

05 基础概念和常见配置项介绍（一）

更新时间：2019-06-21 09:49:29



人生太短，要干的事太多，我要争分夺秒。

——爱迪生

本节介绍下跟 Webpack 配置相关的概念，以及介绍一个简单并且常用的配置项。

webpack.config.js 配置文件

Webpack 是可配置的模块打包工具，我们可以通过修改 Webpack 的配置文件（`webpack.config.js`）来对 Webpack 进行配置，Webpack 的配置文件是遵循 Node.js 的 CommonJS 模块规范的，即：

- 通过 `require()` 语法导入其他文件或者使用 Node.js 内置的模块
- 普通的 JavaScript 编写语法，包括变量、函数、表达式等

说白了，`webpack.config.js` 是一个 Node.js 的模块。

简单的 `webpack.config.js` 示例

```
const path = require('path');

module.exports = {
  mode: 'development',
  entry: './foo.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'foo.bundle.js'
  }
};
```

上面示例中，使用 CommonJS 的 `require` 引入 Node.js 内置的 `path` 模块，然后通过 `module.exports` 将 Webpack 的配置导出。

Tips: Webpack 的配置是一个 Node.js 模块，所以并不只是 JSON 对象。

Webpack 配置支持多种语言

Webpack 不仅仅支持 js 配置，还支持 ts（TypeScript）、CoffeeScript 甚至 JSX 语法的配置，不同语言其实核心配置项都不变，只不过语法不同而已，本专栏都是 JavaScript 语法的配置。

除了配置文件的语法多样之外，对于配置的类型也是多样的，最常见的是直接作为一个对象来使用，除了使用对象，Webpack 还支持函数、Promise 和多配置数组。

函数类型的 Webpack 配置

如果我们只使用一个配置文件来区分生产环境（production）和开发环境（development），则可以使用函数类型的 Webpack 配置，函数类型的配置必须返回一个配置对象，如下面：

```
module.exports = (env, argv) => {
  return {
    mode: env.production ? 'production' : 'development',
    devtool: env.production ? 'source-maps' : 'eval',
    plugins: [
      new TerserPlugin({
        terserOptions: {
          compress: argv['optimize-minimize'] // 只有传入 -p 或 --optimize-minimize
        }
      })
    ]
  };
};
```

Webpack 配置函数接受两个参数 env 和 argv：分别对应着环境对象和 Webpack-CLI 的命令行选项，例如上面代码中的 --optimize-minimize。

Promise 类型的 Webpack 配置

如果需要异步加载一些 Webpack 配置需要做的变量，那么可以使用 Promise 的方式来做 Webpack 的配置，具体方式如下：

```
module.exports = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve({
        entry: './app.js'
        /* ... */
      });
    }, 5000);
  });
};
```

多配置数组

在一些特定的场景，我们可能需要一次打包多次，而多次打包中有一些通用的配置，这时候可以使用配置数组的方式，将两次以上的 Webpack 配置以数组的形式导出：

```
module.exports = [
  {
    mode: 'production'
    // 配置1
  },
  {
    // 配置2
  }
];
```

配置的使用

默认情况下，Webpack 会查找执行目录下面的 `webpack.config.js` 作为配置，如果需要指定某个配置文件，可以使用下面的命令：

```
webpack --config webpack.config.js
```

如果 Webpack 不是全局安装，则可以在项目目录下实行：

```
node ./node_modules/webpack/bin/webpack --config webpack.config.js
```

或者使用 `npx`

```
npx webpack --config webpack.config.js
```

Webpack 常见名词解释

讲完 Webpack 配置文件，下面讲下配置文件中的配置项。当我们谈论 Webpack 的时候，往往会提到下面的名词：

参数	说明
entry	项目入口
module	开发中每一个文件都可以看做 module，模块不局限于 js，也包含 css、图片等
chunk	代码块，一个 chunk 可以由多个模块组成
loader	模块转化器，模块的处理器，对模块进行转换处理
plugin	扩展插件，插件可以处理 chunk，也可以对最后的打包结果进行处理，可以完成 loader 完不成的任务
bundle	最终打包完成的文件，一般就是和 chunk 一一对应的关系，bundle 就是对 chunk 进行任意压缩打包等处理后的产出

mode 模式

Webpack4.0 开始引入了 `mode` 配置，通过配置 `mode=development` 或者 `mode=production` 来制定是开发环境打包，还是生产环境打包，比如生产环境代码需要压缩，图片需要优化，Webpack 默认 `mode` 是生产环境，即 `mode=production`。

