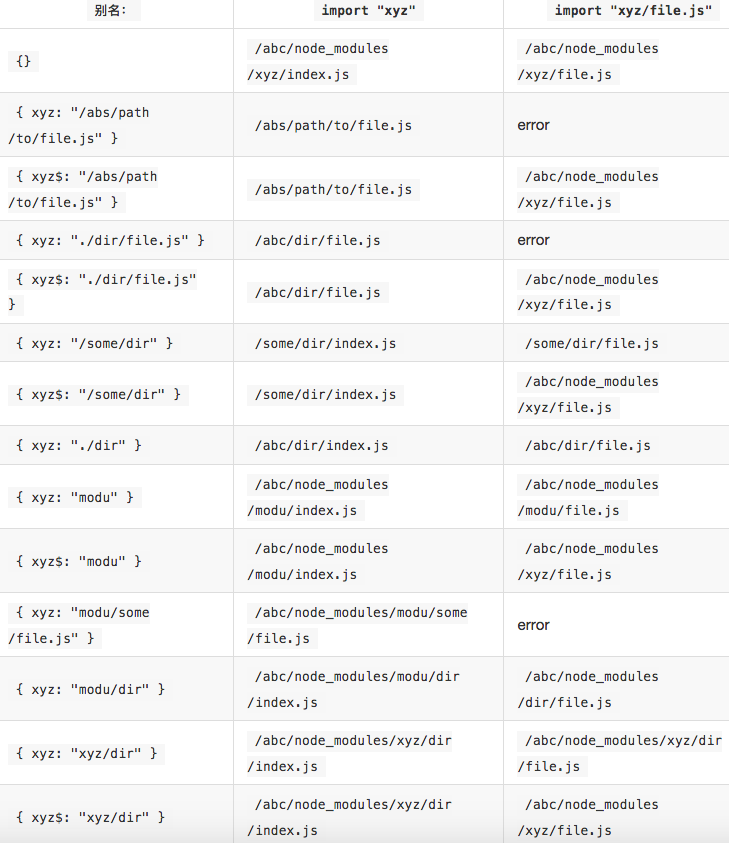
创建 import 或 require 的别名，方便引用。

健末尾加$表示精确匹配

alias: {

xyz$: path.resolve(\_\_dirname, 'path/to/file.js')

}



如果在 package.json 中定义，index.js 可能会被解析为另一个文件。

/abc/node\_modules 也可能在 /node\_modules 中解析。

### root

包含你的模块的目录（绝对路径）。也可能是目录的数组。需要将单个目录添加到搜索路径的情况下，才用这个设置。

var path = require('path');

// ...

resolve: {

root: [

path.resolve('./app/modules'),

path.resolve('./vendor/modules')

]

}

### modules

区分绝对路径和相对路径

### plugins

为什么这里也有个插件？

### modulesDirectories

一个目录数组。这个目录将解析给当前目录以及它的祖先目录，在这里查找模块。它的功能类似于 node 的 node\_modules目录。例如，如果把它设置为["mydir"],webpack 会查找“./mydir”,”../mydir”,”../../mydir”等。

默认值为：["web\_modules", "node\_modules"]

注意：在这里，不必要传入像"../someDir", "app", "." 或者绝对路径之类的值。只需要用一个目录名，不要用路径。当你的层次结构中有 这个文件夹时再用这个，要不然你最好是用 resolve.root 选项。

### fallback

一个目录(或者目录绝对目录的数组)。如果webpack 在 resolve.root 或者 resolve.modulesDirectories 实在找不到某个模块了，会去这个（些）目录中找。

### extensions

自动解析确定的扩展。默认值为：

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

能够使用户在引入模块时不带扩展：

import File from '../path/to/file'

使用此选项，会**覆盖默认数组**，这就意味着 webpack 将不再尝试使用默认扩展来解析模块。对于使用其扩展导入的模块，例如，import SomeFile from "./somefile.ext"，要想正确的解析，一个包含“\*”的字符串必须包含在数组中。

# 插件

plugins: [

new webpack.optimize.UglifyJsPlugin({

compress: {

warnings: false,

},

output: {

comments: false,

},

}),

## ExtractTextPlugin

该插件会提取entry chunk中所有的require('\*.css')，分离出独立的css文件。

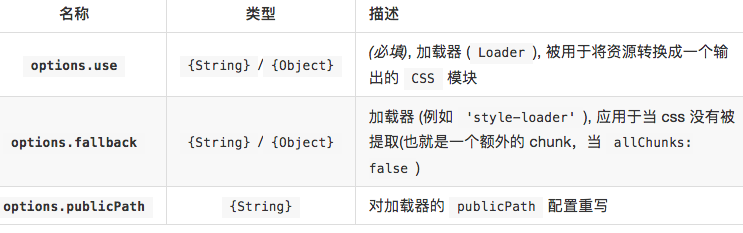
new ExtractTextPlugin([id: string], filename: string, [options])

