<https://blog.csdn.net/keliyxyz/article/details/51571386>

# webpack入门（一）——webpack 介绍

翻译 2016年06月03日 01:03:29

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

如今的网站正在演化为web应用程序:   
1. 越来越多的使用JavaScript。   
2. 现代浏览器提供更广泛的接口。   
3. 整页刷新的情况越来越少，甚至更多代码在同一个页面。（SPA）

因此有很多代码在客户端!   
一个体量庞大的代码库需要好好组织。模块系统提供代码库划分成模块的选项。

## 模块系统风格

目前有多个标准定义依赖和输出：   
1. script标签（不要模块系统）   
2. CommonJS   
3. AMD和它的一些变种   
4. ES 6   
5. 其它

### script 标签的样式

下面这种就是不用模块系统，你会怎么去管理你的代码。

<script src="module1.js"></script>

<script src="module2.js"></script>

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

<script src="module3.js"></script>

* 1
* 2
* 3
* 4

模块接口导出到全局对象,即window对象。模块的接口可以访问全局对象的依赖关系   
**常见问题**

全局冲突   
严重依赖加载的顺序   
开发人员必须人工解决模块/库的依赖关系   
大型项目，script一溜下来可以很长,难以管理

### CommonJs: 同步加载

这种风格用同步require 的方法去加载一个依赖并用暴露一个接口。 一个模块可以通过给export对象添加属性或给module.exports设置值 来指定导出。

require("module");

require("../file.js");

exports.doStuff = function() {};

module.exports = someValue;

* 1
* 2
* 3
* 4

服务器端node.js用的就是这种标准。   
**优点**:   
1. 服务器端模块可以重用   
2. 已经有许多模块用这种风格(npm)。生态圈良好   
3. 非常简单和容易使用。   
**劣势**   
1. 阻塞调用不适用网络。网络请求是异步的。   
2. 没有并行加载机制。   
**哪些在用？**   
1. 服务端 -node.js   
2. browserify   
3. modules-webmake -编译到一个包   
4. wreq -客户端

### AMD: 异步加载

其它模块系统（例如 浏览器） 同步加载有困难(CommonJS) 而引入的一个异步版本（和定义模块和输出值的一种方法 ）。

require(["module", "../file"], function(module, file) { /\* ... \*/ });

define("mymodule", ["dep1", "dep2"], function(d1, d2) {

return someExportedValue;

});

* 1
* 2
* 3
* 4

**优点**:   
1. 适合网络的异步请求的风格   
2. 并行加载多个模块。   
**劣势**   
1. 编码费力，更难读和写   
2. 看起来只是权宜之计。   
**哪些在用？**   
1. require.js   
2. curl

### ES6模式

ES6借鉴其它语言给javascript新加了一些语法结构，有import语法。

import "jquery";

export function doStuff() {}

module "localModule" {}

* 1
* 2
* 3

**优点**:   
1. 静态分析很容易。   
2. 不会过时的ES标准 。   
**劣势**   
1. 浏览器支持需要时间。（迟早的事）   
2. 很少有模块用这种风格。生态圈   
**目前没有公开的方案**

开发者应当自己选择适合自己的风格。允许现有的代码和包能正常工作，可以很容易地添加自定义模块风格。

## 传输

模块应该在客户端执行,所以他们必须从服务器传输到浏览器。   
传输模块有两个极端:   
1. 一个一个地传。   
2. 全部打包在一个里传。

两种用法都泛滥，但是两种都太low了。

**一个一个地传**   
优点：只有确实需要的模块才会传输过去。   
缺点：许多请求意味着很多开销。   
缺点：应用程序启动缓慢,因为请求延迟   
**全部一个地传**   
优点：请求的开销更少,更少的延迟   
缺点：很多暂时不需要的模块给传输过去了。

### 分块传输

更灵活的传输可能会更好。大多数情况下在这两种极端之间的折中比较好。   
=>在编译所有模块时：把模块切分成小块儿（chunks）。   
这样允许多个更小、更快的请求。有些模块不是一开始就需要的，含有这些模块的分块在需要的时候可以加载到。这样加快了初始化速度，但是在需要用那些模块时仍然让你去抓更多的代码。

开发者怎么做“切分点”，可以根据情况自由抉择。   
=》一个代码库是可能的哟。

注意：这些观点来自谷歌 GWT.