除了在配置文件中设置 `mode`：

```
module.exports = {
  mode: 'development'
};
```

还可以在命令行中设置 `mode`：

```
npx webpack --config webpack.config.entry.js --mode development
```

下面的内容及其后续的章节内容，如没有声明，则以 `development` 方式来做演示，这样方便查看输出的结果。

Webpack 的入口（entry）和输出（output）

通过前面的文章我们已经了解到：**webpack** 是一个模块打包工具，能够从一个需要处理的 **JavaScript** 文件开始，构建一个依赖关系图（**dependency graph**），该图映射到了项目中每个模块，然后将这个依赖关系图输出到一个或者多个 **bundle** 中。

从上面文字的认识，可以轻易的得到 **Webpack** 的两个核心概念：**entry** 和 **output**，即入口和输出，**Webpack** 是从指定的入口文件（**entry**）开始，经过加工处理，最终按照 **output** 设定输出固定内容的 **bundle**；而这个加工处理的过程，就用到了 **loader** 和 **plugin** 两个工具；**loader** 是源代码的处理器，**plugin** 解决的是 **loader** 处理不了的事情。今天重点介绍下 **entry** 和 **output**，在后面文章在介绍 **loader** 和 **plugin**。

context

在介绍 **entry** 之前，介绍下 **context**（上下文），**context** 即项目打包的相对路径上下文，如果指定了 **context="/User/test/webpack"**，那么我们设置的 **entry** 和 **output** 的相对路径都是相对于 **/User/test/webpack** 的，包括在 **JavaScript** 中引入模块也是从这个路径开始的。由于 **context** 的作用，决定了 **context** 值必须是一个绝对路径。

```
// webpack.config.js
module.exports = {
  context: '/Users/test/webpack'
};
```

Tips: 在实际开发中 **context** 一般不需要配置，不配置则默认为 **process.cwd()** 即工作目录。

工作目录（英语：**Working directory**），计算机用语。使用者在作业系统内所在的目录，使用者可在此用相对档名存取档案——维基百科。

entry入口

Webpack 的 **entry** 支持多种类型，包括字符串、对象、数组。从作用上来说，包括了单文件入口和多文件入口两种方式。

单文件入口

单文件的用法如下：

```
module.exports = {
  entry: 'path/to/my/entry/file.js'
};
// 或者使用对象方式
module.exports = {
  entry: {
    main: 'path/to/my/entry/file.js'
  }
};
```

单文件入口可以快速创建一个只有单一文件入口的情况，例如 **library** 的封装，但是单文件入口的方式相对来说比较简单，在扩展配置的时候灵活性较低。

entry 还可以传入包含文件路径的数组，当 **entry** 为数组的时候也会合并输出，例如下面的配置：

```
module.exports = {
  mode: 'development',
  entry: ['./src/app.js', './src/home.js'],
  output: {
    filename: 'array.js'
  }
};
```

Tips: 上面配置无论是字符串还是字符串数组的 `entry`，实际上都是只有一个入口，但是在打包产出上会有差异：

- 1.如果直接是 `string` 的形式，那么 `webpack` 就会直接把该 `string` 指定的模块（文件）作为入口模块
- 2.如果是数组 `[string]` 的形式，那么 `webpack` 会自动生成另外一个入口模块，并将数组中每个元素指定的模块（文件）加载进来，并将最后一个模块的 `module.exports` 作为入口模块的 `module.exports` 导出。这部分会在「原理篇：打包产出小节」继续做详细介绍。

