### [webpack入门（三）——webpack 配置](http://blog.csdn.net/keliyxyz/article/details/51527476)

喂一个配置对象给webpack ，它就可以干活儿了。根据你用webpack的用法，有两种途径传入这个对象：

### CLI( 命令行)

如果你用命令行，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。这个文件应该导出一个配置对象，如下：

module.exports = {

// configuration

};

### node.js API

如果 你用node.js API，你需要把配置文件作为一个参数，传给webpack。如下：

webpack({

// configuration

}, callback);

### 混合配置

在两种情况下，你都可以用一个配置对象的数组，它们是并行进程的。它们共用文件系统的缓存和监控，所以这是一种比多次调用webpack更高效的办法。

## config 对象内容

***提示***：记住你不需要写纯粹的JSON到配置文件中。javascript随便用。webpack只是一个node.js 模块…

一个最简单的配置对象如下：

{

context: \_\_dirname + "/app",

entry: "./entry",

output: {

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

filename: "bundle.js"

}

}

下面开始解释这些配置项了。

#### context

这是***entry***配置项的根目录（绝对路径）。如果***output.pathinfo***也设置了，它的pathinfo是基于这个根目录。

#### entry

包的入口。   
如果你传入一个字符串：这个字符串作为主模块的启动点。（本句翻译待考）   
如果你传入一个数组，数组中所有模块都会启动，但最后一个会被导出。

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

如果你传入一个对象：会创建多个入口包。key就是 块（chunk）名字。value就是一个字符串或者一个数组。

{

entry: {

page1: "./page1",

page2: ["./entry1", "./entry2"]

},

output: {

// Make sure to use [name] or [id] in output.filename

// when using multiple entry points

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

chunkFilename: "[id].bundle.js"

}

}

****注意****：你不可能在其它配置项中设置入口点。如果你需要特殊配置入口点，你需要用到上面提到的混合配置。

#### output

影响编译输出的配置项。 ***output***配置项告诉webpack怎么把编译后的文件写入磁盘。注意，如果用多入口配置，只会一个能被指定output。

如果你用了hshing([hash]或者[chunkhash])，请先确保有一个指定的模块排序。用***OccurrenceOrderPlugin*** 或者 ***recordsPath***。

#### output.filename

指定每一个输出文件的存盘文件名。这里你不必指定绝对路径。***output.path***才是用来指定文件路径的配置项。***filename*** 是单独给文件命名的。

单入口

{

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

output: {

filename: 'bundle.js',

path: './built'

}

}

// writes to disk: ./built/bundle.js

多入口   
如果你的配置创建了多个“块”（多入口点或者用了像***CommonsChunkPlugin***），你应该换成下面这样的配置来确保每一个文件都有一个特有的文件名。   
***[name]*** 会被块名替换掉。   
***[hash]***会被编译的hash替换掉。   
***[chunkhash]***会被块的hash替换掉。

{

entry: {

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

search: './src/search.js'

},

output: {

filename: '[name].js',

path: \_\_dirname + '/built'

}

}

// writes to disk: ./built/app.js, ./built/search.js

#### output.path

输出目录，必须是绝对路径。   
***[hash]***会被编译hash替换掉。

#### output.publicPath

当浏览器需要引用输入出文件时， 这个配置项指定输入文件的公共URL地址。在loader中它被嵌入到script 或者 link 标签或者对静态资源的引用里。当文件的href 或者 url()与它在磁盘 上的路径 不一致时publicPath ，就应当用·publicPath (像path一样指定) ，这在你想定义把一些或者所有文件放在不同的主机或CDN上时会非常有用。webpack dev server 也是用publicpath决定输出文件从哪里公开。和 path 一样，你可以用 [hash] 替换缓存文件。   
****config.js****

output: {

path: "/home/proj/public/assets",

publicPath: "/assets/"}

****index.html****

<head>

<link href="/assets/spinner.gif"/></head>

一个更加复杂的例子，使用CDN和hash 资源。

****config.js****

output: {

path: "/home/proj/cdn/assets/[hash]",

publicPath: "http://cdn.example.com/assets/[hash]/"}

注意：如果最终的 publicPath 不能确定在编译时不能确定，你可以留白在运行时在入口文件中动态设置。如果在编译时不知道publicPath你可以忽略它并设置在入口文件中设置 \_\_webpack\_public\_path\_\_。

