**Webpack2官方文档中文翻译**

**相关概念**

Webpack是一个JavaScript应用模块打包器。它的配置非常灵活且强大，然而，在你使用它之前，你应该清楚以下这四个概念。

作为你学习webpack旅程的一部分，这个文档的编写目的是为了以更准确的方式让你了解这些概念，同时为你提供了一些了解这些概念特殊用途的链接。

**Entry:**

Webpack为你的应用建立了一张资源依赖表，这张表的开始部分就是entry。这个入口告诉webpack从哪里开始，并根据文件依赖去打包文件。你可以将这个入口文件看做根目录或者你的应用开始时的第一个文件。

在webpack中我们使用webpack配置对象的entry属性来定义

下面就是一个最简单的例子：

webpack.config.js

module.exports = {

entry: './path/to/my/entry/file.js'

};

根据应用需要可以有多种方式表明入口属性，即一个应用可以有多个入口

**Output:**

当你将所有资源打包在一起的时候，你需要告诉webpack将这些资源打包到哪里，webpack的output属性指定了资源的打包输出路径

webpack.config.js

const path = require('path');

module.exports = {

entry: './path/to/my/entry/file.js',

output: {

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

filename: 'my-first-webpack.bundle.js'

}

};

在上面的例子中，我们使用了output.filename和output.path属性来告诉webpack应用希望被打包输出的路径和打包后的文件名。

Output属性有许多可配置的特性，但是在配置之前，让我们花一点时间来理解一些output属性的最常见使用案例

[Learn more](https://webpack.js.org/concepts/output/)

Loaders:

Loaders的目标是将项目的资源变成webpack的关注点，即将项目所有资源交给webpack进行处理而不是直接交给浏览器。（这并不意味着它们必须被打包到一起）。Webpack将所有文件（css、html、scss、jpg等等）统统视作模块，然而webpack只理解JavaScript语言。

因此loaders负责将这些文件转换为模块加入到资源依赖表中。

从深层次意义来讲，在webpack配置文件中loaders有两个目的。

1. 定义什么文件应该被特定的loader转换（test属性）
2. 转换这些文件以便它能加入到你的依赖表中（并且最终被一起打包）（use属性）

webpack.config.js

const path = require('path');

const config = {

entry: './path/to/my/entry/file.js',

output: {

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

filename: 'my-first-webpack.bundle.js'

},

module: {

rules: [

{test: /\.(js|jsx)$/, use: 'babel-loader'}

]

}

};

module.exports = config;

上面的配置用两个必要的属性test和use定义了一个rules，告诉webpack应用使用的编译器

**Plugins：**

因为loaders仅仅执行了对一个基础文件的转换操功能，因此plugins通常被用来执行对打包模块进行类似编译和分块等操作和一些自定义功能实现。Webpack的插件系统是极其灵活且强大的，并且可根据需要进行自定义设置。

使用一个插件时，你只需要利用require（）函数将它导入进来，并且将它加入plugins数组中即可。大部分插件可以通过option选项进行定义。在一个配置文件中你可以针对不同目的多次使用同一个插件，只要在调用它时使用new创建一个新的对象便可使它立即生效。

webpack.config.js

const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via npm

const webpack = require('webpack'); //to access built-in plugins

const path = require('path');

const config = {

entry: './path/to/my/entry/file.js',

output: {

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

filename: 'my-first-webpack.bundle.js'

},

module: {

rules: [

{test: /\.(js|jsx)$/, use: 'babel-loader'}

]

},

plugins: [

new webpack.optimize.UglifyJsPlugin(),

new HtmlWebpackPlugin({template: './src/index.html'})

]

};

module.exports = config;

webpack提供了很多开包即用的插件，可以浏览[插件列表](https://webpack.js.org/plugins/)来获取更多信息。

在你的webpack配置文件中使用插件是一件比较简单直观的事情，但是也有许多的使用案例值得进一步讨论。

**文档信息**

**介绍**

Webpack是通过一个配置对象进行工作的，它通过哪种方式运行取决于你如何使用webpack：通过terminal终端或者node.js，所有可用的配置选项指定如下：

[Options](https://webpack.js.org/configuration/)

注意在配置文件中从始至终我们使用node的内置path模块和带前缀的\_\_dirname全局变量定义路径，这样可以避免操作系统间的路径定义不统一所带来的问题，并且让相对路径可以按照预期结果执行。

**配置类型**

除了导出单个配置对象，也有几种其他方式可以满足需要。

* **通过用 –env导出一个function**

最终你会发现在webpack.config.js文件中区别development和production是很有必要的，你有两种选择（至少）：

除了导出一个配置对象，你也可以导出一个接受environment变量做参数的函数。当运行webpack的时候你可以通过- -env来指定运行环境，例如

- -env.production 或 - -env.platform=web

+module.exports = function(env) {

+ return {

plugins: [

new webpack.optimize.UglifyJsPlugin({

+ compress: env.production // compress only in production build

})

