

概念

本质上，webpack 是一个现代 JavaScript 应用程序的静态模块打包工具。当 webpack 处理应用程序时，它会在内部构建一个 依赖图(dependency graph)，此依赖图会映射项目所需的每个模块，并生成一个或多个 *bundle*。

可以在 [这里](#) 了解更多关于 JavaScript 模块和 webpack 模块的信息。

从 v4.0.0 开始，webpack 可以不用再引入一个配置文件来打包项目，然而，但它仍然有着高度可配置性，可以很好满足你的需求。

在开始前你需要先理解一些核心概念：

- 入口(entry)
- 输出(output)
- loader
- 插件(plugin)
- 模式(mode)
- 浏览器兼容性(browser compatibility)

本文档旨在给出这些概念的高度概述，同时提供具体概念的详尽相关用例。

为了更好地理解模块打包工具背后的理念，以及在底层它们是如何运作的，请参考以下资源：

- 手动打包一个应用程序
- 实时创建一个简单打包工具
- 一个简单打包工具的详细说明

入口(entry)

入口起点(entry point)指示 webpack 应该使用哪个模块，来作为构建其内部 依赖图(dependency graph) 的开始。进入入口起点后，webpack 会找出有哪些模块和库是入口起点（直接和间接）依赖的。

默认值是 `./src/index.js`，但你可以通过在 `webpack configuration` 中配置 `entry` 属性，来指定一个（或多个）不同的入口起点。例如：

webpack.config.js

```
module.exports = {
  entry: './path/to/my/entry/file.js'
};
```

在 [入口起点](#) 章节可以了解更多信息。

输出(output)

output 属性告诉 webpack 在哪里输出它所创建的 *bundle*, 以及如何命名这些文件。主要输出文件的默认值是 `./dist/main.js`, 其他生成文件默认放置在 `./dist` 文件夹中。

你可以通过在配置中指定一个 `output` 字段, 来配置这些处理过程:

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'
  }
};
```

在上面的示例中, 我们通过 `output.filename` 和 `output.path` 属性, 来告诉 webpack *bundle* 的名称, 以及我们想要 *bundle* 生成(emit)到哪里。可能你想要了解在代码最上面导入的 `path` 模块是什么, 它是一个 [Node.js 核心模块](#), 用于操作文件路径。

`output` 属性还有 [许多可配置的特性](#), 如果你想要了解更多其背后的概念, 你可以通过 [阅读输出章节](#) 来了解。

loader

webpack 只能理解 JavaScript 和 JSON 文件。**loader** 让 webpack 能够去处理其他类型的文件, 并将它们转换为有效 模块, 以供应用程序使用, 以及被添加到依赖图中。

注意, loader 能够 `import` 导入任何类型的模块 (例如 `.css` 文件), 这是 webpack 特有的功能, 其他打包程序或任务执行器的可能并不支持。我们认为这种语言扩展是很有必要的, 因为这可以使开发人员创建出更准确的依赖关系图。

在更高层面, 在 webpack 的配置中 **loader** 有两个属性:

1. `test` 属性, 用于标识出应该被对应的 loader 进行转换的某个或某些文件。
2. `use` 属性, 表示进行转换时, 应该使用哪个 loader。

webpack.config.js

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

module.exports = {
  output: {
    filename: 'my-first-webpack.bundle.js'
  },
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  }
};
```

以上配置中，对一个单独的 `module` 对象定义了 `rules` 属性，里面包含两个必须属性：`test` 和 `use`。这告诉 webpack 编译器(compiler) 如下信息：

“嘿，webpack 编译器，当你碰到「在 `require()`/`import` 语句中被解析为 '.txt' 的路径」时，在你对它打包之前，先使用 `raw-loader` 转换一下。”

重要的是要记住，在 webpack 配置中定义 `rules` 时，要定义在 `module.rules` 而不是 `rules` 中。为了使你受益于此，如果没有按照正确方式去做，webpack 会给出警告。

请记住，使用正则表达式匹配文件时，你不要为它添加引号。也就是说，`/\.\txt$/` 与 `'/\.\txt$/'` 不一样。前者指示 webpack 匹配任何以 `.txt` 结尾的文件，后者指示 webpack 匹配具有绝对路径 `'.txt'` 的单个文件；这可能不符合你的意图。

在使用 loader 时，可以阅读 [loader 章节](#) 查看更深入的自定义配置。

插件(plugin)

loader 用于转换某些类型的模块，而插件则可以用于执行范围更广的任务。包括：打包优化，资源管理，注入环境变量。

插件接口(plugin interface) 功能极其强大，可以用来处理各种各样的任务。

想要使用一个插件，你只需要 `require()` 它，然后把它添加到 `plugins` 数组中。多数插件可以通过选项(option)自定义。你也可以在一个配置文件中因为不同目的而多次使用同一个插件，这时需要通过使用 `new` 操作符来创建它的一个实例。

webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); // 通过 npm 安装
const webpack = require('webpack'); // 用于访问内置插件
```

```
module.exports = {
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};
```

在上面的示例中，`html-webpack-plugin` 为应用程序生成 HTML 一个文件，并自动注入所有生成的 bundle。

webpack 提供许多开箱可用的插件！查阅 [插件列表](#) 获取更多。

在 webpack 配置中使用插件是简单直接的，然而也有很多值得我们进一步探讨的用例。查看[这里了解更多](#)。

模式(mode)

通过选择 `development`, `production` 或 `none` 之中的一个，来设置 `mode` 参数，你可以启用 webpack 内置在相应环境下的优化。其默认值为 `production`。

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

查看 [模式配置](#) 章节了解其详细内容和每个值所作的优化。

浏览器兼容性(browser compatibility)

webpack 支持所有符合 [ES5](#) 标准的浏览器（不支持 IE8 及以下版本）。webpack 的 `import()` 和 `require.ensure()` 需要 `Promise`。如果你想要支持旧版本浏览器，在使用这些表达式之前，还需要 提前加载 [polyfill](#)。

入口起点(entry points)

正如我们在 [起步](#) 中提到的，在 webpack 配置中有多种方式定义 `entry` 属性。除了解释为什么它可能非常有用，我们还将向你展示如何去配置 `entry` 属性。

单个入口（简写）语法

用法: `entry: string|Array<string>`

webpack.config.js

```
module.exports = {
  entry: './path/to/my/entry/file.js'
};
```

`entry` 属性的单个入口语法，是下面的简写：

webpack.config.js

```
module.exports = {
  entry: {
    main: './path/to/my/entry/file.js'
  }
};
```

当你向 `entry` 传入一个数组时会发生什么？向 `entry` 属性传入文件路径数组，将创建出一个 **多主入口(multi-main entry)**。在你想要一次注入多个依赖文件，并且将它们的依赖导向(graph)到一个 chunk 时，这种方式就很有用。

当你正在寻找为「只有一个入口起点的应用程序或工具（即 library）」快速设置 webpack 配置的时候，这会是个很不错的选择。然而，使用此语法在扩展配置时有失灵活性。

对象语法

用法: `entry: {[entryChunkName: string]: string|Array<string>}`

webpack.config.js

```
module.exports = {
  entry: {
    app: './src/app.js',
    adminApp: './src/adminApp.js'
  }
};
```

对象语法会比较繁琐。然而，这是应用程序中定义入口的最可扩展的方式。

“webpack 配置的可扩展”是指，这些配置可以重复使用，并且可以与其他配置组合使用。这是一种流行的技术，用于将关注点从环境(environment)、构建目标(build target)、运行时(runtime)中分离。然后使用专门的工具（如 [webpack-merge](#)）将它们合并起来。

常见场景

以下列出一些入口配置和它们的实际用例：

分离 app(应用程序) 和 vendor(第三方库) 入口

在 webpack < 4 的版本中，通常将 vendor 作为单独的入口起点添加到 entry 选项中，以将其编译为单独的文件（与 CommonsChunkPlugin 结合使用）。而在 webpack 4 中不鼓励这样做。而是使用 `optimization.splitChunks` 选项，将 vendor 和 app(应用程序) 模块分开，并为其创建一个单独的文件。不要为 vendor 或其他不是执行起点创建 entry。

多页面应用程序

webpack.config.js

```
module.exports = {
  entry: {
    pageOne: './src/pageOne/index.js',
    pageTwo: './src/pageTwo/index.js',
    pageThree: './src/pageThree/index.js'
  }
};
```

这是什么？我们告诉 webpack 需要三个独立分离的依赖图（如上面的示例）。

为什么？在多页面应用程序中，服务器会传输一个新的 HTML 文档给你的客户端。页面重新加载此新文档，并且资源被重新下载。然而，这给了我们特殊的机去做很多事：

- 使用 `optimization.splitChunks` 为页面间共享的应用程序代码创建 bundle。由于入口起点增多，多页应用能够复用入口起点之间的大量代码/模块，从而可以极大地从这些技术中受益。

根据经验：每个 HTML 文档只使用一个入口起点。

输出(output)

配置 `output` 选项可以控制 webpack 如何向硬盘写入编译文件。注意，即使可以存在多个 `entry` 起点，但只指定一个 `output` 配置。

用法(Usage)

在 webpack 中配置 `output` 属性的最低要求是，将它的值设置为一个对象，包括以下属性：

- `filename` 用于输出文件的文件名。

webpack.config.js

```
module.exports = {
  output: {
    filename: 'bundle.js',
  }
};
```

此配置将一个单独的 `bundle.js` 文件输出到 `dist` 目录中。

多个入口起点

如果配置创建了多个单独的 "chunk"（例如，使用多个入口起点或使用像 `CommonsChunkPlugin` 这样的插件），则应该使用 [占位符\(substitutions\)](#) 来确保每个文件具有唯一的名称。

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

// 写入到硬盘: ./dist/app.js, ./dist/search.js
```

高级进阶

以下是对资源使用 CDN 和 hash 的复杂示例：

config.js

```
module.exports = {
  //...
  output: {
    path: '/home/proj/cdn/assets/[hash]',
    publicPath: 'http://cdn.example.com/assets/[hash]/'
  }
};
```

如果在编译时，不知道最终输出文件的 `publicPath` 是什么地址，则可以将其留空，并且在运行时通过入口起点文件中的 `__webpack_public_path__` 动态设置。

```
__webpack_public_path__ = myRuntimePublicPath;
// 应用程序入口的其余部分
```

模式(mode)

提供 `mode` 配置选项，告知 webpack 使用相应环境的内置优化。

string

可能的值有：`none`, `development` 或 `production`（默认）。

用法

只需在配置对象中提供 `mode` 选项：

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

或者从 CLI 参数中传递：

```
webpack --mode=production
```

支持以下字符串值：

选项	development
描述	会将 <code>DefinePlugin</code> 中 <code>process.env.NODE_ENV</code> 的值设置为 <code>development</code> 。启用 <code>NamedChunksPlugin</code> 和 <code>NamedModulesPlugin</code> 。
选项	production
描述	会将 <code>DefinePlugin</code> 中 <code>process.env.NODE_ENV</code> 的值设置为 <code>production</code> 。启用 <code>FlagDependencyUsagePlugin</code> , <code>FlagIncludedChunksPlugin</code> , <code>ModuleConcatenationPlugin</code> , <code>NoEmitOnErrorsPlugin</code> , <code>OccurrenceOrderPlugin</code> , <code>SideEffectsFlagPlugin</code> 和 <code>TerserPlugin</code> 。
选项	none
描述	退出任何默认优化选项

如果没有设置，webpack 会将 `mode` 的默认值设置为 `production`。模式支持的值为：

记住，设置 `NODE_ENV` 并不会自动地设置 `mode`。

mode: development

```
// webpack.development.config.js
module.exports = {
+ mode: 'development',
- devtool: 'eval',
- cache: true,
- performance: {
-   hints: false
- },
- output: {
-   pathinfo: true
- },
- optimization: {
-   namedModules: true,
-   namedChunks: true,
-   nodeEnv: 'development',
-   flagIncludedChunks: false,
-   occurrenceOrder: false,
-   sideEffects: false,
-   usedExports: false,
-   concatenateModules: false,
-   splitChunks: {
-     hidePathInfo: false,
-     minSize: 10000,
-     maxAsyncRequests: Infinity,
-     maxInitialRequests: Infinity,
-   },
-   noEmitOnErrors: false,
-   checkWasmTypes: false,
-   minimize: false,
- },
- plugins: [
-   new webpack.NamedModulesPlugin(),
-   new webpack.NamedChunksPlugin(),
-   new webpack.DefinePlugin({ "process.env.NODE_ENV": JSON.stringify("develo
- ]
}
```

mode: production

```
// webpack.production.config.js
module.exports = {
+ mode: 'production',
- performance: {
-   hints: 'warning'
- },
- output: {
-   pathinfo: false
- },
- optimization: {
-   namedModules: false,
-   namedChunks: false,
-   nodeEnv: 'production',
-   flagIncludedChunks: true,
-   occurrenceOrder: true,
```

```

-   sideEffects: true,
-   usedExports: true,
-   concatenateModules: true,
-   splitChunks: {
-     hidePathInfo: true,
-     minSize: 30000,
-     maxAsyncRequests: 5,
-     maxInitialRequests: 3,
-   },
-   noEmitOnErrors: true,
-   checkWasmTypes: true,
-   minimize: true,
- },
- plugins: [
-   new TerserPlugin(/* ... */),
-   new webpack.DefinePlugin({ "process.env.NODE_ENV": JSON.stringify("p
-   new webpack.optimize.ModuleConcatenationPlugin(),
-   new webpack.NoEmitOnErrorsPlugin()
- ]
}

```

mode: none

```

// webpack.custom.config.js
module.exports = {
+ mode: 'none',
- performance: {
-   hints: false
- },
- optimization: {
-   flagIncludedChunks: false,
-   occurrenceOrder: false,
-   sideEffects: false,
-   usedExports: false,
-   concatenateModules: false,
-   splitChunks: {
-     hidePathInfo: false,
-     minSize: 10000,
-     maxAsyncRequests: Infinity,
-     maxInitialRequests: Infinity,
-   },
-   noEmitOnErrors: false,
-   checkWasmTypes: false,
-   minimize: false,
- },
- plugins: []
}

```

如果要根据 webpack.config.js 中的 **mode** 变量更改打包行为，则必须将配置导出为一个函数，而不是导出为一个对象：

```

var config = {
  entry: './app.js'
  //...
};

```

```
module.exports = (env, argv) => {

  if (argv.mode === 'development') {
    config.devtool = 'source-map';
  }

  if (argv.mode === 'production') {
    //...
  }

  return config;
};
```

loader

loader 用于对模块的源代码进行转换。loader 可以使你在 `import` 或"加载"模块时预处理文件。因此，loader 类似于其他构建工具中“任务(task)”，并提供了处理前端构建步骤的强大方法。loader 可以将文件从不同的语言（如 TypeScript）转换为 JavaScript 或将内联图像转换为 data URL。loader 甚至允许你直接在 JavaScript 模块中 `import` CSS文件！

示例

例如，你可以使用 loader 告诉 webpack 加载 CSS 文件，或者将 TypeScript 转为 JavaScript。为此，首先安装相对应的 loader：

```
npm install --save-dev css-loader  
npm install --save-dev ts-loader
```

然后指示 webpack 对每个 `.css` 使用 `css-loader`，以及对所有 `.ts` 文件使用 `ts-loader`：

webpack.config.js

```
module.exports = {  
  module: {  
    rules: [  
      { test: /\.css$/, use: 'css-loader' },  
      { test: /\.ts$/, use: 'ts-loader' }  
    ]  
  }  
};
```

使用 loader

在你的应用程序中，有三种使用 loader 的方式：

- **配置 (推荐)**：在 `webpack.config.js` 文件中指定 loader。
- **内联**：在每个 `import` 语句中显式指定 loader。
- **CLI**：在 shell 命令中指定它们。

配置(configuration)

`module.rules` 允许你在 webpack 配置中指定多个 loader。这种方式是展示 loader 的一种简明方式，并且有助于使代码变得简洁和易于维护。同时让你对各个 loader 有个全局概览：

loader 从右到左地取值(evaluate)/执行(execute)。在下面的示例中，从 sass-loader 开

始执行，然后继续执行 `css-loader`，最后以 `style-loader` 为结束。查看 `loader` 功能了解有关 loader 顺序的更多信息。

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          { loader: 'style-loader' },
          {
            loader: 'css-loader',
            options: {
              modules: true
            }
          },
          { loader: 'sass-loader' }
        ]
      }
    ]
  }
};
```

内联(inline)

可以在 `import` 语句或任何等同于 "`import`" 的方法中指定 loader。使用 `!` 将资源中的 loader 分开。每个部分都会相对于当前目录解析。

```
import Styles from 'style-loader!css-loader?modules!./styles.css';
```

使用 `!` 为整个规则添加前缀，可以覆盖配置中的所有 loader 定义。

选项可以传递查询参数，例如 `?key=value&foo=bar`，或者一个 JSON 对象，例如 `?{"key": "value", "foo": "bar"}`。

尽可能使用 `module.rules`，因为这样可以减少源码中样板文件的代码量，并且可以在出错时，更快地调试和定位 loader 中的问题。

CLI

还可以通过 CLI 使用 loader:

```
webpack --module-bind jade-loader --module-bind 'css=style-loader!css-lc
```

这会对 `.jade` 文件使用 `jade-loader`，以及对 `.css` 文件使用 `style-loader` 和 `css-loader`。

loader 特性

- loader 支持链式传递。链中的每个 loader 会将转换应用在已处理过的资源上。一组链式的 loader 将按照相反的顺序执行。链中的第一个 loader 将其结果（也就是应用过转换后的资源）传递给下一个 loader，依此类推。最后，链中的最后一个 loader，返回 webpack 期望 JavaScript。
- loader 可以是同步的，也可以是异步的。
- loader 运行在 Node.js 中，并且能够执行任何 Node.js 能做到的操作。
- loader 可以通过 `options` 对象配置（仍然支持使用 `query` 参数来设置选项，但是这种方式已被废弃）。
- 除了常见的通过 `package.json` 的 `main` 来将一个 npm 模块导出为 loader，还可以在 `module.rules` 中使用 `loader` 字段直接引用一个模块。
- 插件(plugin)可以为 loader 带来更多特性。
- loader 能够产生额外的任意文件。

通过 (loader) 预处理函数，loader 为 JavaScript 生态系统提供了更多能力。用户现在可以更加灵活地引入细粒度逻辑，例如：压缩、打包、语言翻译和 [更多其他特性](#)。

解析 loader

loader 遵循 [模块解析](#) 标准。多数情况下，loader 将从 [模块路径](#) 加载（通常是从 `npm install, node_modules` 进行加载）。

通常使用 npm 进行管理，但是也可以将自定义 loader 作为应用程序中的文件。按照约定，loader 通常被命名为 `xxx-loader`（例如 `json-loader`）。更多详细信息请查看 [如何编写 loader?](#)。

插件(plugin)

插件是 webpack 的支柱功能。webpack 自身也是构建于，你在 webpack 配置中用到的相同的插件系统之上！

插件目的在于解决 `loader` 无法实现的其他事。

剖析

webpack 插件是一个具有 `apply` 方法的 JavaScript 对象。`apply` 方法会被 webpack compiler 调用，并且 compiler 对象可在整个编译生命周期访问。

ConsoleLogOnBuildWebpackPlugin.js

```
const pluginName = 'ConsoleLogOnBuildWebpackPlugin';

class ConsoleLogOnBuildWebpackPlugin {
  apply(compiler) {
    compiler.hooks.run.tap(pluginName, compilation => {
      console.log('webpack 构建过程开始！');
    });
  }
}
```

compiler hook 的 `tap` 方法的第一个参数，应该是驼峰式命名的插件名称。建议为此使用一个常量，以便它可以在所有 hook 中复用。

用法

由于插件可以携带参数/选项，你必须在 webpack 配置中，向 `plugins` 属性传入 `new` 实例。

根据你使用 webpack 的需要，这里有多种方式使用插件。

配置

webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); //通过 npm 安装
const webpack = require('webpack'); //访问内置的插件
const path = require('path');

module.exports = {
  entry: './path/to/my/entry/file.js',
  output: {
    filename: 'my-first-webpack.bundle.js',
  },
}
```

```
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.js|jsx$/,
        use: 'babel-loader'
      }
    ]
  },
  plugins: [
    new webpack.ProgressPlugin(),
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};
```

Node API

在使用 Node API 时，还可以通过配置中的 `plugins` 属性传入插件。

some-node-script.js

```
const webpack = require('webpack'); //访问 webpack 运行时(runtime)
const configuration = require('./webpack.config.js');

let compiler = webpack(configuration);

new webpack.ProgressPlugin().apply(compiler);

compiler.run(function(err, stats) {
  // ...
});
```

你知道吗：以上看到的示例和 webpack 自身运行时(runtime) 极其类似。wepback 源码中隐藏有大量使用示例，你可以用在自己的配置和脚本中。

配置(configuration)

你可能已经注意到，很少有 webpack 配置看起来完全相同。这是因为 webpack 的配置文件，是一个导出 webpack 配置对象的 JavaScript 文件。然后 webpack 会根据此配置对象上定义的属性进行处理。

因为 webpack 配置是标准的 Node.js CommonJS 模块，你可以做到以下事情：

- 通过 `require(...)` 导入其他文件
- 通过 `require(...)` 使用 npm 的工具函数
- 使用 JavaScript 控制流表达式，例如 `?:` 操作符
- 对常用值使用常量或变量
- 编写和执行函数，来生成部分配置

请在合适的场景下使用这些功能。

虽然技术上可行，但应避免以下做法：

- 在使用 webpack 命令行接口(CLI)时，访问命令行接口(CLI)参数（应该编写自己的命令行接口(CLI)，或使用 `--env`）
- 导出不确定的值（调用 webpack 两次应该产生同样的输出文件）
- 编写很长的配置（应该将配置拆分为多个文件）

这份文档中得出的最重要的收获是，你的 webpack 配置可以有许多不同的格式和风格。关键在于，为了便于你和你的团队易于理解和维护这些配置，需要保证一致性。

接下来的例子展示了 webpack 配置(webpack configuration)如何即具有表现力，又具有可配置性，这是因为配置即是代码：

基本配置

webpack.config.js

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

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

See: [Configuration section](#) for the all supported configuration options

多个 target

除了可以将单个配置导出为 object, function 或 Promise, 还可以将其导出为多个配置。

查看: [导出多个配置](#)

使用其他配置语言

webpack 接受以多种编程和数据语言编写的配置文件。

查看: [配置语言](#)

模块(module)

在 模块化编程 中，开发者将程序分解为功能离散的 chunk(discrete chunks of functionality)，并称之为_模块_。

每个模块具有比完整程序更小的接触面，使得验证、调试、测试轻而易举。精心编写的_模块_提供了可靠的抽象和封装界限，使得应用程序中每个模块，都具备了条理清楚的设计和明确的目的。

Node.js 从最一开始就支持模块化编程。然而，web 的模块化支持正缓慢到来。在 web 存在多种支持 JavaScript 模块化的工具，这些工具各有优势和限制。webpack 基于从这些系统获得的经验教训，并将_模块_的概念应用于项目中的任何文件。

什么是 webpack 模块

与 Node.js 模块 相比，webpack _模块_能够以各种方式表达它们的依赖关系。下面是一些示例：

- ES2015 import 语句
- CommonJS require() 语句
- AMD define 和 require 语句
- css/sass/less 文件中的 @import 语句。
- 样式(url(...))或 HTML 文件()中的图片链接

webpack 1 需要特定的 loader 来转换 ES2015 import，然而在 webpack 2 中，这一切都是开箱即用的。

支持的模块类型

通过 *loader*，webpack 可以支持以各种语言和预处理器语法编写的模块。*loader* 描述了 webpack 如何处理 非 JavaScript _模块_，并且在 *bundle* 中引入这些_依赖_。 webpack 社区已经为各种流行语言和语言处理器创建了 *loader*，包括：

- CoffecScript
- TypeScript
- ESNext (Babel)
- Sass
- Less
- Stylus

其实还有很多！总的来说，webpack 提供了可定制的、强大和丰富的 API，允许

在任何技术栈中使用 webpack，同时在开发、测试和生产环境下的工作流程中做到无侵入性。

有关完整列表，请参考 [loader 列表](#) 或 编写你自己的 loader。

为什么选择 webpack

想要理解为什么要使用 webpack，我们先回顾下历史，在打包工具出现之前，我们是如何在 web 中使用 JavaScript 的。

在浏览器中运行 JavaScript 有两种方法。第一种方式，引用一些脚本来存放每个功能；此解决方案很难扩展，因为加载太多脚本会导致网络瓶颈。第二种方式，使用一个包含所有项目代码的大型 .js 文件，但是这会导致作用域、文件大小、可读性和可维护性方面的问题。

立即调用函数表达式 - Immediately invoked function expressions

IIFE 解决大型项目的作用域问题；当脚本文件被封装在 IIFE 内部时，你可以安全地拼接或安全地组合所有文件，而不必担心作用域冲突。

这种方式产生出 Make, Gulp, Grunt, Broccoli 或 Brunch 等工具。这些工具称为任务执行器，它们将所有项目文件拼接在一起。

但是，修改一个文件意味着必须重新构建整个文件。拼接可以做到很容易地跨文件重用脚本，但是却使构建结果的优化变得更加困难。如何判断代码是否实际被使用？

即使你只用到 lodash 中的某个函数，也必须在构建结果中加入整个库，然后将它们压缩在一起。如何 treeshake 代码依赖？难以大规模地实现延迟加载代码块，这需要开发人员手动地进行大量工作。

感谢 Node.js，JavaScript 模块诞生了

Node.js 是一个 JavaScript 运行时，可以在浏览器环境之外的计算机和服务器中使用。webpack 运行在 Node.js 中。

当 Node.js 发布时，一个新的时代开始了，它带来了新的挑战。既然不是在浏览器中运行 JavaScript，现在已经没有了可以添加到浏览器中的 html 文件和 script 标签。那么 Node.js 应用程序要如何加载新的代码 chunk 呢？

CommonJS 问世并引入了 `require` 机制，它允许你在当前文件中加载和使用某个模块。导入需要的每个模块，这一开箱即用的功能，帮助我们解决了作用域问题。

npm + Node.js + modules - 大规模分发模块

JavaScript 已经成为一种语言、一个平台和一种快速开发和创建快速应用程序的方式，接管了整个 JavaScript 世界。

但 CommonJS 没有浏览器支持。没有 live binding(实时绑定)。循环引用存在问题。同步执行的模块解析加载器速度很慢。虽然 CommonJS 是 Node.js 项目的绝佳解决方案，但浏览器不支持模块。因而创建了 Browserify, RequireJS 和 SystemJS 等打包工具，允许我们编写能够在浏览器中运行的 CommonJS 模块。

ESM - ECMAScript 模块

来自 Web 项目的好消息是，模块正在成为 ECMAScript 标准的官方功能。然而，浏览器支持不完整，版本迭代速度也不够快，目前还是推荐上面那些早期模块实现。

看起来都不是很好.....

是否可以有一种方式，不仅可以让我们编写模块，而且还支持任何模块格式（至少在我们到达 ESM 之前），并且可以同时处理资源和资产？

这就是 webpack 存在的原因。它是一个工具，可以打包你的 JavaScript 应用程序（支持 ESM 和 CommonJS），可以扩展为支持许多不同的资产，例如：images, fonts 和 stylesheets。webpack 关心性能和加载时间；它始终在改进或添加新功能，例如：异步地加载 chunk 和预取，以便为你的项目和用户提供最佳体验。

模块解析(module resolution)

resolver 是一个库(library)，用于帮助找到模块的绝对路径。一个模块可以作为另一个模块的依赖模块，然后被后者引用，如下：

```
import foo from 'path/to/module';
// 或者
require('path/to/module');
```

所依赖的模块可以是来自应用程序代码或第三方的库(library)。resolver 帮助 webpack 从每个如 `require/import` 语句中，找到需要引入到 bundle 中的模块代码。当打包模块时，webpack 使用 `enhanced-resolve` 来解析文件路径。

webpack 中的解析规则

使用 `enhanced-resolve`，webpack 能够解析三种文件路径：

绝对路径

```
import '/home/me/file';
import 'C:\\\\Users\\\\me\\\\file';
```

由于我们已经取得文件的绝对路径，因此不需要进一步再做解析。

相对路径

```
import '../src/file1';
import './file2';
```

在这种情况下，使用 `import` 或 `require` 的资源文件所在的目录，被认为是上下文目录(context directory)。在 `import/require` 中给定的相对路径，会拼接此上下文路径(context path)，以产生模块的绝对路径。

模块路径

```
import 'module';
import 'module/lib/file';
```

模块将在 `resolve.modules` 中指定的所有目录内搜索。你可以替换初始模块路径，此替换路径通过使用 `resolve.alias` 配置选项来创建一个别名。

一旦根据上述规则解析路径后，resolver 将检查路径是否指向文件或目录。如果路径指向一个文件：

- 如果路径具有文件扩展名，则被直接将文件打包。
- 否则，将使用 `[resolve.extensions]` 选项作为文件扩展名来解析，此选项告诉 resolver 在解析中能够接受哪些扩展名（例如 `.js`, `.jsx`）。

如果路径指向一个文件夹，则采取以下步骤找到具有正确扩展名的正确文件：

- 如果文件夹中包含 `package.json` 文件，则按照顺序查找 `resolve.mainFields` 配置选项中指定的字段。通过 `package.json` 中的第一个字段确定文件路径。
- 如果不存在 `package.json` 文件或者 `package.json` 文件中的 `main` 字段没有返回一个有效路径，则按照顺序查找 `resolve.mainFiles` 配置选项中指定的文件名，看是否能在 `import/require` 目录下匹配到一个存在的文件名。
- 文件扩展名通过 `resolve.extensions` 选项，采用类似的方法进行解析。

webpack 根据构建目标(build target)，为这些选项提供了合理的 默认 配置。

解析 loader

loader 的解析规则，也遵循文件解析的特定规则。但是 `resolveLoader` 配置选项，可以用来为 loader 提供独立的解析规则。

缓存

每次文件系统访问都会被缓存，以便更快触发对同一文件的多个并行或串行请求。在 观察模式(watch mode) 下，只有修改过的文件会从缓存中摘出。如果关闭观察模式，会在每次编译前清理缓存。

有关上述配置的更多信息，请查看 [resolve API](#)。

依赖图(dependency graph)

任何时候，一个文件依赖于另一个文件，webpack 就把此视为文件之间有 依赖关系。这使得 webpack 可以接收非代码资源(non-code asset)（例如 images 或 web fonts），并且可以把它们作为 依赖 提供给你的应用程序。

webpack 从命令行或配置文件中定义的一个模块列表开始，处理你的应用程序。从这些 入口起点 开始，webpack 递归地构建一个 依赖图，这个依赖图包含着应用程序所需的每个模块，然后将所有这些模块打包为少量的 *bundle* - 通常只有一个 - 可由浏览器加载。

对于 *HTTP/1.1* 客户端，由 webpack 打包你的应用程序会极其强大，这是因为 在浏览器发起一个新请求时，它能够减少应用程序必须等待的时间。对于 *HTTP/2*，你还可以使用 代码分离 来实现最佳构建结果。de

manifest

在使用 webpack 构建的典型应用程序或站点中，有三种主要的代码类型：

1. 你或你的团队编写的源码。
2. 你的源码会依赖的任何第三方的 library 或 "vendor" 代码。
3. webpack 的 runtime 和 manifest，管理所有模块的交互。

本文将重点介绍这三个部分中的最后部分，runtime 和 manifest，特别是 manifest。

runtime

runtime，以及伴随的 manifest 数据，主要是指：在浏览器运行过程中，webpack 用来连接模块化应用程序所需的所有代码。它包含：在模块交互时，连接模块所需的加载和解析逻辑。包括：已经加载到浏览器中的连接模块逻辑，以及尚未加载模块的延迟加载逻辑。

manifest

在你的应用程序中，形如 `index.html` 文件、一些 bundle 和各种资源，都必须以某种方式加载和链接到应用程序，一旦被加载到浏览器中。在经过打包、压缩、为延迟加载而拆分为细小的 chunk 这些 webpack 优化之后，你精心安排的 `/src` 目录的文件结构都已经不再存在。所以 webpack 如何管理所有所需模块之间的交互呢？这就是 manifest 数据用途的由来……

当 compiler 开始执行、解析和映射应用程序时，它会保留所有模块的详细要点。这个数据集合称为 "manifest"，当完成打包并发送到浏览器时，runtime 会通过 manifest 来解析和加载模块。无论你选择哪种 模块语法，那些 `import` 或 `require` 语句现在都已经转换为 `__webpack_require__` 方法，此方法指向模块标识符 (module identifier)。通过使用 manifest 中的数据，runtime 将能够检索这些标识符，找出每个标识符背后对应的模块。

问题

所以，现在你应该对 webpack 在幕后工作有一点了解。“但是，这对我有什么影响呢？”，你可能会问。答案是大多数情况下没有。runtime 做完成这些工作：一旦你的应用程序加载到浏览器中，使用 manifest，然后所有内容将展现出魔幻般运行结果。然而，如果你决定通过使用浏览器缓存来改善项目的性能，理解这一过程将突然变得极为重要。

通过使用内容散列(content hash)作为 bundle 文件的名称，这样在文件内容修改时，会计算出新的 hash，浏览器会使用新的名称加载文件，从而使缓存无效。一旦你开始这样做，你会立即注意到一些有趣的行为。即使某些内容明显没有修改，某些 hash 还是会改变。这是因为，注入的 runtime 和 manifest 在每次构建后都会发生变化。

查看管理输出指南的 [manifest](#) 部分，了解如何提取 manifest，并阅读下面的指南，以了解更多长效缓存错综复杂之处。

部署目标(target)

因为服务器和浏览器代码都可以用 JavaScript 编写，所以 webpack 提供了多种部署 *target(目标)*，你可以在你的 webpack 配置对象 中进行设置。

webpack 的 `target` 属性，不要和 `output.libraryTarget` 属性混淆。有关 `output` 属性的更多信息，请查看我们的 [指南](#)。

用法

想要设置 `target` 属性，只需要在你的 webpack 配置中设置 `target` 的值。

webpack.config.js

```
module.exports = {  
  target: 'node'  
};
```

在上面例子中，使用 `node`，webpack 会编译为用于类 Node.js 环境（使用 Node.js 的 `require`，而不是使用任意内置模块（如 `fs` 或 `path`）来加载 chunk）。

每个 `target` 都有各种部署(deployment)/环境(environment)特定的附加项，以支持满足其需求。查看 `target` 可用值。

Further expansion for other popular target values

多个 target

虽然 webpack 不支持向 `target` 传入多个字符串，还是可以通过打包两个单独配置，来创建出一个同构的 library:

webpack.config.js

```
const path = require('path');  
const serverConfig = {  
  target: 'node',  
  output: {  
    path: path.resolve(__dirname, 'dist'),  
    filename: 'lib.node.js'  
  }  
  //...  
};  
  
const clientConfig = {  
  target: 'web', // <== 默认是 'web'，可省略  
  output: {  
    path: path.resolve(__dirname, 'dist'),  
  }  
};
```

```
    filename: 'lib.js'  
}  
//...  
};  
  
module.exports = [ serverConfig, clientConfig ];
```

上面的例子将在 `dist` 文件夹下创建 `lib.js` 和 `lib.node.js` 文件。

资源

从上面的选项能够看出，可以选择多种不同的部署 *target*。下面是一个示例列表，以及可以参考的资源。

- **compare-webpack-target-bundles**: 有关「测试和查看」不同的 webpack *target* 的大量资源。也有大量 bug 报告。
- **Boilerplate of Electron-React Application**: 一个 electron 主进程和渲染进程构建过程的很好的例子。

Need to find up to date examples of these webpack targets being used in live code or boilerplates.