多文件入口

多文件入口是使用对象语法来通过支持多个 `entry`，多文件入口的对象语法相对于单文件入口，具有较高的灵活性，例如多页应用、页面模块分离优化。多文件入口的语法如下：

```
module.exports = {
  entry: {
    home: 'path/to/my/entry/home.js',
    search: 'path/to/my/entry/search.js',
    list: 'path/to/my/entry/list.js'
  }
};
```

上面的语法将 `entry` 分成了 3 个独立的入口文件，这样会打包出来三个对应的 `bundle`，在后面的文章还会介绍使用 `splitChunks` 抽离一个项目中多个 `entry` 的公共代码。

Tips: 对于一个 `HTML` 页面，我们推荐只有一个 `entry`，通过统一的入口，解析出来的依赖关系更方便管理和维护。

output 输出

`webpack` 的 `output` 是指定了 `entry` 对应文件编译打包后的输出 `bundle`。`output` 的常用属性是：

- `path`：此选项制定了输出的 `bundle` 存放的路径，比如 `dist`、`output` 等
- `filename`：这个是 `bundle` 的名称
- `publicPath`：指定了一个在浏览器中被引用的 `URL` 地址，后面详细介绍

后面章节还会继续介绍不同项目的 `output` 其他属性，比如我们要使用 `webpack` 作为库的封装工具，会用到 `library` 和 `libraryTarget` 等。

Tips: 当不指定 `output` 的时候，默认输出到 `dist/main.js`，即 `output.path` 是 `dist`，`output.filename` 是 `main`。

一个 `webpack` 的配置，可以包含多个 `entry`，但是只能有一个 `output`。对于不同的 `entry` 可以通过 `output.filename` 占位符语法来区分，比如：

```
module.exports = {
  entry: {
    home: 'path/to/my/entry/home.js',
    search: 'path/to/my/entry/search.js',
    list: 'path/to/my/entry/list.js'
  },
  output: {
    filename: '[name].js',
    path: __dirname + '/dist'
  }
};
```

其中 `[name]` 就是占位符，它对应的是 `entry` 的 `key`（`home`、`search`、`list`），所以最终输出结果是：

```
path/to/my/entry/home.js → dist/home.js
path/to/my/entry/search.js → dist/search.js
path/to/my/entry/list.js → dist/list.js
```

我将 `Webpack` 目前支持的占位符列出来：

占位符	含义
<code>[hash]</code>	模块标识符的 <code>hash</code>
<code>[chunkhash]</code>	<code>chunk</code> 内容的 <code>hash</code>
<code>[name]</code>	模块名称
<code>[id]</code>	模块标识符
<code>[query]</code>	模块的 <code>query</code> ，例如，文件名？后面的字符串
<code>[function]</code>	一个 <code>return</code> 出一个 <code>string</code> 作为 <code>filename</code> 的函数

`[hash]` 和 `[chunkhash]` 的长度可以使用 `[hash:16]`（默认为 `20`）来指定。或者，通过指定 `output.hashDigestLength` 在全局配置长度，那么他们之间有什么区别？

- `[hash]`：是整个项目的 `hash` 值，其根据每次编译内容计算得到，每次编译之后都会生成新的 `hash`，即修改任何文件都会导致所有文件的 `hash` 发生改变；在一个项目中虽然入口不同，但是 `hash` 是相同的；`hash` 无法实现前端静态资源在浏览器上长缓存，这时候应该使用 `chunkhash`；
- `[chunkhash]`：根据不同的入口文件（`entry`）进行依赖文件解析，构建对应的 `chunk`，生成相应的 `hash`；只要组成 `entry` 的模块文件没有变化，则对应的 `hash` 也是不变的，所以一般项目优化时，会将公共库代码拆分到一起，因为公共库代码变动较少的，使用 `chunkhash` 可以发挥最长缓存的作用；
- `[contenthash]`：使用 `chunkhash` 存在一个问题，当在一个 `JS` 文件中引入了 `CSS` 文件，编译后它们的 `hash` 是相同的。而且，只要 `JS` 文件内容发生改变，与其关联的 `CSS` 文件 `hash` 也会改变，针对这种情况，可以把 `CSS` 从 `JS` 中使用 `mini-css-extract-plugin` 或 `extract-text-webpack-plugin` 抽离出来并使用 `contenthash`。