]

+ };

};

* **导出一个promise**

Webpack将会运行由配置文件导出的函数并且等待一个promise被返回。当你需要异步加载配置变量的时候这是很方便的。

module.exports = () => {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve({

entry: './app.js',

/\* ... \*/

})

}, 5000)

})

}

* **导出多个配置**

除了导出单个配置对象或者函数外，你也可以导出多个配置，当运行webpack的时候，所有的配置都会生效，例如，这样构建多个目标库如AMD和CommonJS模块是有效的：

module.exports = [{

output: {

filename: './dist-amd.js',

libraryTarget: 'amd'

},

entry: './app.js',

}, {

output: {

filename: './dist-commonjs.js',

libraryTarget: 'commonjs'

},

entry: './app.js',

}]

**用JavaScript语言的替代品进行配置**

Webpack允许你以任何语言定义配置文件。支持的语言列表可以在node-interpret包里找到。在node-interpret包的帮助下webpack会以你选择的语言运行。

例如，如果你选择使用coffeescript，你的文件将会像下面这样：

Webpack.config.coffee

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

webpack = require('webpack')

path = require('path')

config =

entry: './path/to/my/entry/file.js'

output:

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

filename: 'my-first-webpack.bundle.js'

module: rules: [ {

test: /\.(js|jsx)$/

use: 'babel-loader'

} ]

plugins: [

new (webpack.optimize.UglifyJsPlugin)

new HtmlWebpackPlugin(template: './src/index.html')

]

module.exports = config

**entry和context**

Entry对象是webpack打包开始的地方。Context是包含入口文件的绝对路径字符串。

* **Context**

基本目录（一个绝对字符串），用于解决配置中入口和loaders路径问题。

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

默认情况下则使用这个路径，但是建议在配置文件中明确标注路径，这样可以使得你的配置文件独立于当前工作目录。

* **Entry**

入口可以有一或多个，这是应用开始执行操作的地方，如果是一个数组，则数组内的所有文件都会被执行。

动态加载的模块不是入口点。

简单的规则：每个页面一个入口点。

SPA（单页面应用）：一个入口。

MPA（多页面应用）：多个入口。

entry: {

home: "./home.js",

about: "./about.js",

contact: "./contact.js"

}

当和output.library选项结合起来使用的时候，数组中只有最后一项才是可用的。

**Output**

高级output配置包含webpack应该将资源打包到何处的配置选项。

* **Output.chunkFilename**

这个选项决定按需加载模块的名称，获取更多详细配置信息请查看output.filename选项。

注意，在运行时这些文件名会根据chunks的请求而产生。因此，类似[name]、[chunkhash]这些占位符就需要添加与chunk id的映射，以便在运行时可以替换output模块的占位值，chunk因为大小改变或者失效时，占位符也同样会跟着改变

默认使用[id].js或者output.filename的值（[name]将会被[id]代替或者注入。

* **Output.filename**

这个选项决定每个输出打包文件的名字。打包完成的文件则被写入output.path选项指定的路径。

对于单个的输出，可以是一个简单的静态命名。

filename: "bundle.js"

然而，当通过多入口、代码分隔或各种插件建立多个打包文件的时候，你应该使用以下方式进行替换以给每个打包文件赋予唯一的命名。

使用入口名称

filename: "[name].bundle.js"

使用内部分块id：

filename: "[id].bundle.js"

为每个打包文件建立唯一hash值

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

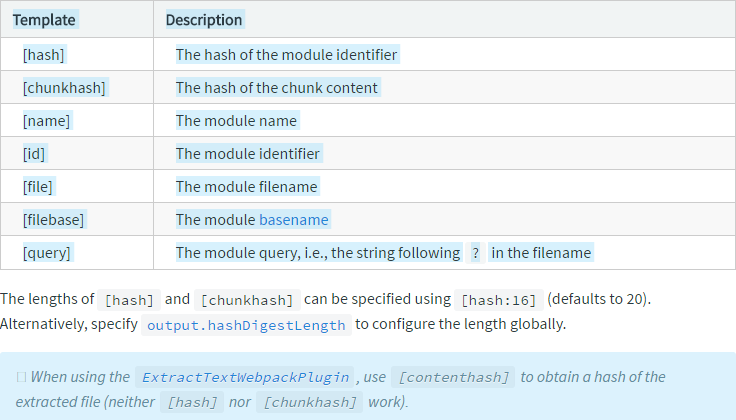
基于每个分块内容建立hash值

filename: "[chunkhash].bundle.js"

更多详情请查看[Caching guide](https://webpack.js.org/guides/caching/)

在上面的配置中，默认值是[name].js，但请注意，虽然选项叫文件名称，但是你仍然可以用带有路径的文件名，比如“js/[name]/bundle.js“来配置此选项。