模块热替换(hot module replacement)

模块热替换(HMR - hot module replacement)功能会在应用程序运行过程中，替换、添加或删除 模块，而无需重新加载整个页面。主要是通过以下几种方式，来显著加快开发速度：

- 保留在完全重新加载页面期间丢失的应用程序状态。
- 只更新变更内容，以节省宝贵的开发时间。
- 在源代码中对 CSS/JS 进行修改，会立刻在浏览器中进行更新，这几乎相当于在浏览器 devtools 直接更改样式。

这一切是如何运行的？

让我们从一些不同的角度观察，以了解 HMR 的工作原理.....

在应用程序中

通过以下步骤，可以做到在应用程序中置换(swap in and out)模块：

1. 应用程序要求 HMR runtime 检查更新。
2. HMR runtime 异步地下载更新，然后通知应用程序。
3. 应用程序要求 HMR runtime 应用更新。
4. HMR runtime 同步地应用更新。

你可以设置 HMR，以使此进程自动触发更新，或者你可以选择要求在用户交互时进行更新。

在 compiler 中

除了普通资源，compiler 需要发出 "update"，将之前的版本更新到新的版本。
"update" 由两部分组成：

1. 更新后的 manifest (JSON)
2. 一个或多个 updated chunk (JavaScript)

manifest 包括新的 compilation hash 和所有的 updated chunk 列表。每个 chunk 都包含着全部更新模块的最新代码（或一个 flag 用于表明此模块需要被移除）。

compiler 会确保在这些构建之间的模块 ID 和 chunk ID 保持一致。通常将这些 ID 存储在内存中（例如，使用 `webpack-dev-server` 时），但是也可能会将它们存储在一个 JSON 文件中。

在模块中

HMR 是可选功能，只会影响包含 HMR 代码的模块。举个例子，通过 `style-loader` 为 style 追加补丁。为了运行追加补丁，`style-loader` 实现了 HMR 接口；当它通过 HMR 接收到更新，它会使用新的样式替换旧的样式。

类似的，当在一个模块中实现了 HMR 接口，你可以描述出当模块被更新后发生了什么。然而在多数情况下，不需要在每个模块中强行写入 HMR 代码。如果一个模块没有 HMR 处理函数，更新就会冒泡(bubble up)。这意味着某个单独处理函数能够更新整个模块树。如果在模块树的一个单独模块被更新，那么整组依赖模块都会被重新加载。

有关 `module.hot` 接口的详细信息，请查看 [HMR API 页面](#) 或 [HMR 指南](#)。

在 HMR runtime 中

这件事情比较有技术性……如果你对其内部不感兴趣，可以随时跳到 [HMR API 页面](#) 或 [HMR 指南](#)。

对于模块系统运行时(module system runtime)，会发出额外代码，来跟踪模块 `parents` 和 `children` 关系。在管理方面，`runtime` 支持两个方法 `check` 和 `apply`。

`check` 方法，发送一个 HTTP 请求来更新 manifest。如果请求失败，说明没有可用更新。如果请求成功，会将 `updated chunk` 列表与当前的 `loaded chunk` 列表进行比较。每个 `loaded chunk` 都会下载相应的 `updated chunk`。当所有更新 `chunk` 完成下载，`runtime` 就会切换到 `ready` 状态。

`apply` 方法，将所有 `updated module` 标记为无效。对于每个无效 `module`，都需要在模块中有一个 `update handler`，或者在此模块的父级模块中有 `update handler`。否则，会进行无效标记冒泡，并且父级也会被标记为无效。继续每个冒泡，直到到达应用程序入口起点，或者到达带有 `update handler` 的 `module`（以最先到达为准，冒泡停止）。如果它从入口起点开始冒泡，则此过程失败。

之后，所有无效 `module` 都会被（通过 `dispose handler`）处理和解除加载。然后更新当前 `hash`，并且调用所有 `accept handler`。`runtime` 切换回 `idle` 状态，一切照常继续。

应用在项目中

在开发环境，可以将 HMR 作为 LiveReload 的替代。`webpack-dev-server` 支持 `hot` 模式，在试图重新加载整个页面之前，`hot` 模式会尝试使用 HMR 来更新。更多细节请查看 [模块热替换 指南](#)。

与许多其他功能一样，`webpack` 的强大之处在于它的可定制化。取决于特定

项目需求，会有许多方式来配置 HMR。然而，对于多数项目的实现目的来说，`webpack-dev-server` 都能够很好适应，可以帮助你在项目中快速应用 HMR。

配置

webpack 开箱即用，可以无需使用任何配置文件。然而，webpack 会假定项目的入口起点为 `src/index`，然后会在 `dist/main.js` 输出结果，并且在生产环境开启压缩和优化。

通常，你的项目还需要继续扩展此能力，为此你可以在项目根目录下创建一个 `webpack.config.js` 文件，webpack 会自动使用它。

下面指定了所有可用的配置选项。

刚开始学习 webpack？请查看我们提供的指南，从 webpack 一些 [核心概念](#) 开始学习吧！

选项

点击下面配置代码中每个选项的名称，跳转到详细的文档。还要注意，带有箭头的项目可以展开，以显示更多示例，在某些情况下可以看到高级配置。

注意整个配置中我们使用 Node 内置的 `path` 模块，并在它前面加上 `_dirname` 这个全局变量。可以防止不同操作系统之间的文件路径问题，并且可以使相对路径按照预期工作。更多「[POSIX 和 Windows](#)」的相关信息请查看 [此章节](#)。

webpack.config.js

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

module.exports = {
  mode: "production", // "production" | "development" | "none"
  // Chosen mode tells webpack to use its built-in optimizations according to the mode.
  entry: "./app/entry", // string | object | array
  // 默认为 './src'
  // 这里应用程序开始执行
  // webpack 开始打包
  output: {
    // webpack 如何输出结果的相关选项
    path: path.resolve(_dirname, "dist"), // string
    // 所有输出文件的目标路径
    // 必须是绝对路径（使用 Node.js 的 path 模块）
    filename: "bundle.js", // string
    // 「入口分块(entry chunk)」的文件名模板
    publicPath: "/assets/", // string
    // 输出解析文件的目录，url 相对于 HTML 页面
    library: "MyLibrary", // string,
    // 导出库(exported library)的名称
    libraryTarget: "umd", // 通用模块定义
    // 导出库(exported library)的类型
  }
}
```

```
►    /* 高级输出配置（点击显示） */
},
module: {
  // 关于模块配置
  rules: [
    // 模块规则（配置 loader、解析器等选项）
    {
      test: /\.jsx?$/,
      include: [
        path.resolve(__dirname, "app")
      ],
      exclude: [
        path.resolve(__dirname, "app/demo-files")
      ],
      // 这里是匹配条件，每个选项都接收一个正则表达式或字符串
      // test 和 include 具有相同的作用，都是必须匹配选项
      // exclude 是必不匹配选项（优先于 test 和 include）
      // 最佳实践：
      // - 只在 test 和 文件名匹配 中使用正则表达式
      // - 在 include 和 exclude 中使用绝对路径数组
      // - 尽量避免 exclude，更倾向于使用 include
      issuer: { test, include, exclude },
      // issuer 条件（导入源）
      enforce: "pre",
      enforce: "post",
      // 标识应用这些规则，即使规则覆盖（高级选项）
      loader: "babel-loader",
      // 应该应用的 loader，它相对上下文解析
      // 为了更清晰，`-loader` 后缀在 webpack 2 中不再是可选的
      // 查看 webpack 1 升级指南。
      options: {
        presets: ["es2015"]
      },
      // loader 的可选项
    },
    {
      test: /\.html$/,
      use: [
        // 应用多个 loader 和选项
        "html-lint-loader",
        {
          loader: "html-loader",
          options: {
            /* ... */
          }
        }
      ]
    },
    { oneOf: [ /* rules */ ] },
    // 只使用这些嵌套规则之一
    { rules: [ /* rules */ ] },
    // 使用所有这些嵌套规则（合并可用条件）
    { resource: { and: [ /* 条件 */ ] } },
    // 仅当所有条件都匹配时才匹配
    { resource: { or: [ /* 条件 */ ] } },
    { resource: [ /* 条件 */ ] },
    // 任意条件匹配时匹配（默认为数组）
    { resource: { not: /* 条件 */ } }
```

```
// 条件不匹配时匹配
],
/* 高级模块配置（点击展示） */
},
resolve: {
    // 解析模块请求的选项
    // （不适用于对 loader 解析）
    modules: [
        "node_modules",
        path.resolve(__dirname, "app")
    ],
    // 用于查找模块的目录
    extensions: [".js", ".json", ".jsx", ".css"],
    // 使用的扩展名
    alias: {
        // 模块别名列表
        "module": "new-module",
        // 起别名: "module" -> "new-module" 和 "module/path/file" -> "new-m
        "only-module$": "new-module",
        // 起别名 "only-module" -> "new-module"，但不匹配 "only-module/path/
        "module": path.resolve(__dirname, "app/third/module.js"),
        // 起别名 "module" -> "./app/third/module.js" 和 "module/file" 会导致
        // 模块别名相对于当前上下文导入
    },
},
/* 可供选择的别名语法（点击展示） */
/* 高级解析选项（点击展示） */
},
performance: {
    hints: "warning", // 枚举
    maxAssetSize: 200000, // 整数类型（以字节为单位）
    maxEntrypointSize: 400000, // 整数类型（以字节为单位）
    assetFilter: function(assetFilename) {
        // 提供资源文件名的断言函数
        return assetFilename.endsWith('.css') || assetFilename.endsWith('
    }
},
devtool: "source-map", // enum
// 通过在浏览器调试工具(browser devtools)中添加元信息(meta info)增强调试
// 牺牲了构建速度的 `source-map` 是最详细的。
context: __dirname, // string (绝对路径！)
// webpack 的主目录
// entry 和 module.rules.loader 选项
// 相对于此目录解析
target: "web", // 枚举
// bundle 应该运行的环境
// 更改 块加载行为(chunk loading behavior) 和 可用模块(available module)
externals: ["react", /^@angular\//],
// 不要遵循/打包这些模块，而是在运行时从环境中请求他们
serve: { //object
    port: 1337,
    content: './dist',
    // ...
},
// 为 webpack-serve 提供选项
stats: "errors-only",
// 精确控制要显示的 bundle 信息
devServer: {
    proxy: { // proxy URLs to backend development server

```

```
'/api': 'http://localhost:3000'  
},  
contentBase: path.join(__dirname, 'public'), // boolean | string |   
compress: true, // enable gzip compression  
historyApiFallback: true, // true for index.html upon 404, object for  
hot: true, // hot module replacement. Depends on HotModuleReplacementPlugin  
https: false, // true for self-signed, object for cert authority  
noInfo: true, // only errors & warns on hot reload  
// ...  
},  
plugins: [  
  // ...  
],  
// 附加插件列表  
► /* 高级配置（点击展示） */  
}
```

Use custom configuration file

If for some reason you want to use custom configuration file depending on certain situations you can change this via command line by using the `--config` flag.

package.json

```
"scripts": {  
  "build": "webpack --config prod.config.js"  
}
```

Configuration file generators

Want to rapidly generate webpack configuration file for your project requirements with just a few clicks away?

[Generate Custom Webpack Configuration](#) is an interactive portal you can play around by selecting custom webpack configuration options tailored for your frontend project. It automatically generates a minimal webpack configuration based on your selection of loaders/plugins, etc.

[Visual tool for creating webpack configs](#) is an online configuration tool for creating webpack configuration file where you can select any combination of features you need. It also generates a full example project based on your webpack configs.

使用不同语言进行配置(configuration languages)

webpack 接受以多种编程和数据语言编写的配置文件。支持的文件扩展名列表，可以在 `node-interpret` 包中找到。使用 `node-interpret`，webpack 可以处理许多不同类型的配置文件。

TypeScript

为了用 `TypeScript` 书写 webpack 的配置文件，必须先安装相关依赖，i.e., `TypeScript` and the relevant type definitions from the `DefinitelyTyped` project:::

```
npm install --save-dev typescript ts-node @types/node @types/webpack
# and, if using webpack-dev-server
npm install --save-dev @types/webpack-dev-server
```

之后就可以使用 `TypeScript` 书写 webpack 的配置文件了：

webpack.config.ts

```
import path from 'path';
import webpack from 'webpack';

const config: webpack.Configuration = {
  mode: 'production',
  entry: './foo.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'foo.bundle.js'
  }
};

export default config;
```

以上示例假定 webpack 版本 ≥ 2.7 ，或者，在 `tsconfig.json` 文件中，具有 `esModuleInterop` 和 `allowSyntheticDefaultImports` 这两个新的编译器选项的较新版本 `TypeScript`。

注意，你还需要核对 `tsconfig.json` 文件。如果 `tsconfig.json` 中的 `compilerOptions` 中的 `module` 字段是 `commonjs`，则配置是正确的，否则 webpack 将因为错误而构建失败。发生这种情况，是因为 `ts-node` 不支持 `commonjs` 以外的任何模块语法。

这个问题有两种解决方案：

- 修改 `tsconfig.json`。

- 安装 `tsconfig-paths`。

第一个选项是指，打开你的 `tsconfig.json` 文件并查找 `compilerOptions`。将 `target` 设置为 "ES5"，以及将 `module` 设置为 "CommonJS"（或者完全移除 `module` 选项）。

第二个选项是指，安装 `tsconfig-paths` 包：

```
npm install --save-dev tsconfig-paths
```

然后，为你的 `webpack` 配置，专门创建一个单独的 TypeScript 配置：

tsconfig-for-webpack-config.json

```
{  
  "compilerOptions": {  
    "module": "commonjs",  
    "target": "es5",  
    "esModuleInterop": true  
  }  
}
```

`ts-node` 可以使用 `tsconfig-path` 提供的环境变量来解析 `tsconfig.json` 文件。

然后，设置 `tsconfig-path` 提供的环境变量 `process.env.TS_NODE_PROJECT`，如下所示：

package.json

```
{  
  "scripts": {  
    "build": "cross-env TS_NODE_PROJECT=\"tsconfig-for-webpack-config.json\""  
  }  
}
```

We had been getting reports that `TS_NODE_PROJECT` might not work with "`TS_NODE_PROJECT`" unrecognized command error. Therefore running it with `cross-env` seems to fix the issue, for more info see [this issue](#).

CoffeeScript

类似的，为了使用 `CoffeeScript` 来书写配置文件，同样需要安装相关的依赖：

```
npm install --save-dev coffee-script
```

之后就可以使用 `Coffecript` 书写配置文件了：

webpack.config.coffee

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

config =
  mode: 'production'
  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 HtmlWebpackPlugin(template: './src/index.html')
  ]

module.exports = config
```

Babel and JSX

在以下的例子中，使用了 JSX（React 形式的 javascript）以及 Babel 来创建 JSON 形式的 webpack 配置文件：

感谢 Jason Miller

首先安装依赖：

```
npm install --save-dev babel-register jsxobj babel-preset-es2015
```

.babelrc

```
{
  "presets": [ "es2015" ]
}
```

webpack.config.babel.js

```
import jsxobj from 'jsxobj';

// example of an imported plugin
const CustomPlugin = config => ({
  ...config,
  name: 'custom-plugin'
});

export default (
  <webpack target="web" watch mode="production">
    <entry path="src/index.js" />
    <resolve>
```

```
<alias {...{
  react: 'preact-compat',
  'react-dom': 'preact-compat'
} } />
</resolve>
<plugins>
  <CustomPlugin foo="bar" />
</plugins>
</webpack>
) ;
```

如果你在其他地方也使用了 Babel 并且把模块(`modules`)设置为了 `false`，那么你要么同时维护两份单独的 `.babelrc` 文件，要么使用 `conts` `jsxobj = require('jsxobj');` 并且使用 `moduel.exports` 而不是新版本的 `import` 和 `export` 语法。这是因为尽管 Node.js 已经支持了许多 ES6 的新特性，然而还无法支持 ES6 模块语法。

多种配置类型(configuration types)

除了导出单个配置对象，还有一些方式满足其他需求。

导出为一个函数

最终，你会发现需要在开发和生产构建之间，消除 `webpack.config.js` 的差异。
(至少)有两种选项：

作为导出一个配置对象的替代，还有一种可选的导出方式是，从 `webpack` 配置文件中导出一个函数。该函数在调用时，可传入两个参数：

- 环境对象(`environment`)作为第一个参数。有关语法示例，请查看[CLI 文档的环境选项](#)。
- 一个选项 `map` 对象 (`argv`) 作为第二个参数。这个对象描述了传递给 `webpack` 的选项，并且具有 `output-filename` 和 `optimize-minimize` 等 key。

```
-module.exports = {
+module.exports = function(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 --optimi
+        }
+      })
+    ]
+  };
};
```

导出一个 Promise

`webpack` 将运行由配置文件导出的函数，并且等待 `Promise` 返回。便于需要异步地加载所需的配置变量。

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

导出多个配置对象

作为导出一个配置对象/配置函数的替代，你可能需要导出多个配置对象（从 webpack 3.1.0 开始支持导出多个函数）。当运行 webpack 时，所有的配置对象都会构建。例如，导出多个配置对象，对于针对多个构建目标（例如 AMD 和 CommonJS）打包一个 library 非常有用。

```
module.exports = [{  
  output: {  
    filename: './dist-amd.js',  
    libraryTarget: 'amd'  
  },  
  name: 'amd',  
  entry: './app.js',  
  mode: 'production',  
}, {  
  output: {  
    filename: './dist-commonjs.js',  
    libraryTarget: 'commonjs'  
  },  
  name: 'commonjs',  
  entry: './app.js',  
  mode: 'production',  
}];
```

If you pass a name to `--config-name` flag, webpack will only build that specific configuration.

入口和上下文(entry and context)

entry 对象是用于 webpack 查找启动并构建 bundle。其上下文是入口文件所处的目录的绝对路径的字符串。

context

string

基础目录，绝对路径，用于从配置中解析入口起点(entry point)和 loader

```
module.exports = {
  //...
  context: path.resolve(__dirname, 'app')
};
```

默认使用当前目录，但是推荐在配置中传递一个值。这使得你的配置独立于 CWD(current working directory - 当前执行路径)。

entry

string | [string] | object { <key>: string | [string] } | (function: () => string | [string] | object { <key>: string | [string] })

起点或是应用程序的起点入口。从这个起点开始，应用程序启动执行。如果传递一个数组，那么数组的每一项都会执行。

动态加载的模块不是入口起点。

简单规则：每个 HTML 页面都有一个入口起点。单页应用(SPA)：一个入口起点，多页应用(MPA)：多个入口起点。

```
module.exports = {
  //...
  entry: {
    home: './home.js',
    about: './about.js',
    contact: './contact.js'
  }
};
```

命名

如果传入一个字符串或字符串数组，chunk 会被命名为 main。如果传入一个对

象，则每个键(key)会是 chunk 的名称，该值描述了 chunk 的入口起点。

动态入口

If a function is passed then it will be invoked on every make event.

Note that the make event triggers when webpack starts and for every invalidation when watching for file changes.

```
module.exports = {
  //...
  entry: () => './demo'
};
```

或

```
module.exports = {
  //...
  entry: () => new Promise((resolve) => resolve(['./demo', './demo2']))
};
```

For example: you can use dynamic entries to get the actual entries from an external source (remote server, file system content or database):

webpack.config.js

```
module.exports = {
  entry() {
    return fetchPathsFromSomeExternalSource(); // returns a promise that
  }
};
```

当结合 output.library 选项时：如果传入数组，则只导出最后一项。

输出(output)

`output` 位于对象最顶级键(key)，包括了一组选项，指示 webpack 如何去输出、以及在哪里输出你的「bundle、asset 和其他你所打包或使用 webpack 载入的任何内容」。

output.auxiliaryComment

string object

在和 `output.library` 和 `output.libraryTarget` 一起使用时，此选项允许用户向导出容器(export wrapper)中插入注释。要为 `libraryTarget` 每种类型都插入相同的注释，将 `auxiliaryComment` 设置为一个字符串：

webpack.config.js

```
module.exports = {
  //...
  output: {
    library: 'someLibName',
    libraryTarget: 'umd',
    filename: 'someLibName.js',
    auxiliaryComment: 'Test Comment'
  }
};
```

将会生成如下：

webpack.config.js

```
(function webpackUniversalModuleDefinition(root, factory) {
  // Test Comment
  if(typeof exports === 'object' && typeof module === 'object')
    module.exports = factory(require('lodash'));
  // Test Comment
  else if(typeof define === 'function' && define.amd)
    define(['lodash'], factory);
  // Test Comment
  else if(typeof exports === 'object')
    exports['someLibName'] = factory(require('lodash'));
  // Test Comment
  else
    root['someLibName'] = factory(root['__']);
})(this, function(__WEBPACK_EXTERNAL_MODULE_1__){
  // ...
});
```

对于 `libraryTarget` 每种类型的注释进行更细粒度地控制，请传入一个对象：

webpack.config.js

```
module.exports = {
  //...
  output: {
    //...
    auxiliaryComment: {
      root: 'Root Comment',
      commonjs: 'CommonJS Comment',
      commonjs2: 'CommonJS2 Comment',
      amd: 'AMD Comment'
    }
  }
};
```

output.chunkFilename

string

此选项决定了非入口(non-entry) chunk 文件的名称。有关可取的值的详细信息，请查看 [output.filename](#) 选项。

注意，这些文件名需要在 runtime 根据 chunk 发送的请求去生成。因此，需要在 webpack runtime 输出 bundle 值时，将 chunk id 的值对应映射到占位符(如 `[name]` 和 `[chunkhash]`)。这会增加文件大小，并且在任何 chunk 的占位符值修改后，都会使 bundle 失效。

默认使用 `[id].js` 或从 [output.filename](#) 中推断出的值 (`[name]` 会被预先替换为 `[id]` 或 `[id].`)。

output.chunkLoadTimeout

integer

chunk 请求到期之前的毫秒数， 默认为 120 000。从 webpack 2.6.0 开始支持此选项。

output.crossOriginLoading

boolean string

只用于 `target` 是 web， 使用了通过 script 标签的 JSONP 来按需加载 chunk。

启用 `cross-origin` 属性 加载 chunk。以下是可接收的值.....

`crossOriginLoading: false` - 禁用跨域加载（默认）

`crossOriginLoading: 'anonymous'` - 不带凭据(credential)启用跨域加载

`crossOriginLoading: 'use-credentials'` - 带凭据(credential)启用跨域加载 with credentials

output.jsonpScriptType

string

允许自定义 `script` 的类型，webpack 会将 `script` 标签注入到 DOM 中以下载异步 chunk。可以使用以下选项：

- '`text/javascript`' (默认)
- '`module`'：与 ES6 就绪代码一起使用。

output.devtoolFallbackModuleFilenameTemplate

string | function(info)

当上面的模板字符串或函数产生重复时使用的备用内容。

查看 [`output.devtoolModuleFilenameTemplate`](#)。

output.devtoolLineToLine

boolean | object

避免使用此选项，因为它们已废弃，并将很快删除。it is **deprecated** and will soon be removed.

对所有或某些模块启用「行到行映射(line to line mapping)」。这将生成基本的源映射(source map)，即生成资源(generated source)的每一行，映射到原始资源(original source)的同一行。这是一个性能优化点，并且应该只需要输入行(input line)和生成行(generated line)相匹配时才使用。

传入 boolean 值，对所有模块启用或禁用此功能（默认 `false`）。对象可有 `test`, `include`, `exclude` 三种属性。例如，对某个特定目录中所有 javascript 文件启用此功能：

webpack.config.js

```
module.exports = {
  //...
  output: {
```

```
    devtoolLineToLine: { test: /\.js$/, include: 'src/utilities' }  
  }  
};
```

output.devtoolModuleFilenameTemplate

string | function(info)

此选项仅在「`devtool` 使用了需要模块名称的选项」时使用。

自定义每个 source map 的 `sources` 数组中使用的名称。可以通过传递模板字符串(template string)或者函数来完成。例如，当使用 `devtool: 'eval'`，默认值是：

webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    devtoolModuleFilenameTemplate: 'webpack://[namespace]/[resource-path]  
  }  
};
```

模板字符串(template string)中做以下替换（通过 webpack 内部的 `ModuleFilenameHelpers`）：

模板 描述	[absolute-resource-path] 绝对路径文件名
模板 描述	[all-loaders] 自动和显式的 loader，并且参数取决于第一个 loader 名称
模板 描述	[hash] 模块标识符的 hash
模板 描述	[id] 模块标识符
模板 描述	[loaders] 显式的 loader，并且参数取决于第一个 loader 名称
模板 描述	[resource] 用于解析文件的路径和用于第一个 loader 的任意查询参数
模板 描述	[resource-path]

描述	一个带任何查询参数，用于解析文件的路径
模板 描述	[namespace] 模块命名空间。在构建成为一个 library 之后，通常也是 library 名称，否则为空

当使用一个函数，同样的选项要通过 `info` 参数并使用驼峰式(camel-cased)：

```
module.exports = {
  //...
  output: {
    devtoolModuleFilenameTemplate: info => {
      return `webpack://${info.resourcePath}?${info.loaders}`;
    }
  }
};
```

如果多个模块产生相同的名称，使用

`output.devtoolFallbackModuleFilenameTemplate` 来代替这些模块。

output.devtoolNamespace

string

此选项确定 `output.devtoolModuleFilenameTemplate` 使用的模块名称空间。未指定时的默认值为：`output.library`。在加载多个通过 webpack 构建的 library 时，用于防止 source map 中源文件路径冲突。

例如，如果你有两个 library，分别使用命名空间 `library1` 和 `library2`，并且都有一个文件 `./src/index.js`（可能具有不同内容），它们会将这些文件暴露为 `webpack://library1./src/index.js` 和 `webpack://library2./src/index.js`。

output.filename

string function

此选项决定了每个输出 bundle 的名称。这些 bundle 将写入到 `output.path` 选项指定的目录下。

对于单个入口起点，filename 会是一个静态名称。

webpack.config.js

```
module.exports = {
  //...
  output: {
```

```
    filename: 'bundle.js'  
  }  
};
```

然而，当通过多个入口起点(entry point)、代码拆分(code splitting)或各种插件(plugin)创建多个 bundle，应该使用以下一种替换方式，来赋予每个 bundle 一个唯一的名字……

使用入口名称：

webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    filename: '[name].bundle.js'  
  }  
};
```

使用内部 chunk id

webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    filename: '[id].bundle.js'  
  }  
};
```

使用每次构建过程中，唯一的 hash 生成

webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    filename: '[name].[hash].bundle.js'  
  }  
};
```

使用基于每个 chunk 内容的 hash:

webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    filename: '[chunkhash].bundle.js'  
  }  
};
```

Using hashes generated for extracted content:

webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    filename: '[contenthash].bundle.css'  
  }  
};
```

Using function to return the filename:

webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    filename: (chunkData) => {  
      return chunkData.chunk.name === 'main' ? '[name].js': '[name]/[nar  
    },  
  }  
};
```

请确保已阅读过 [指南 - 缓存](#) 的详细信息。这里涉及更多步骤，不仅仅是设置此选项。

注意此选项被称为文件名，但是你还是可以使用像 '`js/[name]/bundle.js`' 这样的文件夹结构。

注意，此选项不会影响那些「按需加载 chunk」的输出文件。对于这些文件，请使用 `output.chunkFilename` 选项来控制输出。通过 loader 创建的文件也不受影响。在这种情况下，你必须尝试 loader 特定的可用选项。

可以使用以下替换模板字符串（通过 webpack 内部的 `[TemplatedPathPlugin]`）：

模板	[hash]
描述	模块标识符(module identifier)的 hash
模板	[chunkhash]
描述	chunk 内容的 hash
模板	[name]
描述	模块名称
模板	[id]
描述	模块标识符(module identifier)

模块 描述	[query] 模块的 query, 例如, 文件名 ? 后面的字符串
模板 描述	[function] The function, which can return filename [string]

[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。

如果将这个选项设为一个函数，函数将返回一个包含上面表格中替换信息的对象。

在使用 `ExtractTextWebpackPlugin` 时，可以用 [contenthash] 来获取提取文件的 hash (既不是 [hash] 也不是 [chunkhash])。

output.hashDigest

在生成 hash 时使用的编码方式，默认为 'hex'。支持 Node.js `hash.digest` 的所有编码。对文件名使用 'base64'，可能会出现问题，因为 base64 字母表中具有 / 这个字符(character)。同样的， 'latin1' 规定可以含有任何字符(character)。

output.hashDigestLength

散列摘要的前缀长度， 默认为 20。

output.hashFunction

string|function

散列算法， 默认为 'md4'。支持 Node.JS `crypto.createHash` 的所有功能。从 4.0.0-alpha2 开始， `hashFunction` 现在可以是一个返回自定义 hash 的构造函数。出于性能原因，你可以提供一个不加密的哈希函数(non-crypto hash function)。

```
module.exports = {
  //...
  output: {
    hashFunction: require('metrohash').MetroHash64
  }
};
```

确保 hash 函数有可访问的 `update` and `digest` 方法。

output.hashSalt

一个可选的加盐值，通过 Node.js `hash.update` 来更新哈希。

output.hotUpdateChunkFilename

string function

自定义热更新 chunk 的文件名。可选的值的详细信息，请查看 [`output.filename`](#) 选项。

占位符只能是 `[id]` 和 `[hash]`，默认值是：

webpack.config.js

```
module.exports = {
  //...
  output: {
    hotUpdateChunkFilename: '[id].[hash].hot-update.js'
  }
};
```

这里没有必要修改它。

output.hotUpdateFunction

function

只在 `target` 是 web 时使用，用于加载热更新(hot update)的 JSONP 函数。

JSONP 函数用于异步加载(async load)热更新(hot-update) chunk。

详细请查看 [`output.jsonpFunction`](#)。

output.hotUpdateMainFilename

string function

自定义热更新的主文件名(main filename)。可选的值的详细信息，请查看 [`output.filename`](#) 选项

占位符只能是 `[hash]`，默认值是：

webpack.config.js

```
module.exports = {
  //...
  output: {
```

```
    hotUpdateMainFilename: '[hash].hot-update.json'  
  }  
};
```

这里没有必要修改它。

output.jsonpFunction

string

只在 `target` 是 web 时使用，用于按需加载(load on-demand) chunk 的 JSONP 函数。

JSONP 函数用于异步加载(async load) chunk，或者拼接多个初始 chunk(SplitChunksPlugin, AggressiveSplittingPlugin)。

如果在同一网页中使用了多个（来自不同编译过程(compilation)的） webpack runtime，则需要修改此选项。

如果使用了 `output.library` 选项，library 名称时自动追加的。

output.library

string 或 object (从 webpack 3.1.0 开始；用于 `libraryTarget: 'umd'`)

`output.library` 的值的作用，取决于 `output.libraryTarget` 选项的值；完整的详细信息请查阅该章节。注意，`output.libraryTarget` 的默认选项是 `var`，所以如果使用以下配置选项：

webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    library: 'MyLibrary'  
  }  
};
```

如果生成的输出文件，是在 HTML 页面中作为一个 script 标签引入，则变量 `MyLibrary` 将与入口文件的返回值绑定。

注意，如果将数组作为 `entry`，那么只会暴露数组中的最后一个模块。如果将对象作为 `entry`，还可以使用 array 语法暴露（具体查看[这个示例 for details](#)）。

有关 `output.library` 以及 `output.libraryTarget` 详细信息，请查看[创建](#)

library 指南。

output.libraryExport

string | string[]

Configure which module or modules will be exposed via the `libraryTarget`. It is `undefined` by default, same behaviour will be applied if you set `libraryTarget` to an empty string e.g. '' it will export the whole (namespace) object. The examples below demonstrate the effect of this config when using `libraryTarget: 'var'`.