\_\_webpack\_public\_path\_\_ = myRuntimePublicPath

// rest of your application entry

#### output.chunkFilename

它是 output.path 目录中作为相对路径的非入口chunk的文件名 。   
[id] 会被chunk的id替换掉。   
[name] 会被chunk的name替换掉（如果没有名字，会被id替换）。   
[hash] 会被编译的hash替换掉。   
[chunkhash] 会被 chunk hash替换掉。

#### output.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”***

#### output.devtoolModuleFilenameTemplate

在 sourceMap中生成source array是用函数来做的。这个配置项就是这个函数所在的文件名模板字符。   
[resource] 会被webapck 用来解析文件的路径替换掉，如果有Loader也包括loader最右边的query参数。   
[resource-path]] 跟 resource配置项一样，只是不会带query参数。   
[loaders] loader列表，带最右边的参数（明确的Loader）。   
[[all-loaders] loader列表，带最右边的参数（包括自然生效的loader）。   
[id] 会被 模块id替换掉。   
[hash] 会被 模块唯一标识符替换。   
[hash] 会被 文件的绝对路径文件名替换。

也可以定义为一个函数而不是 字符串模板。这个函数会接收 info 对象，这个Info对象曝露了一下属性：   
1. identifier   
2. shortIdentifier   
3. resource   
4. resourcePath   
5. absoluteResourcePath   
6. allLoaders   
7. query   
8. moduleId   
9. hash

#### output.devtoolFallbackModuleFilenameTemplate

跟 output.devtoolModuleFilenameTemplate 很像。但是是用在混合模块儿标识中。

Default: “webpack:///[resourcePath]?[hash]”

#### output.devtoolLineToLine

能给所有或者指定模块设置为行到行的map模式。行到行map模式用一个简单的 sourcecMap , 在这个sourceMap 中每行生成的文件映射到同一行的源文件。这是一个性能优化。只有当你需要更好的性能或者你确定输入的行和生成 的行匹配，你再这么做。   
true 使它对所有module有效（不推荐）。   
一个对象{test，include,exclude} 同module.loaders 很像，对指定的文件设置有效。

Default:disabled

#### output.hotUpdateChunkFilename

热更新的Chunk 文件名。他们在 output.path 目录中。   
[id] 会被chunk的id替换掉。   
[hash] 会被编译的hash替换掉。（最后一次的hash会被存储在记录中）

Default: “[hash].hot-update.json”

#### output.jsonpFunction

webpack用来异步加载chunk的JSONP 函数。   
一个小函数或许可以减少一点文件的大小。 当一个单页面上有多个webpack实例时，使用不同的标识符。

Default: “webpackJsonp”

#### output.hotUpdateFunction

webpack用来异步加载热替换chunk的JSONP 函数。

Default: “webpackHotUpdate”

#### output.pathinfo

如果设置了，将包导出为库。 output.library 就是库的名字。   
如果你写了一个单库，并且想把它以单文件形式发布的话，可以用它。

#### output.libraryTarget

格式化导出库。   
"var" 通过设置一个变量导出：var Library =xxx (默认)

"this" 通过设置一个this的属性导出：var Library =xxx

"commonjs" 通过设置一个exports 的一个属性导出: exports["Library"] = xxx

"commonjs2" 通过设置一个module.exports 导出: module.exports=xxx

"amd" 导出到AMD(随便命名 -通过library选项来设置 名称)

"umd" 导出到AMD，CommonJS2或者作为root的一个属性。

Default: “var”   
如果 output.library 没设置，但是output.library 设置了除var以外的值，exports对象的每一个属性都 会被复制（除了amd,commonjs2和umd）