注意这个选项并不会影响按需加载的分开。因为按需加载的分块已经由outp.chunkFilename选项指定了。同时它也不会影响由loaders创建的文件，因为这些文件由loaders选项配置。

以下配置选项都可以选择性替代模板字符串（通过webpack内部TemplatedPathPlugin插件。

* **Output.path**

Output目录作为绝对路径。

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

注意参数中的[hash]将会被编译时的hash代替，更多详情请查看[Caching guide](https://webpack.js.org/guides/caching/)。

* **Output.pathinfo**

这个属性是告诉webpack将所属模块信息的注释也一并进行打包。默认是false，并且在生产环境中不应该使用它。但是在开发环境中，这个配置对于调试和阅读源码是非常有用的。

* **Output.publicPath**

在你使用按需加载或者加载类似图片、文件等外部资源时，这是一个非常重要的选项，在加载外部资源时，如果指定了一个错误的值，你将会收到404的错误。

这个选项指定了浏览器引用输出目录的公共URL，URL为相对路径时是相对于HTML页面的。同时也可以使用相对服务器的URLs、带相对协议的的URLs或者绝对路径的URLs，并且有的时候这是很有必要的，比如需要将静态资源部署到CDN上。

这个选项的值是作为运行时loaders产生的URL的前缀。因为在大部分案例中这个值都是以‘/’为末尾结束符的。

默认值是一个空字符串—“”。

简单规则：output.path的URL在页面中就会看起来是下面这样:

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

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

解释一下：上面这种配置，html页面将会打包输出在public/assets下面，而页面中的所有资源路径（由loaders转换而来的资源，例如js、css、图片）将会是https://cdn.example.com/assets/ 加文件名的这种形式。

而对于下面这种配置：

publicPath: "/assets/",

chunkFilename: "[id].chunk.js"

输出的chunk则会类似/assets/4.chunk.js这样。

由loader转换而来的资源在HTML页面中则类似href="/assets/spinner.gif" />这样。至于css中引用的图片，则变成background-image: url(/assets/spinner.gif);

Webpack-dev-server也会对publicPath进行提示，可以利用这个提示来确定服务端输出路径。

注意这个变量中的[hash]值将会被编译后的hash值替代。更多详情请查看[Caching guide](https://webpack.js.org/guides/caching/)。

例子：

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

publicPath: "//cdn.example.com/assets/", // CDN (same protocol)

publicPath: "/assets/", // server-relative

publicPath: "assets/", // relative to HTML page

publicPath: "../assets/", // relative to HTML page

publicPath: "", // relative to HTML page (same directory)

**模块：**

* **Rule.use**

这是对应于应用模块的可用entry列表。每个入口都对应特定的loader来使用。

Loader属性（例如use: [ { loader: "style-loader "} ]）可以简写为一个字符串（例如use: [ "style-loader" ]）。

Loaders可以通过多个loaders进行声明，应用顺序为从右到左（最后一个先执行）。

use: [

{

loader: 'style-loader'

},

{

loader: 'css-loader',

options: {

importLoaders: 1

}

},

{

loader: 'less-loader',

options: {

noIeCompat: true

}

}

]

* **匹配条件**

匹配条件可以是以下任意一条：

一个字符串：以提供的字符串为开始入口，匹配输入。例一个目录的绝对路径，或者文件的绝对路径。

一个正则表达式：用来匹配输入。

一个函数：以输入命名并且必须返回一个可匹配的真值。

一组条件数组：需要至少有一个可匹配。

一个对象：所有属性必须都可匹配，且每个属性都有默认值。

{ test: Condition }：条件必须匹配，通常是正则或正则数组，但不是必选项。

{ include: Condition }：条件必须匹配，通常是字符串或字符串数组，同样也不是必选项。

{ exclude: Condition }：条件不必完全匹配。通常是字符串或字符串数组，不是必选项。

{ and: [Condition] }：所有条件必须匹配。

{ or: [Condition] }：匹配任意一条。

{ not: Condition }：条件需排除。

例子：

{

test: /\.css$/,

include: [

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

path.resolve(\_\_dirname, "vendor/styles")

]

}

**UseEntry**

这个属性的值必须是一个代表loader的字符串。它是根据配置文件中的resolving 选项来加载处理对应的文件。

同时它还有一个以字符串或者对象为值的options属性，这个值将会传递给loader，作为loader的属性值来处理。

也可以设置兼容性的query属性，这是option属性的别名，可以用option替换。

例子：

{

loader: "css-loader",

options: {

modules: true

}

}

注意，webpack需要对所有资源以及loaders包含的选项生成唯一标识符。因此它会对所有options对象用JSON.stringify进行转换，在绝大部分情况下都不会出现问题，但是当你对同一个资源应用拥有不同option选项的loader，并且这个option里面带有已经转换过的值，那么当这个值不能被序列化的时候就可能会出错。针对上述问题，你可以在options属性里带一个标志属性作为唯一标识符，从而避免序列化失败。