The following configurations are supported:

`libraryExport: 'default'` - The **default export of your entry point** will be assigned to the library target:

```
// if your entry has a default export of `MyDefaultModule`  
var MyDefaultModule = _entry_return_.default;
```

`libraryExport: 'MyModule'` - The **specified module** will be assigned to the library target:

```
var MyModule = _entry_return_.MyModule;
```

`libraryExport: ['MyModule', 'MySubModule']` - The array is interpreted as a **path to a module** to be assigned to the library target:

```
var MySubModule = _entry_return_.MyModule.MySubModule;
```

With the `libraryExport` configurations specified above, the resulting libraries could be utilized as such:

```
MyDefaultModule.doSomething();  
MyModule.doSomething();  
MySubModule.doSomething();
```

output.libraryTarget

string: 'var'

配置如何暴露 library。可以使用下面的选项中的任意一个。注意，此选项与分配给 `output.library` 的值一同使用。对于下面的所有示例，都假定将 `output.library` 的值配置为 `MyLibrary`。

注意，下面的示例代码中的 `_entry_return_` 是入口起点返回的值。在 bundle 本身中，它是从入口起点、由 webpack 生成的函数的输出结果。

暴露为一个变量

这些选项将入口起点的返回值（例如，入口起点的任何导出值），在 bundle 包所引入的位置，赋值给 output.library 提供的变量名。

libraryTarget: 'var' - （默认值）当 library 加载完成，入口起点的返回值将分配给一个变量：

```
var MyLibrary = _entry_return_;  
  
// 在一个单独的 script.....  
MyLibrary.doSomething();
```

当使用此选项时，将 output.library 设置为空，会因为没有变量导致无法赋值。

libraryTarget: 'assign' - 这将产生一个隐含的全局变量，可能会潜在地重新分配到全局中已存在的值（谨慎使用）。.

```
MyLibrary = _entry_return_;
```

注意，如果 MyLibrary 在作用域中未在前面代码进行定义，则你的 library 将被设置在全局作用域内。

当使用此选项时，将 output.library 设置为空，将产生一个破损的输出 bundle。

通过在对象上赋值暴露

这些选项将入口起点的返回值（例如，入口起点的任何导出值）赋值给一个特定对象的属性（此名称由 output.library 定义）下。

如果 output.library 未赋值为一个非空字符串，则默认行为是，将入口起点返回的所有属性都赋值给一个对象（此对象由 output.libraryTarget 特定），通过如下代码片段：

```
(function(e, a) { for(var i in a) { e[i] = a[i]; } })(output.libraryTarge
```

注意，不设置 output.library 将导致由入口起点返回的所有属性，都会被赋值给给定的对象；这里并不会检查现有的属性名是否存在。

libraryTarget: "this" - 入口起点的返回值将分配给 this 的一个属性（此名称由 output.library 定义）下，this 的含义取决于你：

```
this['MyLibrary'] = _entry_return_;  
  
// 在一个单独的 script.....
```

```
this.MyLibrary.doSomething();
MyLibrary.doSomething(); // 如果 this 是 window
```

libraryTarget: 'window' - 入口起点的返回值将使用 `output.library` 中定义的值，分配给 `window` 对象的这个属性下。

```
window['MyLibrary'] = _entry_return_;
window.MyLibrary.doSomething();
```

libraryTarget: 'global' - 入口起点的返回值将使用 `output.library` 中定义的值，分配给 `global` 对象的这个属性下。

```
global['MyLibrary'] = _entry_return_;
global.MyLibrary.doSomething();
```

libraryTarget: 'commonjs' - 入口起点的返回值将使用 `output.library` 中定义的值，分配给 `exports` 对象。这个名称也意味着，模块用于 CommonJS 环境：

```
exports['MyLibrary'] = _entry_return_;
require('MyLibrary').doSomething();
```

模块定义系统

这些选项将导致 bundle 带有更完整的模块头部，以确保与各种模块系统的兼容性。根据 `output.libraryTarget` 选项不同，`output.library` 选项将具有不同的含义。

libraryTarget: 'commonjs2' - 入口起点的返回值将分配给 `module.exports` 对象。这个名称也意味着模块用于 CommonJS 环境：

```
module.exports = _entry_return_;
require('MyLibrary').doSomething();
```

注意，`output.library` 会被省略，因此对于此特定的 `output.libraryTarget`，无需再设置 `output.library`。

想要弄清楚 CommonJS 和 CommonJS2 之间的区别？虽然它们很相似，但二者之间存在一些微妙的差异，这通常与 webpack 上下文没有关联。（更多详细信息，请阅读此 [issue](#)。）

libraryTarget: 'amd' - 将你的 library 暴露为 AMD 模块。

AMD 模块要求入口 chunk（例如使用 `<script>` 标签加载的第一个脚本）通过特定的属性定义，例如 `define` 和 `require`，它们通常由 RequireJS 或任何兼容的模块加载器提供（例如 almond）。否则，直接加载生成的 AMD bundle 将导致报

错，如 `define` is not defined。

所以，使用以下配置……

```
module.exports = {
  //...
  output: {
    library: 'MyLibrary',
    libraryTarget: 'amd'
  }
};
```

生成的 `output` 将会使用 "MyLibrary" 作为模块名定义，即

```
define('MyLibrary', [], function() {
  return _entry_return_;
});
```

可以在 `script` 标签中，将 `bundle` 作为一个模块整体引入，并且可以像这样调用 `bundle`：

```
require(['MyLibrary'], function(MyLibrary) {
  // 使用 library 做一些事....
});
```

如果 `output.library` 未定义，将会生成以下内容。

```
define([], function() {
  return _entry_return_; // 此模块返回值，是入口 chunk 返回的值
});
```

如果直接加载 `<script>` 标签，此 `bundle` 无法按预期运行，或者根本无法正常运行（在 almond loader 中）。只能通过文件的实际路径，在 RequireJS 兼容的异步模块加载器中运行，因此在这种情况下，如果这些设置直接暴露在服务器上，那么 `output.path` 和 `output.filename` 对于这个特定的设置可能变得很重要。

`libraryTarget: 'amd-require'` - This packages your output with an immediately-executed AMD `require(dependencies, factory)` wrapper.

The '`amd-require`' target allows for the use of AMD dependencies without needing a separate later invocation. As with the '`amd`' target, this depends on the appropriate `require` function being available in the environment in which the webpack output is loaded.

With this target, the library name is ignored.

`libraryTarget: 'umd'` - 将你的 library 暴露为所有的模块定义下都可运行的方式。它将在 CommonJS, AMD 环境下运行，或将模块导出到 global 下的变量。了解更多请查看 UMD 仓库。

在这个例子中，你需要 `library` 属性来命名你的模块：

```
module.exports = {
  //...
  output: {
    library: 'MyLibrary',
    libraryTarget: 'umd'
  }
};
```

最终输出如下：

```
(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['MyLibrary'] = factory();
  else
    root['MyLibrary'] = factory();
}) (typeof self !== 'undefined' ? self : this, function() {
  return _entry_return_; // 此模块返回值，是入口 chunk 返回的值
});
```

注意，省略 `library` 会导致将入口起点返回的所有属性，直接赋值给 `root` 对象，就像对象分配章节。例如：

```
module.exports = {
  //...
  output: {
    libraryTarget: 'umd'
  }
};
```

输出结果如下：

```
(function webpackUniversalModuleDefinition(root, factory) {
  if(typeof exports === 'object' && typeof module === 'object')
    module.exports = factory();
  else if(typeof define === 'function' && define.amd)
    define([], factory);
  else {
    var a = factory();
    for(var i in a) (typeof exports === 'object' ? exports : root)[i] =
  }
}) (typeof self !== 'undefined' ? self : this, function() {
  return _entry_return_; // 此模块返回值，是入口 chunk 返回的值
});
```

从 webpack 3.1.0 开始，你可以将 `library` 指定为一个对象，用于给每个 target 起不同的名称：

```
module.exports = {
```

```
//...
output: {
  library: {
    root: 'MyLibrary',
    amd: 'my-library',
    commonjs: 'my-common-library'
  },
  libraryTarget: 'umd'
}
};
```

模块验证 library。

其他 Targets

libraryTarget: 'jsonp' - 这将把入口起点的返回值，包裹到一个 jsonp 包装器中

```
MyLibrary(_entry_return_);
```

你的 library 的依赖将由 `externals` 配置定义。

output.path

string

output 目录对应一个绝对路径。

webpack.config.js

```
module.exports = {
  //...
  output: {
    path: path.resolve(__dirname, 'dist/assets')
  }
};
```

注意，`[hash]` 在参数中被替换为编译过程(compilation)的 hash。详细信息请查看指南 - 缓存。

output.pathinfo

boolean

告知 webpack 在 bundle 中引入「所包含模块信息」的相关注释。此选项在 development 模式时的默认值是 `true`，而在 production 模式时的默认值是 `false`。

对于在开发环境(development)下阅读生成代码时，虽然通过这些注释可以提供非常有用的数据信息，但在生产环境(production)下，不应该使用。

webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    pathinfo: true  
  }  
};
```

注意，这些注释也会被添加至经过 tree shaking 后生成的 bundle 中。

output.publicPath

```
string: '' function
```

对于按需加载(on-demand-load)或加载外部资源(external resources)（如图片、文件等）来说，output.publicPath 是很重要的选项。如果指定了一个错误的值，则在加载这些资源时会收到 404 错误。

此选项指定在浏览器中所引用的「此输出目录对应的公开 URL」。相对 URL(relative URL) 会被相对于 HTML 页面（或 `<base>` 标签）解析。相对于服务的 URL(Server-relative URL)，相对于协议的 URL(protocol-relative URL) 或绝对 URL(absolute URL) 也可是可能用到的，或者有时必须用到，例如：当将资源托管到 CDN 时。

该选项的值是以 runtime(运行时) 或 loader(载入时) 所创建的每个 URL 为前缀。因此，在多数情况下，此选项的值都会以 / 结束。

简单规则如下：`output.path` 中的 URL 以 HTML 页面为基准。

webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    path: path.resolve(__dirname, 'public/assets'),  
    publicPath: 'https://cdn.example.com/assets/'  
  }  
};
```

对于这个配置：

webpack.config.js

```
module.exports = {
```

```
//...
output: {
  publicPath: '/assets/',
  chunkFilename: '[id].chunk.js'
}
};
```

对于一个 chunk 请求，看起来像这样 `/assets/4.chunk.js`。

对于一个输出 HTML 的 loader 可能会像这样输出：

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

或者在加载 CSS 的一个图片时：

```
background-image: url(/assets/spinner.gif);
```

webpack-dev-server 也会默认从 `publicPath` 为基准，使用它来决定在哪个目录下启用服务，来访问 webpack 输出的文件。

注意，参数中的 `[hash]` 将会被替换为编译过程(compilation)的 hash。详细信息请查看[指南 - 缓存](#)。

示例：

```
module.exports = {
//...
output: {
  // One of the below
  publicPath: 'https://cdn.example.com/assets/' , // CDN (总是 HTTPS 协议)
  publicPath: '//cdn.example.com/assets/' , // CDN (协议相同)
  publicPath: '/assets/' , // 相对于服务(server-relative)
  publicPath: 'assets/' , // 相对于 HTML 页面
  publicPath: '../assets/' , // 相对于 HTML 页面
  publicPath: '' , // 相对于 HTML 页面(目录相同)
}
};
```

在编译时(compile time)无法知道输出文件的 `publicPath` 的情况下，可以留空，然后在入口文件(entry file)处使用[自由变量\(free variable\)](#) `__webpack_public_path__`，以便在运行时(runtime)进行动态设置。

```
__webpack_public_path__ = myRuntimePublicPath;
// 应用程序入口的其他部分
```

有关 `__webpack_public_path__` 的更多信息，请查看[此讨论](#)。

output.sourceMapFilename

string

此选项会向硬盘写入一个输出文件，只在 `devtool` 启用了 SourceMap 选项时才使用。

配置 source map 的命名方式。默认使用 '`[file].map`'。

可以使用 `#output-filename` 中的 `[name]`, `[id]`, `[hash]` 和 `[chunkhash]` 替换符号。除此之外，还可以使用以下替换符号。`[file]` 占位符会被替换为原始文件的文件名。我们建议只使用 `[file]` 占位符，因为其他占位符在非 chunk 文件(non-chunk files)生成的 SourceMap 时不起作用。

模板	<code>[file]</code>
描述	模块文件名称
模板	<code>[filebase]</code>
描述	模块 <code>basename</code>

output.sourcePrefix

string

修改输出 bundle 中每行的前缀。

webpack.config.js

```
module.exports = {
  //...
  output: {
    sourcePrefix: '\t'
  }
};
```

注意，默认情况下使用空字符串。使用一些缩进会看起来更美观，但是可能导致多行字符串中的问题。

这里没有必要修改它。

output.strictModuleExceptionHandling

boolean

如果一个模块是在 `require` 时抛出异常，告诉 webpack 从模块实例缓存(`require.cache`)中删除这个模块。

出于性能原因， 默认为 `false`。

当设置为 `false` 时， 该模块不会从缓存中删除， 这将造成仅在第一次 `require` 调用时抛出异常（会导致与 node.js 不兼容）。

例如， 设想一下 `module.js`:

```
throw new Error('error');
```

将 `strictModuleExceptionHandling` 设置为 `false`， 只有第一个 `require` 抛出异常：

```
// with strictModuleExceptionHandling = false
require('module'); // <- 抛出
require('module'); // <- 不抛出
```

相反， 将 `strictModuleExceptionHandling` 设置为 `true`， 这个模块所有的 `require` 都抛出异常：

```
// with strictModuleExceptionHandling = true
require('module'); // <- 抛出
require('module'); // <- 仍然抛出
```

output/umdNamedDefine

`boolean`

当使用了 `libraryTarget: "umd"`， 设置：

```
module.exports = {
  //...
  output: {
    umdNamedDefine: true
  }
};
```

会对 UMD 的构建过程中的 AMD 模块进行命名。否则就使用匿名的 `define`。

模块(module)

这些选项决定了如何处理项目中的不同类型的模块。

module.noParse

RegExp [RegExp] function(resource) string [string]

防止 webpack 解析那些任何与给定正则表达式相匹配的文件。忽略的文件中不应该含有 `import`, `require`, `define` 的调用, 或任何其他导入机制。忽略大型的 library 可以提高构建性能。

webpack.config.js

```
module.exports = {  
  //...  
  module: {  
    noParse: /jquery|lodash/,  
  }  
};  
  
module.exports = {  
  //...  
  module: {  
    noParse: (content) => /jquery|lodash/.test(content)  
  }  
};
```

module.rules

[Rule]

创建模块时, 匹配请求的规则数组。这些规则能够修改模块的创建方式。这些规则能够对模块(module)应用 loader, 或者修改解析器(parser)。

Rule

object

每个规则可以分为三部分 - 条件(condition), 结果(result)和嵌套规则(nested rule)。

Rule 条件

条件有两种输入值:

1. resource: 请求文件的绝对路径。它已经根据 `resolve` 规则解析。
2. issuer: 被请求资源(requested the resource)的模块文件的绝对路径。是导入时的位置。

例如: 从 `app.js` 导入 `'./style.css'`, `resource` 是 `/path/to/style.css`. `issuer` 是 `/path/to/app.js`。

在规则中, 属性 `test`, `include`, `exclude` 和 `resource` 对 `resource` 匹配, 并且属性 `issuer` 对 `issuer` 匹配。

当使用多个条件时, 所有条件都匹配。

小心! `resource` 是文件的 解析 路径, 这意味着符号链接的资源是真正的路径, 而不是 符号链接位置。在使用工具来符号链接包的时候(如 `npm link`)比较好记, 像 `/node_modules/` 等常见条件可能会不小心错过符号链接的文件。注意, 可以通过 `resolve.symlinks` 关闭符号链接解析(以便将资源解析为符号链接路径)。

Rule 结果

规则结果只在规则条件匹配时使用。

规则有两种输入值:

1. 应用的 loader: 应用在 `resource` 上的 loader 数组。
2. Parser 选项: 用于为模块创建解析器的选项对象。

这些属性会影响 loader: `loader`, `options`, `use`。

也兼容这些属性: `query`, `loaders`。

`enforce` 属性会影响 loader 种类。不论是普通的, 前置的, 后置的 loader。

`parser` 属性会影响 parser 选项。

嵌套的 Rule

可以使用属性 `rules` 和 `oneOf` 指定嵌套规则。

这些规则用于在规则条件(rule condition)匹配时进行取值。

Rule.enforce

string

可能的值有: "pre" | "post"

指定 loader 种类。没有值表示是普通 loader。

还有一个额外的种类"行内 loader"， loader 被应用在 import/require 行内。

所有一个接一个地进入的 loader， 都有两个阶段：

1. **pitching** 阶段： loader 上的 pitch 方法，按照 后置(post)、行内(normal)、普通(inline)、前置(pre) 的顺序调用。更多详细信息，请查看 [越过 loader\(pitching loader\)](#)。
2. **normal** 阶段： loader 上的 常规方法，按照 前置(pre)、行内(normal)、普通(inline)、后置(post) 的顺序调用。模块源码的转换，发生在这个阶段。

所有普通 loader 可以通过在请求中加上 ! 前缀来忽略（覆盖）。

所有普通和前置 loader 可以通过在请求中加上 -! 前缀来忽略（覆盖）。

所有普通，后置和前置 loader 可以通过在请求中加上 !! 前缀来忽略（覆盖）。

不应该使用行内 loader 和 ! 前缀，因为它们是非标准的。它们可在由 loader 生成的代码中使用。

Rule.exclude

Rule.exclude 是 Rule.resource.exclude 的简写。如果你提供了 Rule.exclude 选项，就不能再提供 Rule.resource。详细请查看 [Rule.resource](#) 和 [Condition.exclude](#)。

Rule.include

Rule.include 是 Rule.resource.include 的简写。如果你提供了 Rule.include 选项，就不能再提供 Rule.resource。详细请查看 [Rule.resource](#) 和 [Condition.include](#)。

Rule.issuer

一个条件，用来与被发布的 request 对应的模块项匹配。在以下示例中，a.js request 的发布者(issuer)是 index.js 文件的路径。

index.js

```
import A from './a.js';
```

这个选项可以用来将 loader 应用到一个特定模块或一组模块的依赖中。

Rule.loader

Rule.loader 是 Rule.use: [{ loader }] 的简写。详细请查看 [Rule.use](#) 和 [UseEntry.loader](#)。

Rule.loaders

由于需要支持 Rule.use，此选项已废弃。

Rule.loaders 是 Rule.use 的别名。详细请查看 [Rule.use](#)。

Rule.oneOf

规则数组，当规则匹配时，只使用第一个匹配规则。

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.css$/,
        oneOf: [
          {
            resourceQuery: /inline/, // foo.css?inline
            use: 'url-loader'
          },
          {
            resourceQuery: /external/, // foo.css?external
            use: 'file-loader'
          }
        ]
      }
    ]
  }
};
```

Rule.options / Rule.query

Rule.options 和 Rule.query 是 Rule.use: [{ options }] 的简写。详细请查看 [Rule.use](#) 和 [UseEntry.options](#)。

由于需要支持 `Rule.options` 和 `UseEntry.options`, `Rule.use`, `Rule.query` 已废弃。

Rule.parser

解析选项对象。所有应用的解析选项都将合并。

解析器(parser)可以查阅这些选项，并相应地禁用或重新配置。大多数默认插件，会如下解释值：

- 将选项设置为 `false`, 将禁用解析器。
- 将选项设置为 `true`, 或不修改将其保留为 `undefined`, 可以启用解析器。

然而，一些解析器(parser)插件可能不光只接收一个布尔值。例如，内部的 `NodeStuffPlugin` 差距，可以接收一个对象，而不是 `true`, 来为特定的规则添加额外的选项。

示例（默认的插件解析器选项）：

```
module.exports = {
  //...
  module: {
    rules: [
      {
        //...
        parser: {
          amd: false, // 禁用 AMD
          commonjs: false, // 禁用 CommonJS
          system: false, // 禁用 SystemJS
          harmony: false, // 禁用 ES2015 Harmony import/export
          requireInclude: false, // 禁用 require.include
          requireEnsure: false, // 禁用 require.ensure
          requireContext: false, // 禁用 require.context
          browserify: false, // 禁用特殊处理的 browserify bundle
          requireJs: false, // 禁用 requirejs.*
          node: false, // 禁用 dirname, filename, module, require.ex
          node: {...} // 在模块级别(module level)上重新配置 node 层(layer)
        }
      }
    ]
  }
}
```

Rule.resource

条件会匹配 resource。既可以提供 `Rule.resource` 选项，也可以使用快捷选项 `Rule.test`, `Rule.exclude` 和 `Rule.include`。在 `Rule` 条件中查看详细。

Rule.resourceQuery

A [Condition](#) matched with the resource query. This option is used to test against the query section of a request string (i.e. from the question mark onwards). If you were to import Foo from './foo.css?inline', the following condition would match:

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.css$/,
        resourceQuery: /inline/,
        use: 'url-loader'
      }
    ]
  }
};
```

Rule.rules

规则数组，当规则匹配时使用。

Rule.sideEffects

bool

标示出模块的哪些部分包含外部作用(side effect)。更多详细信息，请查看 [tree shaking](#)。

Rule.test

Rule.test 是 Rule.resource.test 的简写。如果你提供了一个 Rule.test 选项，就不能再提供 Rule.resource。详细请查看 [Rule.resource](#) 和 [Condition.test](#)。

Rule.type

string

Possible values: 'javascript/auto' | 'javascript/dynamic' |
'javascript/esm' | 'json' | 'webassembly/experimental'

`Rule.type` sets the type for a matching module. This prevents `defaultRules` and their default importing behaviors from occurring. For example, if you want to load a `.json` file through a custom loader, you'd need to set the `type` to `'javascript/auto'` to bypass webpack's built-in json importing. (See [v4.0 changelog](#) for more details)

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      //...
      {
        test: /\.json$/,
        type: 'javascript/auto',
        loader: 'custom-json-loader'
      }
    ]
  }
};
```

Rule.use

```
[UseEntry] function(info)
```

```
[UseEntry]
```

`Rule.use` 可以是一个应用于模块的 `UseEntries` 数组。每个入口(entry)指定使用一个 loader。

传递字符串（如：`use: ['style-loader']`）是 `loader` 属性的简写方式（如：`use: [{ loader: 'style-loader' }]`）。

Loaders can be chained by passing multiple loaders, which will be applied from right to left (last to first configured).

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        //...
        use: [
          'style-loader',
          {
            loader: 'css-loader',
            options: {
              importLoaders: 1
            }
          }
        ]
      }
    ]
  }
};
```

```

        }
    },
    {
        loader: 'less-loader',
        options: {
            noIeCompat: true
        }
    }
]
}
]
}
};

function(info)

```

`Rule.use` can also be a function which receives the `object` argument describing the module being loaded, and must return an array of `UseEntry` items.

The `info` object parameter has the following fields:

- `compiler`: The current webpack compiler (can be `undefined`)
- `issuer`: The path to the module that is importing the module being loaded
- `realResource`: Always the path to the module being loaded
- `resource`: The path to the module being loaded, it is usually equal to `realResource` except when the resource name is overwritten via `!=!` in request string

The same shortcut as an array can be used for the return value (i.e. `use: ['style-loader']`).

webpack.config.js

```

module.exports = {
    //...
    module: {
        rules: [
            {
                use: (info) => ([
                    {
                        loader: 'custom-svg-loader'
                    },
                    {
                        loader: 'svgo-loader',
                        options: {
                            plugins: [
                                cleanupIDs: { prefix: basename(info.resource) }
                            ]
                        }
                    }
                ])
            }
        ]
    }
};

```

```
};
```

详细信息请查看 [UseEntry](#)。

条件

条件可以是这些之一：

- 字符串：匹配输入必须以提供的字符串开始。是的。目录绝对路径或文件绝对路径。
- 正则表达式：test 输入值。
- 函数：调用输入的函数，必须返回一个真值(truthy value)以匹配。
- 条件数组：至少一个匹配条件。
- 对象：匹配所有属性。每个属性都有一个定义行为。

{ test: Condition }: 匹配特定条件。一般是提供一个正则表达式或正则表达式的数组，但这不是强制的。

{ include: Condition }: 匹配特定条件。一般是提供一个字符串或者字符串数组，但这不是强制的。

{ exclude: Condition }: 排除特定条件。一般是提供一个字符串或字符串数组，但这不是强制的。

{ and: [Condition] }: 必须匹配数组中的所有条件

{ or: [Condition] }: 匹配数组中任何一个条件

{ not: [Condition] }: 必须排除这个条件

示例：

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.css$/,
        include: [
          path.resolve(__dirname, 'app/styles'),
          path.resolve(__dirname, 'vendor/styles')
        ]
      }
    ]
  }
};
```

UseEntry

```
object function(info)
```

object

必须有一个 `loader` 属性是字符串。它使用 loader 解析选项 (`resolveLoader`)，相对于配置中的 `context` 来解析。

可以有一个 `options` 属性为字符串或对象。值可以传递到 loader 中，将其理解为 loader 选项。

由于兼容性原因，也可能有 `query` 属性，它是 `options` 属性的别名。使用 `options` 属性替代。

注意，webpack 需要生成资源和所有 loader 的独立模块标识，包括选项。它尝试对选项对象使用 `JSON.stringify`。这在 99.9% 的情况下是可以的，但是如果将相同的 loader 应用于相同资源的不同选项，并且选项具有一些带字符的值，则可能不是唯一的。

如果选项对象不被字符化（例如循环 JSON），它也会中断。因此，你可以在选项对象使用 `ident` 属性，作为唯一标识符。

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        loader: 'css-loader',
        options: {
          modules: true
        }
      }
    ]
  }
};
```

function(info)

A `UseEntry` can also be a function which receives the object argument describing the module being loaded, and must return an options object. This can be used to vary the loader options on a per-module basis.

The `info` object parameter has the following fields:

- `compiler`: The current webpack compiler (can be undefined)
- `issuer`: The path to the module that is importing the module being loaded
- `realResource`: Always the path to the module being loaded

- `resource`: The path to the module being loaded, it is usually equal to `realResource` except when the resource name is overwritten via `!=!` in request string

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        loader: 'file-loader',
        options: {
          outputPath: 'svgs'
        }
      },
      (info) => ({
        loader: 'svgo-loader',
        options: {
          plugins: [{ cleanupIDs: { prefix: basename(info.resource) } }]
        }
      })
    ]
  }
};
```

模块上下文(module context)

避免使用这些选项，因为它们已废弃，并将很快删除。

这些选项描述了当遇到动态依赖时，创建上下文的默认设置。

例如，未知的(`unknown`) 动态依赖: `require`。

例如，表达式(`expr`) 动态依赖: `require(expr)`。

例如，包裹的(`wrapped`) 动态依赖: `require('./templates/' + expr)`。

以下是其默认值的可用选项

webpack.config.js

```
module.exports = {
  //...
  module: {
    exprContextCritical: true,
    exprContextRecursive: true,
    exprContextRegExp: false,
    exprContextRequest: '.',
    unknownContextCritical: true,
    unknownContextRecursive: true,
```

```
unknownContextRegExp: false,  
unknownContextRequest: '.',  
wrappedContextCritical: false,  
wrappedContextRecursive: true,  
wrappedContextRegExp: /\.*/,  
strictExportPresence: false // since webpack 2.3.0  
}  
};
```

你可以使用 `ContextReplacementPlugin` 来修改这些单个依赖的值。这也会删除警告。

几个用例：

- **动态依赖的警告:** `wrappedContextCritical: true`。
- **require(expr) 应该包含整个目录:** `exprContextRegExp: /^\.\\//`
- **require("./templates/" + expr) 不应该包含默认子目录:** `wrappedContextRecursive: false`
- **strictExportPresence makes missing exports an error instead of warning**

解析(resolve)

这些选项能设置模块如何被解析。webpack 提供合理的默认值，但是还是会修改一些解析的细节。关于 resolver 具体如何工作的更多解释说明，请查看模块解析。

resolve

object

配置模块如何解析。例如，当在 ES2015 中调用 `import 'lodash'`，`resolve` 选项能够对 webpack 查找 `'lodash'` 的方式去做修改（查看模块）。

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    // configuration options
  }
};
```

resolve.alias

object

创建 `import` 或 `require` 的别名，来确保模块引入变得更容易。例如，一些位于 `src/` 文件夹下的常用模块：

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    alias: {
      Utilities: path.resolve(__dirname, 'src/utilities/'),
      Templates: path.resolve(__dirname, 'src/templates/')
    }
  }
};
```

现在，替换「在导入时使用相对路径」这种方式，就像这样：

```
import Utility from '../..utilities/utility';
```

你可以这样使用别名：

```
import Utility from 'Utilities/utility';
```

也可以在给定对象的键后的末尾添加 \$，以表示精准匹配：

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    alias: {
      xyz$: path.resolve(__dirname, 'path/to/file.js')
    }
  }
};
```

这将产生以下结果：

```
import Test1 from 'xyz'; // 精确匹配，所以 path/to/file.js 被解析和导入
import Test2 from 'xyz/file.js'; // 非精确匹配，触发普通解析
```

下面的表格展示了一些其他情况：

别名:	{ }
import 'xyz'	/abc/node_modules/xyz/index.js
import 'xyz/file.js'	/abc/node_modules/xyz/file.js
别名:	{ xyz: '/abs/path/to/file.js' }
import 'xyz'	/abs/path/to/file.js
import 'xyz/file.js'	error
别名:	{ xyz\$: '/abs/path/to/file.js' }
import 'xyz'	/abs/path/to/file.js
import 'xyz/file.js'	/abc/node_modules/xyz/file.js
别名:	{ xyz: './dir/file.js' }
import 'xyz'	/abc/dir/file.js
import 'xyz/file.js'	error
别名:	{ xyz\$: './dir/file.js' }
import 'xyz'	/abc/dir/file.js
import 'xyz/file.js'	/abc/node_modules/xyz/file.js
别名:	{ xyz: '/some/dir' }
import 'xyz'	/some/dir/index.js
import 'xyz/file.js'	/some/dir/file.js
别名:	{ xyz\$: '/some/dir' }
.	.

import 'xyz'	/some/dir/index.js
import 'xyz/file.js'	/abc/node_modules/xyz/file.js
别名:	{ xyz: './dir' }
import 'xyz'	/abc/dir/index.js
import 'xyz/file.js'	/abc/dir/file.js
别名:	{ xyz: 'modu' }
import 'xyz'	/abc/node_modules/modu/index.js
import 'xyz/file.js'	/abc/node_modules/modu/file.js
别名:	{ xyz\$: 'modu' }
import 'xyz'	/abc/node_modules/modu/index.js
import 'xyz/file.js'	/abc/node_modules/xyz/file.js
别名:	{ xyz: 'modu/some/file.js' }
import 'xyz'	/abc/node_modules/modu/some/file.js
import 'xyz/file.js'	error
别名:	{ xyz: 'modu/dir' }
import 'xyz'	/abc/node_modules/modu/dir/index.js
import 'xyz/file.js'	/abc/node_modules/dir/file.js
别名:	{ xyz: 'xyz/dir' }
import 'xyz'	/abc/node_modules/xyz/dir/index.js
import 'xyz/file.js'	/abc/node_modules/xyz/dir/file.js
别名:	{ xyz\$: 'xyz/dir' }
import 'xyz'	/abc/node_modules/xyz/dir/index.js
import 'xyz/file.js'	/abc/node_modules/xyz/file.js

如果在 package.json 中定义，index.js 可能会被解析为另一个文件。

/abc/node_modules 也可能在 /node_modules 中解析。

resolve.aliasFields

[string]: ['browser']

指定一个字段，例如 browser，根据此规范进行解析。默认：

webpack.config.js

```
module.exports = {
  // ...
}
```

```
    resolve: {
      aliasFields: ['browser']
    }
};
```

resolve.cacheWithContext

boolean (从 webpack 3.1.0 开始)

如果启用了不安全缓存, 请在缓存键(cache key)中引入 `request.context`。这个选项被 `enhanced-resolve` 模块考虑在内。从 webpack 3.1.0 开始, 在配置了 `resolve` 或 `resolveLoader` 插件时, 解析缓存(resolve caching)中的上下文(context)会被忽略。这解决了性能衰退的问题。

resolve.descriptionFiles

[string]: ['package.json']

用于描述的 JSON 文件。默认:

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    descriptionFiles: ['package.json']
  }
};
```

resolve.enforceExtension

boolean: false

如果是 `true`, 将不允许无扩展名(extension-less)文件。默认如果 `./foo` 有 `.js` 扩展, `require('./foo')` 可以正常运行。但如果启用此选项, 只有 `require('./foo.js')` 能够正常工作。默认:

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    enforceExtension: false
  }
};
```

resolve.enforceModuleExtension

```
boolean: false
```

对模块是否需要使用的扩展（例如 loader）。默认：

webpack.config.js

```
module.exports = {  
  //...  
  resolve: {  
    enforceModuleExtension: false  
  }  
};
```

resolve.extensions

```
[string]: ['.wasm', '.mjs', '.js', '.json']
```

自动解析确定的扩展。默认值为：

webpack.config.js

```
module.exports = {  
  //...  
  resolve: {  
    extensions: ['.wasm', '.mjs', '.js', '.json']  
  }  
};
```

能够使用户在引入模块时不带扩展：

```
import File from '../path/to/file';
```

使用此选项，会覆盖默认数组，这就意味着 webpack 将不再尝试使用默认扩展来解析模块。对于使用其扩展导入的模块，例如，`import SomeFile from './somefile.ext'`，要想正确的解析，一个包含“*”的字符串必须包含在数组中。

resolve.mainFields

```
[string]
```

当从 npm 包中导入模块时（例如，`import * as D3 from 'd3'`），此选项将决定在 `package.json` 中使用哪个字段导入模块。根据 webpack 配置中指定的 `target` 不同，默认值也会有所不同。

当 `target` 属性设置为 `webworker`, `web` 或者没有指定，默认值为：

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    mainFields: ['browser', 'module', 'main']
  }
};
```

对于其他任意的 target (包括 node) , 默认值为:

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    mainFields: ['module', 'main']
  }
};
```

例如, 考虑任意一个名为 upstream 的 library, 其 package.json 包含以下字段:

```
{
  "browser": "build/upstream.js",
  "module": "index"
}
```

在我们 import * as Upstream from 'upstream' 时, 这实际上会从 browser 属性解析文件。在这里 browser 属性是最优先选择的, 因为它是 mainFields 的第一项。同时, 由 webpack 打包的 Node.js 应用程序首先会尝试从 module 字段中解析文件。

resolve.mainFiles

```
[string]: ['index']
```

解析目录时要使用的文件名。

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    mainFiles: ['index']
  }
};
```

resolve.modules

```
[string]: ['node_modules']
```

告诉 webpack 解析模块时应该搜索的目录。

绝对路径和相对路径都能使用，但是要知道它们之间有一点差异。

通过查看当前目录以及祖先路径（即 `./node_modules`, `../node_modules` 等等），相对路径将类似于 Node 查找 '`node_modules`' 的方式进行查找。

使用绝对路径，将只在给定目录中搜索。

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    modules: ['node_modules']
  }
};
```

如果你想要添加一个目录到模块搜索目录，此目录优先于 `node_modules/` 搜索：

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    modules: [path.resolve(__dirname, 'src'), 'node_modules']
  }
};
```

resolve.unsafeCache

```
regex array boolean: true
```

启用，会主动缓存模块，但并不安全。传递 `true` 将缓存一切。默认：

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    unsafeCache: true
  }
};
```

正则表达式，或正则表达式数组，可以用于匹配文件路径或只缓存某些模块。例如，只缓存 `utilities` 模块：

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    unsafeCache: '/src\\utilities/
```

```
};
```

修改缓存路径可能在极少数情况下导致失败。

resolve.plugins

```
[Plugin]
```

应该使用的额外的解析插件列表。它允许插件，如 `DirectoryNamedWebpackPlugin`。

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    plugins: [
      new DirectoryNamedWebpackPlugin()
    ]
  }
};
```

resolve.symlinks

```
boolean: true
```

是否将符号链接(symlink)解析到它们的符号链接位置(symlink location)。默认：

启用时，符号链接(symlink)的资源，将解析为其真实路径，而不是其符号链接(symlink)位置。注意，当使用符号链接 package 包工具时（如 `npm link`），可能会导致模块解析失败。

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    symlinks: true
  }
};
```

resolve.cachePredicate