## 怎么可能只有javascript

为什么一个模块系统只能帮程序猿们解决javascript问题呢？还有许多其他资源需要处理:   
样式表   
图片   
web字体   
html模板   
等等

或者 一些预编译和后编译语言   
coffeescript → javascript   
elm → javascript   
less 样式→ css 样式   
jade 模板→ javascript 生成 html   
i18n files → something   
等等

这些东西应该也像下面这样容易：

require("./style.css");

* 1

require("./style.less");

require("./template.jade");

require("./image.png");

* 1
* 2
* 3

## 静态分析

当编译所有这些模块时,一个静态分析试图找到自己的依赖。   
传统上这只能找到简单的东西没有表达 。但是   
require("./template/" + templateName + ".jade") 这样是常见的结构。   
有些库是用一些不一样的风格写的。它们有些很奇怪（不可思议）。

### 策略

聪明的解析办法允许现存代码能跑起来，如果程序猿用了一些怪异的东西，它能试图找到兼容的解决方案。

# 什么是webpack

webpack是一个模块打包器。webpack把模块(s)连同它的依赖一起打包生成包含这些模块的静态资源。

## 为什么另用一个打包器？

现有的模块打包器不适合大项目（大单页面应用）程序。代码分割和静态资源无缝模块化的迫切需求，催生了一个新的模块打包器。   
我试图扩展现有的模块打包器，但是它没能实现所有的目标。   
目标如下：   
1. 把依赖树切分成块，实现按需加载。   
2. 保持低初始加载时间   
3. 每个静态资源都能是一个模块   
4. 具备把第三方库集成为模块的能力   
5. 具备打包器每个部分几乎都能自己定制的特点。   
6. 适合大型项目。

## webpack有什么不同？

### 代码分割

webpack依赖树中有两个依赖类型：同步和异步。异步模块切分成一个新的的块。在块树（chunk tree）优化之后，文件会为每个chunk发文件。

### loader

webpack可以处理javascript本身，但loader用来将其它资源转换为javascript。这样以来，所有资源都被格式化成模块了。

### 智能解析

webpack有一个智能解析器，它能处理几乎所有的第三方库。它甚至允许你在依赖中你像这样加表达式 require("./templates/" + name + ".jade") 。它可以处理最常见的模块化标准风格：CommonJS和AMD。

# 安装

### node.js

安装node.js   
包管理工具 npm会一起装上。

### webapck

webpack 可以用用npm 命令来装

$ npm install webpack -g

* 1

webpack 已经全局安装了，现在 webpack 命令可用了。

### 项目中使用webpack

你的项目最好也有webpack 依赖。 这样你可以在项目中自由决定用webpack哪个版本而必去用全局那个webpack。   
用 npm 命令添加一个 package.json文件

$ npm init

* 1

如果你不发布npm包，Init过程中的问题不重要，都可以忽略。   
安装webpack 并添加到package.json中：

$ npm install webpack --save-dev

* 1

### 版本

有两个webpack版本可用。稳定版本和beta版。beta版 在版本字符中标记为 -beta 。beta版本可能包含脆弱的或者实验功能，都没进行过多少测试。正式场景下应该用稳定版。

$ npm install webpack@1.2.x --save-dev

* 1

### 开发工具

如果你想用开发工具，先安装它

$ npm install webpack-dev-server --save-dev

<https://blog.csdn.net/keliyxyz/article/details/51577905>

# webpack入门（二）——webpack使用

翻译 2016年06月03日 12:19:30

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

# 安装

可以用 npm 安装

npm install webpack -g

* 1

注意： 这里全局安装是出于演示的目的。在真实的项目中，建议安装为你的项目依赖。

# 开始

首先，我们将只使用webpack的命令行界面学习基本webpack 。

## 新建一个 模块化的javascript 项目

cats.js

var cats = ['dave', 'henry', 'martha'];

module.exports = cats;