`[hash]`、`[chunkhash]` 和 `[contenthash]` 都支持 `[xxx:length]` 的语法。

Tips: 占位符是可以组合使用的，例如 `[name]-[hash:8]`

output.publicPath

对于使用 `<script>` 和 `<link>` 标签时，当文件路径不同于他们的本地磁盘路径（由 `output.path` 指定）时，`output.publicPath` 被用来作为 `src` 或者 `link` 指向该文件。这种做法在需要将静态文件放在不同的域名或者 `CDN` 上面的时候是很有用的。


```
module.exports = {
  output: {
    path: '/home/git/public/assets',
    publicPath: '/assets/'
  }
};
```

则输出:

```
<head>
  <link href="/assets/logo.png" />
</head>
```

上面的 `/assets/logo.png` 就是根据 `publicPath` 输出的, `output.path` 制定了输出到本地磁盘的路径, 而 `output.publicPath` 则作为实际上线到服务器之后的 url 地址。所以我们在上 CDN 的时候可以这样配置:

```
module.exports = {
  output: {
    path: '/home/git/public/assets',
    publicPath: 'http://cdn.example.com/assets/'
  }
};
```

则输出:

```
<head>
  <link href="http://cdn.example.com/assets/logo.png" />
</head>
```

output.library

如果我们打包的目的是生成一个供别人使用的库, 那么可以使用 `output.library` 来指定库的名称, 库的名称支持占位符和普通字符串:

```
module.exports = {
  output: {
    library: 'myLib' // '[name]'
  }
};
```

output.libraryTarget

使用 `output.library` 确定了库的名称之后, 还可以使用 `output.libraryTarget` 指定库打包出来的规范, `output.libraryTarget` 取值范围为: `var`、`assign`、`this`、`window`、`global`、`commonjs`、`commonjs2`、`commonjs-module`、`amd`、`umd`、`umd2`、`jsonp`, 默认是 `var`, 下面通过打包后的代码不同, 来看下差别。

```
// var config
{
  output: {
    library: 'myLib',
    filename: 'var.js',
    libraryTarget: 'var'
  }
}
// output
var myLib = (function(modules) {})([
  './src/index.js': function(module, exports) {}
]);
// =====
// assign config
{
  output: {
    library: 'myLib',
```

```

        filename: 'assign.js',
        libraryTarget: 'assign'
    }
}
// output: 少了个 var
myLib = (function(modules) {}){(
    './src/index.js': function(module, exports) {}
)};
// =====
// this config
{
    output: {
        library: 'myLib',
        filename: 'this.js',
        libraryTarget: 'this'
    }
}
// output
this["myLib"] = (function(modules) {}){(
    './src/index.js': function(module, exports) {}
)};
// =====
// window config
{
    output: {
        library: 'myLib',
        filename: 'window.js',
        libraryTarget: 'window'
    }
}
// output
window["myLib"] = (function(modules) {}){(
    './src/index.js': function(module, exports) {}
)};
// =====
// global config
{
    output: {
        library: 'myLib',
        filename: 'global.js',
        libraryTarget: 'global'
    }
}
// output: 注意 target=node 的时候才是 global, 默认 target=web下global 为 window
window["myLib"] = (function(modules) {}){(
    './src/index.js': function(module, exports) {}
)};
// =====
// commonjs config
{
    output: {
        library: 'myLib',
        filename: 'commonjs.js',
        libraryTarget: 'commonjs'
    }
}
// output
exports["myLib"] = (function(modules) {}){(
    './src/index.js': function(module, exports) {}
)};
// =====
// amd config
{
    output: {
        library: 'myLib',
        filename: 'amd.js',
        libraryTarget: 'amd'
    }
}
// output
define('myLib', [], function() {
    return (function(modules) {}){(