```
function: function (module) { return true; }
```

决定请求是否应该被缓存的函数。函数传入一个带有 `path` 和 `request` 属性的对象。必须返回一个 boolean 值。

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    cachePredicate: (module) => {
      // additional logic
      return true;
    }
  }
};
```

resolveLoader

`object`

这组选项与上面的 `resolve` 对象的属性集合相同，但仅用于解析 webpack 的 `loader` 包。默认：

webpack.config.js

```
module.exports = {
  //...
  resolveLoader: {
    modules: ['node_modules'],
    extensions: ['.js', '.json'],
    mainFields: ['loader', 'main']
  }
};
```

注意，这里你可以使用别名，并且其他特性类似于 `resolve` 对象。例如，
`{ txt: 'raw-loader' }` 会使用 `raw-loader` 去 shim(填充)
`txt!templates/demo.txt`。

resolveLoader.moduleExtensions

`[string]`

解析 loader 时，用到扩展名(extensions)/后缀(suffixes)。从 webpack 2 开始，我们强烈建议 使用全名，例如 `example-loader`，以尽可能清晰。然而，如果你确实想省略 `-loader`，也就是说只使用 `example`，则可以使用此选项来实现：

webpack.config.js

```
module.exports = {
  //...
  resolveLoader: {
    moduleExtensions: ['-loader']
  }
};
```


优化(optimization)

从 webpack 4 开始，会根据你选择的 `mode` 来执行不同的优化，不过所有的优化还是可以手动配置和重写。

`optimization.minimize`

boolean

告知 webpack 使用 `TerserPlugin` 压缩 bundle。

`production` 模式下，这里默认是 `true`。

`webpack.config.js`

```
module.exports = {
  //...
  optimization: {
    minimize: false
  }
};
```

了解 `mode` 工作机制。

`optimization.minimizer`

[`<plugin>`] and or [function (`compiler`)]

允许你通过提供一个或多个定制过的 `TerserPlugin` 实例，覆盖默认压缩工具 (minimizer)。

`webpack.config.js`

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        cache: true,
        parallel: true,
        sourceMap: true, // Must be set to true if using source-maps in
        terserOptions: {
          // https://github.com/webpack-contrib/terser-webpack-plugin#te
        }
      }),
    ],
  }
};
```

```
};
```

Or, as function:

```
module.exports = {
  optimization: {
    minimizer: [
      (compiler) => {
        const TerserPlugin = require('terser-webpack-plugin');
        new TerserPlugin({ /* your config */ }).apply(compiler);
      }
    ],
  }
};
```

optimization.splitChunks

object

对于动态导入模块， 默认使用 webpack v4+ 提供的全新的通用分块策略(common chunk strategy)。请在 [SplitChunksPlugin](#) 页面中查看配置其行为的可用选项。

optimization.runtimeChunk

object string boolean

将 `optimization.runtimeChunk` 设置为 `true` 或 `"multiple"`，会为每个仅含有 `runtime` 的入口起点添加一个额外 chunk。此设置是如下设置的别名：

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    runtimeChunk: {
      name: entrypoint => `runtime~${entrypoint.name}`
    }
  }
};
```

值 `"single"` 会创建一个在所有生成 chunk 之间共享的运行时文件。此设置是如下设置的别名：

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    runtimeChunk: {
      name: 'runtime'
```

```
        }
    }
};
```

通过将 `optimization.runtimeChunk` 设置为 `object`, 对象中可以设置只有 `name` 属性, 其中属性值可以是名称或者返回名称的函数, 用于为 runtime chunks 命名。

默认值是 `false`: 每个入口 chunk 中直接嵌入 runtime。

对于每个 runtime chunk, 导入的模块会被分别初始化, 因此如果你在同一个页面中引用多个入口起点, 请注意此行为。你或许应该将其设置为 `single`, 或者使用其他只有一个 runtime 实例的配置。

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    runtimeChunk: {
      name: entrypoint => `runtimechunk~${entrypoint.name}`
    }
  }
};
```

optimization.noEmitOnErrors

boolean

在编译出错时, 使用 `optimization.noEmitOnErrors` 来跳过生成阶段(emitting phase)。这可以确保没有生成出错误资源。而 stats 中所有 assets 中的 `emitted` 标记都是 `false`。

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    noEmitOnErrors: true
  }
};
```

如果你正在使用 webpack CLI, 在此插件开启时, webpack 处理过程不会因为错误代码而退出。如果你希望在使用 CLI 时 webpack "失败(fail)", 请查看 `bail` 选项。

optimization.namedModules

```
boolean: false
```

告知 webpack 使用可读取模块标识符(readable module identifiers)，来帮助更好地调试。webpack 配置中如果没有设置此选项，默认会在 `mode development` 启用，在 `mode production` 禁用。

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    namedModules: true
  }
};
```

optimization.namedChunks

```
boolean: false
```

告知 webpack 使用可读取 chunk 标识符(readable chunk identifiers)，来帮助更好地调试。webpack 配置中如果没有设置此选项，默认会在 `mode development` 启用，在 `mode production` 禁用。

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    namedChunks: true
  }
};
```

optimization.moduleIds

```
bool: false string: natural, named, hashed, size, total-size
```

Tells webpack which algorithm to use when choosing module ids. Setting `optimization.moduleIds` to `false` tells webpack that none of built-in algorithms should be used, as custom one can be provided via plugin. By default `optimization.moduleIds` is set to `false`.

The following string values are supported:

Option	natural
Description	Numeric ids in order of usage.
Option	named

Description	Readable ids for better debugging.
Option	hashed
Description	Short hashes as ids for better long term caching.
Option	size
Description	Numeric ids focused on minimal initial download size.
Option	total-size
Description	numeric ids focused on minimal total download size.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    moduleIds: 'hashed'
  }
};
```

optimization.chunkIds

bool: false string: natural, named, size, total-size

Tells webpack which algorithm to use when choosing chunk ids. Setting `optimization.chunkIds` to `false` tells webpack that none of built-in algorithms should be used, as custom one can be provided via plugin. There are couple of defaults for `optimization.chunkIds`:

- if `optimization.occurrenceOrder` is enabled `optimization.chunkIds` is set to '`total-size`'
- Disregarding previous if, if `optimization.namedChunks` is enabled `optimization.chunkIds` is set to '`named`'
- if none of the above, `optimization.namedChunks` will be defaulted to '`natural`'

The following string values are supported:

Option	'natural'
Description	Numeric ids in order of usage.
Option	'named'
Description	Readable ids for better debugging.
Option	'size'

Description	Numeric ids focused on minimal initial download size.
Option	'total-size'
Description	numeric ids focused on minimal total download size.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    chunkIds: 'named'
  }
};
```

optimization.nodeEnv

string bool: false

告知 webpack 将 `process.env.NODE_ENV` 设置为一个给定字符串。如果 `optimization.nodeEnv` 不是 `false`，则会使用 [DefinePlugin](#)，`optimization.nodeEnv` 默认值取决于 `mode`，如果为 falsy 值，则会回退到 "production"。

可能的值有：

- 任何字符串：用于设置 `process.env.NODE_ENV` 的值。
- `false`：不修改/设置 `process.env.NODE_ENV` 的值。

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    nodeEnv: 'production'
  }
};
```

optimization.mangleWasmImports

bool: false

在设置为 `true` 时，告知 webpack 通过将导入修改为更短的字符串，来减少 WASM 大小。这会破坏模块和导出名称。

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    mangleWasmImports: true
  }
};
```

optimization.removeAvailableModules

bool: true

如果模块已经包含在所有父级模块中，告知 webpack 从 chunk 中检测出这些模块，或移除这些模块。将 `optimization.removeAvailableModules` 设置为 `false` 以禁用这项优化。

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    removeAvailableModules: false
  }
};
```

optimization.removeEmptyChunks

bool: true

如果 chunk 为空，告知 webpack 检测或移除这些 chunk。将 `optimization.removeEmptyChunks` 设置为 `false` 以禁用这项优化。

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    removeEmptyChunks: false
  }
};
```

optimization.mergeDuplicateChunks

bool: true

告知 webpack 合并含有相同模块的 chunk。将

`optimization.mergeDuplicateChunks` 设置为 `false` 以禁用这项优化。

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    mergeDuplicateChunks: false
  }
};
```

optimization.flagIncludedChunks

bool

告知 webpack 确定和标记出作为其他 chunk 子集的那些 chunk，其方式是在已经加载过较大的 chunk 之后，就不再去加载这些 chunk 子集。`optimization.flagIncludedChunks` 默认会在 `production mode` 中启用，其他情况禁用。

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    flagIncludedChunks: true
  }
};
```

optimization.occurrenceOrder

bool

Tells webpack to figure out an order of modules which will result in the smallest initial bundle. By default `optimization.occurrenceOrder` is enabled in `production mode` and disabled elsewhere.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    occurrenceOrder: false
  }
};
```

optimization.providedExports

bool

Tells webpack to figure out which exports are provided by modules to generate more efficient code for `export * from ...`. By default `optimization.providedExports` is enabled.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    providedExports: false
  }
};
```

optimization.usedExports

bool

Tells webpack to determine used exports for each module. This depends on `optimization.providedExports`. Information collected by `optimization.usedExports` is used by other optimizations or code generation i.e. exports are not generated for unused exports, export names are mangled to single char identifiers when all usages are compatible. Dead code elimination in minimizers will benefit from this and can remove unused exports. By default `optimization.usedExports` is enabled in `production mode` and disabled elsewhere.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    usedExports: true
  }
};
```

optimization.concatenateModules

bool

Tells webpack to find segments of the module graph which can be safely concatenated into a single module. Depends on `optimization.providedExports` and `optimization.usedExports`. By default `optimization.concatenateModules` is enabled in `production mode` and disabled elsewhere.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    concatenateModules: true
  }
};
```

optimization.sideEffects

bool

Tells webpack to recognise the `sideEffects` flag in `package.json` or rules to skip over modules which are flagged to contain no side effects when exports are not used.

package.json

```
{
  "name": "awesome npm module",
  "version": "1.0.0",
  "sideEffects": false
}
```

Please note that `sideEffects` should be in the npm module's `package.json` file and doesn't mean that you need to set `sideEffects` to `false` in your own project's `package.json` which requires that big module.

`optimization.sideEffects` depends on `optimization.providedExports` to be enabled. This dependency has a build time cost, but eliminating modules has positive impact on performance because of less code generation. Effect of this optimization depends on your codebase, try it for possible performance wins.

By default `optimization.sideEffects` is enabled in production `mode` and disabled elsewhere.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    sideEffects: true
  }
};
```

optimization.portableRecords

bool

`optimization.portableRecords` tells webpack to generate records with relative paths

to be able to move the context folder.

By default `optimization.portableRecords` is disabled. Automatically enabled if at least one of the records options provided to webpack config: `recordsPath`, `recordsInputPath`, `recordsOutputPath`.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    portableRecords: true
  }
};
```

插件(plugins)

`plugins` 选项用于以各种方式自定义 webpack 构建过程。webpack 附带了各种内置插件，可以通过 `webpack.[plugin-name]` 访问这些插件。请查看 [插件页面](#) 获取插件列表和对应文档，但请注意这只是其中一部分，社区中还有许多插件。

注意：本页面仅讨论使用插件，如果你有兴趣编写自己的插件，请访问 [编写一个插件](#) 页面。

plugins

[Plugin]

一组 webpack 插件。例如，`DefinePlugin` 允许你创建可在编译时配置的全局常量。这对需要再开发环境构建和生产环境构建之间产生不同行为来说非常有用。

webpack.config.js

```
module.exports = {
  //...
  plugins: [
    new webpack.DefinePlugin({
      // Definitions...
    })
  ]
};
```

一个复杂示例，使用多个插件，可能看起来就像这样：

webpack.config.js

```
var webpack = require('webpack');
// 导入非 webpack 自带默认插件
var ExtractTextPlugin = require('extract-text-webpack-plugin');
var DashboardPlugin = require('webpack-dashboard/plugin');

// 在配置中添加插件
module.exports = {
  //...
  plugins: [
    new ExtractTextPlugin({
      filename: 'build.min.css',
      allChunks: true,
    }),
    new webpack.IgnorePlugin(/^\.\/locale$/, /moment$/),
    // 编译时 (compile time) 插件
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': '"production"',
    }),
  ]
};
```

```
// webpack-dev-server 强化插件
new DashboardPlugin(),
new webpack.HotModuleReplacementPlugin(),
]
};
```

开发中 server(devServer)

[webpack-dev-server](#) 能够用于快速开发应用程序。起步请查看 [开发指南](#)。

此页面描述影响 webpack-dev-server(简写为： dev-server) 行为的选项。

与 [webpack-dev-middleware](#) 兼容的选项旁边有 。

devServer

object

通过来自 [webpack-dev-server](#) 的这些选项，能够用多种方式改变其行为。这里有一个简单的例子，会 gzip(压缩) 和 serve(服务) 所有来自项目根路径下 `dist/` 目录的文件：

webpack.config.js

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

module.exports = {
  //...
  devServer: {
    contentBase: path.join(__dirname, 'dist'),
    compress: true,
    port: 9000
  }
};
```

当服务器启动时，在解析模块列表之前会有一条消息：

```
http://localhost:9000/
webpack 的服务路径是 /build/
非 webpack 的内容的服务路径是 /path/to/dist/
```

这将给出一些背景知识，就能知道服务器的访问位置，并且知道服务已启动。

如果你通过 Node.js API 来使用 dev-server，`devServer` 中的选项将被忽略。将选项作为第二个参数传入：`new WebpackDevServer(compiler, {...})`。关于如何通过 Node.js API 使用 webpack-dev-server 的示例，请 [查看此处](#)。

请注意，在 [导出多个配置](#) 时，只会使用第一个配置中的 `devServer` 选项，并将其用于数组中的其他所有配置。

如果遇到问题，导航到 `/webpack-dev-server` 路径，可以显示出文件的服务位置。例如，`http://localhost:9000/webpack-dev-server`。

devServer.after

```
function (app, server)
```

在服务内部的所有其他中间件之后， 提供执行自定义中间件的功能。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    after: function(app, server) {
      // 做些有趣的事
    }
  }
};
```

devServer.allowedHosts

array

此选项允许你添加白名单服务， 允许一些开发服务器访问。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    allowedHosts: [
      'host.com',
      'subdomain.host.com',
      'subdomain2.host.com',
      'host2.com'
    ]
  }
};
```

模仿 django 的 ALLOWED_HOSTS， 以 . 开头的值可以用作子域通配符。.host.com 将会匹配 host.com, www.host.com 和 host.com 的任何其他子域名。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    // 这实现了与第一个示例相同的效果,
    // 如果新的子域名需要访问 dev server,
    // 则无需更新您的配置
    allowedHosts: [
      '.host.com',
      'host2.com'
    ]
  }
};
```

```
        ]  
    }  
};
```

想要在 CLI 中使用这个选项，请向 `--allowed-hosts` 选项传入一个以逗号分隔的字符串。

```
webpack-dev-server --entry /entry/file --output-path /output/path --allow-hosts
```

devServer.before

```
function (app, server)
```

在服务内部的所有其他中间件之前，提供执行自定义中间件的功能。这可以用来配置自定义处理程序，例如：

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    before: function(app, server) {  
      app.get('/some/path', function(req, res) {  
        res.json({ custom: 'response' });  
      });  
    }  
  }  
};
```

devServer.bonjour

此选项在启动时，通过 [ZeroConf](#) 网络广播服务

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    bonjour: true  
  }  
};
```

CLI 用法

```
webpack-dev-server --bonjour
```

devServer.clientLogLevel

```
string: 'none' | 'info' | 'error' | 'warning'
```

当使用内联模式(*inline mode*)时，会在开发工具(DevTools)的控制台(console)显示消息，例如：在重新加载之前，在一个错误之前，或者 模块热替换(Hot Module Replacement) 启用时。默认值是 `info`。

`devServer.clientLogLevel` 可能会显得很繁琐，你可以通过将其设置为 '`none`' 来关闭 log。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    clientLogLevel: 'none'
  }
};
```

CLI 用法

```
webpack-dev-server --client-log-level none
```

devServer.color - 只用于命令行工具(CLI)

boolean

启用/禁用控制台的彩色输出。

```
webpack-dev-server --color
```

devServer.compress

boolean

一切服务都启用 `gzip` 压缩：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    compress: true
  }
};
```

CLI 用法

```
webpack-dev-server --compress
```

devServer.contentBase

boolean: false string [string] number

告诉服务器从哪个目录中提供内容。只有在你想要提供静态文件时才需要。`devServer.publicPath` 将用于确定应该从哪里提供 bundle，并且此选项优先。

推荐使用一个绝对路径。

默认情况下，将使用当前工作目录作为提供内容的目录。将其设置为 `false` 以禁用 `contentBase`。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    contentBase: path.join(__dirname, 'public')
  }
};
```

也可以从多个目录提供内容：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    contentBase: [path.join(__dirname, 'public'), path.join(__dirname,
    )]
  }
};
```

CLI 用法

```
webpack-dev-server --content-base /path/to/content/dir
```

devServer.disableHostCheck

boolean

设置为 `true` 时，此选项绕过主机检查。不建议这样做，因为不检查主机的应用程序容易受到 DNS 重新连接攻击。

webpack.config.js

```
module.exports = {
  //...
```

```
devServer: {  
  disableHostCheck: true  
}  
};
```

CLI 用法

```
webpack-dev-server --disable-host-check
```

devServer.filename □

string

在 lazy mode(惰性模式) 中，此选项可减少编译。默认在 lazy mode(惰性模式)，每个请求结果都会产生全新的编译。使用 `filename`，可以只在某个文件被请求时编译。

如果 `output.filename` 设置为 '`bundle.js`'，`devServer.filename` 用法如下：

webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    filename: 'bundle.js'  
  },  
  devServer: {  
    lazy: true,  
    filename: 'bundle.js'  
  }  
};
```

现在只有在请求 `/bundle.js` 时候，才会编译 `bundle`。

`filename` 在不使用 lazy mode(惰性模式) 时没有效果。

devServer.headers □

object

在所有响应中添加头部内容：

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    headers: {  
      'X-Custom-Foo': 'bar'  
    }  
  }  
};
```

```
        }
    }
};
```

devServer.historyApiFallback

boolean object

当使用 [HTML5 History API](#) 时，任意的 404 响应都可能需要被替代为 index.html。 devServer.historyApiFallback 默认禁用。通过传入以下启用：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    historyApiFallback: true
  }
};
```

通过传入一个对象，比如使用 rewrites 这个选项，此行为可进一步地控制：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    historyApiFallback: {
      rewrites: [
        { from: /^\/$/, to: '/views/landing.html' },
        { from: /^\/subpage/, to: '/views/subpage.html' },
        { from: './', to: '/views/404.html' }
      ]
    }
  }
};
```

当路径中使用点(dot)（常见于 Angular），你可能需要使用 disableDotRule：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    historyApiFallback: {
      disableDotRule: true
    }
  }
};
```

CLI 用法

```
webpack-dev-server --history-api-fallback
```

更多选项和信息，查看 [connect-history-api-fallback](#) 文档。

devServer.host

string

指定使用一个 host。默认是 `localhost`。如果你希望服务器外部可访问，指定如下：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    host: '0.0.0.0'
  }
};
```

CLI 用法

```
webpack-dev-server --host 0.0.0.0
```

devServer.hot

boolean

启用 webpack 的 模块热替换 功能：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    hot: true
  }
};
```

注意，必须有 `webpack.HotModuleReplacementPlugin` 才能完全启用 HMR。如果 `webpack` 或 `webpack-dev-server` 是通过 `--hot` 选项启动的，那么这个插件会被自动添加，所以你可能不需要把它添加到 `webpack.config.js` 中。关于更多信息，请查看 [HMR 概念](#) 页面。

devServer.hotOnly

boolean

Enables Hot Module Replacement (see `devServer.hot`) without page refresh as fallback in case of build failures.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    hotOnly: true
  }
};
```

CLI 用法

```
webpack-dev-server --hot-only
```

devServer.https

boolean object

默认情况下，dev-server 通过 HTTP 提供服务。也可以选择带有 HTTPS 的 HTTP/2 提供服务：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    https: true
  }
};
```

以上设置使用了自签名证书，但是你可以提供自己的：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    https: {
      key: fs.readFileSync('/path/to/server.key'),
      cert: fs.readFileSync('/path/to/server.crt'),
      ca: fs.readFileSync('/path/to/ca.pem'),
    }
  }
};
```

此对象直接传递到 Node.js HTTPS 模块，所以更多信息请查看 [HTTPS 文档](#)。

CLI 用法

```
webpack-dev-server --https
```

想要向 CLI 传入你自己的证书，请使用以下选项

```
webpack-dev-server --https --key /path/to/server.key --cert /path/to/se
```

devServer.index

string

被作为索引文件的文件名。

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    index: 'index.html'  
  }  
};
```

devServer.info - 只用于命令行工具(CLI)

boolean

输出 cli 信息。默认启用。

```
webpack-dev-server --info=false
```

devServer.inline

boolean

在 dev-server 的两种不同模式之间切换。默认情况下，应用程序启用 **内联模式** (*inline mode*)。这意味着一段处理实时重载的脚本被插入到你的包(bundle)中，并且构建消息将会出现在浏览器控制台。

也可以使用 **iframe 模式**，它在通知栏下面使用 `<iframe>` 标签，包含了关于构建的消息。切换到 **iframe 模式**：

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    inline: false  
  }  
};
```

```
};
```

CLI 用法

```
webpack-dev-server --inline=false
```

推荐使用 模块热替换 的内联模式，因为它包含来自 websocket 的 HMR 触发器。轮询模式可以作为替代方案，但需要一个额外的入口点：'webpack/hot/poll?1000'。

devServer.lazy

boolean

当启用 `devServer.lazy` 时，dev-server 只有在请求时才编译包(bundle)。这意味着 webpack 不会监视任何文件改动。我们称之为惰性模式。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    lazy: true
  }
};
```

CLI 用法

```
webpack-dev-server --lazy
```

watchOptions 在使用惰性模式时无效。

如果使用命令行工具(CLI)，请确保内联模式(**inline mode**)被禁用。

devServer.noInfo

boolean

告诉 dev-server 隐藏 webpack bundle 信息之类的消息。`devServer.noInfo` 默认禁用。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    noInfo: true
  }
};
```

```
};
```

devServer.open

```
boolean string
```

告诉 dev-server 在 server 启动后打开浏览器。默认禁用。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    open: true
  }
};
```

If no browser is provided (as shown above), your default browser will be used. To specify a different browser, just pass its name instead of boolean:

```
module.exports = {
  //...
  devServer: {
    open: 'Google Chrome'
  }
};
```

CLI 用法

```
webpack-dev-server --open
```

Or with specified browser:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    open: 'Chrome'
  }
};
```

And via the CLI

```
webpack-dev-server --open 'Chrome'
```

The browser application name is platform dependent. Don't hard code it in reusable modules. For example, '`Chrome`' is Google Chrome on macOS, '`google-chrome`' on Linux and '`chrome`' on Windows.

devServer.openPage

string

指定打开浏览器时的导航页面。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    openPage: '/different/page'
  }
};
```

CLI 用法

```
webpack-dev-server --open-page "/different/page"
```

devServer.overlay

boolean object: { boolean errors, boolean warnings }

当出现编译器错误或警告时，在浏览器中显示全屏覆盖层。默认禁用。如果你想要只显示编译器错误：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    overlay: true
  }
};
```

如果想要显示警告和错误：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    overlay: {
      warnings: true,
      errors: true
    }
  }
};
```

devServer.pfx

string

当CLI用法时，路径是一个 .pfx 后缀的 SSL 文件。如果用在选项中，它应该是 .pfx 文件的字节流(bytestream)。

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    pfx: '/path/to/file.pfx'  
  }  
};
```

CLI 用法

```
webpack-dev-server --pfx /path/to/file.pfx
```

devServer.pfxPassphrase

string

SSL PFX文件的密码。

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    pfxPassphrase: 'passphrase'  
  }  
};
```

CLI 用法

```
webpack-dev-server --pfx-passphrase passphrase
```

devServer.port

number

指定要监听请求的端口号：

webpack.config.js

```
module.exports = {
```

```
//...
devServer: {
  port: 8080
}
};
```

CLI 用法

```
webpack-dev-server --port 8080
```

devServer.proxy

object [object, function]

如果你有单独的后端开发服务器 API，并且希望在同域名下发送 API 请求，那么代理某些 URL 会很有用。

dev-server 使用了非常强大的 [http-proxy-middleware](#) 包。更多高级用法，请查阅其 [文档](#)。Note that some of `http-proxy-middleware`'s features do not require a `target` key, e.g. its `router` feature, but you will still need to include a `target` key in your config here, otherwise `webpack-dev-server` won't pass it along to `http-proxy-middleware`).

在 `localhost:3000` 上有后端服务的话，你可以这样启用代理：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    proxy: {
      '/api': 'http://localhost:3000'
    }
  }
};
```

请求到 `/api/users` 现在会被代理到请求 `http://localhost:3000/api/users`。

如果你不想始终传递 `/api`，则需要重写路径：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    proxy: {
      '/api': {
        target: 'http://localhost:3000',
        pathRewrite: {'^/api' : ''}
      }
    }
  }
};
```

```
    }  
};
```

默认情况下，不接受运行在 HTTPS 上，且使用了无效证书的后端服务器。如果你想要接受，修改配置如下：

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    proxy: {  
      '/api': {  
        target: 'https://other-server.example.com',  
        secure: false  
      }  
    }  
  }  
};
```

有时你不想代理所有的请求。可以基于一个函数的返回值绕过代理。

在函数中你可以访问请求体、响应体和代理选项。必须返回 `false` 或路径，来跳过代理请求。

例如：对于浏览器请求，你想要提供一个 HTML 页面，但是对于 API 请求则保持代理。你可以这样做：

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    proxy: {  
      '/api': {  
        target: 'http://localhost:3000',  
        bypass: function(req, res, proxyOptions) {  
          if (req.headers.accept.indexOf('html') !== -1) {  
            console.log('Skipping proxy for browser request.');//  
            return '/index.html';  
          }  
        }  
      }  
    }  
  }  
};
```

如果你想要代理多个路径特定到同一个 target 下，你可以使用由一个或多个「具有 `context` 属性的对象」构成的数组：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    proxy: [
      {
        context: ['/auth', '/api'],
        target: 'http://localhost:3000',
      }
    ]
  }
};
```

注意， 默认情况下， 根请求不会被代理。要启用根代理， 应该将 `devServer.index` 选项指定为 falsy 值：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    index: '', // specify to enable root proxying
    host: '...',
    contentBase: '...',
    proxy: {
      context: () => true,
      target: 'http://localhost:1234'
    }
  }
};
```

The origin of the host header is kept when proxying by default, you can set `changeOrigin` to `true` to override this behaviour. It is useful in some cases like using name-based virtual hosted sites.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    proxy: {
      '/api': 'http://localhost:3000',
      changeOrigin: true
    }
  }
};
```

devServer.progress - 只用于命令行工具(CLI)

boolean

将运行进度输出到控制台。

```
webpack-dev-server --progress
```

devServer.public

string

当使用内联模式(*inline mode*)并代理 dev-server 时，内联的客户端脚本并不总是知道要连接到什么地方。它会尝试根据 `window.location` 来猜测服务器的 URL，但是如果失败，你需要使用这个配置。

例如，dev-server 被代理到 nginx，并且在 `myapp.test` 上可用：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    public: 'myapp.test:80'
  }
};
```

CLI 用法

```
webpack-dev-server --public myapp.test:80
```

devServer.publicPath □

string

此路径下的打包文件可在浏览器中访问。

假设服务器运行在 `http://localhost:8080` 并且 `output.filename` 被设置为 `bundle.js`。默认 `devServer.publicPath` 是 `'/'`，所以你的包(bundle)可以通过 `http://localhost:8080/bundle.js` 访问。

修改 `devServer.publicPath`，将 bundle 放在指定目录下：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    publicPath: '/assets/'
  }
};
```

现在可以通过 `http://localhost:8080/assets/bundle.js` 访问 bundle。

确保 `devServer.publicPath` 总是以斜杠(/)开头和结尾。

也可以使用一个完整的 URL。这是 模块热替换 所必需的。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    publicPath: 'http://localhost:8080/assets/'
  }
};
```

可以通过 `http://localhost:8080/assets/bundle.js` 访问 bundle。

`devServer.publicPath` 和 `output.publicPath` 一样被推荐。

devServer. quiet

boolean

启用 `devServer.quiet` 后，除了初始启动信息之外的任何内容都不会被打印到控制台。这也意味着来自 webpack 的错误或警告在控制台不可见。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    quiet: true
  }
};
```

CLI 用法

```
webpack-dev-server --quiet
```

devServer. setup

```
function (app, server)
```

此选项 已废弃，并将在 v3.0.0 中被删除。应当使用 `devServer.before`。

这里你可以访问 Express 应用程序对象，并且添加你的自定义中间件。例如，想要为一些路径定义自定义处理函数：

webpack.config.js

```
module.exports = {
  //...
```

```
devServer: {  
  setup: function(app, server) {  
    app.get('/some/path', function(req, res) {  
      res.json({ custom: 'response' });  
    });  
  }  
};
```

devServer.socket

string

用于监听的 Unix socket (而不是 host)。

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    socket: 'socket'  
  }  
};
```

CLI 用法

```
webpack-dev-server --socket socket
```

devServer.staticOptions

可以用于对 `contentBase` 路径下提供的静态文件，进行高级选项配置。有关可能的选项，请查看 [Express 文档](#)。

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    staticOptions: {  
      redirect: false  
    }  
  }  
};
```

这只有在使用 `devServer.contentBase` 是一个 string 时才有效。

devServer.stats □

```
string: 'none' | 'errors-only' | 'minimal' | 'normal' | 'verbose'
```

object

通过此选项，可以精确控制要显示的 bundle 信息。如果你想要显示一些打包信息，但又不是显示全部，这可能是一个不错的妥协。

想要在 bundle 中只显示错误：

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    stats: 'errors-only'  
  }  
};
```

关于更多信息，请查看 [stats 文档](#)。

此选项在配置 quiet 或 noInfo 时无效。

devServer.stdin - 只用于命令行工具(CLI)

boolean

此选项在 stdin 结束时关闭服务。

```
webpack-dev-server --stdin
```

devServer.useLocalIp

boolean

此选项允许浏览器使用本地 IP 打开。

webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    useLocalIp: true  
  }  
};
```

CLI 用法

```
webpack-dev-server --useLocalIp
```

devServer.watchContentBase

boolean

告知 dev-server, serve(服务) `devServer.contentBase` 选项下的文件。开启此选项后，在文件修改之后，会触发一次完整的页面重载。

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    watchContentBase: true
  }
};
```

CLI 用法

```
webpack-dev-server --watch-content-base
```

devServer.watchOptions □

object

与监视文件相关的控制选项。

webpack 使用文件系统(file system)获取文件改动的通知。在某些情况下，不会正常工作。例如，当使用 Network File System (NFS) 时。Vagrant 也有很多问题。在这些情况下，请使用轮询：

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    watchOptions: {
      poll: true
    }
  }
};
```

如果这对文件系统来说太重了的话，你可以修改间隔时间（以毫秒为单位），将其设置为一个整数。

更多选项请查看 [WatchOptions](#)。

devServer.writeToDisk □

```
boolean: false function (filePath)
```

Tells devServer to write generated assets to the disk.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    writeToDisk: true
  }
};
```

