# Webpack

## 安装

### 前提条件

在开始之前，请确保安装了 [Node.js](https://nodejs.org/en/) 的最新版本。使用 Node.js 最新的长期支持版本(LTS - Long Term Support)，是理想的起步。使用旧版本，你可能遇到各种问题，因为它们可能缺少 webpack 功能以及/或者缺少相关 package 包。

### 本地安装

要安装最新版本或特定版本，请运行以下命令之一：

npm install --save-dev webpack

npm install --save-dev webpack@<version>

如果你使用 webpack 4+ 版本，你还需要安装 CLI。

对于大多数项目，我们建议本地安装。这可以使我们在引入破坏式变更(breaking change)的依赖时，更容易分别升级项目。通常，webpack 通过运行一个或多个 [npm scripts](https://docs.npmjs.com/misc/scripts)，会在本地 node\_modules 目录中查找安装的 webpack：

"scripts": {

"start": "webpack --config webpack.config.js"

}

*当你在本地安装 webpack 后，你能够从 node\_modules/.bin/webpack 访问它的 bin 版本。*

### 全局安装

以下的 NPM 安装方式，将使 webpack 在全局环境下可用：

npm install --global webpack

*不推荐全局安装 webpack。这会将你项目中的 webpack 锁定到指定版本，并且在使用不同的 webpack 版本的项目中，可能会导致构建失败。*

## 起步

webpack 用于编译 JavaScript 模块。一旦完成[安装](https://www.webpackjs.com/guides/installation)，你就可以通过 webpack 的 [CLI](https://www.webpackjs.com/api/cli) 或 [API](https://www.webpackjs.com/api/node) 与其配合交互。如果你还不熟悉 webpack，请阅读[核心概念](https://www.webpackjs.com/concepts)和[打包器对比](https://www.webpackjs.com/comparison)，了解为什么你要使用 webpack，而不是社区中的其他工具。

首先我们创建一个目录，初始化 npm，然后 [在本地安装 webpack](https://www.webpackjs.com/guides/installation#local-installation)，接着安装 webpack-cli（此工具用于在命令行中运行 webpack）：

mkdir webpack-demo && cd webpack-demo

npm init -y

npm install webpack webpack-cli --save-dev

*贯穿整个指南的是，我们将使用 diff 块，来显示我们对目录、文件和代码所做的更改。*

现在我们将创建以下目录结构、文件和内容：

**Project**

webpack-demo

|- package.json

+ |- index.html

+ |- /src

+ |- index.js

**src/index.js**

function component() {

var element = document.createElement('div');

// Lodash（目前通过一个 script 脚本引入）对于执行这一行是必需的

element.innerHTML = \_.join(['Hello', 'webpack'], ' ');

return element;

}

document.body.appendChild(component());

**index.html**

<!doctype html>

<html>

<head>

<title>起步</title>

<script src="https://unpkg.com/lodash@4.16.6"></script>

</head>

<body>

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

</body>

</html>

我们还需要调整 package.json 文件，以便确保我们安装包是私有的(private)，并且移除 main 入口。这可以防止意外发布你的代码。

*如果你想要了解 package.json 内在机制的更多信息，我们推荐阅读 [npm 文档](https://docs.npmjs.com/files/package.json)。*

**package.json**

{

"name": "webpack-demo",

"version": "1.0.0",

"description": "",

+ "private": true,

- "main": "index.js",

"scripts": {

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

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"webpack": "^4.0.1",

"webpack-cli": "^2.0.9"

},

"dependencies": {}

}

在此示例中，<script> 标签之间存在隐式依赖关系。index.js 文件执行之前，还依赖于页面中引入的 lodash。之所以说是隐式的是因为 index.js 并未显式声明需要引入 lodash，只是假定推测已经存在一个全局变量 \_。

使用这种方式去管理 JavaScript 项目会有一些问题：

* 无法立即体现，脚本的执行依赖于外部扩展库(external library)。
* 如果依赖不存在，或者引入顺序错误，应用程序将无法正常运行。
* 如果依赖被引入但是并没有使用，浏览器将被迫下载无用代码。

让我们使用 webpack 来管理这些脚本。

### 创建一个bundle文件

首先，我们稍微调整下目录结构，将“源”代码(/src)从我们的“分发”代码(/dist)中分离出来。“源”代码是用于书写和编辑的代码。“分发”代码是构建过程产生的代码最小化和优化后的“输出”目录，最终将在浏览器中加载：

webpack-demo

|- package.json

+ |- /dist

+ |- index.html

- |- index.html

|- /src

|- index.js

要在 index.js 中打包 lodash 依赖，我们需要在本地安装 library：

npm install --save lodash

*在安装一个要打包到生产环境的安装包时，你应该使用 npm install --save，如果你在安装一个用于开发环境的安装包（例如，linter, 测试库等），你应该使用 npm install --save-dev。请在 [npm 文档](https://docs.npmjs.com/cli/install) 中查找更多信息。*