```



```

        './src/index.js': function(module, exports) {}
    });
});

// =====
// umd config
{
    output: {
        library: 'myLib',
        filename: 'umd.js',
        libraryTarget: 'umd'
    }
}
// output
(function webpackUniversalModuleDefinition(root, factory) {
    if (typeof exports === 'object' && typeof module === 'object') module.exports = factory();
    else if (typeof define === 'function' && define.amd) define([], factory);
    else if (typeof exports === 'object') exports['myLib'] = factory();
    else root['myLib'] = factory();
})(window, function() {
    return (function(modules) {})(({
        './src/index.js': function(module, exports) {}
    }));
});

// =====
// commonjs2 config
{
    output: {
        library: 'myLib',
        filename: 'commonjs2.js',
        libraryTarget: 'commonjs2'
    }
}
// output
module.exports = (function(modules) {})(({
    './src/index.js': function(module, exports) {}
}));

// =====
// umd2 config
{
    output: {
        library: 'myLib',
        filename: 'umd2.js',
        libraryTarget: 'umd2'
    }
}
// output
(function webpackUniversalModuleDefinition(root, factory) {
    if (typeof exports === 'object' && typeof module === 'object') module.exports = factory();
    else if (typeof define === 'function' && define.amd) define([], factory);
    else if (typeof exports === 'object') exports['myLib'] = factory();
    else root['myLib'] = factory();
})(window, function() {
    return (function(modules) {})(({
        './src/index.js': function(module, exports) {
        }
    }));
});

// =====
// commonjs-module config
{
    output: {
        library: 'myLib',
        filename: 'commonjs-module.js',
        libraryTarget: 'commonjs-module'
    }
}
// =====
// output
module.exports = (function(modules) {})(({
    './src/index.js': function(module, exports) {}
}));

// =====

```

```
// jsonp config
{
  output: {
    library: 'myLib',
    filename: 'jsonp.js',
    libraryTarget: 'jsonp'
  }
}
// output
myLib((function(modules) {}))({
  './src/index.js': function(module, exports) {}
}));
```

注意：`libraryTarget=global` 的时候，如果 `target=node` 才是 `global`，默认 `target=web` 下 `global` 为 `window`，保险起见可以使用 `this`。

下面介绍下跟 `output` 输出相关的三个配置项：`externals`，`target` 和 `devtool`

externals

`externals` 配置项用于去除输出的打包文件中依赖的某些第三方 `js` 模块（例如 `jquery`，`vue` 等等），减小打包文件的体积。该功能通常在开发自定义 `js` 库（`library`）的时候用到，用于去除自定义 `js` 库依赖的其他第三方 `js` 模块。这些被依赖的模块应该由使用者提供，而不应该包含在 `js` 库文件中。例如开发一个 `jQuery` 插件或者 `Vue` 扩展，不需要把 `jQuery` 和 `Vue` 打包进我们的 `bundle`，引入库的方式应该交给使用者。

所以，这里就有个重要的问题，使用者应该怎么提供这些被依赖的模块给我们的 `js` 库（`library`）使用呢？这就要看我们的 `js` 库的导出方式是什么，以及使用者采用什么样的方式使用我们的库。例如：

js library 导出方式	output.libraryTarget	使用者引入方式	使用者提供给被依赖模块的方式
默认的导出方式	output.libraryTarget='var'	只能以 <code><script></code> 标签的形式引入我们的库	只能以全局变量的形式提供这些被依赖的模块
commonjs	output.libraryTarget='commonjs'	只能按照 commonjs 的规范引入我们的库	被依赖模块需要按照 commonjs 规范引入
amd	output.libraryTarget='amd'	只能按照 amd 规范引入	被依赖模块需要按照 amd 规范引入
umd	output.libraryTarget='umd'	可以用 <code><script></code> 、commonjs、amd 引入	被依赖模块需要按照对应方式引入

如果不是在开发一个 `js` 库，即没有设置 `output.library`，`output.libraryTarget` 等配置信息，那么我们生成的打包文件只能以 `<script>` 标签的方式在页面中引入，因此那些被去除的依赖模块也只能以全局变量的方式引入。

target

在项目开发中，我们不仅仅是开发 `web` 应用，还可能开发的是 `Node.js` 服务应用、或者 `electron` 这类跨平台桌面应用，这时候因为对应的宿主环境不同，所以在构建的时候需要特殊处理。`webpack` 中可以通过设置 `target` 来指定构建的目标（`target`）。

```
module.exports = {
  target: 'web' // 默认是 web，可以省略
};
```