Providing a Function to devServer.writeToDisk can be used for filtering. The function follows the same premise as [Array#filter](#) in which a boolean return value tells if the file should be written to disk.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    writeToDisk: (filePath) => {
      return /superman\.css$/.test(filePath);
    }
  }
};
```

devtool

此选项控制是否生成，以及如何生成 source map。

使用 `SourceMapDevToolPlugin` 进行更细粒度的配置。查看 `source-map-loader` 来处理已有的 source map。

devtool

`string false`

选择一种 `source map` 格式来增强调试过程。不同的值会明显影响到构建(build)和重新构建(rebuild)的速度。

webpack 仓库中包含一个 显示所有 `devtool` 变体效果的示例。这些例子或许会有助于你理解这些差异之处。

你可以直接使用 `SourceMapDevToolPlugin/EvalSourceMapDevToolPlugin` 来替代使用 `devtool` 选项，因为它有更多的选项。切勿同时使用 `devtool` 选项和 `SourceMapDevToolPlugin/EvalSourceMapDevToolPlugin` 插件。`devtool` 选项在内部添加过这些插件，所以你最终将应用两次插件。

devtool	(none)
构建速度	+++
重新构建速度	+++
生产环境	yes
品质(quality)	打包后的代码
devtool	eval
构建速度	+++
重新构建速度	+++
生产环境	no
品质(quality)	生成后的代码
devtool	cheap-eval-source-map
构建速度	+
重新构建速度	++
生产环境	no
品质(quality)	转换过的代码（仅限行）
devtool	cheap-module-eval-source-map
构建速度	○

重新构建速度	+
生产环境	++
品质(quality)	no 原始源代码（仅限行）
devtool	eval-source-map
构建速度	--
重新构建速度	+
生产环境	no
品质(quality)	原始源代码
devtool	cheap-source-map
构建速度	+
重新构建速度	o
生产环境	yes
品质(quality)	转换过的代码（仅限行）
devtool	cheap-module-source-map
构建速度	o
重新构建速度	-
生产环境	yes
品质(quality)	原始源代码（仅限行）
devtool	inline-cheap-source-map
构建速度	+
重新构建速度	o
生产环境	no
品质(quality)	转换过的代码（仅限行）
devtool	inline-cheap-module-source-map
构建速度	o
重新构建速度	-
生产环境	no
品质(quality)	原始源代码（仅限行）
devtool	source-map
构建速度	--
重新构建速度	--
生产环境	yes
品质(quality)	原始源代码
devtool	inline-source-map

构建速度	--
重新构建速度	--
生产环境	no
品质(quality)	原始源代码
devtool	hidden-source-map
构建速度	--
重新构建速度	--
生产环境	yes
品质(quality)	原始源代码
devtool	nosources-source-map
构建速度	--
重新构建速度	--
生产环境	yes
品质(quality)	无源代码内容

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

其中一些值适用于开发环境，一些适用于生产环境。对于开发环境，通常希望更快的 source map，需要添加到 bundle 中以增加体积为代价，但是对于生产环境，则希望更精准的 source map，需要从 bundle 中分离并独立存在。

Chrome 中的 source map 有一些问题。我们需要你的帮助！。

查看 `output.sourceMapFilename` 自定义生成的 source map 的文件名。

品质说明(quality)

打包后的代码 - 将所有生成的代码视为一大块代码。你看不到相互分离的模块。

生成后的代码 - 每个模块相互分离，并用模块名称进行注释。可以看到 webpack 生成的代码。示例：你会看到类似 `var module__WEBPACK_IMPORTED_MODULE_1__ = __webpack_require__(42); module__WEBPACK_IMPORTED_MODULE_1__.a();`，而不是 `import {test} from "module"; test();`。

转换过的代码 - 每个模块相互分离，并用模块名称进行注释。可以看到 webpack 转换前、loader 转译后的代码。示例：你会看到类似 `import {test} from "module"; var A = function(_test) { ... }(test);`，而不是 `import {test} from "module"; class A extends test {}`。

原始源代码 - 每个模块相互分离，并用模块名称进行注释。你会看到转译之前的代码，正如编写它时。这取决于 loader 支持。

无源代码内容 - source map 中不包含源代码内容。浏览器通常会尝试从 web 服务器或文件系统加载源代码。你必须确保正确设置 `output.devtoolModuleFilenameTemplate`, 以匹配源代码的 url。

(仅限行) - source map 被简化为每行一个映射。这通常意味着每个语句只有一个映射 (假设你使用这种方式)。这会妨碍你在语句级别上调试执行, 也会妨碍你在每行的一些列上设置断点。与压缩后的代码组合后, 映射关系是不可能实现的, 因为压缩工具通常只会输出一行。

对于开发环境

以下选项非常适合开发环境:

`eval` - 每个模块都使用 `eval()` 执行, 并且都有 `//@ sourceURL`。此选项会非常快地构建。主要缺点是, 由于会映射到转换后的代码, 而不是映射到原始代码 (没有从 loader 中获取 source map), 所以不能正确的显示行数。

`eval-source-map` - 每个模块使用 `eval()` 执行, 并且 source map 转换为 DataUrl 后添加到 `eval()` 中。初始化 source map 时比较慢, 但是会在重新构建时提供比较快的速度, 并且生成实际的文件。行数能够正确映射, 因为会映射到原始代码中。它会生成用于开发环境的最佳品质的 source map。

`cheap-eval-source-map` - 类似 `eval-source-map`, 每个模块使用 `eval()` 执行。这是 "cheap(低开销)" 的 source map, 因为它没有生成列映射(column mapping), 只是映射行数。它会忽略源自 loader 的 source map, 并且仅显示转译后的代码, 就像 `eval devtool`。

`cheap-module-eval-source-map` - 类似 `cheap-eval-source-map`, 并且, 在这种情况下, 源自 loader 的 source map 会得到更好的处理结果。然而, loader source map 会被简化为每行一个映射(mapping)。

特定场景

以下选项对于开发环境和生产环境并不理想。他们是一些特定场景下需要的, 例如, 针对一些第三方工具。

`inline-source-map` - source map 转换为 DataUrl 后添加到 bundle 中。

`cheap-source-map` - 没有列映射(column mapping)的 source map, 忽略 loader source map。

`inline-cheap-source-map` - 类似 `cheap-source-map`, 但是 source map 转换为 DataUrl 后添加到 bundle 中。

`cheap-module-source-map` - 没有列映射(column mapping)的 source map, 将 loader

source map 简化为每行一个映射(mapping)。

`inline-cheap-module-source-map` - 类似 `cheap-module-source-map`, 但是 source map 转换为 DataUrl 添加到 bundle 中。

对于生产环境

这些选项通常用于生产环境中:

`(none)` (省略 `devtool` 选项) - 不生成 source map。这是一个不错的选择。

`source-map` - 整个 source map 作为一个单独的文件生成。它为 bundle 添加了一个引用注释, 以便开发工具知道在哪里可以找到它。

你应该将你的服务器配置为, 不允许普通用户访问 source map 文件!

`hidden-source-map` - 与 `source-map` 相同, 但不会为 bundle 添加引用注释。如果你只想 source map 映射那些源自错误报告的错误堆栈跟踪信息, 但不想为浏览器开发工具暴露你的 source map, 这个选项会很有用。

你不应将 source map 文件部署到 web 服务器。而是只将其用于错误报告工具。

`nosources-source-map` - 创建的 source map 不包含 `sourcesContent` (源代码内容)。它可以用来映射客户端上的堆栈跟踪, 而无须暴露所有的源代码。你可以将 source map 文件部署到 web 服务器。

这仍然会暴露反编译后的文件名和结构, 但它不会暴露原始代码。

在使用 `terser-webpack-plugin` 时, 你必须提供 `sourceMap: true` 选项来启用 source map 支持。

构建目标(targets)

webpack 能够为多种环境或 *target* 构建编译。想要理解什么是 *target* 的详细信息，请阅读 [target 概念页面](#)。

target

string | function (compiler)

告知 webpack 为目标(target)指定一个环境。

string

通过 [WebpackOptionsApply](#)，可以支持以下字符串值：

选项	async-node
描述	编译为类 Node.js 环境可用（使用 fs 和 vm 异步加载分块）
选项	electron-main
描述	编译为 Electron 主进程。
选项	electron-renderer
描述	编译为 Electron 渲染进程，使用 JsonpTemplatePlugin, FunctionModulePlugin 来为浏览器环境提供目标，使用 NodeTargetPlugin 和 ExternalPlugin 为 CommonJS 和 Electron 内置模块提供目标。
选项	node
描述	编译为类 Node.js 环境可用（使用 Node.js require 加载 chunk）
选项	node-webkit
描述	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）
选项	web
描述	编译为类浏览器环境里可用（默认）
选项	webworker
描述	编译成一个 WebWorker

例如，当 target 设置为 "electron-main"，webpack 引入多个 electron 特定的变量。有关使用哪些模板和 externals 的更多信息，你可以直接参考 webpack 源码。

function

如果传入一个函数，此函数调用时会传入一个 compiler 作为参数。如果以上列表中没有一个预定义的目标(target)符合你的要求，请将其设置为一个函数。

例如，如果你不需要使用以上任何插件：

```
const options = {
  target: () => undefined
};
```

或者可以使用你想要指定的插件

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

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

watch 和 watchOptions

webpack 可以监听文件变化，当它们修改后会重新编译。这个页面介绍了如何启用这个功能，以及当 watch 无法正常运行的时候你可以做的一些调整。

watch

boolean: false

启用 Watch 模式。这意味着在初始构建之后，webpack 将继续监听任何已解析文件的更改。

webpack.config.js

```
module.exports = {  
  //...  
  watch: true  
};
```

[webpack-dev-server](#) 和 [webpack-dev-middleware](#) 里 Watch 模式默认开启。

watchOptions

object

一组用来定制 Watch 模式的选项：

webpack.config.js

```
module.exports = {  
  //...  
  watchOptions: {  
    aggregateTimeout: 300,  
    poll: 1000  
  }  
};
```

watchOptions.aggregateTimeout

number: 300

当第一个文件更改，会在重新构建前增加延迟。这个选项允许 webpack 将这段时间内进行的任何其他更改都聚合到一次重新构建里。以毫秒为单位：

```
module.exports = {
```

```
//...
watchOptions: {
  aggregateTimeout: 600
}
};
```

watchOptions.ignored

RegExp anymatch

对于某些系统，监听大量文件系统会导致大量的 CPU 或内存占用。这个选项可以排除一些巨大的文件夹，例如 `node_modules`:

webpack.config.js

```
module.exports = {
  //...
  watchOptions: {
    ignored: /node_modules/
  }
};
```

也可以使用多种 anymatch 模式:

webpack.config.js

```
module.exports = {
  //...
  watchOptions: {
    ignored: ['files/**/*.js', 'node_modules']
  }
};
```

如果你使用 `require.context`，webpack 会观察你的整个目录。你应该忽略一些文件和/或(and/or)目录，以便那些不需要监听的文件修改，不会触发重新构建。

watchOptions.poll

boolean: false number

通过传递 `true` 开启 polling，或者指定毫秒为单位进行轮询。

webpack.config.js

```
module.exports = {
  //...
  watchOptions: {
    poll: 1000 // 每秒检查一次变动
  }
};
```

```
};
```

如果监听没生效，试试这个选项吧。Watch 在 NFS 和 VirtualBox 机器上不适用。

info-verbosity

```
string: 'none', 'info', 'verbose'
```

控制生命周期消息的详细程度，例如 Started watching files(开始监听文件)… 日志。将 `info-verbosity` 设置为 `verbose`，还会额外在增量构建的开始和结束时，向控制台发送消息。`info-verbosity` 默认设置为 `info`。

```
webpack --watch --info-verbosity verbose
```

故障排除

如果您遇到任何问题，请查看以下注意事项。对于 webpack 为何会忽略文件修改，这里有多种原因。

发现修改，但并未做处理

在运行 webpack 时，通过使用 `--progress` 标志，来验证文件修改后，是否没有通知 webpack。如果进度显示保存，但没有输出文件，则可能是配置问题，而不是文件监视问题。

```
webpack --watch --progress
```

没有足够的文件观察者

确认系统中有足够多的文件观察者。如果这个值太低，webpack 中的文件观察者将无法识别修改：

```
cat /proc/sys/fs/inotify/max_user_watches
```

Arch 用户，请将 `fs.inotify.max_user_watches=524288` 添加到 `/etc/sysctl.d/99-sysctl.conf` 中，然后执行 `sudo sysctl --system`。Ubuntu 用户（可能还有其他用户）请执行：`echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf && sudo sysctl -p`。

macOS fsevents Bug

在 macOS 中，某些情况下文件夹可能会损坏。请参阅这篇文章。

Windows Paths

因为 webpack 期望获得多个配置选项的绝对路径（如 `__dirname + '/app/folder'`），所以 Windows 的路径分隔符 \ 可能会破坏某些功能。

使用正确的分隔符。即 `path.resolve(__dirname, 'app/folder')` 或 `path.join(__dirname, 'app', 'folder')`。

Vim

在某些机器上，Vim 预先将 `backupcopy` 选项设置为 `auto`。这可能会导致系统的文件监视机制出现问题。将此选项设置为 `yes` 可以确保创建文件的副本，并在保存时覆盖原始文件。

```
:set backupcopy=yes
```

在 WebStorm 中保存

使用 JetBrains WebStorm IDE 时，你可能会发现保存修改过的文件，并不会按照预期触发观察者。尝试在设置中禁用安全写入(`safe write`)选项，该选项确定在原文件被覆盖之前，文件是否先保存到临时位置：取消选中 `File > {Settings | Preferences} > Appearance & Behavior > System Settings > Use "safe write" (save changes to a temporary file first)`。

外部扩展(externals)

`externals` 配置选项提供了「从输出的 bundle 中排除依赖」的方法。相反，所创建的 bundle 依赖于那些存在于用户环境(consumer's environment)中的依赖。此功能通常对 `library` 开发人员来说是最有用的，然而也会有各种各样的应用程序用到它。

用户(consumer)，在这里是指，引用了「使用 webpack 打包的 library」的所有终端用户的应用程序(end user application)。

externals

string object function regex

防止将某些 `import` 的包(package)打包到 bundle 中，而是在运行时(runtime)再去从外部获取这些扩展依赖(*external dependencies*)。

例如，从 CDN 引入 `jQuery`，而不是把它打包：

index.html

```
<script
  src="https://code.jquery.com/jquery-3.1.0.js"
  integrity="sha256-slogkvB1K3V0kzAI8QITxV3VzpOnkeNVsKvtkYLMjf
  crossorigin="anonymous">
</script>
```

webpack.config.js

```
module.exports = {
  //...
  externals: {
    jquery: 'jQuery'
  }
};
```

这样就剥离了那些不需要改动的依赖模块，换句话，下面展示的代码还可以正常运行：

```
import $ from 'jquery';

$('.my-element').animate(* ... *);
```

具有外部依赖(*external dependency*)的 bundle 可以在各种模块上下文(module context)中使用，例如 `CommonJS`, `AMD`, 全局变量和 `ES2015` 模块。外部 library 可能是以下任何一种形式：

- **root**: 可以通过一个全局变量访问 library (例如, 通过 script 标签)。
- **commonjs**: 可以将 library 作为一个 CommonJS 模块访问。
- **commonjs2**: 和上面的类似, 但导出的是 `module.exports.default`。
- **amd**: 类似于 `commonjs`, 但使用 AMD 模块系统。

可以接受各种语法.....

string

请查看上面的例子。属性名称是 `jquery`, 表示应该排除 `import $ from 'jquery'` 中的 `jquery` 模块。为了替换这个模块, `jQuery` 的值将被用来检索一个全局的 `jQuery` 变量。换句话说, 当设置为一个字符串时, 它将被视为全局的 (定义在上面和下面)。

array

```
module.exports = [
  //...
  externals: {
    subtract: ['./math', 'subtract']
  }
};
```

`subtract: ['./math', 'subtract']` 转换为父子结构, 其中 `./math` 是父模块, 而 `bundle` 只引用 `subtract` 变量下的子集。

object

An object with { `root`, `amd`, `commonjs`, ... } is only allowed for `libraryTarget: 'umd'`. It's not allowed for other library targets.

```
module.exports = {
  //...
  externals : {
    react: 'react'
  },
  // 或者

  externals : {
    lodash : {
      commonjs: 'lodash',
      amd: 'lodash',
      root: '_' // 指向全局变量
    }
  },
  // 或者

  externals : {
```

```
subtract : {
  root: ['math', 'subtract']
}
};

};
```

此语法用于描述外部 library 所有可用的访问方式。这里 `lodash` 这个外部 library 可以在 AMD 和 CommonJS 模块系统中通过 `lodash` 访问，但在全局变量形式下用 `_` 访问。`subtract` 可以通过全局 `math` 对象下的属性 `subtract` 访问（例如 `window['math']['subtract']`）。

function

对于 webpack 外部化，通过定义函数来控制行为，可能会很有帮助。例如，`webpack-node-externals` 能够排除 `node_modules` 目录中所有模块，还提供一些选项，比如白名单 package(whitelist package)。

基本配置如下：

```
module.exports = {
  //...
  externals: [
    function(context, request, callback) {
      if (/^yourregex$/.test(request)){
        return callback(null, 'commonjs ' + request);
      }
      callback();
    }
  ]
};

'commonjs'+ request 定义了需要外部化的模块类型。
```

regex

匹配给定正则表达式的每个依赖，都将从输出 bundle 中排除。

```
module.exports = {
  //...
  externals: /^ jquery | \$ /i
};
```

这个示例中，所有名为 `jQuery` 的依赖（忽略大小写），或者 `$`，都会被外部化。

Combining syntaxes

Sometimes you may want to use a combination of the above syntaxes. This can be done in the following manner:

```
module.exports = {
//...
externals: [
{
    // String
    react: 'react',
    // Object
    lodash : {
        commonjs: 'lodash',
        amd: 'lodash',
        root: '_' // indicates global variable
    },
    // Array
    subtract: ['./math', 'subtract']
},
// Function
function(context, request, callback) {
    if (/^yourregex$/.test(request)){
        return callback(null, 'commonjs ' + request);
    }
    callback();
},
// Regex
/^ jquery|\.$/i
]
};
```

关于如何使用此 `externals` 配置的更多信息，请参考[如何编写 library](#)。

Node.js

这些选项可以配置是否 polyfill 或 mock 某些 Node.js 全局变量和模块。这可以使最初为 Node.js 环境编写的代码，在其他环境（如浏览器）中运行。

此功能由 webpack 内部的 `NodeStuffPlugin` 插件提供。如果 target 是 "web"（默认）或 "webworker"，那么 `NodeSourcePlugin` 插件也会被激活。

node

object

是一个对象，其中每个属性都是 Node.js 全局变量或模块的名称，每个 value 是以下其中之一……

- `true`: 提供 polyfill。
- `"mock"`: 提供 mock 实现预期接口，但功能很少或没有。
- `"empty"`: 提供空对象。
- `false`: 什么都不提供。预期获取此对象的代码，可能会因为获取不到此对象，触发 `ReferenceError` 而崩溃。尝试使用 `require('modulename')` 导入模块的代码，可能会触发 `Cannot find module "modulename"` 错误。

注意，不是每个 Node 全局变量都支持所有选项。对于不支持的键值组合 (property-value combination)，compiler 会抛出错误。更多细节请查看接下来的章节。

这里是默认值：

```
module.exports = {
  //...
  node: {
    console: false,
    global: true,
    process: true,
    __filename: 'mock',
    __dirname: 'mock',
    Buffer: true,
    setImmediate: true
    // 更多选项，请查看“其他 Node.js 核心库”。
  }
};
```

从 webpack 3.0.0 开始，`node` 选项可能被设置为 `false`，以完全关闭 `NodeStuffPlugin` 和 `NodeSourcePlugin` 插件。

node.console

boolean | "mock"

默认值: false

浏览器提供一个 `console` 对象，具有非常类似 Node.js `console` 的接口，所以通常不需要 polyfill。

node.process

boolean | "mock"

默认值: true

node.global

boolean

默认值: true

关于此对象的准确行为，请查看源码。

node.__filename

boolean | "mock"

默认值: "mock"

选项:

- true: 输入文件的文件名，是相对于 `context` 选项。
- false: 常规的 Node.js `__filename` 行为。在 Node.js 环境中运行时，输出文件的文件名。
- "mock": value 填充为 "index.js".

node.__dirname

boolean | "mock"

默认值: "mock"

选项:

- `true`: 输入文件的目录名，是相对于 `context` 选项。
- `false`: 常规的 Node.js `__dirname` 行为。在 Node.js 环境中运行时，输出文件的目录名。
- `"mock"`: value 填充为 `/`。

node.Buffer

`boolean | "mock"`

默认值: `true`

node.setImmediate

`boolean | "mock" | "empty"`

默认值: `true`

其他 Node.js 核心库(Node.js core libraries)

`boolean | "mock" | "empty"`

只有当 target 是未指定、`"web"` 或 `"webworker"` 这三种情况时，此选项才会被激活（通过 `NodeSourcePlugin`）。

当 `NodeSourcePlugin` 插件启用时，则会使用 `node-labs-browser` 来对 Node.js 核心库 polyfill。请查看 `Node.js` 核心库及其 polyfills 列表。

默认情况下，如果有一个已知的 polyfill，webpack 会对每个 library 进行 polyfill，如果没有，则 webpack 不会执行任何操作。在后一种情况下，如果模块名称配置为 `false` 值，webpack 表现为不会执行任何操作。

为了导入内置的模块，使用 `__non_webpack_require__`，例如，使用 `__non_webpack_require__('modulename')` 而不是 `require('modulename')`。

示例:

```
module.exports = {
  //...
  node: {
    dns: 'mock',
    fs: 'empty',
    path: true,
```

```
    url: false
  }
};
```

性能(performance)

这些选项可以控制 webpack 如何通知「资源(asset)和入口起点超过指定文件限制」。此功能受到 [webpack 性能评估](#) 的启发。

performance

object

配置如何展示性能提示。例如，如果一个资源超过 250kb，webpack 会对此输出一个警告来通知你。

performance.hints

false | "error" | "warning"

打开/关闭提示。此外，当找到提示时，告诉 webpack 抛出一个错误或警告。此属性默认设置为 "warning"。

给定一个创建后超过 250kb 的资源：

```
module.exports = {  
  //...  
  performance: {  
    hints: false  
  }  
};
```

不展示警告或错误提示。

```
module.exports = {  
  //...  
  performance: {  
    hints: 'warning'  
  }  
};
```

将展示一条警告，通知你这是体积大的资源。在开发环境，我们推荐这样。

```
module.exports = {  
  //...  
  performance: {  
    hints: 'error'  
  }  
};
```

将展示一条错误，通知你这是体积大的资源。在生产环境构建时，我们推荐使用

`hints: "error"`, 有助于防止把体积巨大的 bundle 部署到生产环境, 从而影响网页的性能。

performance.maxEntrypointSize

int

入口起点表示针对指定的入口, 对于所有资源, 要充分利用初始加载时(initial load time)期间。此选项根据入口起点的最大体积, 控制 webpack 何时生成性能提示。默认值是: 250000 (bytes)。

```
module.exports = {
  //...
  performance: {
    maxEntrypointSize: 400000
  }
};
```

performance.maxAssetSize

int

资源(asset)是从 webpack 生成的任何文件。此选项根据单个资源体积, 控制 webpack 何时生成性能提示。默认值是: 250000 (bytes)。

```
module.exports = {
  //...
  performance: {
    maxAssetSize: 100000
  }
};
```

performance.assetFilter

Function

此属性允许 webpack 控制用于计算性能提示的文件。默认函数如下:

```
function assetFilter(assetFilename) {
  return !(/\.map$/.test(assetFilename));
}
```

你可以通过传递自己的函数来覆盖此属性:

```
module.exports = {
  //...
  performance: {
    assetFilter: function(assetFilename) {
```

```
        return assetFilename.endsWith('.js');  
    }  
}  
};
```

以上示例将只给出 .js 文件的性能提示。

统计信息(stats)

如果你不希望使用 `quiet` 或 `noInfo` 这样的不显示信息，而是又不想得到全部的信息，只是想要获取某部分 bundle 的信息，使用 stats 选项是比较好的折衷方式。

对于 webpack-dev-server，这个属性要放在 `devServer` 对象里。

在使用 Node.js API 时，此选项无效。

stats

object string

有一些预设选项，可作为快捷方式。像这样使用它们：

```
module.exports = {
  //...
  stats: 'errors-only'
};
```

Preset	"errors-only"
Alternative	<code>none</code>
Description	只在发生错误时输出
Preset	"minimal"
Alternative	<code>none</code>
Description	只在发生错误或有新的编译时输出
Preset	"none"
Alternative	<code>false</code>
Description	没有输出
Preset	"normal"
Alternative	<code>true</code>
Description	标准输出
Preset	"verbose"
Alternative	<code>none</code>
Description	全部输出

对于更加精细的控制，下列这些选项可以准确地控制并展示你想要的信息。请注意，此对象中的所有选项都是可选的。

```
module.exports = {
//...
stats: {
  // 未定义选项时, stats 选项的备用值(fallback value) (优先级高于 webpack 本身)
  all: undefined,
  // 添加资源信息
  assets: true,
  // 对资源按指定的字段进行排序
  // 你可以使用 `!field` 来反转排序。
  // Some possible values: 'id' (default), 'name', 'size', 'chunks',
  // For a complete list of fields see the bottom of the page
  assetsSort: "field",
  // 添加构建日期和构建时间信息
  builtAt: true,
  // 添加缓存(但未构建)模块的信息
  cached: true,
  // 显示缓存的资源(将其设置为 `false` 则仅显示输出的文件)
  cachedAssets: true,
  // 添加 children 信息
  children: true,
  // 添加 chunk 信息(设置为 `false` 能允许较少的冗长输出)
  chunks: true,
  // 添加 namedChunkGroups 信息
  chunkGroups: true,
  // 将构建模块信息添加到 chunk 信息
  chunkModules: true,
  // 添加 chunk 和 chunk merge 来源的信息
  chunkOrigins: true,
  // 按指定的字段, 对 chunk 进行排序
  // 你可以使用 `!field` 来反转排序。默认是按照 `id` 排序。
  // Some other possible values: 'name', 'size', 'chunks', 'failed',
  // For a complete list of fields see the bottom of the page
  chunksSort: "field",
  // 用于缩短 request 的上下文目录
  context: "../src/",
  // `webpack --colors` 等同于
  colors: false,
  // 显示每个模块到入口起点的距离(distance)
  depth: false,
  // 通过对应的 bundle 显示入口起点
  entrypoints: false,
  // 添加 --env information
```

```
env: false,  
  
// 添加错误信息  
errors: true,  
  
// 添加错误的详细信息（就像解析日志一样）  
errorDetails: true,  
  
// 将资源显示在 stats 中的情况排除  
// 这可以通过 String, RegExp, 获取 assetName 的函数来实现  
// 并返回一个布尔值或如下所述的数组。  
excludeAssets: "filter" | /filter/ | (assetName) => true | false |  
  ["filter"] | [/filter/] | [(assetName) => true|false],  
  
// 将模块显示在 stats 中的情况排除  
// 这可以通过 String, RegExp, 获取 moduleSource 的函数来实现  
// 并返回一个布尔值或如下所述的数组。  
excludeModules: "filter" | /filter/ | (moduleSource) => true | false |  
  ["filter"] | [/filter/] | [(moduleSource) => true|false],  
  
// 查看 excludeModules  
exclude: "filter" | /filter/ | (moduleSource) => true | false |  
  ["filter"] | [/filter/] | [(moduleSource) => true|false],  
  
// 添加 compilation 的哈希值  
hash: true,  
  
// 设置要显示的模块的最大数量  
maxModules: 15,  
  
// 添加构建模块信息  
modules: true,  
  
// 按指定的字段，对模块进行排序  
// 你可以使用 `!field` 来反转排序。默认是按照 `id` 排序。  
// Some other possible values: 'name', 'size', 'chunks', 'failed',  
// For a complete list of fields see the bottom of the page  
modulesSort: "field",  
  
// 显示警告/错误的依赖和来源（从 webpack 2.5.0 开始）  
moduleTrace: true,  
  
// 当文件大小超过 `performance.maxAssetSize` 时显示性能提示  
performance: true,  
  
// 显示模块的导出  
providedExports: false,  
  
// 添加 public path 的信息  
publicPath: true,  
  
// 添加模块被引入的原因  
reasons: true,  
  
// 添加模块的源码  
source: false,  
  
// 添加时间信息
```

```

timings: true,
// 显示哪个模块导出被用到
usedExports: false,
// 添加 webpack 版本信息
version: true,
// 添加警告
warnings: true,
// 过滤警告显示 (从 webpack 2.4.0 开始) ,
// 可以是 String, Regexp, 一个获取 warning 的函数
// 并返回一个布尔值或上述组合的数组。第一个匹配到的为胜 (First match wins.)。
warningsFilter: "filter" | /filter/ | ["filter", /filter/] | (warning: Boolean)
}
}

```

If you want to use one of the pre-defined behaviours e.g. 'minimal' but still override one or more of the rules, see [the source code](#). You would want to copy the configuration options from `case 'minimal': ...` and add your additional rules while providing an object to `stats`.

webpack.config.js

```

module.exports = {
  //..
  stats: {
    // copied from `minimal`
    all: false,
    modules: true,
    maxModules: 0,
    errors: true,
    warnings: true,
    // our additional options
    moduleTrace: true,
    errorDetails: true
  }
};

```

Sorting fields

For `assetsSort`, `chunksSort` and `moduleSort` there are several possible fields that you can sort items by:

- `id` is the item's id;
- `name` - a item's name that was assigned to it upon importing;
- `size` - a size of item in bytes;
- `chunks` - what chunks the item originates from (for example, if there are multiple subchunks for one chunk - the subchunks will be grouped together according to their main chunk);
- `errors` - amount of errors in items;

- `warnings` - amount of warnings in items;
- `failed` - whether the item has failed compilation;
- `cacheable` - whether the item is cacheable;
- `built` - whether the asset has been built;
- `prefetched` - whether the asset will be prefetched;
- `optional` - whether the asset is optional;
- `identifier` - identifier of the item;
- `index` - item's processing index;
- `index2`
- `profile`
- `issuer` - an identifier of the issuer;
- `issuerId` - an id of the issuer;
- `issuerName` - a name of the issuer;
- `issuerPath` - a full issuer object. There's no real need to sort by this field;

Colors

You can specify your own terminal output colors using [ANSI escape sequences](#)

```
module.exports = {
  //...
  colors: {
    green: '\u001b[32m',
  },
};
```

其它选项(other options)

这里是 webpack 支持的其它选项。

寻求帮助：这个页面还在更新中，如果你发现本页面内有描述不准确或者不完整，请在 [webpack 的文档仓库](#) 中创建 issue 或者 pull request

amd

object

设置 `require.amd` 或 `define.amd` 的值：

webpack.config.js

```
module.exports = {
  //...
  amd: {
    jQuery: true
  }
};
```

某些流行的模块是按照 AMD 规范编写的，最引人瞩目的 jQuery 版本在 1.7.0 到 1.9.1，如果 loader 提示它对页面包含的多个版本采取了特殊许可时，才会注册为 AMD 模块。

许可权限是具有「限制指定版本注册」或「支持有不同定义模块的不同沙盒」的能力。

此选项允许将模块查找的键(key)设置为真值(truthy value)。发生这种情况时，webpack 中的 AMD 支持将忽略定义的名称。

bail

boolean

在第一个错误出现时抛出失败结果，而不是容忍它。默认情况下，当使用 HMR 时，webpack 会将在终端以及浏览器控制台中，以红色文字记录这些错误，但仍然继续进行打包。要启用它：

webpack.config.js

```
module.exports = {
  //...
  bail: true
```

```
};
```

这将迫使 webpack 退出其打包过程。

cache

boolean object

缓存生成的 webpack 模块和 chunk，来改善构建速度。缓存默认在观察模式(watch mode)启用。禁用缓存只需简单传入：

webpack.config.js

```
module.exports = {  
  //...  
  cache: false  
};
```

如果传递一个对象，webpack 将使用这个对象进行缓存。保持对此对象的引用，将可以在 compiler 调用之间共享同一缓存：

webpack.config.js

```
let SharedCache = {};  
  
module.exports = {  
  //...  
  cache: SharedCache  
};
```

不要在不同选项的调用之间共享缓存。

Elaborate on the warning and example - calls with different configuration options?

loader

object

在 loader 上下文中暴露自定义值。

Add an example...

parallelism

number: 100

Limit the number of parallel processed modules. Can be used to fine tune performance or to get more reliable profiling results.

profile

boolean

捕获一个应用程序"配置文件"，包括统计和提示，然后可以使用 [Analyze](#) 分析工具进行详细分析。

使用 [StatsPlugin](#) 可以更好地控制生成的配置文件。

Combine with `parallelism: 1` for better results.

recordsPath

string

开启这个选项可以生成一个 JSON 文件，其中含有 webpack 的 "records" 记录 - 即「用于存储跨多次构建(across multiple builds)的模块标识符」的数据片段。可以使用此文件来跟踪在每次构建之间的模块变化。只要简单的设置一下路径,就可以生成这个 JSON 文件：

webpack.config.js

```
module.exports = {
  //...
  recordsPath: path.join(__dirname, 'records.json')
};
```

如果你使用了代码分离(code splitting)这样的复杂配置，records 会特别有用。这些数据用于确保拆分 bundle，以便实现你需要的缓存(caching)行为。

注意，虽然这个文件是由编译器(compiler)生成的，但你可能仍然希望在源代码管理中追踪它，以便随时记录它的变化情况。

设置 `recordsPath` 本质上会把 `recordsInputPath` 和 `recordsOutputPath` 都设置成相同的路径。通常来讲这也是符合逻辑的，除非你决定改变记录文件的名称。可以查看下面的实例：

recordsInputPath

string

指定读取最后一条记录的文件的名称。这可以用来重命名一个记录文件，可以查看下面的实例：

recordsOutputPath

string

指定记录要写入的位置。以下示例描述了如何用这个选项和 `recordsInputPath` 来重命名一个记录文件：

webpack.config.js

```
module.exports = {
  //...
  recordsInputPath: path.join(__dirname, 'records.json'),
  recordsOutputPath: path.join(__dirname, 'newRecords.json')
};
```

name

string

Name of the configuration. Used when loading multiple configurations.

webpack.config.js

```
module.exports = {
  //...
  name: 'admin-app'
};
```

引导

可以使用各种接口来定制化编译过程。一些特性会在几个接口之间重叠，例如，其中一些配置选项可能会从 CLI 标记(flag)中获取，而另一些配置选项，则只能从单个接口获取。以下高级信息可以帮助你起步。

CLI

命令行接口(Command Line Interface - CLI)，用来对构建(build)进行配置和交互。这在早期的原型设计和概要分析时特别有用。在大多数情况下，CLI 仅用于使用配置文件和几个标记(flag)（例如 `--env`）启动该进程。

[了解更多关于 CLI 的信息！](#)

模块

当使用 webpack 处理模块时，理解不同的模块语法（特别是模块方法和模块变量）是很重要的。- 这些模块语法 webpack 都可以支持。

[了解更多关于模块的信息！](#)

Node

虽然大多数用户只要用到配置文件足矣，然而对编译的更细粒度控制，则需要通过 Node 接口实现。包括传递多个配置文件、可编程方式的编译执行或观察文件，以及收集概要信息。

[了解更多关于 Node API 的信息！](#)

loader

loader 是转译模块源代码的转换规则。loader 被编写为，接受源代码作为参数的函数，并返回这些转换过的新版本代码。

[了解更多关于 loader 的信息！](#)

plugin

插件接口可以帮助用户直接接触到编译过程(compilation process)。插件可以将处理函数(handler)注册到编译过程中的不同事件点上运行的生命周期钩子函数上。

当执行每个钩子时， 插件能够完全访问到编译(compilation)的当前状态。

[了解更多关于插件的信息！](#)

命令行接口

为了更合适且方便地使用配置，可以在 `webpack.config.js` 中对 webpack 进行配置。CLI 中传入的任何参数会在配置文件中映射为对应的参数。

如果你还没有安装过 webpack 和 CLI，请先阅读 [安装指南](#)。

使用配置文件的用法

```
webpack [--config webpack.config.js]
```

配置文件中的相关选项，请参阅[配置](#)。

不使用配置文件的用法

```
webpack <entry> [<entry>] -o <output>
```

<entry>

一个文件名或一组被命名的文件名，作为构建项目的入口起点。你可以传递多个入口（每个入口在启动时加载）。如果传递一个形式为 `<name> = <request>` 的键值对，则可以创建一个额外的入口起点。它将被映射到配置选项(configuration option)的 `entry` 属性。

<output>

要保存的 bundled 文件的路径和文件名。它将映射到配置选项 `output.path` 和 `output.filename`。

示例

假设你的项目结构像下面这样：

```
.  
└── dist  
└── index.html  
└── src  
    └── index.js  
    └── index2.js  
    └── others.js
```

```
webpack src/index.js -o dist/bundle.js
```

打包源码，入口为 `index.js`，并且输出文件的路径为 `dist`，文件名为 `bundle.js`

Asset	Size	Chunks	Chunk Names
-------	------	--------	-------------

```
|-----|-----|-----|-----|
| bundle.js | 1.54 kB | 0 [emitted] | index
[0] ./src/index.js 51 bytes {0} [built]
[1] ./src/others.js 29 bytes {0} [built]

webpack index=./src/index.js entry2=./src/index2.js dist/bundle.js
```

以多个入口的方式打包文件

Asset	Size	Chunks	Chunk Names
bundle.js	1.55 kB	0,1 [emitted]	index, entry2
[0] ./src/index.js	51 bytes	{0} [built]	
[0] ./src/index2.js	54 bytes	{1} [built]	
[1] ./src/others.js	29 bytes	{0} {1} [built]	

常用配置

注意，命令行接口(Command Line Interface)参数的优先级，高于配置文件参数。例如，如果将 `--mode="production"` 传入 webpack CLI，而配置文件使用的是 `development`，最终会使用 `production`。

列出命令行所有可用的配置选项