现在，在我们的脚本中 import lodash：

**src/index.js**

+ import \_ from 'lodash';

+

function component() {

var element = document.createElement('div');

- // Lodash, currently included via a script, is required for this line to work

+ // Lodash, now imported by this script

element.innerHTML = \_.join(['Hello', 'webpack'], ' ');

return element;

}

document.body.appendChild(component());

现在，由于通过打包来合成脚本，我们必须更新 index.html 文件。因为现在是通过 import 引入 lodash，所以将 lodash <script> 删除，然后修改另一个 <script> 标签来加载 bundle，而不是原始的 /src 文件：

**dist/index.html**

<!doctype html>

<html>

<head>

<title>起步</title>

- <script src="https://unpkg.com/lodash@4.16.6"></script>

</head>

<body>

- <script src="./src/index.js"></script>

+ <script src="main.js"></script>

</body>

</html>

在这个设置中，index.js 显式要求引入的 lodash 必须存在，然后将它绑定为 \_（没有全局作用域污染）。通过声明模块所需的依赖，webpack 能够利用这些信息去构建依赖图，然后使用图生成一个优化过的，会以正确顺序执行的 bundle。

可以这样说，执行 npx webpack，会将我们的脚本作为[入口起点](https://www.webpackjs.com/concepts/entry-points)，然后 [输出](https://www.webpackjs.com/concepts/output) 为 main.js。Node 8.2+ 版本提供的 npx 命令，可以运行在初始安装的 webpack 包(package)的 webpack 二进制文件（./node\_modules/.bin/webpack）：

npx webpack

Hash: dabab1bac2b940c1462b

Version: webpack 4.0.1

Time: 3003ms

Built at: 2018-2-26 22:42:11

Asset Size Chunks Chunk Names

main.js 69.6 KiB 0 [emitted] main

Entrypoint main = main.js

[1] (webpack)/buildin/module.js 519 bytes {0} [built]

[2] (webpack)/buildin/global.js 509 bytes {0} [built]

[3] ./src/index.js 256 bytes {0} [built]

+ 1 hidden module

WARNING in configuration(配置警告)

The 'mode' option has not been set. Set 'mode' option to 'development' or 'production' to enable defaults for this environment.('mode' 选项还未设置。将 'mode' 选项设置为 'development' 或 'production'，来启用环境默认值。)

*输出可能会稍有不同，但是只要构建成功，那么你就可以继续。并且不要担心，稍后我们就会解决。*

在浏览器中打开 index.html，如果一切访问都正常，你应该能看到以下文本：'Hello webpack'。

### 模块

[ES2015](https://babeljs.io/learn-es2015/) 中的 [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) 和 [export](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export) 语句已经被标准化。虽然大多数浏览器还无法支持它们，但是 webpack 却能够提供开箱即用般的支持。

事实上，webpack 在幕后会将代码“转译”，以便旧版本浏览器可以执行。如果你检查 dist/bundle.js，你可以看到 webpack 具体如何实现，这是独创精巧的设计！除了 import 和 export，webpack 还能够很好地支持多种其他模块语法，更多信息请查看[模块 API](https://www.webpackjs.com/api/module-methods)。

注意，webpack 不会更改代码中除 import 和 export 语句以外的部分。如果你在使用其它 [ES2015 特性](http://es6-features.org/)，请确保你在 webpack 的 [loader 系统](https://www.webpackjs.com/concepts/loaders/)中使用了一个像是 [Babel](https://babeljs.io/) 或 [Bublé](https://buble.surge.sh/guide/) 的[转译器](https://www.webpackjs.com/loaders/" \l "transpiling)。

### 使用一个配置文件

在 webpack 4 中，可以无须任何配置使用，然而大多数项目会需要很复杂的设置，这就是为什么 webpack 仍然要支持 [配置文件](https://www.webpackjs.com/concepts/configuration)。这比在终端(terminal)中手动输入大量命令要高效的多，所以让我们创建一个取代以上使用 CLI 选项方式的配置文件：

**project**

webpack-demo

|- package.json

+ |- webpack.config.js

|- /dist

|- index.html

|- /src

|- index.js

**webpack.config.js**

const path = require('path');

module.exports = {

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

output: {

filename: 'bundle.js',

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

}

};

现在，让我们通过新配置文件再次执行构建：

npx webpack --config webpack.config.js

Hash: dabab1bac2b940c1462b

Version: webpack 4.0.1

Time: 328ms

Built at: 2018-2-26 22:47:43

Asset Size Chunks Chunk Names

bundle.js 69.6 KiB 0 [emitted] main

Entrypoint main = bundle.js

[1] (webpack)/buildin/module.js 519 bytes {0} [built]

[2] (webpack)/buildin/global.js 509 bytes {0} [built]

[3] ./src/index.js 256 bytes {0} [built]

+ 1 hidden module

WARNING in configuration(配置警告)