* 1
* 2

app.js (入口文件)

cats = require('./cats.js');

console.log(cats);

* 1
* 2

入口文件就是你项目启动点。也是webpack追踪模块之间依赖关系的入口点。

## 5秒 打包

给webpack指定一个入口文件（app.js）和输出文件（app.bundle.js）

webpack ./app.js app.bundle.js

* 1

webpack将读取和分析入口点及其依赖(包括传递的依赖）。然后它会把它们都打包到app.bundle.js。   
现在你打好的包可以跑起来了。 运行一下 node app.bundle.js ，你会发现，哇，你有好多猫。

node app.bundle.js

["dave", "henry", "martha"]

* 1
* 2

你也可以在 浏览器端使用打好的包。

## 来点严肃的

webpack是一个非常灵活的模块打包器。它提供了很多高级功能，但并非所有功能都通过命令行界面实现。要获得全部webpack的灵活性，我们需要创建一个 config 文件

### 项目结构

在真实的webpack项目中，我们会把源文件和打包文件用文件夹分开。在这个例子中，我们把源文件放在src中，把打包后的文件放在bin中。   
结构如下

project

└─webpack.config.js

└─src

└─app.js

└─app.js

└─node\_modules

└─bin

└─app.bundle.js

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

大千世界，无奇不有。有不少不一样的结构。有些项目用app而不是src，用些用dist 或者 build而不是bin.项目测试学用 test,tests,spec,specs或者把测试文件放在源文件夹一起。

1.创建 bin 和 src 文件夹。

mkdir bin

mkdir src

* 1
* 2

2.原始源文件移动到src文件夹

mv app.js cats.js src

* 1

3.初始化一个新项目。

npm init # (answer the questions)

* 1

4.安装webpack为你的项目开发依赖。这将声明你的项目适用webpack版本。

npm install --save-dev webpack

* 1

### 迁移到config文件

随着你项目越来越大，你的配置文件也会越来越复杂。从命令行配置webpack就显得很不科学。我们来创建一个配置文件。   
1.创建 webpack.config.js

module.exports = {

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

output: {

path: './bin',

filename: 'app.bundle.js'

}

};

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

webpack 配置文件是 node 风格的模式。所以您可以运行任何类型的代码，只要一个配置对象，导出了配置模块。

2.在放了配置文件的地方，你可以简单地运行一下 webpack:

webpack

* 1

3.运行 bin/app.bundle.js ,你又可以拿到你的猫了

node bin/app.bundle.js

["dave", "henry", "martha"]

* 1
* 2

你也可以 require 用npm安装过的模块，而不需要额外的配置。

### 使用loader

webpack原生只支持javascript模式。但是许多人会用像es 2015 ，coffeeScript ，typeScript 等语言。这些语言是可以用的，要用“loader”处理它们。   
loader是一种用来加载其它语言到javascript（webapck能够理解）的特殊的模块 。例如，babel-loader 用Babel 加载 ES2015文件。json-loader 加载JSON文件（简单地在前面加上 module.exports= 来把它转换成 CommonJS模式）   
。loader可以链式调用，而且你通常需要这么做。例如，yaml-loader 只把 YAML转换为 JSON,因此你需要 把它链到 json-loader 这样就可以用啦。

### 用 babel-loader 转换 ES2015

在这个例子中， 我们会告诉 webpack　用　babel 跑我们的源码，这样就可以用 ES2015新特性了。   
1.先安装 babel 和 预设。

npm intall --save-dev babel-core babel-preset-es2015

* 1

2.安装 babel-loader

npm install --save-dev babel-loader

* 1

3.添加 .babelrc 文件来配置 babel 使用 规范。

npm install --save-dev babel-loader

* 1

4.修改 webpack.config.js 用babel-loader 处理所以 .js文件。

module.exports = {

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

output: {

path: './bin',

filename: 'app.bundle.js',

},

module: {

loaders: [{

test: /\.js$/,

exclude: /node\_modules/,

loader: 'babel-loader',

}]