```
webpack --help
webpack -h
```

使用配置文件进行构建

指定其它的配置文件。配置文件默认为 `webpack.config.js`，如果你想使用其它配置文件，可以加入这个参数。

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

以 JSON 格式输出 webpack 的运行结果

```
webpack --json
webpack --json > stats.json
```

在其他每个情况下，webpack 会打印一组统计信息，用于显示 bundle, chunk 和用时等详细信息。使用此选项，输出可以是 JSON 对象。此输出文件(response)可被 webpack 的分析工具，或 chrisbateman 的 webpack 可视化工具，或 th0r 的 webpack bundle 分析工具接收后进行分析。分析工具将接收 JSON 并以图形形式提供构建的所有细节。

环境选项

当 webpack 配置对象导出为一个函数时，可以向其传入一个"环境对象 (environment)"。

```
webpack --env.production      # 设置 env.production == true  
webpack --env.platform=web   # 设置 env.platform == "web"
```

--env 参数具有多种语法 accepts various syntaxes:

Invocation	webpack --env prod
Resulting environment	"prod"
Invocation	webpack --env.prod
Resulting environment	{ prod: true }
Invocation	webpack --env.prod=1
Resulting environment	{ prod: 1 }
Invocation	webpack --env.prod=foo
Resulting environment	{ prod: "foo" }
Invocation	webpack --env.prod --env.min
Resulting environment	{ prod: true, min: true }
Invocation	webpack --env.prod --env min
Resulting environment	[{ prod: true }, "min"]
Invocation	webpack --env.prod=foo --env.prod=bar
Resulting environment	{prod: ["foo", "bar"]}

See the [environment variables](#) guide for more information on its usage.

配置选项

参数	--config
说明	配置文件的路径
输入类型	string
默认值	webpack.config.js 或 webpackfile.js
参数	--config-register, -r
说明	在 webpack 配置文件加载前先预加载一个或多个模块
输入类型	array
默认值	
参数	--config-name
说明	要使用的配置名称
输入类型	string

默认值

参数

--env

说明

当配置文件是一个函数时，会将环境变量传给这个函数

输入类型

参数

--mode

说明

用到的模式，"development" 或 "production" 之中的一个

输入类型

string

默认值

输出配置

通过以下这些配置，你可以调整构建流程的某些输出参数。

参数

--output-chunk-filename

说明

输出的附带 chunk 的文件名

输入类型

string

默认值

含有 [id] 的文件名，而不是 [name] 或者 [id] 作为前缀

参数

--output-filename

说明

打包文件的文件名

输入类型

string

默认值

[name].js

参数

--output-jsonp-function

说明

加载 chunk 时使用的 JSONP 函数名

输入类型

string

默认值

webpackJsonp

参数

--output-library

说明

以库的形式导出入口文件

输入类型

string

默认值

参数

--output-library-target

说明

以库的形式导出入口文件时，输出的类型

输入类型

string

默认值

var

参数	--output-path
说明	输出的路径（在公共路径的基础上）
输入类型	string
默认值	当前目录
参数	--output-pathinfo
说明	加入一些依赖信息的注解
输入类型	boolean
默认值	false
参数	--output-public-path
说明	The 输出文件时使用的公共路径
输入类型	string
默认值	/
参数	--output-source-map-filename
说明	生成的 SourceMap 的文件名
输入类型	string
默认值	[name].map or [outputFilename].map
参数	--build-delimiter
说明	在构建输出之后，显示的自定义文本
输入类型	string
默认值	默认字符串是 null。你可以提供一个 === Build done === 这样的字符串

示例用法

```
webpack index=./src/index.js index2=./src/index2.js --output-path='./dist'
```

```
| Asset                                | Size     | Chunks      | Chunk 1
|-----|-----|-----|-----|
| index2740fdca26e9348bedbec.bundle.js | 2.6 kB  | 0 [emitted] | index2
| index740fdca26e9348bedbec.bundle.js  | 2.59 kB | 1 [emitted] | index2
|   [0] ./src/others.js 29 bytes {0} {1} [built]
|   [1] ./src/index.js 51 bytes {1} [built]
|   [2] ./src/index2.js 54 bytes {0} [built]
```

```
webpack.js index=./src/index.js index2=./src/index2.js --output-path='./dist'
```

```
| Asset                                | Size     | Chunks      | Chunk 1
|-----|-----|-----|-----|
| index2740fdca26e9348bedbec.bundle.js | 2.76 kB | 0 [emitted] | index2
| index740fdca26e9348bedbec.bundle.js  | 2.74 kB | 1 [emitted] | index2
|   index2123.map | 2.95 kB | 0 [emitted] | index2
|   index123.map | 2.95 kB | 1 [emitted] | index2
```

```
[0] ./src/others.js 29 bytes {0} {1} [built]
[1] ./src/index.js 51 bytes {1} [built]
[2] ./src/index2.js 54 bytes {0} [built]
```

Debug 配置

以下这些配置可以帮助你在 Webpack 编译过程中更好地 debug。

参数	--debug
说明	把 loader 设置为 debug 模式
输入类型	boolean
默认值	false
参数	--devtool
说明	为打包好的资源定义 [source map 的类型]
输入类型	string
默认值	-
参数	--progress
说明	打印出编译进度的百分比值
输入类型	boolean
默认值	false
参数	--display-error-details
说明	展示错误细节
输入类型	boolean
默认值	false

模块配置

这些配置可以用于绑定 Webpack 允许的模块。

参数	--module-bind
说明	为 loader 绑定一个文件扩展
用法	--module-bind js=babel-loader
参数	--module-bind-post
说明	为 post loader 绑定一个文件扩展
用法	
参数	--module-bind-pre
说明	为 pre loader 绑定一个文件扩展
用法	

Watch 选项

这些配置可以用于观察依赖文件的变化，一旦有变化，则可以重新执行构建流程。

参数	--watch, -w
说明	观察文件系统的变化
参数	--watch-aggregate-timeout
说明	指定一个毫秒数，在这个时间内，文件若发送了多次变化，会被合并
参数	--watch-poll
说明	轮询观察文件变化的时间间隔（同时会打开轮询机制）
参数	--watch-stdin, --stdin
说明	当 stdin 关闭时，退出进程

性能优化配置

在生产环境的构建时，这些配置可以用于调整的一些性能相关的配置。

参数	--optimize-max-chunks
说明	限制 chunk 的数量
使用的插件	LimitChunkCountPlugin
参数	--optimize-min-chunk-size
说明	限制 chunk 的最小体积
使用的插件	MinChunkSizePlugin
参数	--optimize-minimize
说明	压缩混淆 javascript，并且把 loader 设置为 minimizing
使用的插件	TerserPlugin & LoaderOptionsPlugin

Resolve 配置

这些配置可以用于设置 webpack `resolver` 时使用的别名(alias)和扩展名(extension)。

参数	--resolve-alias
----	-----------------

说明	<code>--alias</code>	指定模块的别名
示例		<code>--resolve-alias jquery-plugin=jquery.plugin</code>
参数	<code>--resolve-extensions</code>	
说明		指定需要被处理的文件的扩展名
示例		<code>--resolve-extensions .es6 .js .ts</code>
参数	<code>--resolve-loader-alias</code>	
说明		最小化 JavaScript，并且将 loader 切换到最简
示例		

统计数据配置

以下选项用于配置 Webpack 在控制台输出的统计数据，以及这些数据的样式。

参数	<code>--color, --colors</code>	
说明		强制在控制台开启颜色 [默认：仅对 TTY 输出启用]
Type	<code>boolean</code>	
参数	<code>--no-color, --no-colors</code>	
说明		强制在控制台关闭颜色
Type	<code>boolean</code>	
参数	<code>--display</code>	
说明		选择显示预设(verbose - 繁琐, detailed - 细节, normal - 正常, minimal - 最小, errors-only - 仅错误, none - 无; 从 webpack 3.0.0 开始)
Type	<code>string</code>	
参数	<code>--display-cached</code>	
说明		在输出中显示缓存的模块
Type	<code>boolean</code>	
参数	<code>--display-cached-assets</code>	
说明		在输出中显示缓存的 assets
Type	<code>boolean</code>	
参数	<code>--display-chunks</code>	
说明		在输出中显示 chunks
Type	<code>boolean</code>	
参数	<code>--display-depth</code>	

说明 Type	--display-entrypoints 显示从入口起点到每个模块的距离 boolean
参数 说明 Type	--display-error-details 在输出中显示入口文件 boolean
参数 说明 Type	--display-exclude 显示详细的错误信息 boolean
参数 说明 Type	--display-exclude 在输出中显示被排除的文件 boolean
参数 说明 Type	--display-max-modules 设置输出中可见模块的最大数量 number
参数 说明 Type	--display-modules 在输出中显示所有模块，包括被排除的模块 boolean
参数 说明 Type	--display-optimization-bailout 作用域提升回退触发器(Scope hoisting fallback trigger)（从 webpack 3.0.0 开始） boolean
参数 说明 Type	--display-originals 在输出中显示最初的 chunk boolean
参数 说明 Type	--display-provided-exports 显示有关从模块导出的信息 boolean
参数 说明 Type	--display-reasons 显示模块包含在输出中的原因 boolean
参数 说明 Type	--display-used-exports 显示模块中被使用的接口（Tree Shaking） boolean
参数	--display-module-map

参数	--quiet-modules
说明	隐藏关于模块的信息
Type	boolean
参数	--sort-assets-by
说明	对 assets 列表以某种属性排序
Type	string
参数	--sort-chunks-by
说明	对 chunks 列表以某种属性排序
Type	string
参数	--sort-modules-by
说明	对模块列表以某种属性排序
Type	string
参数	--verbose
说明	显示更多信息
Type	boolean

高级配置

参数	--bail
说明	一旦发生错误，立即终止
用法	
参数	--cache
说明	开启缓存 [watch 时会默认打开]
用法	--cache=false
参数	--define
说明	定义 bundle 中的任意自由变量，查看 shimming
用法	--define process.env.NODE_ENV="'development'"
参数	--hot
说明	开启模块热替换
用法	--hot=true
参数	--labeled-modules
说明	开启模块标签 [使用 LabeledModulesPlugin]
用法	
参数	--plugin

说明
用法

加载某个插件

参数
说明
用法

--prefetch
预加载某个文件
--prefetch=./files.js

参数
说明
用法

--provide
在所有模块中将这些模块提供为自由变量，查看 shimming
--provide jquery=jquery

参数
说明
用法

--records-input-path
记录文件的路径（读取）

参数
说明
用法

--records-output-path
记录文件的路径（写入）

参数
说明
用法

--records-path
记录文件的路径

参数
说明
用法

--target
目标的执行环境
--target='node'

简写

简写
含义

-d
--debug --devtool cheap-module-eval-source-map --output-pathinfo

简写
含义

-p
--optimize-minimize --define
process.env.NODE_ENV="production", see
building for production

Profiling

--profile 选项捕获编译时每个步骤的时间信息，并且将这些信息包含在输出

中。

```
webpack --profile

:
[0] ./src/index.js 90 bytes {0} [built]
  factory:22ms building:16ms = 38ms
```

For each module, the following details are included in the output as applicable:

- **factory**: time to collect module metadata (e.g. resolving the filename)
- **building**: time to build the module (e.g. loaders and parsing)
- **dependencies**: time to identify and connect the module's dependencies

Paired with `--progress`, `--profile` gives you an in depth idea of which step in the compilation is taking how long. This can help you optimise your build in a more informed manner.

```
webpack --progress --profile

30ms building modules
1ms sealing
1ms optimizing
0ms basic module optimization
1ms module optimization
1ms advanced module optimization
0ms basic chunk optimization
0ms chunk optimization
1ms advanced chunk optimization
0ms module and chunk tree optimization
1ms module reviving
0ms module order optimization
1ms module id optimization
1ms chunk reviving
0ms chunk order optimization
1ms chunk id optimization
10ms hashing
0ms module assets processing
13ms chunk assets processing
1ms additional chunk assets processing
0ms recording
0ms additional asset processing
26ms chunk asset optimization
1ms asset optimization
6ms emitting
:
```

包含统计数据的文件

通过 webpack 编译源文件时，用户可以生成包含有关于模块的统计数据的 JSON 文件。这些统计数据不仅可以帮助开发者来分析应用的依赖图表，还可以优化编译的速度。这个 JSON 文件可以通过以下的命令来生成：

```
webpack --profile --json > compilation-stats.json
```

这个标识是告诉 webpack `compilation-stats.json` 要包含依赖的图表以及各种其他的编译信息。一般来说，也会把 `--profile` 一起加入，这样每一个包含自身编译数据的模块对象(`modules object`) 都会添加 `profile`。

结构 (Structure)

最外层的输出 JSON 文件比较容易理解，但是其中还是有一小部分嵌套的数据不是那么容易理解。不过放心，这其中的每一部分都在后面有更详细的解释，并且注释中还附带有超链接可以直接跳入相应的章节。

```
{
  "version": "1.4.13", // 用来编译的 webpack 的版本
  "hash": "11593e3b3ac85436984a", // 编译使用的 hash
  "time": 2469, // 编译耗时 (ms)
  "filteredModules": 0, // 当 `exclude` 传入 `toJson` 函数时，统计被无视的模块
  "outputPath": "/", // path to webpack 输出目录的 path 路径
  "assetsByChunkName": {
    // 用作映射的 chunk 的名称
    "main": "web.js?h=11593e3b3ac85436984a",
    "named-chunk": "named-chunk.web.js",
    "other-chunk": [
      "other-chunk.js",
      "other-chunk.css"
    ]
  },
  "assets": [
    // asset 对象 (asset objects) 的数组
  ],
  "chunks": [
    // chunk 对象 (chunk objects) 的数组
  ],
  "modules": [
    // 模块对象 (module objects) 的数组
  ],
  "errors": [
    // 错误字符串 (error string) 的数组
  ],
  "warnings": [
    // 警告字符串 (warning string) 的数组
  ]
}
```

Asset 对象 (Asset Objects)

每一个 assets 对象都表示一个编译出的 output 文件。 assets 都会有一个共同的结构：

```
{  
  "chunkNames": [], // 这个 asset 包含的 chunk  
  "chunks": [ 10, 6 ], // 这个 asset 包含的 chunk 的 id  
  "emitted": true, // 表示这个 asset 是否会让它输出到 output 目录  
  "name": "10.web.js", // 输出的文件名  
  "size": 1058 // 文件的大小  
}
```

Chunk 对象 (Chunk Objects)

每一个 chunks 表示一组称为 chunk 的模块。每一个对象都满足以下的结构。

```
{  
  "entry": true, // 表示这个 chunk 是否包含 webpack 的运行时  
  "files": [  
    // 一个包含这个 chunk 的文件名的数组  
  ],  
  "filteredModules": 0, // 见上文的 结构  
  "id": 0, // 这个 chunk 的 id  
  "initial": true, // 表示这个 chunk 是开始就要加载还是 懒加载(lazy-loading)  
  "modules": [  
    // 模块对象 (module objects) 的数组  
    "web.js?h=11593e3b3ac85436984a"  
  ],  
  "names": [  
    // 包含在这个 chunk 内的 chunk 的名字的数组  
  ],  
  "origins": [  
    // 下文详述  
  ],  
  "parents": [], // 父 chunk 的 ids  
  "rendered": true, // 表示这个 chunk 是否会参与进编译  
  "size": 188057 // chunk 的大小(byte)  
}
```

chunks 对象还会包含一个 来源 (origins) , 来表示每一个 chunk 是从哪里来的。来源 (origins) 是以下的形式

```
{  
  "loc": "", // 具体是哪行生成了这个chunk  
  "module": "(webpack)\\test\\browsertest\\lib\\index.web.js", // 模块的全路径  
  "moduleId": 0, // 模块的 ID  
  "moduleIdentifier": "(webpack)\\test\\browsertest\\lib\\index.web.js",  
  "moduleName": "./lib/index.web.js", // 模块的相对地址  
  "name": "main", // chunk 的名称  
  "reasons": [  
    // 模块对象中 `reason` 的数组  
  ]  
}
```

模块对象 (Module Objects)

缺少了对实际参与进编译的模块的描述，这些数据又有什么意义呢。每一个在依赖图表中的模块都可以表示成以下的形式。

```
{  
  "assets": [  
    // asset 对象 (asset objects) 的数组  
  ],  
  "built": true, // 表示这个模块会参与 Loaders , 解析，并被编译  
  "cacheable": true, // 表示这个模块是否会被缓存  
  "chunks": [  
    // 包含这个模块的 chunks 的 id  
  ],  
  "errors": 0, // 处理这个模块发现的错误的数量  
  "failed": false, // 编译是否失败  
  "id": 0, // 这个模块的 ID (类似于 `module.id`)  
  "identifier": "(webpack)\\test\\browsertest\\lib\\index.web.js", // we  
  "name": "./lib/index.web.js", // 实际文件的地址  
  "optional": false, // 每一个对这个模块的请求都会包裹在 `try... catch` 内 (与  
  "prefetched": false, // 表示这个模块是否会被 prefetched  
  "profile": {  
    // 有关 `--profile` flag 的这个模块特有的编译数据 (ms)  
    "building": 73, // 载入和解析  
    "dependencies": 242, // 编译依赖  
    "factory": 11 // 解决依赖  
  },  
  "reasons": [  
    // 见下文描述  
  ],  
  "size": 3593, // 预估模块的大小 (byte)  
  "source": "// Should not break it...\r\nif(typeof...)", // 字符串化的输入  
  "warnings": 0 // 处理模块时警告的数量  
}
```

每一个模块都包含一个 理由 (reasons) 对象，这个对象描述了这个模块被加入依赖图表的理由。每一个 理由 (reasons) 都类似于上文 chunk objects 中的 来源 (origins)：

```
{  
  "loc": "33:24-93", // 导致这个被加入依赖图标的代码行数  
  "module": "./lib/index.web.js", // 所基于模块的相对地址 context  
  "moduleId": 0, // 模块的 ID  
  "moduleIdentifier": "(webpack)\\test\\browsertest\\lib\\index.web.js",  
  "moduleName": "./lib/index.web.js", // 可读性更好的模块名称 (用于 "更好的打  
  "type": "require.context", // 使用的请求的种类 (type of request)  
  "userRequest": ".../..../cases" // 用来 `import` 或者 `require` 的源字符串  
}
```

错误与警告

错误 (errors) 和 警告 (warnings) 会包含一个字符串数组。每个字符串包含了信息和栈的追溯：

```
./cases/parsing/browserify/index.js
Critical dependencies:
2:114-121 This seem to be a pre-built javascript file. Even while this :
@ ./cases/parsing/browserify/index.js 2:114-121
```

需要注意的是，当错误详情为`false(errorDetails:false)`传入`toJson`函数时，对栈的追溯就不会被显示。错误详情(`errorDetails`)默认值为`true`

Node.js API

webpack 提供了 Node.js API，可以在 Node.js 运行时下直接使用。

当你需要自定义构建或开发流程时，Node.js API 非常有用，因为此时所有的报告和错误处理都必须自行实现，webpack 仅仅负责编译的部分。所以 `stats` 配置选项不会在 `webpack()` 调用中生效。

安装(Installation)

开始使用 webpack 的 Node.js API 之前，首先你需要安装 webpack：

```
npm install --save-dev webpack
```

然后在 Node.js 脚本中 `require` webpack module：

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

或者如果你喜欢 ES2015：

```
import webpack from 'webpack';
```

webpack()

导入的 `webpack` 函数需要传入一个 webpack 配置对象，当同时传入回调函数时就会执行 webpack compiler：

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

webpack({
  // 配置对象
}, (err, stats) => {
  if (err || stats.hasErrors()) {
    // 在这里处理错误
  }
  // 处理完成
});
```

编译错误不在 `err` 对象内，而是需要使用 `stats.hasErrors()` 单独处理，你可以在指南的 [错误处理](#) 部分查阅到更多细节。`err` 对象只会包含 webpack 相关的问题，比如配置错误等。

你可以向 `webpack` 函数提供一个由配置选项对象构成的数组。更多详细信息，请查看 [MultiCompiler](#) 章节。

Compiler 实例(Compiler Instance)

如果你不向 `webpack` 执行函数传入回调函数，就会得到一个 `webpack Compiler` 实例。你可以通过它手动触发 `webpack` 执行器，或者是让它执行构建并监听变更。和 [CLI API](#) 很类似。`Compiler` 实例提供了以下方法：

- `.run(callback)`
- `.watch(watchOptions, handler)`

通常情况下，虽然可以创建一些子 `compiler` 来代理到特定任务，然而只会创建一个主要 `Compiler` 实例。`Compiler` 基本上只是执行最低限度的功能，以维持生命周期运行的功能。它将所有的加载、打包和写入工作，都委托到注册过的插件上。

`Compiler` 实例上的 `hooks` 属性，用于将一个插件，注册到 `Compiler` 的生命周期中的所有钩子事件上。`webpack` 使用 `[WebpackOptionsDefaulter]` (<https://github.com/webpack/webpack/blob/master/lib/WebpackOptionsDefaulter.js>) 和 `WebpackOptionsApply` 这两个工具，通过所有内置插件，来配置 `Compiler` 实例。

`run` 方法用于触发所有编译时工作。完成之后，执行给定的 `callback` 函数。最终记录下来的概括信息(`stats`)和错误(`errors`)，应该在这个 `callback` 函数中获取。

这个 API 一次只支持一个并发编译。当使用 `run` 时，会等待它完成后，然后才能再次调用 `run` 或 `watch`。当使用 `watch` 时，调用 `close`，等待它完成后，然后才能再次调用 `run` 或 `watch`。多个并发编译会损坏输出文件。

执行(run)

调用 `Compiler` 实例的 `run` 方法跟上文提到的快速执行方法很相似：

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

const compiler = webpack({
  // 配置对象
});

compiler.run((err, stats) => {
  // ...
});
```

监听(watching)

调用 `watch` 方法会触发 `webpack` 执行器，但之后会监听变更（很像 CLI 命令：`webpack --watch`），一旦 `webpack` 检测到文件变更，就会重新执行编译。该方法返回一个 `Watching` 实例。

```
watch(watchOptions, callback);

const webpack = require("webpack");

const compiler = webpack({
  // 配置对象
});

const watching = compiler.watch({
  // watchOptions 示例
  aggregateTimeout: 300,
  poll: undefined
}, (err, stats) => {
  // 在这里打印 watch/build 结果...
  console.log(stats);
});
```

`watching` 配置选项的细节可以在 [这里](#) 查阅。

文件系统不正确的问题，可能会对单次修改触发多次构建。因此，在上面的示例中，一次修改可能会多次触发 `console.log` 语句。用户应该预知此行为，并且可能需要检查 `stats.hash` 来查看文件哈希是否确实变更。-

关闭 `watching`(Close watching)

`watch` 方法返回一个 `Watching` 实例，它会暴露一个 `.close(callback)` 方法。调用该方法将会结束监听：

```
watching.close(() => {
  console.log('Watching Ended.');
});
```

不允许在当前监听器已经关闭或失效前再次监听或执行。

作废 `watching`(Invalidate watching)

使用 `watching.invalidate`，你可以手动使当前编译循环(compiling round)无效，而不会停止监视进程：

```
watching.invalidate();
```

Stats 对象(Stats Object)

`stats` 对象会被作为 `webpack()` 回调函数的第二个参数传入，可以通过它获取到代码编译过程中的有用信息，包括：

- 错误和警告（如果有的话）
- 计时信息
- module 和 chunk 信息

[webpack CLI](#) 正是基于这些信息在控制台展示友好的格式输出。

当使用 [MultiCompiler](#) 时，会返回一个 `MultiStats` 实例，它实现与 `stats` 相同的接口，也就是下面描述的方法。

`stats` 对象暴露了以下方法：

`stats.hasErrors()`

可以用来检查编译期是否有错误，返回 `true` 或 `false`。

`stats.hasWarnings()`

可以用来检查编译期是否有警告，返回 `true` 或 `false`。

`stats.toJson(options)`

以 JSON 对象形式返回编译信息。`options` 可以是一个字符串（预设值）或是颗粒化控制的对象：

```
stats.toJson("minimal"); // 更多选项如: "verbose" 等.
```

```
stats.toJson({
  assets: false,
  hash: true
});
```

所有可用的配置选项和预设值都可查询 `stats` 文档。

这里有一个该函数输出的示例。

`stats.toString(options)`

以格式化的字符串形式返回描述编译信息（类似 [CLI](#) 的输出）。

配置对象与 `stats.toJson(options)` 一致，除了额外增加的一个选项：

```
stats.toString({
  // 增加控制台颜色开关
  colors: true
});
```

下面是 `stats.toString()` 用法的示例：

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

webpack({
  // 配置对象
```

```

}, (err, stats) => {
  if (err) {
    console.error(err);
    return;
  }

  console.log(stats.toString({
    chunks: false, // 使构建过程更静默无输出
    colors: true   // 在控制台展示颜色
  }));
}) ;
})

```

Multicompiler

`Multicompiler` 模块可以让 `webpack` 在单个 `compiler` 中执行多个配置。如果传给 `webpack` 的 `Node.js API` 的 `options` 参数，是一个由配置对象构成的数组，则 `webpack` 会应用单独 `compiler`，并且在每次 `compiler` 执行结束时，都会调用 `callback` 方法。

```

var webpack = require('webpack');

webpack([
  { entry: './index1.js', output: { filename: 'bundle1.js' } },
  { entry: './index2.js', output: { filename: 'bundle2.js' } }
], (err, stats) => {
  process.stdout.write(stats.toString() + "\n");
})

```

多个配置对象在执行时，不会并行执行。每个配置都只会在前一个处理结束后，才进行处理。想要并行处理，你可以使用第三方解决方案，例如 [parallel-webpack](#)。

错误处理(error handling)

完备的错误处理中需要考虑以下三种类型的错误：

- 致命的 `wepback` 错误（配置出错等）
- 编译错误（缺失的 module，语法错误等）
- 编译警告

下面是一个覆盖这些场景的示例：

```

const webpack = require("webpack");

webpack({
  // 配置对象
}, (err, stats) => {
  if (err) {
    console.error(err.stack || err);
    if (err.details) {

```

```

        console.error(err.details);
    }
    return;
}

const info = stats.toJson();

if (stats.hasErrors()) {
    console.error(info.errors);
}

if (stats.hasWarnings()) {
    console.warn(info.warnings);
}

// 记录结果...
));

```

自定义文件系统(Custom File Systems)

默认情况下，webpack 使用普通文件系统来读取文件并将文件写入磁盘。但是，还可以使用不同类型的文件系统（内存(memory)，webDAV 等）来更改输入或输出行为。为了实现这一点，可以改变 `inputFileSystem` 或 `outputFileSystem`。例如，可以使用 `memory-fs` 替换默认的 `outputFileSystem`，以将文件写入到内存中，而不是写入到磁盘：

```

const MemoryFS = require('memory-fs');
const webpack = require('webpack');

const fs = new MemoryFS();
const compiler = webpack({ /* options */ });

compiler.outputFileSystem = fs;
compiler.run((err, stats) => {
    // 之后读取输出:
    const content = fs.readFileSync('...');
});

```

值得一提的是，被 `webpack-dev-server` 及众多其他包依赖的 `webpack-dev-middleware` 就是通过这种方式，将你的文件神秘地隐藏起来，但却仍然可以用它们为浏览器提供服务！

你指定的输出文件系统需要兼容 Node 自身的 `fs` 模块接口，接口需要提供 `mkdirp` 和 `join` 工具方法。

模块热替换

如果已经通过 `HotModuleReplacementPlugin` 启用了模块热替换(Hot Module Replacement)，则它的接口将被暴露在 `module.hot` 属性下面。通常，用户先要检查这个接口是否可访问，然后再开始使用它。举个例子，你可以这样 `accept` 一个更新的模块：

```
if (module.hot) {  
  module.hot.accept('./library.js', function() {  
    // 使用更新过的 library 模块执行某些操作...  
  });  
}
```

支持以下方法.....

模块 API

accept

接受(accept)给定依赖模块(dependencies)的更新，并触发一个回调函数来对这些更新做出响应。

```
module.hot.accept(  
  dependencies, // 可以是一个字符串或字符串数组  
  callback // 用于在模块更新后触发的函数  
) ;
```

当使用 ESM `import` 时，所有引用依赖模块(dependencies)的导入符号都会被自动更新。注意：依赖模块字符串必须和 `import` 中的 `from` 字符串相匹配。在一些情况下 `callback` 可以省略。在 `callback` 中使用的 `require()` 在这里没有任何意义。

在使用 CommonJS 时，你应该通过 `callback` 中的 `require()` 手动更新依赖模块。这时省略 `callback` 在这里没有任何意义。

accept（自身）

接受自身更新。

```
module.hot.accept(  
  errorHandler // 在计算新版本时处理错误的函数  
) ;
```

在此模块或依赖模块更新时，在不通知父母的情况下，可以对此模块处理和重新取值。如果此模块没有导出（或以其他方式更新的导出），这是有意义的。

当对此模块（或依赖模块）进行取值而引发异常时，会触发 `errorHandler`。

decline

拒绝给定依赖模块的更新，使用 '`decline`' 方法强制更新失败。

```
module.hot.decline(  
  dependencies // 可以是一个字符串或字符串数组  
) ;
```

将依赖模块标记为不可更新(not-update-able)。在处理「依赖的导出正在更新」或「尚未实现处理」时，这是有意义的。取决于你的 HMR 管理代码，此依赖模块（或其未接受的依赖模块）更新，通常会导致页面被完全重新加载。

decline (自身)

拒绝自身更新。

```
module.hot.decline();
```

将依赖模块标记为不可更新(not-update-able)。当此模块具有无法避免的外部作用(side-effect)，或者尚未对此模块进行 HMR 处理时，这是有意义的。取决于你的 HMR 管理代码，此依赖模块（或其未接受的依赖模块）更新，通常会导致页面被完全重新加载。

dispose (或 addDisposeHandler)

添加一个处理函数，在当前模块代码被替换时执行。此函数应该用于移除你声明或创建的任何持久资源。如果要将状态传入到更新过的模块，请添加给定 `data` 参数。更新后，此对象在更新之后可通过 `module.hot.data` 调用。

```
module.hot.dispose(data => {  
  // 清理并将 data 传递到更新后的模块.....  
});
```

removeDisposeHandler

删除由 `dispose` 或 `addDisposeHandler` 添加的回调函数。

```
module.hot.removeDisposeHandler(callback);
```

管理 API

status

取得模块热替换进程的当前状态。

```
module.hot.status() ; // 返回以下字符串之一.....
```

Status	idle
Description	该进程正在等待调用 <code>check</code> (见下文)
Status	check
Description	该进程正在检查以更新
Status	prepare
Description	该进程正在准备更新 (例如, 下载已更新的模块)
Status	ready
Description	此更新已准备并可用
Status	dispose
Description	该进程正在调用将被替换模块的 <code>dispose</code> 处理函数
Status	apply
Description	该进程正在调用 <code>accept</code> 处理函数, 并重新执行自我接受(self-accepted)的模块
Status	abort
Description	更新已中止, 但系统仍处于之前的状态
Status	fail
Description	更新已抛出异常, 系统状态已被破坏

check

测试所有加载的模块以进行更新, 如果有更新, 则应用它们。

```
module.hot.check(autoApply).then(outdatedModules => {
  // 超时的模块.....
}).catch(error => {
  // 捕获错误
});
```

`autoApply` 参数可以是布尔值, 也可以是 `options`, 当被调用时可以传递给 `apply` 方法。

apply

继续更新进程（只要 `module.hot.status() === 'ready'`）。

```
module.hot.apply(options).then(outdatedModules => {
  // 超时的模块.....
}).catch(error => {
  // 捕获错误
});
```

可选的 `options` 对象可以包含以下属性：

- `ignoreUnaccepted (boolean)`: Ignore changes made to unaccepted modules.
- `ignoreDeclined (boolean)`: Ignore changes made to declined modules.
- `ignoreErrored (boolean)`: Ignore errors throw in accept handlers, error handlers and while reevaluating module.
- `onDeclined (function(info))`: Notifier for declined modules
- `onUnaccepted (function(info))`: Notifier for unaccepted modules
- `onAccepted (function(info))`: Notifier for accepted modules
- `onDisposed (function(info))`: Notifier for disposed modules
- `onErrored (function(info))`: Notifier for errors

The `info` parameter will be an object containing some of the following values:

```
{
  type: "self-declined" | "declined" |
    "unaccepted" | "accepted" |
    "disposed" | "accept-errored" |
    "self-accept-errored" | "self-accept-error-handler-errored",
  moduleId: 4, // The module in question.
  dependencyId: 3, // For errors: the module id owning the accept handle
  chain: [1, 2, 3, 4], // For declined/accepted/unaccepted: the chain f...
  parentId: 5, // For declined: the module id of the declining parent
  outdatedModules: [1, 2, 3, 4], // For accepted: the modules that are o...
  outdatedDependencies: { // For accepted: The location of accept handle...
    5: [4]
  },
  error: new Error(...), // For errors: the thrown error
  originalError: new Error(...) // For self-accept-error-handler-errored:
                                // the error thrown by the module before
}
```

addStatusHandler

注册一个函数来监听 `status` 的变化。

```
module.hot.addStatusHandler(status => {
  // 响应当前状态.....
});
```

removeStatusHandler

移除一个注册的状态处理函数。

```
module.hot.removeStatusHandler(callback);
```

loader API

所谓 loader 只是一个导出为函数的 JavaScript 模块。`loader runner` 会调用这个函数，然后把上一个 loader 产生的结果或者资源文件(resource file)传入进去。函数的 `this` 上下文将由 webpack 填充，并且 `loader runner` 具有一些有用方法，可以使 loader 改变为异步调用方式，或者获取 query 参数。

第一个 loader 的传入参数只有一个：资源文件(resource file)的内容。`compiler` 需要得到最后一个 loader 产生的处理结果。这个处理结果应该是 `String` 或者 `Buffer`（被转换为一个 `string`），代表了模块的 JavaScript 源码。另外还可以传递一个可选的 `SourceMap` 结果（格式为 JSON 对象）。

如果是单个处理结果，可以在同步模式中直接返回。如果有多个处理结果，则必须调用 `this.callback()`。在异步模式中，必须调用 `this.async()`，来指示 `loader runner` 等待异步结果，它会返回 `this.callback()` 回调函数，随后 loader 必须返回 `undefined` 并且调用该回调函数。