#### output.umdNamedDefine

如果 output.libraryTarget 设置为umd 而且output.library 也设置了。这个设为true，将为AMD模块命名。

#### output.sourcePrefix

给bundle资源的每一行前面加上这个字符串。

Default: “\t”

#### output.crossOriginLoading

这一项的设置可以允许跨域加载 chunks。   
可能的值为：   
false 不允许跨域加载。   
anonymous 启用跨域加载。当用anonymous 时，请求中没有安全证书发送。   
use-credentials 启用跨域加载。请求中安全证书会发送。

Default: false

#### module

影响正常模块的选项

#### module.loaders

一个自动运用的 loader数组。   
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。

一个条件可以 是正则表达式(tested against absolute path)，或者是一个包含绝对路径的字符串，或者一个函数function(absPath): bool ，或者一个数组 用“and”连接这些的数组。

重要：这里Loader的resolved相对于它们应用的的资源的路径。这意味着它们不是相对 配置文件的路径 。如果你已经用npm 安装过loader而且 你的 node\_ｍoudles文件夹不在所有源文件的父文件夹中，webpack会找不到 loader.你需要添加 node-modules文件夹,作为 resolveLoader.root 选项的绝对路径。   
例如：

module: {

loaders: [

{

// "test" is commonly used to match the file extension

test: /\.jsx$/,

// "include" is commonly used to match the directories

include: [

path.resolve(\_\_dirname, "app/src"),

path.resolve(\_\_dirname, "app/test")

],

// "exclude" should be used to exclude exceptions

// try to prefer "include" when possible

// the "loader"

loader: "babel-loader"

}

]

}

#### module.preLoaders, module.postLoaders

语法跟 module.loaders 一样。   
一个在运用loader之前（后）的loader数组。

#### resolve

影响模块的解决方案。

#### resolve.alias

用其它模块或者路径替换一个模块。   
预期是一个对象，这个对象的key是模块的名称，value是一个新的path。它类似于一个替换，但更聪明一点。   
如果key 以结尾,只有精确的部分（不包括）会被替换掉。   
如果value 是一个相对路径，模块将与包含它的文件的路径有关。

| **alias** | **require(“xyz”)** | **require(“xyz/file.js”)** |
| --- | --- | --- |
| {} | /abc/node\_modules/xyz/index.js | /abc/node\_modules/xyz/file.js |
| { xyz: “/absolute/path/to/file.js” } | /absolute/path/to/file.js | /abc/node\_modules/xyz/file.js |
| { xyz$: “/absolute/path/to/file.js” } | /absolute/path/to/file.js | error |
| { xyz: “./dir/file.js” } | /abc/dir/file.js | /abc/node\_modules/xyz/file.js |
| { xyz$: “./dir/file.js” } | /abc/dir/file.js | error |
| { xyz: “/some/dir” } | /some/dir/index.js | /some/dir/file.js |

如果在package.json中有定义入口文件，index.js或许会被查到到其它文件。

#### resolve.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。

注意： 它****必须****是绝对路径，请不要传像./app/modules 这样的相对路径。   
例：

var path = require('path');

// ...

resolve: {

root: [

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

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

]

}

#### resolve.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。

#### resolve.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。

#### resolve.extensions

一个包含模块扩展名的数组。例如，为了发现CoffeeScript 文件，你的数组应该包含字符串".coffee"。

Default: ["", ".webpack.js", ".web.js", ".js"]   
****注意****：设置了这个选项，会取代默认的模块扩展名。重要的事情说三遍：意味着webpack不再用默认扩展名查找模块，意味着webpack不再用默认扩展名查找模块，意味着webpack不再用默认扩展名查找模块。如果你想正确加载一个带有扩展名的模块，你必须把一个空字符串放在你的数组里。如果你想不要扩展名来加载一个js文件，你需要将“.js”加入你的数组。

#### resolve.packageMains

在package.json中查找符合这些字段的文件

Default: [“webpack”, “browser”, “web”, “browserify”, [“jam”, “main”], “main”]