一个entry生成一个文件，当多个entry的时候，可以用[name]/[id]/[contenthash]来生成多个文件  
**id** 插件实例的唯一标识，自动生成  
**filename** 输出的filename，可以通过[name]/[id]/[contenthash]自定义filename  
**options.allChunks** 提取所有的chunk（默认只提取initial chunk）  
**options.disable** disable该插件

**ExtractTextPlugin.extract**([notExtractLoader], loader, [options])

该方法将已经存在的loader转换成一个提取loader  
notExtractLoader 不提取css时需要使用的loader  
**loader** 用来将资源文件转换为css输出模块的loader  
options.publicPath 覆盖loader的publicPath设置

新版：



const ExtractTextPlugin = require("extract-text-webpack-plugin");

module.exports = {

module: {

rules: [

{

test: /\.css$/,

use: ExtractTextPlugin.extract({

fallback: "style-loader",

**use: "css-loader"**

})

}

]

},

plugins: [

**new ExtractTextPlugin("styles.css"),**

]

}

复杂情况

plugins:[

new ExtractTextPlugin({

+ filename: "bundle.css",

+ disable: false,

+ allChunks: true

+ }),

]

实例：

（这是旧的写法）

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

// multiple extract instances

let extractCSS = new ExtractTextPlugin('stylesheets/**[name].css**');

let extractLESS = new ExtractTextPlugin('stylesheets/[name].less');

module.exports = {

...

module: {

loaders: [

{ test: /\.scss$/i, loader: **extractCSS**.extract(['css', 'sass']) },

{ test: /\.less$/i, loader: extractLESS.extract(['css', 'less']) },

...

]

},

plugins: [

**extractCSS,**

extractLESS

]

};

## loader-options-plugin

?> 需要 Review

loader-options-plugin 和其他插件不同。它的用途是帮助人们从 webpack 1 迁移至 webpack 2。在 webpack 2 中对 webpack.config.js 的结构要求变得更加严格；**不再开放扩展给其他的加载器/插件**。**webpack 2 推荐的使用方式是直接传递 options 给加载器/插件。换句话说，配置选项将不是全局/共享的。**

不过，在某个加载器升级为依靠直接传递给它的配置选项运行之前，可以使用 loader-options-plugin 来抹平差异。你可以通过这个插件配置全局/共享的加载器配置，使所有的加载器都能收到这些配置。

在将来这个插件可能会被移除。

ew webpack.LoaderOptionsPlugin(options)

* options.debug (boolean)：加载器是否为 debug 模式。debug 在 webpack 3 中将被移除。
* options.minimize (boolean)：加载器是否要切换到优化模式。
* options.options (object)：一个配置对象，用来配置旧的加载器 - 将使用和 webpack.config.js 相同的结构。**(自定义属性写在这里)**
* options.options.context (string)：配置加载器时使用的上下文。
* 其他选项和在 webpack.config.js 中一样……

plugins: {  
new webpack.LoaderOptionsPlugin({  
**options: {  
postcss: function () {  
return [precss, autoprefixer];  
},**devServer: {  
contentBase: "./public", //本地服务器所加载的页面所在的目录  
colors: true, //终端中输出结果为彩色  
historyApiFallback: true, //不跳转  
inline: true //实时刷新  
}  
}  
})

## DefinePlugin

允许你创建一个在编译时可以配置的全局常量。这可能会对开发模式和发布模式的构建允许不同的行为非常有用。比如,你可能会用一个全局的常量来决定 log 在开发模式触发而不是发布模式。

new webpack.DefinePlugin({

PRODUCTION: JSON.stringify(true),

VERSION: JSON.stringify("5fa3b9"),

BROWSER\_SUPPORTS\_HTML5: true,

TWO: "1+1",

"typeof window": JSON.stringify("object")

})