}

}

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

我们排除 node\_modules中的文件，因为要不然的话所有外部库都会通过babel编译，将会降低 编译速度。

5.安装你想用的第三方库。（例如 jQuery）

npm install --save jquery babel-polyfill

* 1

这里用 –save 而不是 –save-dev，　是因为这些库在执行时会用到。我们也安装了babel-polyfill 这样在一些老式浏览中可以用 ES2015了。

6.编辑 src/app.js

import 'babel-polyfill';

import cats from './cats';

import $ from 'jquery';

$('<h1>Cats</h1>').appendTo('body');

const ul = $('<ul></ul>').appendTo('body');

for (const cat of cats) {

$('<li></li>').text(cat).appendTo(ul);

}

7.用webpack 打包

webpack

* 1

8.添加 index.html，这样app能跑起来。

<!DOCTYPE html><body>

<script src="bin/app.bundle.js"></script>

* 1
* 2

当你打开 index.html 时你应该能看到你的猫了。

有许多不同的loders你可以用在app打包中，包括 css 和 image　loader。

### 用插件

通常你需要在你的打包工作流中做一些额外的事情。一个简单的例子是你会压缩文件来使客户端加载起来更快，这就可以用插件来做。我们将添加 uglify 插件到配置中。

const webpack = require('webpack');

module.exports = {

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

output: {

path: './bin',

filename: 'app.bundle.js',

},

module: {

loaders: [{

test: /\.jsx?$/,

exclude: /node\_modules/,

loader: 'babel',

}]

},

plugins: [

new webpack.optimize.UglifyJsPlugin({

compress: {

warnings: false,

},

output: {

comments: false,

},

}),

]

}

uglify 插件在webpack包中已经包含了，所以你不需要另外添加，但也有可能出现例外。你可以编写自己的自定义插件。在这个例子中， uglify 插件就把文件从1618b压缩到了308b。

<https://blog.csdn.net/keliyxyz/article/details/51527476>

# webpack入门（三）——webpack 配置

翻译 2016年05月28日 23:52:00

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

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

};

* 1
* 2
* 3

### 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);

* 1
* 2
* 3
* 4

### 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"

}

}

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

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"]

* 1

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"

}

}

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

**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

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

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

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

#### 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/"

}

* 1
* 2
* 3
* 4

**index.html**

<head>

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

</head>

* 1
* 2
* 3

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]/"

}

* 1
* 2
* 3
* 4

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

* 1
* 2
* 3

#### 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"

}

]

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

#### 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')

]

}

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

#### 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"]

}

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

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"))

]

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23

| **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=...

* 1
* 2
* 3
* 4
* 5

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",

}

}

* 1
* 2
* 3
* 4
* 5

#### 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

}

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

#### 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。

<https://blog.csdn.net/keliyxyz/article/details/51579078>

# webpack入门（四）——webpack loader 和plugin

翻译 2016年06月03日 15:09:46

* 标签：
* [webapck](http://so.csdn.net/so/search/s.do?q=webapck&t=blog) /
* [loader](http://so.csdn.net/so/search/s.do?q=loader&t=blog) /
* [plugin](http://so.csdn.net/so/search/s.do?q=plugin&t=blog)

## 什么是loader

loaders是你用在app源码上的转换元件。他们是用node.js运行的，把源文件作为参数，返回新的资源的函数。   
例如，你可以用loaders告诉webpack加载 coffeeScript或者JSX。   
loaders 特点：   
1. 可以链式拼接。他们用在通向文件的管道，最后一个loader预期返回一个javascript，其它Loader可以返回任意格式给下一个loader。   
2. loaders可以是同步的，也可以是异步的。   
3. loaders是用node.js来跑，可以做一切可能的事情。   
4. loaders接收query参数。这些参数会传入 loaders内部作为配置来用。   
5. 在webpack config 中 loaders可以绑定到 扩展名/正则 。   
6. loaders 可以用npm 发布或者安装。   
7. 除了用package.json 的main导出的 loader外， 一般的模块也可以导出一个loader。   
8. 装载机可以访问配置。   
9. plugin 可以给loaders更多功能。   
10. loader可以发出更多的任意文件。