示例

以下部分提供了不同类型的 loader 的一些基本示例。注意，`map` 和 `meta` 参数是可选的，查看下面的 `this.callback`。

同步 loader

无论是 `return` 还是 `this.callback` 都可以同步地返回转换后的 `content` 内容：

`sync-loader.js`

```
module.exports = function(content, map, meta) {
  return someSyncOperation(content);
};
```

`this.callback` 方法则更灵活，因为它允许传递多个参数，而不仅仅是 `content`。

`sync-loader-with-multiple-results.js`

```
module.exports = function(content, map, meta) {
  this.callback(null, someSyncOperation(content), map, meta);
  return; // 当调用 callback() 时总是返回 undefined
};
```

异步 loader

对于异步 loader，使用 `this.async` 来获取 `callback` 函数：

async-loader.js

```
module.exports = function(content, map, meta) {
  var callback = this.async();
  someAsyncOperation(content, function(err, result) {
    if (err) return callback(err);
    callback(null, result, map, meta);
  });
};
```

async-loader-with-multiple-results.js

```
module.exports = function(content, map, meta) {
  var callback = this.async();
  someAsyncOperation(content, function(err, result, sourceMaps, meta) {
    if (err) return callback(err);
    callback(null, result, sourceMaps, meta);
  });
};
```

loader 最初被设计为可以在同步 loader pipeline (如 Node.js，使用 [enhanced-require](#))，以及在异步 pipeline (如 webpack) 中运行。然而在 Node.js 这样的单线程环境下进行耗时长的同步计算不是个好主意，我们建议尽可能地使你的 loader 异步化。但如果计算量很小，同步 loader 也是可以的。

"Raw" loader

默认情况下，资源文件会被转化为 UTF-8 字符串，然后传给 loader。通过设置 `raw`，loader 可以接收原始的 `Buffer`。每一个 loader 都可以用 `String` 或者 `Buffer` 的形式传递它的处理结果。Complier 将会把它们在 loader 之间相互转换。

raw-loader.js

```
module.exports = function(content) {
  assert(content instanceof Buffer);
  return someSyncOperation(content);
  // 返回值也可以是一个 `Buffer`
  // 即使不是 raw loader 也没问题
};
module.exports.raw = true;
```

越过 loader(Pitching loader)

loader 总是从右到左地被调用。有些情况下，loader 只关心 request 后面的元数据 (**metadata**)，并且忽略前一个 loader 的结果。在实际（从右到左）执行 loader 之前，会先从左到右调用 loader 上的 `pitch` 方法。对于以下 `use` 配置：

```
module.exports = {
  //...
  module: {
```

```

rules: [
  {
    //...
    use: [
      'a-loader',
      'b-loader',
      'c-loader'
    ]
  }
]
};

}
;

```

将会发生这些步骤:

```

|- a-loader `pitch`
|- b-loader `pitch`
  |- c-loader `pitch`
    |- requested module is picked up as a dependency
    |- c-loader normal execution
  |- b-loader normal execution
|- a-loader normal execution

```

那么，为什么 loader 可以利用 "跳跃(pitching)" 阶段呢？

首先，传递给 pitch 方法的 data，在执行阶段也会暴露在 this.data 之下，并且可以用于在循环时，捕获和共享前面的信息。

```

module.exports = function(content) {
  return someSyncOperation(content, this.data.value);
};

module.exports.pitch = function(remainingRequest, precedingRequest, data) {
  data.value = 42;
};

```

其次，如果某个 loader 在 pitch 方法中给出一个结果，那么这个过程会回过身来，并跳过剩下的 loader。在我们上面的例子中，如果 b-loader 的 pitch 方法返回了一些东西：

```

module.exports = function(content) {
  return someSyncOperation(content);
};

module.exports.pitch = function(remainingRequest, precedingRequest, data) {
  if (someCondition()) {
    return 'module.exports = require(' + JSON.stringify('!-' + remainingRequest) + ')';
  }
};

```

上面的步骤将被缩短为：

```

|- a-loader `pitch`
  |- b-loader `pitch` returns a module

```

```
| - a-loader normal execution
```

查看 [bundle-loader](#)，了解如何以更有意义的方式使用此过程。

loader 上下文

loader context 表示在 loader 内使用 `this` 可以访问的一些方法或属性。

假设我们这样请求加载别的模块： 在 `/abc/file.js` 中：

```
require('./loader1?xyz!loader2!/resource?rrr');
```

`this.version`

loader API 的版本号。目前是 `2`。这对于向后兼容性有一些用处。通过这个版本号，你可以为不同版本间的破坏性变更编写不同的逻辑，或做降级处理。

`this.context`

模块所在的目录。可以用作解析其他模块路径的上下文。

在我们的例子中：这个属性为 `/abc`，因为 `resource.js` 在这个目录中

`this.rootContext`

从 webpack 4 开始，原先的 `this.options.context` 被改进为 `this.rootContext`。

`this.request`

被解析出来的 request 字符串。

在我们的例子中：`"/abc/loader1.js?xyz!/abc/node_modules/loader2/index.js!/abc/resource.js?rrr"`

`this.query`

1. 如果这个 loader 配置了 `options` 对象的话，`this.query` 就指向这个 option 对象。
2. 如果 loader 中没有 `options`，而是以 query 字符串作为参数调用时，`this.query` 就是一个以 `?` 开头的字符串。

使用 `loader-utils` 中提供的 `getOptions` 方法来提取给定 loader 的 option。

this.callback

一个可以同步或者异步调用的可以返回多个结果的函数。预期的参数是：

```
this.callback(  
  err: Error | null,  
  content: string | Buffer,  
  sourceMap?: SourceMap,  
  meta?: any  
) ;
```

1. 第一个参数必须是 `Error` 或者 `null`
2. 第二个参数是一个 `string` 或者 `Buffer`。
3. 可选的：第三个参数必须是一个可以被这个模块解析的 `source map`。
4. 可选的：第四个选项，会被 `webpack` 忽略，可以是任何东西（例如一些元数据）。

可以将抽象语法树(abstract syntax tree - AST)（例如 `ESTree`）作为第四个参数 (`meta`)，如果你想在多个 loader 之间共享通用的 AST，这样做有助于加速编译时间。

如果这个函数被调用的话，你应该返回 `undefined` 从而避免含糊的 loader 结果。

this.async

告诉 `loader-runner` 这个 loader 将会异步地回调。返回 `this.callback`。

this.data

在 pitch 阶段和正常阶段之间共享的 `data` 对象。

this.cacheable

设置是否可缓存标志的函数：

```
cacheable(flag = true: boolean)
```

默认情况下，loader 的处理结果会被标记为可缓存。调用这个方法然后传入 `false`，可以关闭 loader 的缓存。

一个可缓存的 loader 在输入和相关依赖没有变化时，必须返回相同的结果。这意味着 loader 除了 `this.addDependency` 里指定的以外，不应该有其它任何外部依赖。

this.loaders

所有 loader 组成的数组。它在 pitch 阶段的时候是可以写入的。

```
loaders = [{request: string, path: string, query: string, module: funct:
```

在我们的示例中：

```
[  
  {  
    request: '/abc/loader1.js?xyz',  
    path: '/abc/loader1.js',  
    query: '?xyz',  
    module: [Function]  
  },  
  {  
    request: '/abc/node_modules/loader2/index.js',  
    path: '/abc/node_modules/loader2/index.js',  
    query: '',  
    module: [Function]  
  }  
];
```

this.loaderIndex

当前 loader 在 loader 数组中的索引。

在我们的示例中： loader1 中得到： 0， loader2 中得到： 1

this.resource

request 中的资源部分，包括 query 参数。

在我们的示例中： "/abc/resource.js?rrr"

this.resourcePath

资源文件的路径。

在我们的示例中： "/abc/resource.js"

this.resourceQuery

资源的 query 参数。

在我们的示例中： "?rrr"

this.target

编译的目标。从配置选项中传递过来的。

示例: "web", "node"

this.webpack

如果是由 webpack 编译的，这个布尔值会被设置为真。

loader 最初被设计为可以同时当 Babel transform 用。如果你编写了一个 loader 可以同时兼容二者，那么可以使用这个属性了解是否存在可用的 loaderContext 和 webpack 特性。

this.sourceMap

应该生成一个 source map。因为生成 source map 可能会非常耗时，你应该确认 source map 确实有必要请求。

this.emitWarning

```
emitWarning(warning: Error)
```

发出一个警告，在输出中显示如下：

```
WARNING in ./src/lib.js (./src/loader.js!./src/lib.js)
Module Warning (from ./src/loader.js):
Here is a Warning!
@ ./src/index.js 1:0-25
```

Note that the warnings will not be displayed if stats.warnings is set to false, or some other omit setting is used to stats such as none or errors-only. See the [stats configuration](#).

this.emitError

```
emitError(error: Error)
```

发出一个错误，在输出中显示如下：

```
ERROR in ./src/lib.js (./src/loader.js!./src/lib.js)
Module Error (from ./src/loader.js):
Here is an Error!
@ ./src/index.js 1:0-25
```

Unlike throwing an Error directly, it will NOT interrupt the compilation process of the current module.

this.loadModule

```
loadModule(request: string, callback: function(err, source, sourceMap, r
```

解析给定的 request 到一个模块，应用所有配置的 loader，并且在回调函数中传入生成的 source、sourceMap 和 模块实例（通常是 `NormalModule` 的一个实例）。如果你需要获取其他模块的源代码来生成结果的话，你可以使用这个函数。

this.resolve

```
resolve(context: string, request: string, callback: function(err, result)
```

像 require 表达式一样解析一个 request。

this.addDependency

```
addDependency(file: string)  
dependency(file: string) // 简写
```

加入一个文件作为产生 loader 结果的依赖，使它们的任何变化可以被监听到。例如，`html-loader` 就使用了这个技巧，当它发现 `src` 和 `src-set` 属性时，就会把这些属性上的 url 加入到被解析的 html 文件的依赖中。

this.addContextDependency

```
addContextDependency(directory: string)
```

把文件夹作为 loader 结果的依赖加入。

this.clearDependencies

```
clearDependencies()
```

移除 loader 结果的所有依赖。甚至自己和其它 loader 的初始依赖。考虑使用 `pitch`。

this.emitFile

```
emitFile(name: string, content: Buffer|string, sourceMap: {...})
```

产生一个文件。这是 webpack 特有的。

this.fs

用于访问 `compilation` 的 `inputFileSystem` 属性。

废弃的上下文属性

强烈建议不要使用这些属性，因为我们打算移除它们。它们仍然列在此处用

于文档目的。

this.exec

```
exec(code: string, filename: string)
```

以模块的方式执行一些代码片段。如果需要，请查看[这里的评论](#)以获取替换方法。

this.resolveSync

```
resolveSync(context: string, request: string) -> string
```

像 require 表达式一样解析一个 request。

this.value

向下一个 loader 传值。如果你知道了作为模块执行后的结果，请在这里赋值（以单元素数组的形式）。

this inputValue

从上一个 loader 那里传递过来的值。如果你会以模块的方式处理输入参数，建议预先读入这个变量（为了性能因素）。

this.options

`options` 属性，在 webpack 3 中已经废弃(deprecated)，在 webpack 4 中已经移除(removed)。

this.debug

一个布尔值，当处于 debug 模式时为 true。

this.minimize

决定处理结果是否应该被压缩。

this._compilation

一种 hack 写法。用于访问 webpack 的 Compilation 对象。

this._compiler

一种 hack 写法。用于访问 webpack 的 Compiler 对象。

this._module

一种 hack 写法。用于访问当前加载的 Module 对象。

Error Reporting

You can report errors from inside a loader by:

- Using `this.emitError`. Will report the errors without interrupting module's compilation.
- Using `throw` (or other uncaught exception). Throwing an error while a loader is running will cause current module compilation failure.
- Using `callback` (in async mode). Pass an error to the callback will also cause module compilation failure.

For example:

./src/index.js

```
require('./loader!./lib');
```

Throwing an error from loader:

./src/loader.js

```
module.exports = function(source) {
  throw new Error('This is a Fatal Error!');
};
```

Or pass an error to the callback in async mode:

./src/loader.js

```
module.exports = function(source) {
  const callback = this.async();
  //...
  callback(new Error('This is a Fatal Error!'), source);
};
```

The module will get bundled like this:

```
/* */ "./src/loader.js!./src/lib.js":
/* !*****!*\
 !*** ./src/loader.js!./src/lib.js ***!
 \*****/
/*! no static exports found */
/* */ (function(module, exports) {
```

```
throw new Error("Module build failed (from ./src/loader.js):\nError: Th:  
/****/ })
```

Then the build output will also display the error (Similar to `this.emitError`):

```
ERROR in ./src/lib.js (./src/loader.js!./src/lib.js)  
Module build failed (from ./src/loader.js):  
Error: This is a Fatal Error!  
    at Object.module.exports (/workspace/src/loader.js:2:9)  
    @ ./src/index.js 1:0-25
```

As you can see below, not only error message, but also details about which loader and module are involved:

- the module path: `ERROR in ./src/lib.js`
- the request string: `(./src/loader.js!./src/lib.js)`
- the loader path: `(from ./src/loader.js)`
- the caller path: `@ ./src/index.js 1:0-25`

The loader path in the error is displayed since webpack 4.12

All the errors and warnings will be recorded into `stats`. Please see [Stats Data](#).

Inline matchResource

A new inline request syntax was introduced in webpack v4. Prefixing `<match-resource>!!=!` to a request will set the `matchResource` for this request.

It is not recommended to use this syntax in application code. Inline request syntax is intended to only be used by loader generated code. Not following this recommendation will make your code webpack-specific and non-standard.

A relative `matchResource` will resolve relative to the current context of the containing module.

When a `matchResource` is set, it will be used to match with the `module.rules` instead of the original resource. This can be useful if further loaders should be applied to the resource, or if the module type need to be changed. It's also displayed in the stats and used for matching `Rule.issuer` and `test` in `splitChunks`.

Example:

file.js

```
/* STYLE: body { background: red; } */  
console.log('yep');
```

A loader could transform the file into the following file and use the `matchResource` to apply the user-specified CSS processing rules:

file.js (transformed by loader)

```
import './file.js.css'!=!extract-style-loader/getStyles!./file.js';
console.log('yep');
```

This will add a dependency to `extract-style-loader/getStyles!./file.js` and treat the result as `file.js.css`. Because `module.rules` has a rule matching `/\.css$/` and it will apply to this dependency.

The loader could look like this:

extract-style-loader/index.js

```
const stringifyRequest = require('loader-utils').stringifyRequest;
const getRemainingRequest = require('loader-utils').getRemainingRequest,
const getStylesLoader = require.resolve('./getStyle');

module.exports = function (source) {
  if (STYLES_REGEXP.test(source)) {
    source = source.replace(STYLES_REGEXP, '');
    const remReq = getRemainingRequest(this);
    return `import ${stringifyRequest(`.${this.resource}.css!=!${getStyle}`)};`;
  }
  return source;
};
```

extract-style-loader/getStyles.js

```
module.exports = function(source) {
  const match = STYLES_REGEXP.match(source);
  return match[0];
};
```

模块方法

本节涵盖了使用 webpack 编译代码的所有方法。在 webpack 打包应用程序时，你可以选择各种模块语法风格，包括 [ES6](#), [CommonJS](#) 和 [AMD](#)。

虽然 webpack 支持多种模块语法，但我们建议尽量遵循一致的语法，避免一些奇怪的行为和 bug。这是一个混合使用了 ES6 和 CommonJS 的示例，但我们确定还有其他的 BUG 会产生。

ES6 (推荐)

webpack 2 支持原生的 ES6 模块语法，意味着你可以无须额外引入 babel 这样的工具，就可以使用 `import` 和 `export`。但是注意，如果使用其他的 ES6+ 特性，仍然需要引入 babel。webpack 支持以下的方法：

`import`

通过 `import` 以静态的方式，导入另一个通过 `export` 导出的模块。

```
import MyModule from './my-module.js';
import { NamedExport } from './other-module.js';
```

这里的关键词是静态的。标准的 `import` 语句中，模块语句中不能以「具有逻辑或含有变量」的动态方式去引入其他模块。关于 `import` 的更多信息和 `import()` 动态用法，请查看[这里的说明](#)。

`export`

默认导出整个模块，或具名导出模块

```
// 具名导出
export var Count = 5;
export function Multiply(a, b) {
  return a * b;
}
```

```
// 默认导出
export default {
  // Some data...
};
```

`import()`

```
import('path/to/module') -> Promise
```

动态地加载模块。调用 `import()` 之处，被作为分离的模块起点，意思是，被请求的模块和它引用的所有子模块，会分离到一个单独的 chunk 中。

[ES2015 loader 规范](#) 定义了 `import()` 方法，可以在运行时动态地加载 ES2015 模块。

```
if ( module.hot ) {
  import('lodash').then(_ => {
    // Do something with lodash (a.k.a '_')...
  });
}
```

`import()` 特性依赖于内置的 [Promise](#)。如果想在低版本浏览器使用 `import()`，记得使用像 [es6-promise](#) 或者 [promise-polyfill](#) 这样 polyfill 库，来预先填充 (shim) `Promise` 环境。

Magic Comments

Inline comments to make features work. By adding comments to the `import` we can do things such as name our chunk or select different modes. For a full list of these magic comments see the code below followed by an explanation of what these comments do.

```
// 单个目标
import(
  /* webpackChunkName: "my-chunk-name" */
  /* webpackMode: "lazy" */
  'module'
);

// 多个可能目标
import(
  /* webpackInclude: /\.json$/ */
  /* webpackExclude: /\.noimport\.json$/ */
  /* webpackChunkName: "my-chunk-name" */
  /* webpackMode: "lazy" */
  /* webpackPrefetch: true */
  /* webpackPreload: true */
  `./locale/${language}`
);
import(/* webpackIgnore: true */ 'ignored-module.js');
```

`webpackIgnore`: Disables dynamic import parsing when set to `true`.

Note that setting `webpackIgnore` to `true` opts out of code splitting.

`webpackChunkName`: 新 chunk 的名称。从 webpack 2.6.0 开始，`[index]` 和 `[request]` 占位符，分别支持赋予一个递增的数字和实际解析的文件名。Adding this comment will cause our separate chunk to be named `[my-chunk-name].js` instead of `[id].js`.

`webpackMode`: 从 webpack 2.6.0 开始，可以指定以不同的模式解析动态导入。支持以下选项：

- "lazy"（默认）：为每个 `import()` 导入的模块，生成一个可延迟加载(lazy-loadable) chunk。
- "lazy-once": 生成一个可以满足所有 `import()` 调用的单个可延迟加载(lazy-loadable) chunk。此 chunk 将在第一次 `import()` 调用时获取，随后的 `import()` 调用将使用相同的网络响应。注意，这种模式仅在部分动态语句中有意义，例如 `import(`./locales/${language}.json`)`，其中可能含有多个被请求的模块路径。
- "eager": 不会生成额外的 chunk，所有模块都被当前 chunk 引入，并且没有额外的网络请求。仍然会返回 `Promise`，但是是 `resolved` 状态。和静态导入相对比，在调用 `import()` 完成之前，该模块不会被执行。
- "weak": 尝试加载模块，如果该模块函数已经以其他方式加载（即，另一个 chunk 导入过此模块，或包含模块的脚本被加载）。仍然会返回 `Promise`，但是只有在客户端上已经有该 chunk 时才成功解析。如果该模块不可用，`Promise` 将会是 `rejected` 状态，并且网络请求永远不会执行。当需要的 chunks 始终在（嵌入在页面中的）初始请求中手动提供，而不是在应用程序导航在最初没有提供的模块导入的情况下触发，这对于通用渲染（SSR）是非常有用的。

`webpackPrefetch`: Tells the browser that the resource is probably needed for some navigation in the future. Check out the guide for more information on [how webpackPrefetch works](#).

`webpackPreload`: Tells the browser that the resource might be needed during the current navigation. Check out the guide for more information on [how webpackPreload works](#).

注意，所有这些选项都可以组合起来使用，如 `/* webpackMode: "lazy-once", webpackChunkName: "all-i18n-data" */`，这会按没有花括号的 JSON5 对象去解析。它会被包裹在 JavaScript 对象中，并使用 node VM 执行。所有你不需要添加花括号。

`webpackInclude`: 在导入解析(import resolution)过程中，用于匹配的正则表达式。只有匹配到的模块才会被打包。

`webpackExclude`: 在导入解析(import resolution)过程中，用于匹配的正则表达式。所有匹配到的模块都不会被打包。

注意，`webpackInclude` 和 `webpackExclude` 选项不会影响到前缀，例如：`./locale`。

完全动态的语句（如 `import(foo)`），因为 webpack 至少需要一些文件的路径信息，而 `foo` 可能是系统或项目中任何文件的任何路径，因此 `foo` 将会解析失败。`import()` 必须至少包含模块位于何处的路径信息，所以打包应当限

制在一个指定目录或一组文件中。

调用 `import()` 时，包含在其中的动态表达式 `request`，会潜在的请求的每个模块。例如，`import(`./locale/${language}.json`)` 会导致 `./locale` 目录下的每个 `.json` 文件，都被打包到新的 `chunk` 中。在运行时，当计算出变量 `language` 时，任何文件（如 `english.json` 或 `german.json`）都可能会被用到。Using the `webpackInclude` and `webpackExclude` options allows us to add regex patterns that reduce the files that webpack will bundle for this import.

在 `webpack` 中使用 `System.import` 不符合提案规范，所以在 `2.1.0-beta.28` 后被弃用，并且建议使用 `import()`。

CommonJS

CommonJS 致力于为浏览器之外的 JavaScript 指定一个生态系统。`webpack` 支持以下的 CommonJS 方法：

`require`

```
require(dependency: String);
```

以同步的方式检索其他模块的导出。由编译器(compiler)来确保依赖项在最终输出 `bundle` 中可用。

```
var $ = require('jquery');
var myModule = require('my-module');
```

以异步的方式使用，可能不会达到预期的效果。

`require.resolve`

```
require.resolve(dependency: String);
```

以同步的方式获取模块的 ID。由编译器(compiler)来确保依赖项在最终输出 `bundle` 中可用。更多关于模块的信息，请点击这里 [module.id](#)。

`webpack` 中模块 ID 是一个数字（而在 NodeJS 中是一个字符串 -- 也就是文件名）。

`require.cache`

多处引用同一个模块，最终只会产生一次模块执行和一次导出。所以，会在运行时(runtime)中会保存一份缓存。删除此缓存，会产生新的模块执行和新的导出。

只有很少数的情况需要考虑兼容性！

```
var d1 = require('dependency');
require('dependency') === d1;
delete require.cache[require.resolve('dependency')];
require('dependency') !== d1;

// in file.js
require.cache[module.id] === module;
require('./file.js') === module.exports;
delete require.cache[module.id];
require.cache[module.id] === undefined;
require('./file.js') !== module.exports; // 这是理论上的操作不相等；在实际运行
require.cache[module.id] !== module;
```

require.ensure

`require.ensure()` 是 webpack 特有的，已经被 `import()` 取代。

```
require.ensure(
  dependencies: String[],
  callback: function(require),
  errorCallback: function(error),
  chunkName: String
)
```

给定 `dependencies` 参数，将其对应的文件拆分到一个单独的 bundle 中，此 bundle 会被异步加载。当使用 CommonJS 模块语法时，这是动态加载依赖的唯一方法。意味着，可以在模块执行时才运行代码，只有在满足某些条件时才加载依赖项。

这个特性依赖于内置的 `Promise`。如果想在低版本浏览器使用 `require.ensure`，记得使用像 `es6-promise` 或者 `promise-polyfill` 这样 polyfill 库，来预先填充(shim) `Promise` 环境。

```
var a = require('normal-dep');

if ( module.hot ) {
  require.ensure(['b'], function(require) {
    var c = require('c');

    // Do something special...
  });
}
```

按照上面指定的顺序，webpack 支持以下参数：

- `dependencies`: 字符串构成的数组，声明 `callback` 回调函数中所需的所有模块。
- `callback`: 只要加载好全部依赖，webpack 就会执行此函数。`require` 函数的实现，作为参数传入此函数。当程序运行需要依赖时，可以使用 `require()` 来加载依赖。函数体可以使用此参数，来进一步执行 `require()` 模块。
- `errorCallback`: 当 webpack 加载依赖失败时，会执行此函数。

- `chunkName`: 由 `require.ensure()` 创建出的 chunk 的名字。通过将同一个 `chunkName` 传递给不同的 `require.ensure()` 调用，我们可以将它们的代码合并到一个单独的 chunk 中，从而只产生一个浏览器必须加载的 bundle。

虽然我们将 `require` 的实现，作为参数传递给回调函数，然而如果使用随意的名字，例如 `require.ensure([], function(request) { request('someModule'); })` 则无法被 webpack 静态解析器处理，所以还是请使用 `require`，例如 `require.ensure([], function(require) { require('someModule'); })`。

AMD

AMD(Asynchronous Module Definition) 是一种定义了写入模块接口和加载模块接口的 JavaScript 规范。webpack 支持以下的 AMD 方法：

`define` (通过 factory 方法导出)

```
define([name: String], [dependencies: String[]], factoryMethod: function
```

如果提供 `dependencies` 参数，将会调用 `factoryMethod` 方法，并（以相同的顺序）传入每个依赖项的导出。如果未提供 `dependencies` 参数，则调用 `factoryMethod` 方法时传入 `require, exports` 和 `module`（用于兼容）。如果此方法返回一个值，则返回值会作为此模块的导出。由编译器(compiler)来确保依赖项在最终输出 bundle 中可用。

注意：webpack 会忽略 `name` 参数。

```
define(['jquery', 'my-module'], function($, myModule) {
  // 使用 $ 和 myModule 做一些操作......

  // 导出一个函数
  return function doSomething() {
    // ...
  };
});
```

此 `define` 导出方式不能在异步函数中调用。

`define` (通过 value 导出)

```
define(value: !Function)
```

只会将提供的 `value` 导出。这里的 `value` 可以是除函数外的任何值。

```
define({
  answer: 42
});
```

此 `define` 导出方式不能在异步函数中调用。

`require` (AMD 版本)

```
require(dependencies: String[], [callback: function(...)])
```

与 `require.ensure` 类似，给定 `dependencies` 参数，将其对应的文件拆分到一个单独的 bundle 中，此 bundle 会被异步加载。然后会调用 `callback` 回调函数，并传入 `dependencies` 数组中每一项的导出。

这个特性依赖于内置的 `Promise`。如果想在低版本浏览器使用 `require.ensure`，记得使用像 `es6-promise` 或者 `promise-polyfill` 这样 polyfill 库，来预先填充(shim) `Promise` 环境。

```
require(['b'], function(b) {
  var c = require('c');
});
```

这里没有提供命名 chunk 名称的选项。

标签模块(Labeled Modules)

webpack 内置的 `LabeledModulesPlugin` 插件，允许使用下面的方法导出和导入模块：

`export` 标签

导出给定的 `value`。`export` 标记可以出现在函数声明或变量声明之前。函数名或变量名是导出值的标识符。

```
export: var answer = 42;
export: function method(value) {
  // 做一些操作.....
};
```

以异步的方式使用，可能不会达到预期的效果。

`require` 标签

使当前作用域下，可访问所依赖模块的所有导出。`require` 标签可以放置在一个字符串之前。依赖模块必须使用 `export` 标签导出值。CommonJS 或 AMD 模块无法通过这种方式，使用标签模块的导出。

`some-dependency.js`

```
export: var answer = 42;
```

```
export: function method(value) {
  // 执行一些操作.....
};

require: 'some-dependency';
console.log(answer);
method(...);
```

webpack

webpack 除了支持上述的语法之外，还可以使用一些 webpack 特定的方法：

require.context

```
require.context(
  directory: String,
  includeSubdirs: Boolean /* 可选的，默认值是 true */,
  filter: RegExp /* 可选的，默认值是 /^\.\/.*$/, 所有文件 */,
  mode: String /* 可选的，'sync' | 'eager' | 'weak' | 'lazy' | 'lazy-once' */
)
```

指定一系列完整的依赖关系，通过一个 `directory` 路径、一个 `includeSubdirs` 选项、一个 `filter` 更细粒度的控制模块引入和一个 `mode` 定义加载方式。然后可以很容易地解析模块：

```
var context = require.context('components', true, /\.html$/);
var componentA = context.resolve('componentA');
```

If `mode` is specified as "lazy", the underlying modules will be loaded asynchronously:

```
var context = require.context('locales', true, /\.json$/, 'lazy');
context('localeA').then(locale => {
  // do something with locale
});
```

The full list of available modes and its behavior is described in [import\(\)](#) documentation.

require.include

```
require.include(dependency: String)
```

引入一个不需要执行的依赖，这可以用于优化输出 chunk 中的依赖模块的位置。

```
require.include('a');
require.ensure(['a', 'b'], function(require) { /* ... */ });
require.ensure(['a', 'c'], function(require) { /* ... */ });
```

这会产生以下输出：

- entry chunk: file.js and a

- anonymous chunk: b
- anonymous chunk: c

如果不使用 `require.include('a')`，输出的两个匿名 chunk 都有模块 a。

require.resolveWeak

与 `require.resolve` 类似，但是这不会将 `module` 引入到 `bundle` 中。这就是所谓的"弱(weak)"依赖。

```
if(__webpack_modules__[require.resolveWeak('module')]) {  
    // 模块可用时，执行一些操作.....  
}  
if(require.cache[require.resolveWeak('module')]) {  
    // 在模块被加载之前，执行一些操作.....  
}  
  
// 你可以像执行其他 require/import 方法一样，  
// 执行动态解析（“上下文”）。  
const page = 'Foo';  
__webpack_modules__[require.resolveWeak(`./page/${page}`)];
```

`require.resolveWeak` 是通用渲染 (SSR + 代码分离) 的基础，例如在 `react-universal-component` 等包中的用法。它允许代码在服务器端和客户端初始页面的加载上同步渲染。它要求手动或以某种方式提供 chunk。它可以在不需要指示应该被打包的情况下引入模块。它与 `import()` 一起使用，当用户导航触发额外的导入时，它会被接管。

模块变量

本章节涵盖了使用 webpack 编译的代码中所有的变量。模块将通过 `module` 和其他变量，来访问编译过程中的某些数据。

`module.loaded` (NodeJS)

`false` 表示该模块正在执行，`true` 表示同步执行已经完成。

`module.hot` (webpack 特有变量)

表示 模块热替换(Hot Module Replacement) 是否启用，并给进程提供一个接口。详细说明请查看 [模块热替换 API 页面](#)

`module.id` (CommonJS)

当前模块的 ID。

```
module.id === require.resolve('./file.js');
```

`module.exports` (CommonJS)

调用者通过 `require` 对模块进行调用时返回的值（默认为一个新对象）。

```
module.exports = function doSomething() {
  // 做一些操作.....
};
```

无法在异步函数中访问该变量

`exports` (CommonJS)

该变量默认值为 `module.exports` (即一个对象)。如果 `module.exports` 被重写的话，`exports` 不再会被导出。

```
exports.someValue = 42;
exports.anObject = {
  x: 123
};
exports.aFunction = function doSomething() {
  // Do something
};
```

`global` (NodeJS)

见 [Node.js global](#).

process (NodeJS)

见 [Node.js process](#).

__dirname (NodeJS)

取决于 `node.__dirname` 配置选项:

- `false`: Not defined
- `mock`: equal `"/"`
- `true`: [Node.js __dirname](#)

如果在一个被 Parser 解析的表达式内部使用，则配置选项会被当作 `true` 处理。

__filename (NodeJS)

取决于 `node.__filename` 配置选项:

- `false`: Not defined
- `mock`: equal `"/index.js"`
- `true`: [Node.js __filename](#)

如果在一个被 Parser 解析的表达式内部使用，则配置选项会被当作 `true` 处理。

__resourceQuery (webpack 特有变量)

当前模块的资源查询(resource query)。如果进行了如下的 `require` 调用，那么查询字符串(query string)在 `file.js` 中可访问。

```
require('file.js?test');
```

file.js

```
__resourceQuery === '?test';
```

__webpack_public_path__ (webpack 特有变量)

等同于 `output.publicPath` 配置选项.

__webpack_require__ (webpack 特有变量)

原始 `require` 函数。这个表达式不会被解析器解析为依赖。

__webpack_chunk_load__ (webpack 特有变量)

内部 chunk 载入函数，有两个输入参数：

- `chunkId` 需要载入的 chunk id。
- `callback(require)` chunk 载入后调用的回调函数。

__webpack_modules__ (webpack 特有变量)

访问所有模块的内部对象。

__webpack_hash__ (webpack 特有变量)

这个变量只有在启用 `HotModuleReplacementPlugin` 或者 `ExtendedAPIPlugin` 时才生效。这个变量提供对编译过程中(`compilation`)的 hash 信息的获取。

__non_webpack_require__ (webpack 特有变量)

生成一个不会被 webpack 解析的 `require` 函数。配合全局可以获取到的 `require` 函数，可以完成一些酷炫操作。

DEBUG (webpack 特有变量)

等同于配置选项中的 `debug`。

Plugin API

插件是 webpack 生态系统的重要组成部分，为社区用户提供了一种强大方式来直接触及 webpack 的编译过程(compilation process)。插件能够 钩入(hook) 到在每个编译(compilation)中触发的所有关键事件。在编译的每一步，插件都具备完全访问 `compiler` 对象的能力，如果情况合适，还可以访问当前 `compilation` 对象。

对于编写插件的高度概括，请从编写一个插件开始。

我们首先回顾 `tapable` 工具，它提供了 webpack 插件接口的支柱。

Tapable

`tapable` 这个小型 library 是 webpack 的一个核心工具，但也可用于其他地方，以提供类似的插件接口。webpack 中许多对象扩展自 `Tapable` 类。这个类暴露 `tap`, `tapAsync` 和 `tapPromise` 方法，可以使用这些方法，注入自定义的构建步骤，这些步骤将在整个编译过程中不同时机触发。

请查看 文档 了解更多信息。理解三种 `tap` 方法以及提供这些方法的钩子至关重要。要注意到，扩展自 `Tapable` 的对象（例如 `compiler` 对象）、它们提供的钩子和每个钩子的类型（例如 `SynchHook`）。

插件类型(plugin types)

根据所使用的 钩子(hook) 和 `tap` 方法，插件可以以多种不同的方式运行。这个工作方式与 `Tapable` 提供的 `hooks` 密切相关。`compiler hooks` 分别记录了 `Tapable` 内在的钩子，指出哪些 `tap` 方法可用。

因此，根据你触发到 `tap` 事件，插件可能会以不同的方式运行。例如，当钩入 `compile` 阶段时，只能使用同步的 `tap` 方法：

```
compiler.hooks.compile.tap('MyPlugin', params => {
  console.log('以同步方式触及 compile 钩子。');
})
```

然而，对于能够使用了 `AsyncHook`(异步钩子) 的 `run`，我们可以使用 `tapAsync` 或 `tapPromise` (以及 `tap`)：

```
compiler.hooks.run.tapAsync('MyPlugin', (source, target, routesList, callback) =>
  console.log('以异步方式触及 run 钩子。');
  callback();
);

compiler.hooks.run.tapPromise('MyPlugin', (source, target, routesList) =>
  return new Promise(resolve => setTimeout(resolve, 1000)).then(() => {
```

```

        console.log('以具有延迟的异步方式触及 run 钩子。');
    });
});

compiler.hooks.run.tapPromise('MyPlugin', async (source, target, routes) =>
    await new Promise(resolve => setTimeout(resolve, 1000));
    console.log('以具有延迟的异步方式触及 run 钩子。');
);

```

这些需求(story)的含义在于，可以有多种方式将 hook 钩入到 compiler 中，可以让各种插件都以合适的方式去运行。

自定义的钩子函数(custom hooks)

为了给其他插件的编译添加一个新的钩子，来 tap(触及) 到这些插件的内部，直接从 tapable 中 require 所需的钩子类(hook class)，然后创建：

```

const SyncHook = require('tapable').SyncHook;

// 具有 `apply` 方法.....
if (compiler.hooks.myCustomHook) throw new Error('Already in use');
compiler.hooks.myCustomHook = new SyncHook(['a', 'b', 'c']);

// 在你想要触发钩子的位置/时机下调用.....
compiler.hooks.myCustomHook.call(a, b, c);

```

再次声明，查看 tapable 文档 来，了解更多不同的钩子类(hook class)，以及它们是如何工作的。

Reporting Progress

Plugins can report progress via [ProgressPlugin](#), which prints progress messages to stderr by default. In order to enable progress reporting, pass a `--progress` argument when running the [webpack CLI](#).