#### resolve.packageAlias

在package.json中查询对象里的字段，键值对是按照这个规范的别名来进行的

Not set by default

比如”browser”会检查browser字段

#### resolve.unsafeCache

启用不安全的缓存来解析一部分文件。改变缓存路径也许会导致出错（罕见情况下）。 一个正则表达式数组里，只有一个正则或只有一个为true（对应全部文件）是最好的实践 。如果解析路径匹配，就会被缓存。

Default: []

#### resolveLoader

跟 resolve很像，但是是为loaders准备的。

// Default:

{

modulesDirectories: ["web\_loaders", "web\_modules", "node\_loaders", "node\_modules"],

extensions: ["", ".webpack-loader.js", ".web-loader.js", ".loader.js", ".js"],

packageMains: ["webpackLoader", "webLoader", "loader", "main"]

}

注意，你可以用alias，其他特性和resolve相似。例如 设置了alias中的{txt: ‘raw-loader’｝， txt!templates/demo.txt 结果会用 raw-loader解析。

#### resolveLoader.moduleTemplates

这是resolveLoader 唯一的属性。它描述了尝试的模块名称的替代名

Default: [“***-webpack-loader”, “***-web-loader”, “***-loader”, “***“]

#### externals

指定的依赖不会被webpack解析，但会成为bundle里的依赖。output.libraryTarget.决定着依赖的类型。值是对象，字符串，函数，正则，数组都会被接受。

字符串：一个精确匹配的依赖会变成externals依赖，同一字符串会被用于externals依赖。   
对象：如果依赖精确匹配到了对象的一个属性，属性值就会被当作externals依赖。属性值可以包含一个依赖型的前缀，用一个空格隔开。如果属性值为true，则使用该属性名。如果属性值为false，外部测试失败，这个依赖是内部依赖。见下面的例子。

函数：function(context, request, callback(err, result))。函数会在每个依赖中调用。如果结果被传递到回调函数里，这个值就会被像处理对象属性值那样处理。

正则表达式：每个被匹配的依赖都会成为外部依赖。匹配的文本会被用作外部依赖的请求。因为请求是用于生成外部代码钩子的确切代码，如果你匹配到一个cmd的包(比如 ‘../some/package.js’),相反使用外部function的策略。你可以通过callback(null, “require(‘” + request + “’)”引入包，这个包生成module.exports = require(‘../some/package.js’);使用要求在webpack上下文外。

数组：这个表的多个值(递归)

{

output: { libraryTarget: "commonjs" },

externals: [

{

a: false, // a is not external

b: true, // b is external (require("b"))

"./c": "c", // "./c" is external (require("c"))

"./d": "var d" // "./d" is external (d)

},

// Every non-relative module is external

// abc -> require("abc")

/^[a-z\-0-9]+$/,

function(context, request, callback) {

// Every module prefixed with "global-" becomes external

// "global-abc" -> abc

if(/^global-/.test(request))

return callback(null, "var " + request.substr(7));

callback();

},

"./e" // "./e" is external (require("./e"))

]

}

| **type** | **value** | **resulting import code** |
| --- | --- | --- |
| “var” | “abc” | module.exports = abc; |
| “var” | “abc.def” | module.exports = abc.def; |
| “this” | “abc” | (function() { module.exports = this[“abc”]; }()); |
| “this” | [“abc”, “def”] | (function() { module.exports = this[“abc”][“def”]; }()); |
| “commonjs” | “abc” | module.exports = reqaluire(“abc”); |
| “commonjs” | [“abc”, “def”] | module.exports = require(“abc”).def; |
| “amd” | “abc” | define([“abc”], function(X) { module.exports = X; }) |
| “umd” | “abc” | everything above |