## UglifyJsPlugin

UglifyJsPlugin 的 sourceMap 配置项现在默认为 false 而不是 true。

UglifyJsPlugin 的 compress.warnings 配置项现在默认为 false 而不是 true。

UglifyJsPlugin 不再压缩 loaders。在未来很长一段时间里，需要通过设置 minimize:true 来压缩 loaders。

aders 的压缩模式将在 webpack 3 或更高的版本中被取消。

## DedupePlugin

不再需要 webpack.optimize.DedupePlugin。请从配置中移除。

有些JS库有自己的依赖树，并且这些库可能有交叉的依赖，DedupePlugin可以找出他们并删除重复的依赖。

new **webpack.optimize.DedupePlugin**()

## NoErrorsPlugin

跳过编译时出错的代码并记录，使编译后运行时的包不会发生错误。

new **webpack.NoErrorsPlugin**()

## ProvidePlugin

definitions 定义标识符，当遇到指定标识符的时候，自动加载模块。

new **webpack.ProvidePlugin**({

$: "jquery"

})

// in a module

$("#item") // <= just works

// $ is automatically set to the exports of module "jquery"

## CommonsChunkPlugin

提取代码中的公共模块，然后将公共模块打包到一个独立的文件中，以便在其他的入口和模块中使用。别忘了在html中单独引入抽离出来的公共模块。

https://segmentfault.com/a/1190000012828879

new webpack.optimize.CommonsChunkPlugin(options)。

**配置：**

{

name: string, // or

names: string[],

// 这是 common chunk 的名称。已经存在的 chunk 可以通过传入一个已存在的 chunk 名称而被选择。

// 如果一个字符串数组被传入，这相当于插件针对每个 chunk 名被多次调用

// 如果该选项被忽略，同时 `options.async` 或者 `options.children` 被设置，所有的 chunk 都会被使用，否则 `options.filename` 会用于作为 chunk 名。

filename: string,

// common chunk 的文件名模板。可以包含与 `output.filename` 相同的占位符。

// 如果被忽略，原本的文件名不会被修改(通常是 `output.filename` 或者 `output.chunkFilename`)

minChunks: number|Infinity|function(module, count) -> boolean,

// 在传入 公共chunk(commons chunk) 之前所需要包含的最少数量的 chunks 。

// 数量必须大于等于2，或者少于等于 chunks的数量

// 传入 `Infinity` 会马上生成 公共chunk，但里面没有模块。

// **你可以传入一个 `function` ，以添加定制的逻辑（默认是 chunk 的数量）**

**chunks: string[],**

// 通过 chunk name 去选择 chunks 的来源。chunk 必须是 公共chunk 的子模块。

// 如果被忽略，所有的，所有的 入口chunk (entry chunk) 都会被选择。

children: boolean,

// 如果设置为 `true`，所有 公共chunk 的子模块都会被选择

async: boolean|string,

// 如果设置为 `true`，一个异步的 公共chunk 会作为 `options.name` 的子模块，和 `options.chunks` 的兄弟模块被创建。

// 它会与 `options.chunks` 并行被加载。可以通过提供想要的字符串，而不是 `true` 来对输出的文件进行更换名称。

minSize: number,

// 在 公共chunk 被创建立之前，所有 公共模块 (common module) 的最少大小。

}

## HtmlWebpackPlugin

这个插件将为您**生成一个HTML5文件**，该文件的 body 中使用script 标记引用了所有 webpack bundle。只需如下添加插件到您的webpack配置中：

这对于被更改文件的文件名中包含每次编译哈希(hash) 的webpack bundle尤其有用。

您可以让插件为您生成一个HTML文件，使用lodash templates提供您自己的模板，或使用自己的加载器(loader)。

如果你有任何**CSS资源在webpack的 output 中**，（例如，使用 [ExtractTextPlugin](http://www.css88.com/doc/webpack2/plugins/extract-text-webpack-plugin)提取的CSS）那么这些资源将被包含在HTML head 中，使用<link>标记引用。

例子1

new HtmlWebpackPlugin(), // Generates default index.html

new HtmlWebpackPlugin({ // Also generate a test.html

filename: 'test.html',

template: 'src/assets/test.html'

})

例子2

new HtmlWebpackPlugin({

title: 'Custom template',

template: 'my-index.ejs', // Load a custom template (ejs by default see the FAQ for details)

})