## resoloving loaders

loader 的resolve跟 模块很像。一个loader预期导出一个函数，而且是用兼容javascript的nodepgn 的。一般情况下，用npm管理loader,但是你也可以在自己app内有loader文件。

### 引用loader

为了方便，虽然不是必须的， loaders一般命名为xxx-loader， xxx就是loader的实义名。例如 json-loader。   
或许你已经用全名引用了loader(例如 json-loader)，如果没有你可以用它的短名（例如 json）。   
装载机命名惯例和优先级搜索顺序由webpack 配置API中的resolveLoader.moduleTemplates 定义。   
装载机的命名约定可能在用require 语法引用时会有用。看下面的用法。

### 安装 loader

如果 npm 上有 这个Loader，你可以像这下面这样安装

$ nppm install xxx-loader --save

* 1

或者

$ npm install xxx-loader --save-dev

* 1

## 用法

有很多种办法在你的app中用loader：   
1. 显示地在 require 中添加。   
2. 在配置文件中 配置。   
3. 在命令行配置。

### 在require 中用

注意：如果你不知道你的代码在哪个环境（node.js和browser）中用尽量， 避免使用这个。 使用下一节那样的配置。

在require语句中（或者 define, require.ensure等）你可以指定装载机。只需要用 “！”将资源和Loader分开。每一部分会相对于当前文件夹来resolve。它可能覆盖config 文件中用 “！”规定的全部loader。

require("./loader!./dir/file.txt");

// => uses the file "loader.js" in the current directory to transform

// "file.txt" in the folder "dir".

require("jade!./template.jade");

// => uses the "jade-loader" (that is installed from npm to "node\_modules")

// to transform the file "template.jade"

// If configuration has some transforms bound to the file, they will still be applied.

require("!style!css!less!bootstrap/less/bootstrap.less");

// => the file "bootstrap.less" in the folder "less" in the "bootstrap"

// module (that is installed from github to "node\_modules") is

// transformed by the "less-loader". The result is transformed by the

// "css-loader" and then by the "style-loader".

// If configuration has some transforms bound to the file, they will not be applied.

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

### config配置

你可以在config文件中把loader绑定到 一个正则。

{

module: {

loaders: [

{ test: /\.jade$/, loader: "jade" },

// => "jade" loader is used for ".jade" files

{ test: /\.css$/, loader: "style!css" },

// => "style" and "css" loader is used for ".css" files

// Alternative syntax:

{ test: /\.css$/, loaders: ["style", "css"] },

]

}

}

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

### 命令行

命令行中，你也可以把Loader绑定到扩展名上。

$ webpack --module-bind jade --module-bind 'css=style!css'

* 1

上面的意思是用jade文件 用“jade-loader”加载，css文件用 “style-loader”和“css-loader”加载。

### 参数

loader可以传入query字符作为参数（像浏览器中那样），query字符串跟在 Loader后面。例如 url-loader?mimetype=image/png。   
注意：查询字符串的格式是由加载程序决定。请去具体的loader文档中查看。许多加载机可以接收正常的query字符串（例如 ?key=value&key2=value2）   
，也可以接收JSON对象(?{“key”:”value”,”key1”:”value1”})。

看以下四种写法。

require("url-loader?mimetype=image/png!./file.png");

* 1

{ test: /\.png$/, loader: "url-loader?mimetype=image/png" }

* 1

{

test: /\.png$/,

loader: "url-loader",

query: { mimetype: "image/png" }

}

* 1
* 2
* 3
* 4
* 5

webpack --module-bind "png=url-loader?mimetype=image/png"