The 'mode' option has not been set. Set 'mode' option to 'development' or 'production' to enable defaults for this environment.('mode' 选项还未设置。将 'mode' 选项设置为 'development' 或 'production'，来启用环境默认值。)

*注意，当在 windows 中通过调用路径去调用 webpack 时，必须使用反斜线( )。例如 node\_modules\.bin\webpack --config webpack.config.js。*

*如果 webpack.config.js 存在，则 webpack 命令将默认选择使用它。我们在这里使用 --config 选项只是向你表明，可以传递任何名称的配置文件。这对于需要拆分成多个文件的复杂配置是非常有用。*

比起 CLI 这种简单直接的使用方式，配置文件具有更多的灵活性。我们可以通过配置方式指定 loader 规则(loader rules)、插件(plugins)、解析选项(resolve options)，以及许多其他增强功能。了解更多详细信息，请查看[配置文档](https://www.webpackjs.com/configuration)。

### NPM脚本（NPM Script）

考虑到用 CLI 这种方式来运行本地的 webpack 不是特别方便，我们可以设置一个快捷方式。在 package.json 添加一个 [npm 脚本(npm script)](https://docs.npmjs.com/misc/scripts)：

**package.json**

{

"name": "webpack-demo",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

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

+ "build": "webpack"

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"webpack": "^4.0.1",

"webpack-cli": "^2.0.9",

"lodash": "^4.17.5"

}

}

现在，可以使用 npm run build 命令，来替代我们之前使用的 npx 命令。注意，使用 npm 的 scripts，我们可以像使用 npx 那样通过模块名引用本地安装的 npm 包。这是大多数基于 npm 的项目遵循的标准，因为它允许所有贡献者使用同一组通用脚本（如果必要，每个 flag 都带有 --config 标志）。

现在运行以下命令，然后看看你的脚本别名是否正常运行：

npm run build

Hash: dabab1bac2b940c1462b

Version: webpack 4.0.1

Time: 323ms

Built at: 2018-2-26 22:50:25

Asset Size Chunks Chunk Names

bundle.js 69.6 KiB 0 [emitted] main

Entrypoint main = bundle.js

[1] (webpack)/buildin/module.js 519 bytes {0} [built]

[2] (webpack)/buildin/global.js 509 bytes {0} [built]

[3] ./src/index.js 256 bytes {0} [built]

+ 1 hidden module

WARNING in configuration(配置警告)

The 'mode' option has not been set. Set 'mode' option to 'development' or 'production' to enable defaults for this environment.('mode' 选项还未设置。将 'mode' 选项设置为 'development' 或 'production'，来启用环境默认值。)

*通过向 npm run build 命令和你的参数之间添加两个中横线，可以将自定义参数传递给 webpack，例如：npm run build -- --colors。*

### 结论

现在，你已经实现了一个基本的构建过程，你应该移至下一章节的 [Asset Management](https://www.webpackjs.com/guides/asset-management) 指南，以了解如何通过 webpack 来管理资源，例如图片、字体。此刻你的项目应该和如下类似：

**project**

webpack-demo

|- package.json

|- webpack.config.js

|- /dist

|- bundle.js

|- index.html

|- /src

|- index.js

|- /node\_modules

*如果你使用的是 npm 5，你可能还会在目录中看到一个 package-lock.json 文件。*

如果你想要了解 webpack 的设计思想，你应该查看 [basic concepts](https://www.webpackjs.com/concepts) 和 [configuration](https://www.webpackjs.com/configuration) 页面。此外，[API](https://www.webpackjs.com/api) 章节可以深入了解 webpack 提供的各种接口。

压缩图片（**image-webpack-loader/img-loader**）

注意：配置内容当中的 **quality** 属性配置的值越大，那么图片将会被压缩的越小那么图片的清晰度也会随之变模糊

Css 代码压缩（optimize-css-assets-webpack-plugin插件配合预处理器 cssnano）

HTML压缩（html-webpack-plugin）

Js 压缩（uglifyjs-webpack-plugin------ webpck4内置会默认压缩）

开启GZIP （CompressionWebpackPlugin 需要nginx配合）

代码分割：

1. 配置多入口

缺点：如果入口文件包含多个相同模块会打包多次

1. 动态引入

基于es6 import（）语法，import（） 会返回一个promise对象，意味着当前模块和其子模块都会被分割成一个单独的bundle（用到vue异步组件特性）

自动刷新 （browser-sync）

（webpack-dev-server ）

自动编译和自动刷新浏览器

如何配置多页面应用(AutoWebPlugin)

常用的loader

css-loader读取 合并CSS 文件style-loader把 CSS 内容注入到 JavaScript 里sass-loader 解析sass文件（安装sass-loader，node-sass）postcss-loader自动添加浏览器兼容前缀（postcss.config配置）url-loader将文件转换为base64 URI。vue-loader处理vue文件。

解释webpack关键词：

bundle：是由webpack打包出来的文件

chunk：是指webpack在进行模块依赖分析的时候，代码分割出来的代码块

module：是开发中的单个模块