例子3

new HtmlWebpackPlugin({

template: 'index.html',

minify: {

removeComments: true,

collapseWhitespace: true,

removeRedundantAttributes: true,

useShortDoctype: true,

removeEmptyAttributes: true,

removeStyleLinkTypeAttributes: true,

keepClosingSlash: true,

minifyJS: true,

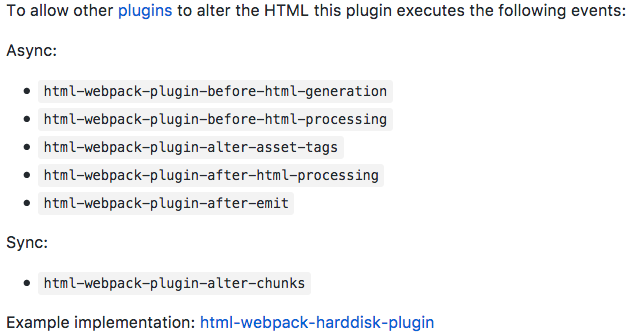
minifyCSS: true,

minifyURLs: true

}

}),

**该插件支持插件**



传入的template路径是相对工作目录而言，即使配置文件在子目录里

## CopyWebpackPlugin

## ExtractTextPlugin

https://www.jianshu.com/p/ed3c98ba5474

ExtractTextPlugin . extract(loader)

把css抽离成单独的文件。

filename，allChunks

publicPath，use，fallback。

一般把style-loader放在callback里面，用来在编译后提取css。

# 配置对象

## context

基础目录，**绝对路径**，用于从配置中解析入口起点(entry point)和加载器(loader)

context: path.resolve(\_\_dirname, "app")

默认使用当前目录，但是推荐在配置中传递一个值。这使得你的配置独立于 CWD(current working directory)。

dev

# webpack-dev-server

首先安装webpack-dev-server

[?](http://www.jb51.net/article/110367.htm)

|  |  |
| --- | --- |
| 1 | npm install --save-dev webpack-dev-server |

然后修改package.json配置文件中：

[?](http://www.jb51.net/article/110367.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | "scripts": {   "start": "webpack-dev-server",   "build": "webpack"   } |

这里如果使用webpack-dev-server 命令来启动就必须全局安装 devServer:

[?](http://www.jb51.net/article/110367.htm)

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

在webpack的配置文件中可以对devServer进行配置

[?](http://www.jb51.net/article/110367.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // 配置devServer各种参数   devServer: {   contentBase: "./", // 本地服务器所加载的页面所在的目录   historyApiFallback: true, // 不跳转   inline: true // 实时刷新   } |

此时可以监听入口文件的改变，实时刷新页面，然而非入口文件的改变则不会被监听到，需要手动进行刷新。并且目标文件不包括index.html。这里使用html-webpack-plugin插件。

[?](http://www.jb51.net/article/110367.htm)

|  |  |
| --- | --- |
| 1 | npm install html-webpack-plugin --save-dev |

修改webpack配置文件，添加以下配置：

[?](http://www.jb51.net/article/110367.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | ...  const HtmlWebpackPlugin = require('html-webpack-plugin');    module.exports = {   ....,   plugins: [   new HtmlWebpackPlugin({    template: './index.html' // 模版文件   })   ]  } |

# 常见情况

## 默认搜索

Webpack 在执行的时候，除了在命令行传入参数，还可以通过指定的配置文件来执行。默认情况下，会搜索当前目录的 webpack.config.js 文件，这个文件是一个 node.js 模块，返回一个 json 格式的配置信息对象，或者通过 --config 选项来指定配置文件。

## 打印错误信息

$ webpack --display-error-details

## 路径兼容windows

Webpack 中涉及路径配置最好使用绝对路径，建议通过 **path.resolve(\_\_dirname, "app/folder")** 或 **path.join(\_\_dirname, "app", "folder")** 的方式来配置，以兼容 Windows 环境。

## Loaders

多个时记得用loaders，不是loader

cssLoaders = ['style-loader', 'css-loader', 'postcss-loader'];

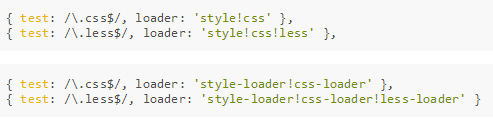
{ test: /\.css$/**,loaders**:cssLoaders},

{test:[/\.js$/,/\.jsx?$/],exclude:/(node\_modules)/,**loader**:'react-hot!babel'},

### 省略写法

### -loader 可以省略

v2以后不能省略。



### css-loader和style-loader

我们需要在js文件里，通过require的方式来引入css，我们来看具体的方法，首先需要安装css-loader，   
style-loader(安装style-loader的目的是为了在html中以style的方式嵌入css)。

## 权限问题

不能写入文件，不能创建文件夹。

Sudo

## Npm

### 镜像

cnpm方式

npm install -g cnpm --registry=https://registry.npm.taobao.org