如果没有作为amd/umd的目标解析，将会用一个external 值强制执行amd或者umd。

注意，如果用umd你可以指定一个对象的额外值，属性为 commonjs, commonjs2, amd和root会被设置不同的值。

#### target

* “web” 在浏览器中使用的编译环境（默认值）
* “webworker” 被作为webworker编译
* “node” 在nodejs环境下编译(用require加载chunks)
* “async-node” 在nodejs环境下编译(用fs和vm异步加载chunks)
* “node-webkit” 在webkit下使用jsonp加载chunk，也支持在node中加入require(“nw.gui”) （实验性质）
* “electron” 为 Electron 编译。
* “electron-renderer” 为 Electron 渲染进程编译。

#### bail

报告第一个错误为硬性错误，不可忽略。

#### profile

定时在每个模块捕捉信息。

提示，用analyze tool让它可视化，–json 或者 stats.toJson()会给你JSON的统计。

#### cache

缓存生成模块和和chunks来提高混合增l量编译时的性能。   
启用watch模式会默认启动它。   
可以传入 false禁用它。   
你可以传递一个对象启用它并且让webpack把传递的对象作为缓存。用这种办法，你可以在混合编译器的编译回调中共享缓存。注意：不要在多配置的回调中共享缓存。

#### debug

把loader的模式切到debug。

#### devtool

选一个开发工具来加快调试

* eval 每个模块都用eval执行
* source-map 触发SourceMap，详情看output.sourceMapFilename
* hidden-source-map 同上，但不会在包中添加引用注释。
* inline-source-map SourceMap被作为dataurl加入到js文件中。
* eval-source-map 每个模块都用eval执行，并且SourceMap被作为dataurl加入到eval中。
* cheap-source-map 没有映射的sourcemap，loaders的sourcemap不会启用。

cheap-module-source-map 没有映射的sourcemap，sourcemap就简单的映射到每一行。

@,#或者#@前缀 将强行显示编译指示。（webpack@1版本中默认用@，webpack@2版本中默认用#，建议用#）   
例如： cheap-module-inline-source-map, cheap-eval-source-map, #@source-map

如果你的模块已经包含了sourcemap，你需要用source-map-loader合并被触发的sourcemap

| **devtool** | **打包速度** | **二次打包速度** | **是否建议生产中使用** | **特点｜** |
| --- | --- | --- | --- | --- |
| eval | +++ | +++ | 否 | 生成代码 |
| cheap-eval-source-map | + | ++ | 否 | 转化代码（一行） |
| cheap-source-map | + | o | 是 | 转化代码（一行） |
| cheap-module-eval-source-map | o | ++ | 否 | 源码（一行） |
| cheap-module-source-map | o | - | 是 | 源码（一行） |
| eval-source-map | - | + | 否 | 源码 |
| source-map | - | - | 是 | 源码 |

例如：

{

devtool: "#inline-source-map"

}// =>//# sourceMappingURL=...

注意，下一个大的版本默认加-d，将会变为cheap-module-eval-source-map。

#### devServer

当这个webpack config 传给webpack-dev-server命令行时，这个选项用来 配置 webpack-dev-server的一些行为。   
例如

{

devServer: {

contentBase: "./build",

}

}

#### node

为不同节点包含polufills或者mocks。

* console: true or false
* global: true or false
* process: true, “mock” or false
* \_\_filename: true (real filename), “mock” (“/index.js”) or false
* \_\_dirname: true (real dirname), “mock” (“/”) or false
* < node buildin >: true, “mock”, “empty” or false

// Default:

{

console: false,

global: true,

process: true,

Buffer: true,

\_\_filename: "mock",

\_\_dirname: "mock",

setImmediate: true

}

#### amd

设置require.amd和 define.amd的值   
例如： amd:{jquery:true} （为旧的 1.x AMD版本的jQuery）

#### loader

在loader上下文中可以自定义。

#### recordsPath, recordsInputPath, recordsOutputPath

从一个JSON文件中存入或者读取 编译器 state。这将导致 模块或chunk的id持久化。   
期望是绝对路径，recordsPath 被用于recordsInputPath 和recordsOutputPath，如果他俩未定义。   
当使用热替换之间的多个调用编译器的时候，这个选项是必须的。

#### plugins

向编译器添加额外的插件。