* 1

# 使用插件

在webpack中通常使用插件添加与 bundles相关的功能。例如 BellOnBundlerErrorPlugin 会在打包进程中给你通知错误消息。

## 内置插件

直接用配置项的plugins属性。

// webpack should be in the node\_modules directory, install if not.

var webpack = require("webpack");

module.exports = {

plugins: [

new webpack.ResolverPlugin([

new webpack.ResolverPlugin.DirectoryDescriptionFilePlugin("bower.json", ["main"])

], ["normal", "loader"])

]

};

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

## 其它插件

如果不是内置插件，在npm 上发布过的插件一般可以通过npm 安装安装，如果没有发布，你要用其它方法了

npm install component-webpack-plugin

* 1

然后像像下面这样使用

var ComponentPlugin = require("component-webpack-plugin");

module.exports = {

plugins: [

new ComponentPlugin()

]

}

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

如果你用npm装第三方插件，建议用 webpack-load-plugins .它能检测所有第你安装过并且在保存在package中的三方插件，在你需要用的时候会加载进来。

<https://blog.csdn.net/keliyxyz/article/details/51579983>

# webpack入门（五）——webpack 故障处理

翻译 2016年06月03日 16:37:59

* 标签：
* [webpack](http://so.csdn.net/so/search/s.do?q=webpack&t=blog) /
* [故障](http://so.csdn.net/so/search/s.do?q=故障&t=blog) /
* [watch](http://so.csdn.net/so/search/s.do?q=watch&t=blog) /
* [window](http://so.csdn.net/so/search/s.do?q=window&t=blog) /
* [os-x](http://so.csdn.net/so/search/s.do?q=os-x&t=blog)

## resolveing

一般问题的解决   
1. –display-error-details 提供给你更多的出错信息。   
2. 查问 配置文档关于 resolve 的部分。loader有它自己的 resolveing配置 resolveLoader

npm 链接的模块找不到它们的依赖。   
node.js 模块的resolve步骤非常简单： 在模块的父文件夹中的node\_modules中依次向上查找模块.如果找到根目录还没找到，说明模块确实找不到了 (你可以认为用nodejs写的peerDependencies坏掉了 ) 。注意：一个应用的依赖（即使这是个不完美的设计）也是一种相关依赖，即使它没有在 package.json中列出来。   
但是用webpack你可以很容易避开这个问题，添加 node\_modules文件夹到 resolve paths中。这样做有两个配置选项 resolve.fallback 和 resolveLoader.fallback。如例：

module.exports = {

resolve: { fallback: path.join(\_\_dirname, "node\_modules") },

resolveLoader: { fallback: path.join(\_\_dirname, "node\_modules") }

};

* 1
* 2
* 3
* 4

## watch

**开了watch ，但 是修改了文件，没有编译。**   
**文件的变化看到了，只是没有文件被更新。**   
确定webpack 有打开通知（用–progress标志）。如果 开了了但是文件没有输出，可能是配置的问题，而不是watch的问题。

webpack --watch --progress

* 1

**没有足够的watchers**   
确定你有足够的观察者在你的系统中。如果此值太低，文件查看器在Webpack认不出的变化 。

cat /proc/sys/fs/inotify/max\_user\_watches

* 1

Arch 用户添加fs.inotify.max\_user\_watches=524288 到 /etc/sysctl.d/99-sysctl.conf ，然后运行 sysctl --system 。

Ubuntu 用户（或者其它） : echo fs.inotify.max\_user\_watches=524288 | sudo tee -a /etc/sysctl.conf && sudo sysctl -p. 。

**OS-X fsevents bug**   
在 OS-X上文件夹可以获取损坏文件。、

**Windows 路径**   
webpack许多配置项的路径预期都是绝对路径。\_\_dirname+"/app/folder" 这样写是错的，因为windows用 反斜杠切分路径。这样容易出问题。   
使用正确的分隔符例如path.resolve(\_\_dirname, "app/folder") 或者 path.join(\_\_dirname, "app", "folder")。