`target` 的值有两种类型：`string` 和 `function`。

`string` 类型支持下面的七种：

- `web`：默认，编译为类浏览器环境里可用；

- `node`：编译为类 Node.js 环境可用（使用 Node.js require 加载 chunk）；
- `async-node`：编译为类 Node.js 环境可用（使用 fs 和 vm 异步加载分块）；
- `electron-main`：编译为 Electron 主进程；
- `electron-renderer`：编译为 Electron 渲染进程；
- `node-webkit`：编译为 Webkit 可用，并且使用 jsonp 去加载分块。支持 Node.js 内置模块和 nw.gui 导入（实验性质）；
- `webworker`：编译成一个 WebWorker。

后面章节介绍 webpack 特殊项目类型配置的时候还会介绍 `target` 相关的用法。

除了 `string` 类型，`target` 还支持 `function` 类型，这个函数接收一个 `compiler` 作为参数，如下面代码可以用来增加插件：

```
const webpack = require('webpack');

const options = {
  target: compiler => {
    compiler.apply(new webpack.JsonpTemplatePlugin(options.output), new webpack.LoaderTargetPlugin('web'));
  }
};
```

devtool

`devtool` 是用来控制怎么显示 `sourcemap`，通过 `sourcemap` 我们可以快速还原代码的错误位置。

但是由于 `sourcemap` 包含的数据量较大，而且生成算法需要计算量支持，所以 `sourcemap` 的生成会消耗打包的时间，下面的表格整理了不同的 `devtool` 值对应不同的 `sourcemap` 类型对应打包速度和特点。

devtool	构建速度	重新构建速度	生产环境	品质(quality)
留空, none	+++	+++	yes	打包后的代码
eval	+++	+++	no	生成后的代码
cheap-eval-source-map	+	++	no	转换过的代码（仅限行）
cheap-module-eval-source-map	o	++	no	原始源代码（仅限行）
eval-source-map	-	+	no	原始源代码
cheap-source-map	+	o	no	转换过的代码（仅限行）
cheap-module-source-map	o	-	no	原始源代码（仅限行）
inline-cheap-source-map	+	o	no	转换过的代码（仅限行）
inline-cheap-module-source-map	o	-	no	原始源代码（仅限行）
source-map	-	-	yes	原始源代码
inline-source-map	-	-	no	原始源代码
hidden-source-map	-	-	yes	原始源代码
nosources-source-map	-	-	yes	无源代码内容

+++ 非常快速, ++ 快速, + 比较快, o 中等, - 比较慢, -- 慢

一般在实际项目中，我个人推荐生产环境不使用或者使用 `source-map`（如果有 `Sentry` 这类错误跟踪系统），开发环境使用 `cheap-module-eval-source-map`。

小结

本小节从 `webpack` 的配置文件 `webpack.config.js` 基本语法开始，分别介绍了配置的基本用法，`mode`、`context`、`entry`、`output`、`target` 等 `webpack` 中的基础概念。希望大家读完本小节内容之后，能够动手实际操作一下。在记忆方面，可以

本小节 Webpack 相关面试题：

1. Webpack 的配置有几种写法，分别可以应用到什么场景？
2. 我们要开发一个 jQuery 插件、Vue 组件等，需要怎么配置 Webpack？
3. Webpack 的占位符 `[hash]`、`[chunkhash]` 和 `[contenthash]` 有什么区别和联系？最佳实践是什么？
4. Webpack 的 SourceMap 有几种形式？分别有什么特点？SourceMap 配置的最佳实践是什么？
5. 什么是 bundle，什么是 chunk，什么是 module？

专栏代码已经整理好给大家共享出来：

<https://github.com/ksky521/webpack-tutorial>



04 使用 webpack-cli 体验零配置打包

06 基础概念和常见配置项介绍
(二)



无淘宝 无论坛 被骗 勿扰 更多 超清 加密 视频 资料 联系 沫沫酱 qq 672990394 价格 原价*0.1