**vim**   
有些机器上vim 的 backupcopy 默认设置为 auto了。这可能会导致文件监视机制的问题。把这个选项切换到 yes 确保有一个文件备份，源文件在保存的时候会被重写。

:set backupcopy=yes

<https://blog.csdn.net/keliyxyz/article/details/51513114>

# webpack入门（六）——html-webpack-plugin

翻译 2016年05月27日 10:15:30

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

# html-webpack-plugin

该插件可以简化创建调用webpack bundles的html文件。在每次编译后，文件名会包含有hash值的bundles 特别有用。你可以让插件为您生成一个HTML文件，也可以提供您自己使用lodash模板的模板，或使用您自己的装载机。   
维护者：Jan Nicklas @jantimon。

## 安装

用npm安装这个插件

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

* 1

## 基本配置

该插件将为您生成一个HTML5文件，这个文件用script标签引用所有的webpack包。只需将插件添加到您的webpack配置，如下：

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

var webpackConfig = {

entry: 'index.js',

output: {

path: 'dist',

filename: 'index\_bundle.js'

},

plugins: [new HtmlWebpackPlugin()]

};

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

这样就会生成一个文件 **dist/index.html**，如下：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Webpack App</title>

</head>

<body>

<script src="index\_bundle.js"></script>

</body>

</html>

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

如果您有多个webpack入口点，他们都将包括在生成的HTML文件script标签中。   
如果你有用webpack产出css文件（例如用ExtractTextPlugin提取的css文件），那么html-webpack-plugin会在html的head中插件link标签引入这些css文件。

## 完整配置

你可以传一个配置选项的 散列到 **HtmlWebpackPlugin**，允许的值如下：

**title** : 用于生成的HTML文件的标题。

**filename** : 用于生成的HTML文件的名称，默认是index.html。你可以在这里指定子目录（例如:assets/admin.html）

**template** : 模板的路径。支持加载器，例如 html!./index.html。

**inject** :true | ‘head’ | ‘body’ | false 。把所有产出文件注入到给定的 template 或templateContent。当传入 true或者 ‘body’时所有javascript资源将被放置在body元素的底部，“head”则会放在head元素内。

**favicon** : 给定的图标路径，可将其添加到输出html中。

**minify** : {…} | false 。传一个**html-minifier** 配置object来压缩输出。

**hash** : true | false。如果是true，会给所有包含的script和css添加一个唯一的webpack编译hash值。这对于缓存清除非常有用。

**cache** : true | false 。如果传入true（默认），只有在文件变化时才 发送（emit）文件。

**showErrors** : true | false 。如果传入true（默认），错误信息将写入html页面。

**chunks** : 只允许你添加chunks 。（例如：只有单元测试块 ）

**chunksSortMode** : 在chunk被插入到html之前，你可以控制它们的排序。允许的值 ‘none’ | ‘auto’ | ‘dependency’ | {function} 默认为‘auto’.

**excludeChunks** : 允许你跳过一些chunks（例如，不要单元测试的 chunk）.

**xhtml** : 用于生成的HTML文件的标题。

**title** : true | false。如果是true，把link标签渲染为自闭合标签，XHTML要这么干的。默认false。

下面是一个示例webpack配置说明如何使用这些选项：

{

entry: 'index.js',

output: {

path: 'dist',

filename: 'index\_bundle.js'

},

plugins: [

new HtmlWebpackPlugin({

title: 'My App',

filename: 'assets/admin.html'

})

]

}

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

## 生成多个html文件

生成多个html文件，多次声明这个插件在plugins 数组中。如下：

{

entry: 'index.js',

output: {

path: 'dist',

filename: 'index\_bundle.js'

},

plugins: [

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

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

filename: 'test.html',

template: 'src/assets/test.html'

})

]

}

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

## 自定义模板

如果默认生成的HTML不能满足你的需求，你可以自己写模板。最简单的方法是使用插入选项，并传入一个自定义的html文件。html-webpack-plugin将自动注入所需的css, js, manifest 和 favicon 到标签中。

plugins: [

new HtmlWebpackPlugin({

title: 'Custom template',

template: 'my-index.ejs', // Load a custom template (ejs by default but can be changed)

inject: 'body' // Inject all scripts into the body (this is the default so you can skip it)

})

]

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

my-index.ejs:

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-type" content="text/html; charset=utf-8"/>

<title><%= htmlWebpackPlugin.options.title %></title>

</head>

<body>

</body>

</html>

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

如果你已经有模板的loader，你可以用它来解析模板。请注意，如果您指定html-loader 并且用了 .html文件作为模板，它也会发生。

module: {

loaders: [

{ test: /\.hbs$/, loader: "handlebars" }

]

},

plugins: [

new HtmlWebpackPlugin({

title: 'Custom template using Handlebars',

template: 'my-index.hbs'

})

]

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

你可以用开开箱即用的lodash语法。如果inject 性能还不能满足你的需求，而且你想完全控制资源放到哪里，你可以用[html-webpack-template project](https://github.com/jaketrent/html-webpack-template)的[默认模板](https://github.com/jaketrent/html-webpack-template/blob/86f285d5c790a6c15263f5cc50fd666d51f974fd/index.html) 模板作为启动点。   
下列这些变量可以用在模板中：   
**htmlWebpackPlugin**:这个插件的特定数据   
**htmlWebpackPlugin.files** 它包含一个从入口点名称映射到包的文件名   
  
"htmlWebpackPlugin": {   
"files": {   
"css": [ "main.css" ],   
"js": [ "assets/head\_bundle.js", "assets/main\_bundle.js"],   
"chunks": {   
"head": {   
"entry": "assets/head\_bundle.js",   
"css": [ "main.css" ]   
},   
"main": {   
"entry": "assets/main\_bundle.js",   
"css": []   
},   
}   
}   
}   
  
如果你在webpack 配置文件中设置了publicPath。htmlWebpackPlugin.files将会正确映射到 资源散列。   
**htmlWebpackPlugin.options**：传给插件的 配置项。除了插件本身使用这个些配置项以外，你也可以在模板中使用这些配置项。   
**webpack**：webpack的统计对象。注意：这是stats对象，因为它是在HTML模板时发出，因此wepback运行完成后可能没有完整的数据集可用。   
**webpackConfig**：插件编译用的webpack 配置项。例如它可以用来获取publicPath （webpackConfig.output.publicPath）。

## 过滤Filtering chunks

只包括某些你模块（chunk），你可以限制这些模块的使用。

plugins: [

new HtmlWebpackPlugin({

chunks: ['app']

})

]

* 1
* 2
* 3
* 4
* 5

通过设置excludeChunks选项还可以排除某些块：

plugins: [

new HtmlWebpackPlugin({

excludeChunks: ['dev-helper']

})

]

* 1
* 2
* 3
* 4
* 5

## 事件

允许其它插件篡改这个插件执行的以下方法：   
1. html-webpack-plugin-before-html-generation   
2. html-webpack-plugin-before-html-processing   
3. html-webpack-plugin-after-html-processing   
4. html-webpack-plugin-after-emit

例如用[html-webpack-harddisk-plugin](https://github.com/jantimon/html-webpack-harddisk-plugin)   
用法：

/ MyPlugin.js

function MyPlugin(options) {

// Configure your plugin with options...

}

MyPlugin.prototype.apply = function(compiler) {

// ...

compiler.plugin('compilation', function(compilation) {

console.log('The compiler is starting a new compilation...');

compilation.plugin('html-webpack-plugin-before-html-processing', function(htmlPluginData, callback) {

htmlPluginData.html += 'The magic footer';

callback(null, htmlPluginData);

});

});

};

module.exports = MyPlugin;

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

然后在webpack.config.js中这样写：

plugins: [

new MyPlugin({options: ''})

]

* 1
* 2
* 3

注意：这个回调函数必须传htmlPluginData ,这是为了把它传给其它也监听“html-webpack-plugin-before-html-processing”这个同一事件的插件。