It is possible to customize the printed output by passing different arguments to the `reportProgress` function of [ProgressPlugin](#).

To report progress, a plugin must tap into a hook using the `context: true` option:

```

compiler.hooks.emit.tapAsync({
    name: 'MyPlugin',
    context: true
}, (context, compiler, callback) => {
    const reportProgress = context && context.reportProgress;
    if (reportProgress) reportProgress(0.95, 'Starting work');
    setTimeout(() => {
        if (reportProgress) reportProgress(0.95, 'Done work');
        callback();
    }, 1000);
}

```

```
} );
```

The `reportProgress` function may be called with these arguments:

```
reportProgress(percentage, ...args);
```

- `percentage`: This argument is unused; instead, `ProgressPlugin` will calculate a percentage based on the current hook.
- `...args`: Any number of strings, which will be passed to the `ProgressPlugin` handler to be reported to the user.

Note that only a subset of compiler and compilation hooks support the `reportProgress` function. See [ProgressPlugin](#) for a full list.

下一步

查看 [compiler hooks](#) 部分，了解所有可用的 `compiler` 钩子和其所需的参数的详细列表。

compiler 钩子

`Compiler` 模块是 webpack 的支柱引擎，它通过 [CLI](#) 或 [Node API](#) 传递的所有选项，创建出一个 `compilation` 实例。它扩展(`extend`)自 `Tapable` 类，以便注册和调用插件。大多数面向用户的插件首，会先在 `Compiler` 上注册。

此模块会暴露在 `webpack.Compiler`，可以直接通过这种方式使用。关于更多信息，请查看[这个示例](#)。

在为 webpack 开发插件时，你可能需要知道每个钩子函数是在哪里调用的。想要了解这些，请在 webpack 源码中搜索 `hooks.<hook name>.call`。

监听(watching)

`Compiler` 支持可以监控文件系统的监听(watching)机制，并且在文件修改时重新编译。当处于监听模式(watch mode)时，`compiler` 会触发诸如 `watchRun`, `watchClose` 和 `invalid` 等额外的事件。通常用于开发环境中使用，也常常会在 `webpack-dev-server` 这些工具的底层之下调用，由此开发人员无须每次都使用手动方式重新编译。还可以通过 [CLI](#) 进入监听模式。

相关钩子

以下生命周期钩子函数，是由 `compiler` 暴露，可以通过如下方式访问：

```
compiler.hooks.someHook.tap /* ... */;
```

取决于不同的钩子类型，也可以在某些钩子上访问 `tapAsync` 和 `tapPromise`。

关于钩子类型的描述，请查看 [Tapable 文档](#)。

`entryOption`

`SyncBailHook`

在 webpack 选项中的 `entry` 配置项 处理过之后，执行插件。

`afterPlugins`

`SyncHook`

设置完初始插件之后，执行插件。

参数：`compiler`

afterResolvers

SyncHook

resolver 安装完成之后，执行插件。

参数： compiler

environment

SyncHook

environment 准备好之后，执行插件。

afterEnvironment

SyncHook

environment 安装完成之后，执行插件。

beforeRun

AsyncSeriesHook

compiler.run() 执行之前，添加一个钩子。

参数： compiler

run

AsyncSeriesHook

开始读取 records 之前，钩入(hook into) compiler。

参数： compiler

watchRun

AsyncSeriesHook

监听模式下，一个新的编译(compilation)触发之后，执行一个插件，但是是在实际编译开始之前。

参数： compiler

normalModuleFactory

SyncHook

NormalModuleFactory 创建之后，执行插件。

参数： normalModuleFactory

contextModuleFactory

ContextModuleFactory 创建之后，执行插件。

参数： contextModuleFactory

beforeCompile

AsyncSeriesHook

编译(compilation)参数创建之后，执行插件。

参数： compilationParams

compile

SyncHook

一个新的编译(compilation)创建之后，钩入(hook into) compiler。

参数： compilationParams

thisCompilation

SyncHook

触发 compilation 事件之前执行（查看下面的 compilation）。

参数： compilation

compilation

SyncHook

编译(compilation)创建之后，执行插件。

参数： compilation

make

AsyncParallelHook

...

参数: compilation

afterCompile

AsyncSeriesHook

...

参数: compilation

shouldEmit

SyncBailHook

此时返回 true/false。

参数: compilation

emit

AsyncSeriesHook

生成资源到 output 目录之前。

参数: compilation

afterEmit

AsyncSeriesHook

生成资源到 output 目录之后。

参数: compilation

done

AsyncSeriesHook

编译(compilation)完成。

参数: stats

failed

SynCHook

编译(compilation)失败。

参数: error

invalid

SynCHook

监听模式下，编译无效时。

参数: fileName, changeTime

watchClose

SynCHook

监听模式停止。

compilation 钩子

Compilation 模块会被 Compiler 用来创建新的编译（或新的构建）。compilation 实例能够访问所有的模块和它们的依赖（大部分是循环依赖）。它会对应应用程序的依赖图中所有模块进行字面上的编译(literal compilation)。在编译阶段，模块会被加载.loaded)、封存(sealed)、优化(optimized)、分块(chunked)、哈希(hashed)和重新创建(restored)。

Compilation 类扩展(extend)自 Tappable，并提供了以下生命周期钩子。可以按照 compiler 钩子的相同方式，调用 tap:

```
compilation.hooks.someHook.tap(/* ... */);
```

和 compiler 用法相同，取决于不同的钩子类型，也可以在某些钩子上访问 tapAsync 和 tapPromise。

buildModule

SyncHook

在模块构建开始之前触发。

参数: module

rebuildModule

SyncHook

在重新构建一个模块之前触发。

参数: module

failedModule

SyncHook

模块构建失败时执行。

参数: module error

succeededModule

SyncHook

模块构建成功时执行。

参数: module

finishModules

SyncHook

所有模块都完成构建。

参数: modules

finishRebuildingModule

SyncHook

一个模块完成重新构建。

参数: module

seal

SyncHook

编译(compilation)停止接收新模块时触发。

unseal

SyncHook

编译(compilation)开始接收新模块时触发。

optimizeDependenciesBasic

SyncBailHook

...

参数: modules

optimizeDependencies

SyncBailHook

依赖优化开始时触发。

参数: modules

optimizeDependenciesAdvanced

SyncBailHook

...

参数: modules

afterOptimizeDependencies

SyncHook

...

参数: modules

optimize

SyncHook

优化阶段开始时触发。

optimizeModulesBasic

SyncBailHook

...

参数: modules

optimizeModules

SyncBailHook

...

参数: modules

optimizeModulesAdvanced

SyncBailHook

...

参数: modules

afterOptimizeModules

SyncHook

...

参数: modules

optimizeChunksBasic

SyncBailHook

...

参数: chunks

optimizeChunks

SyncBailHook

优化 chunk。

参数: chunks

optimizeChunksAdvanced

SyncBailHook

...

参数: chunks

afterOptimizeChunks

SyncHook

chunk 优化完成之后触发。

参数: chunks

optimizeTree

AsyncSeriesHook

异步优化依赖树。

参数: chunks modules

afterOptimizeTree

SyncHook

...

参数: chunks modules

optimizeChunkModulesBasic

SyncBailHook

...

参数: chunks modules

optimizeChunkModules

SyncBailHook

...

参数: chunks modules

optimizeChunkModulesAdvanced

SyncBailHook

...

参数: chunks modules

afterOptimizeChunkModules

SyncHook

...

参数: chunks modules

shouldRecord

SyncBailHook

...

reviveModules

SyncHook

从 records 中恢复模块信息。

参数: modules records

optimizeModuleOrder

SyncHook

将模块从最重要的到最不重要的进行排序。

参数: modules

advancedOptimizeModuleOrder

SyncHook

...

参数: modules

beforeModuleIds

SyncHook

...

参数: modules

moduleIds

SyncHook

...

参数: modules

optimizeModuleIds

SyncHook

...

参数: chunks

afterOptimizeModuleIds

SyncHook

...

参数: chunks

reviveChunks

SyncHook

从 records 中恢复 chunk 信息。

参数: modules records

optimizeChunkOrder

SyncHook

将 chunk 从最重要的到最不重要的进行排序。

参数: chunks

beforeOptimizeChunkIds

SyncHook

chunk id 优化之前触发。

参数: chunks

optimizeChunkIds

SyncHook

优化每个 chunk 的 id。

参数: chunks

afterOptimizeChunkIds

SyncHook

chunk id 优化完成之后触发。

参数: chunks

recordModules

SyncHook

将模块信息存储到 records。

参数: modules records

recordChunks

SyncHook

将 chunk 信息存储到 records。

参数: chunks records

beforeHash

SyncHook

在编译被哈希(hashed)之前。

afterHash

SyncHook

在编译被哈希(hashed)之后。

recordHash

SyncHook

...

参数: records

record

SyncHook

将 compilation 相关信息存储到 records 中。

参数: compilation records

beforeModuleAssets

SyncHook

...

shouldGenerateChunkAssets

SyncBailHook

...

beforeChunkAssets

SyncHook

在创建 chunk 资源(asset)之前。

additionalChunkAssets

SyncHook

为 chunk 创建附加资源(asset)

参数: chunks

records

SyncHook

...

参数: compilation records

additionalAssets

AsyncSeriesHook

为编译(compilation)创建附加资源(asset)。这个钩子可以用来下载图像，例如：

```
compilation.hooks.additionalAssets.tapAsync('MyPlugin', callback => {
  download('https://img.shields.io/npm/v/webpack.svg', function(resp) {
    if(resp.status === 200) {
      compilation.assets['webpack-version.svg'] = toAsset(resp);
      callback();
    } else {
      callback(new Error('[webpack-example-plugin] Unable to download t'))
    }
  });
});
```

optimizeChunkAssets

AsyncSeriesHook

优化所有 chunk 资源(asset)。资源(asset)会被存储在 compilation.assets。每个 chunk 都有一个 files 属性，指向这个 chunk 创建的所有文件。附加资源(asset)被存储在 compilation.additionalChunkAssets 中。

参数: chunks

以下是在每个 chunk 添加 banner 的简单示例。

```
compilation.hooks
.optimizeChunkAssets
.tapAsync('MyPlugin', (chunks, callback) => {
  chunks.forEach(chunk => {
    chunk.files.forEach(file => {
      compilation.assets[file] = new ConcatSource(
        '/**Sweet Banner**/',
        '\n',
        compilation.assets[file]
      );
    });
  });
  callback();
});
```

afterOptimizeChunkAssets

SyncHook

chunk 资源(asset)已经被优化。

参数: chunks

这里是一个来自 [@boopathi](#) 的示例插件，详细地输出每个 chunk 里有什么。

```
compilation.hooks.afterOptimizeChunkAssets.tap('MyPlugin', chunks => {
  chunks.forEach(chunk => {
    console.log({
      id: chunk.id,
      name: chunk.name,
      includes: chunk.modules.map(module => module.request)
    });
  });
});
```

optimizeAssets

AsyncSeriesHook

优化存储在 compilation.assets 中的所有资源(asset)。

参数: assets

afterOptimizeAssets

SyncHook

资源优化已经结束。

参数: assets

needAdditionalSeal

SyncBailHook

...

afterSeal

AsyncSeriesHook

...

chunkHash

SyncHook

...

参数: chunk chunkHash

moduleAsset

SyncHook

一个模块中的一个资源被添加到编译中。

参数: module filename

chunkAsset

SyncHook

一个 chunk 中的一个资源被添加到编译中。

参数: chunk filename

assetPath

SyncWaterfallHook

...

参数: filename data

needAdditionalPass

SyncBailHook

...

childCompiler

SyncHook

...

参数: childCompiler compilerName compilerIndex

normalModuleLoader

SynchHook

普通模块 loader，真正（一个接一个地）加载模块图(graph)中所有模块的函数。

参数: loaderContext module

dependencyReference

SyncWaterfallHook

Compilation.hooks.dependencyReference(depRef, dependency, module) allows to change the references reported by dependencies.

Parameters: depRef dependency module

resolver

resolver 是由 `enhanced-resolve` package 创建出来的。Resolver 类继承了 `tapable` 类，并且使用 `tapable` 提供的一些钩子。可以直接使用 `enhanced-resolve` package 创建一些新的 resolver，然而，所有的 `compiler` 实例都有一些可以接触 (tap into) 到的 resolver 实例。

在继续阅读之前，请确保至少了解过 `enhanced-resolve` 和 `tapable` 文档。

类型

`compiler` 类有三种类型的内置 resolver:

- Normal: 通过绝对路径或相对路径，解析一个模块。
- Context: 通过给定的 context 解析一个模块。
- Loader: 解析一个 webpack loader。

根据需要，所有这些 `compiler` 用到的内置 resolver，都可以通过插件进行自定义：

```
compiler.resolverFactory.plugin('resolver [type]', resolver => {
  resolver.hooks.resolve.tapAsync('MyPlugin', params => {
    // ...
  });
});
```

其中 `[type]` 是上面提到的三个 resolver 之一，指定为：

- `normal`
- `context`
- `loader`

完整的钩子和描述列表，请查看 `enhanced-resolve` 文档。

配置选项

上面提到的 resolver，也可以通过在配置文件使用 `resolve` 或 `resolveLoader` 选项来自定义。这些选项允许用户通过各种选项（包括解析 `plugins`），来改变解析行为。

resolver 插件（例如 `[DirectoryNamedPlugin]` (<https://github.com/shaketbaby/directory-named-webpack-plugin>)）可以直接包含在 `resolve.plugins` 中，而不是使用标准插件用法。注意，`resolve` 配置会影响 `normal` 和 `context` 这两个 resolver，而 `resolveLoader` 则用于修改 loader

resolver。

parser

`parser` 实例，是用来解析由 webpack 处理过的每个模块。`parser` 也是扩展自 `tapable` 的 webpack 类，并且提供多种 `tapable` 钩子，插件作者可以使用它来自定义解析过程。

以下示例中，`parser` 位于 `normalModuleFactory` 这个中，因此需要调用额外钩子来进行获取：

```
compiler.hooks.normalModuleFactory.tap('MyPlugin', factory => {
  factory.hooks.parser.for('javascript/auto').tap('MyPlugin', (parser, ...) =>
    parser.hooks.someHook.tap(/* ... */);
  );
});
```

和 `compiler` 用法相同，取决于不同的钩子类型，也可以在某些钩子上访问 `tapAsync` 和 `tapPromise`。

相关钩子

以下生命周期钩子函数，是由 `parser` 暴露，可以通过如下方式访问：

evaluateTypeof

SyncBailHook

取值标识符(identifier)的类型。 (译注：取值(evaluate)是一个动词，表示对参数进行求值并返回)

参数: `expression`

evaluate

SyncBailHook

取值一个表达式(expression)

参数: `expression`

evaluateIdentifier

SyncBailHook

取值一个自由变量标识符。

参数: expression

evaluateDefinedIdentifier

SyncBailHook

取值一个定义变量标识符。

参数: expression

evaluateCallExpressionMember

SyncBailHook

进行一次「成功取值表达式的成员函数(member function of a successfully evaluated expression)」调用取值。

参数: expression param

statement

SyncBailHook

通用钩子，在从代码片段中解析语句时调用。

参数: statement

statementIf

SyncBailHook

...

参数: statement

label

SyncBailHook

...

参数: statement

import

SyncBailHook

...

参数: statement source

importSpecifier

SyncBailHook

...

参数: statement source exportName identifierName

export

SyncBailHook

...

参数: statement

exportImport

SyncBailHook

...

参数: statement source

exportDeclaration

SyncBailHook

...

参数: statement declaration

exportExpression

SyncBailHook

...

参数: statement declaration

exportSpecifier

SyncBailHook

...

参数: statement identifierName exportName index

exportImportSpecifier

SyncBailHook

...

参数: statement source identifierName exportName index

varDeclaration

SyncBailHook

...

参数: declaration

varDeclarationLet

SyncBailHook

...

参数: declaration

varDeclarationConst

SyncBailHook

...

参数: declaration

varDeclarationVar

SyncBailHook

...

参数: declaration

canRename

SyncBailHook

...

参数: initExpression

rename

SyncBailHook

...

参数: initExpression

assigned

SyncBailHook

...

参数: expression

assign

SyncBailHook

...

参数: expression

typeof

SyncBailHook

...

参数: expression

call

SyncBailHook

...

参数: expression

callAnyMember

SyncBailHook

...

参数: expression

new

SyncBailHook

...

参数: expression

expression

SyncBailHook

...

参数: expression

expressionAnyMember

SyncBailHook

...

参数: expression

expressionConditionalOperator

SyncBailHook

...

参数: expression

program

SyncBailHook

访问代码片段的抽象语法树(Abstract syntax tree - AST)

参数: ast comments

指南

本指南章节包含有关理解和掌握 webpack 提供的各种工具和特性。首先，通过安装进行简单引入。

指南会逐步带你由浅入深。本章节更多是作为一个切入点，一旦阅读完成后，你就会更加容易深入到实际的 配置 文档中。

指南中运行 webpack 后显示的输出，可能和新版本的输出略有不同。这是意料之中的事情。只要 bundle 看起来接近，而且运行正常，那就没有问题。如果你遇到在新版本中，示例无法良好运行，请创建一个 issue，我们将尽力解决版本差异。

安装

本指南介绍了安装 webpack 的各种方法。

预先准备

在开始之前，请确保安装了 [Node.js](#) 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，或者缺少相关 package。

本地安装

最新的 webpack 正式版本是：

webpack v4.39.1

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 v4+ 版本，你还需要安装 [CLI](#)。

```
npm install --save-dev webpack-cli
```

对于大多数项目，我们建议本地安装。这可以在引入突破式变更(breaking change)版本时，更容易分别升级项目。通常会通过运行一个或多个 [npm scripts](#) 以在本地 `node_modules` 目录中查找安装的 webpack，来运行 webpack：

```
"scripts": {
  "build": "webpack --config webpack.config.js"
}
```

想要运行本地安装的 webpack，你可以通过 `node_modules/.bin/webpack` 来访问它的 bin 版本。

全局安装

通过以下 NPM 安装方式，可以使 `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。

最新体验版本

如果你热衷于使用最新版本的 webpack，你可以使用以下命令安装 beta 版本，或者直接从 webpack 的仓库中安装：

```
npm install webpack@beta  
npm install webpack/webpack#<tagname/branchname>
```

在安装这些最新体验版本时要小心！它们可能仍然包含 bug，因此不应该用于生产环境。

起步

webpack 用于编译 JavaScript 模块。一旦完成 安装，你就可以通过 webpack [CLI](#) 或 [API](#) 与其配合交互。如果你还不熟悉 webpack，请阅读 [核心概念](#) 和 [对比](#)，了解为什么要使用 webpack，而不是社区中的其他工具。

基本安装

首先我们创建一个目录，初始化 npm，然后 [在本地安装 webpack](#)，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，来展示对目录、文件和代码所做的修改。

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() {  
  let element = document.createElement('div');  
  
  // lodash (目前通过一个 script 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 `private`(私有的)，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。

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": "^4.20.2",
    "webpack-cli": "^3.1.2"
  },
  "dependencies": {}
}
```

在此示例中，`<script>` 标签之间存在隐式依赖关系。在 `index.js` 文件执行之前，还需要在页面中先引入 `lodash`。这是因为 `index.js` 并未显式声明它需要 `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：

- 无法直接体现，脚本的执行依赖于外部库。
- 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

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(`output`) 目录，最终将在浏览器中加载：

project

```
webpack-demo
|- package.json
+ |- /dist
+   |- index.html
- |- index.html
|- /src
  |- index.js
```

要在 `index.js` 中装入 `lodash` 依赖，我们需要在本地安装 library：

```
npm install --save lodash
```

在安装一个 package，而此 package 要打包到生产环境 bundle 中时，你应该使用 `npm install --save`。如果你在安装一个用于开发环境目的的 package 时（例如，linter，测试库等），你应该使用 `npm install --save-dev`。在 [npm 文档](#) 中查找更多信息。

现在，在我们的 script 中 import `lodash`：

src/index.js

```
+ import _ from 'lodash';
+
function component() {
  let element = document.createElement('div');

- // lodash (目前通过一个 script 引入) 对于执行这一行是必需的
  element.innerHTML = _.join(['Hello', 'webpack'], ' ');

  return element;
}

document.body.appendChild(component());
```

现在，我们将会打包所有脚本，我们必须更新 `index.html` 文件。由于现在是通过 `import` 引入 `lodash`，所以要将 `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会以正确顺序执行的优化 bundle。

可以这样说，执行 `npx webpack`，会将我们的脚本 `src/index.js` 作为入口起点，也会生成 `dist/main.js` 作为输出。Node 8.2/npm 5.2.0 以上版本提供的 `npx` 命令，可以运行在开始安装的 webpack package 中的 webpack 二进制文件（即 `./node_modules/.bin/webpack`）：

```
npx webpack
```

```
...
Built at: 13/06/2018 11:52:07
 Asset      Size  Chunks      Chunk Names
main.js    70.4 KiB       0  [emitted]  main
...
WARNING in configuration (配置警告)
The 'mode' option has not been set, webpack will fallback to 'production'.
You can also set it to 'none' to disable any default behavior. Learn more about this in the documentation.
```

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

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

模块

ES2015 中的 `import` 和 `export` 语句已经被标准化，并且多数浏览器已经能够支持。一些旧版本浏览器虽然无法支持它们，但是通过 webpack 开箱即用的模块支持，我们也可以使用这些 ES2015 模块标准。

在幕后，webpack 实际上会将代码进行 transpile(转译)，因此旧版本浏览器也可以执行。如果检查 `dist/main.js`，你就可以看到 webpack 是如何实现，这是独创精巧的设计！除了 `import` 和 `export`，webpack 还能够很好地支持各种其他模块语法，更多信息请查看 模块 API。

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，请确保你在 webpack loader 系统 中使用了一个像是 Babel 或 Bublé 的 transpiler(转译器)。

使用一个配置文件

在 webpack v4 中，可以无须任何配置，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。这比在 terminal(终端) 中手动输入大

量命令要高效的多，所以让我们创建一个配置文件：

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: 'main.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

现在，让我们通过新的配置文件再次执行构建：

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

```
...
Asset      Size  Chunks      Chunk Names
main.js  70.4 KiB      0  [emitted]  main
...
```

WARNING in configuration(配置警告)

The 'mode' option has not been set, webpack will fallback to 'production'. You can also set it to 'none' to disable any default behavior. Learn more about this at [https://webpack.js.org/configuration/mode/](#).

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 rule)、插件(plugin)、resolve 选项，以及许多其他增强功能。更多详细信息请查看 配置文档。

npm scripts

考虑到用 CLI 这种方式来运行本地的 webpack 副本并不是特别方便，我们可以设置一个快捷方式。调整 `package.json` 文件，添加在 `npm scripts` 中添加一个 npm 命令：

package.json

```
{  
  "name": "webpack-demo",  
  "version": "1.0.0",  
  "description": "",  
  "scripts": {  
    "test": "echo \\"$Error: no test specified\\" && exit 1",  
+    "build": "webpack"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "webpack": "^4.20.2",  
    "webpack-cli": "^3.1.2"  
  },  
  "dependencies": {  
    "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 scripts`，我们可以像使用 `npx` 那样通过模块名引用本地安装的 `npm packages`。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  
  
...  
Asset      Size  Chunks      Chunk Names  
main.js   70.4 KiB       0  [emitted]  main  
...  
  
WARNING in configuration (配置警告)  
The 'mode' option has not been set, webpack will fallback to 'production'.  
You can also set it to 'none' to disable any default behavior. Learn more  
at https://webpack.js.org/configuration	mode/
```

通过在 `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，你已经有了一个基础构建配置，你应该移至下一章节 [资源管理指南](#)，以了解如何通过 `webpack` 来管理资源，例如 `images`、`fonts`。此刻你的项目看起来应该如下：

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
  |- main.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 设计思想，你应该看下 [基本概念](#) 和 [配置](#) 页面。此外，[API](#) 章节可以深入了解 webpack 提供的各种接口。

管理资源

如果你是从开始一直遵循着指南的示例，现在会有一个小项目，显示 "Hello webpack"。现在我们尝试混合一些其他资源，比如 images，看看 webpack 如何处理。

在 webpack 出现之前，前端开发人员会使用 grunt 和 gulp 等工具来处理资源，并将它们从 `/src` 文件夹移动到 `/dist` 或 `/build` 目录中。JavaScript 模块也遵循同样方式，但是，像 webpack 这样的工具，将动态打包所有依赖（创建所谓的 依赖图 (dependency graph)）。这是极好的创举，因为现在每个模块都可以明确表述它自身的依赖，可以避免打包未使用的模块。

webpack 最出色的功能之一就是，除了引入 JavaScript，还可以通过 loader 引入任何其他类型的文件。也就是说，以上列出的那些 JavaScript 的优点（例如显式依赖），同样可以用来构建 web 站点或 web 应用程序中的所有非 JavaScript 内容。让我们从 CSS 开始起步，或许你可能已经熟悉了下面这个设置。

安装

在开始之前，让我们对项目做一个小的修改：

dist/index.html

```
<!doctype html>
<html>
  <head>
-   <title>起步</title>
+   <title>管理资源</title>
  </head>
  <body>
-   <script src="./main.js"></script>
+   <script src="./bundle.js"></script>
  </body>
</html>
```

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
-    filename: 'main.js',
+    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

加载 CSS

为了在 JavaScript 模块中 `import` 一个 CSS 文件，你需要安装 `style-loader` 和 `css-loader`，并在 `module` 配置中添加这些 loader：

```
npm install --save-dev style-loader css-loader
```

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
+  module: {
+    rules: [
+      {
+        test: /\.css$/,
+        use: [
+          'style-loader',
+          'css-loader'
+        ]
+      }
+    ]
+  }
};
```

webpack 根据正则表达式，来确定应该查找哪些文件，并将其提供给指定的 loader。在这个示例中，所有以 `.css` 结尾的文件，都将被提供给 `style-loader` 和 `css-loader`。

这使你可以在依赖于此样式的 js 文件中 `import './style.css'`。现在，在此模块执行过程中，含有 CSS 字符串的 `<style>` 标签，将被插入到 html 文件的 `<head>` 中。

我们尝试一下，通过在项目中添加一个新的 `style.css` 文件，并将其 `import` 到我们的 `index.js` 中：

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
+  |- style.css
```

```
| - index.js  
|- /node_modules
```

src/style.css

```
.hello {  
  color: red;  
}
```

src/index.js

```
import _ from 'lodash';  
+ import './style.css';  
  
function component() {  
  var element = document.createElement('div');  
  
  // lodash 是由当前 script 脚本 import 导入进来的  
  element.innerHTML = _.join(['Hello', 'webpack'], ' ');  
+   element.classList.add('hello');  
  
  return element;  
}  
  
document.body.appendChild(component());
```

现在运行 build 命令：

```
npm run build
```

```
...  
Asset      Size  Chunks      Chunk Names  
bundle.js  76.4 KiB    0  [emitted]  main  
Entrypoint main = bundle.js  
...
```

再次在浏览器中打开 `index.html`，你应该看到 Hello webpack 现在的样式是红色。要查看 webpack 做了什么，请检查页面（不要查看页面源代码，因为它不会显示结果），并查看页面的 `head` 标签。它应该包含 `style` 块元素，也就是我们在 `index.js` 中 `import` 的 `css` 文件中的样式。

注意，在多数情况下，你也可以进行 CSS 提取，以便在生产环境中节省加载时间。最重要的是，现有的 loader 可以支持任何你可以想到的 CSS 风格 - `postcss`, `sass` 和 `less` 等。

加载 images 图像

假想，现在我们正在下载 CSS，但是像 `background` 和 `icon` 这样的图像，要如何处理呢？使用 `file-loader`，我们可以轻松地将这些内容混合到 CSS 中：

```
npm install --save-dev file-loader
```

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\.(png|svg|jpg|gif)$/,
        use: [
          'file-loader'
        ]
      }
    ]
  }
};
```

现在，在`import MyImage from './my-image.png'`时，此图像将被处理并添加到`output`目录，并且`_MyImage`变量将包含该图像在处理后的最终url。在使用`css-loader`时，如前所示，会使用类似过程处理你的CSS中的`url('./my-image.png')`。`loader`会识别这是一个本地文件，并将`'./my-image.png'`路径，替换为`output`目录中图像的最终路径。而`html-loader`以相同的方式处理``。

我们向项目添加一个图像，然后看它是如何工作的，你可以使用任何你喜欢的图像：

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
  |- /src
+   |- icon.png
   |- style.css
   |- index.js
|- /node_modules
```

src/index.js

```
import _ from 'lodash';
import './style.css';
+ import Icon from './icon.png';

function component() {
  var element = document.createElement('div');

  // lodash, 现在由此脚本导入
  element.innerHTML = _.join(['Hello', 'webpack'], ' ');
  element.classList.add('hello');

+ // 将图像添加到我们已经存在的 div 中。
+ var myIcon = new Image();
+ myIcon.src = Icon;
+
+ element.appendChild(myIcon);

  return element;
}

document.body.appendChild(component());
```

src/style.css

```
.hello {
  color: red;
+ background: url('./icon.png');
}
```

重新构建并再次打开 index.html 文件：

```
npm run build
```

```
...
          Asset      Size  Chunks   [emitted]  [big]
da4574bb234ddc4bb47cbe1ca4b20303.png  3.01 MiB
bundle.js        76.7 KiB    0  [emitted]
Entrypoint main = bundle.js
...
```

如果一切顺利，你现在应该看到你的 icon 图标成为了重复的背景图，以及 Hello webpack 文本旁边的 img 元素。如果检查此元素，你将看到实际的文件名已更改为 5c999da72346a995e7e2718865d019c8.png。这意味着 webpack 在 src 文件夹中找到我们的文件，并对其进行处理！

合乎逻辑下一步是，压缩和优化你的图像。查看 [image-webpack-loader](#) 和 [url-loader](#)，以了解更多关于如何增强加载处理图像功能。

加载 fonts 字体

那么，像字体这样的其他资源如何处理呢？file-loader 和 url-loader 可以接收并加载任何文件，然后将其输出到构建目录。这就是说，我们可以将它们用于任何类型的文件，也包括字体。让我们更新 webpack.config.js 来处理字体文件：

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\.(png|svg|jpg|gif)$/,
        use: [
          'file-loader'
        ]
      },
      {
        test: /\.(woff|woff2|eot|ttf|otf)$/,
        use: [
          'file-loader'
        ]
      }
    ]
  }
};
```

在项目中添加一些字体文件：

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
+  |- my-font.woff
+  |- my-font.woff2
  |- icon.png
  |- style.css
  |- index.js
```

```
| - /node_modules
```

配置好 loader 并将字体文件放在合适的位置后，你可以通过一个 `@font-face` 声明将其混合。本地的 `url(...)` 指令会被 webpack 获取处理，就像它处理图片一样：

src/style.css

```
+ @font-face {  
+   font-family: 'MyFont';  
+   src: url('./my-font.woff2') format('woff2'),  
+       url('./my-font.woff') format('woff');  
+   font-weight: 600;  
+   font-style: normal;  
+ }  
  
.hello {  
  color: red;  
+ font-family: 'MyFont';  
  background: url('./icon.png');  
}
```

现在，让我们重新构建，然后看下 webpack 是否处理了我们的字体：

```
npm run build
```

```
...  
Asset           Size    Chunks  
5439466351d432b73fdb518c6ae9654a.woff2 19.5 KiB      [emitted]  
387c65cc923ad19790469cfb5b7cb583.woff  23.4 KiB      [emitted]  
da4574bb234ddc4bb47cbe1ca4b20303.png   3.01 MiB     [emitted]  [b:  
bundle.js        77 KiB      0  [emitted]  
Entrypoint main = bundle.js  
...
```

重新打开 `index.html` 看看我们的 `Hello webpack` 文本显示是否换上了新的字体。如果一切顺利，你应该能看到变化。

加载数据

此外，可以加载的有用资源还有数据，如 JSON 文件，CSV、TSV 和 XML。类似于 NodeJS，JSON 支持实际上是内置的，也就是说 `import Data from './data.json'` 默认将正常运行。要导入 CSV、TSV 和 XML，你可以使用 `csv-loader` 和 `xml-loader`。让我们处理加载这三类文件：

```
npm install --save-dev csv-loader xml-loader
```

webpack.config.js

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

```

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\.(png|svg|jpg|gif)$/,
        use: [
          'file-loader'
        ]
      },
      {
        test: /\.(woff|woff2|eot|ttf|otf)$/,
        use: [
          'file-loader'
        ]
      },
      {
        test: /\.csv$/,
        use: [
          'csv-loader'
        ]
      },
      {
        test: /\.xml$/,
        use: [
          'xml-loader'
        ]
      }
    ]
  }
};

```

在项目中添加一些数据文件:

project

```

webpack-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
+  |- data.xml
  |- my-font.woff
  |- my-font.woff2

```

```
| - icon.png  
| - style.css  
| - index.js  
|- /node_modules
```

src/data.xml

```
<?xml version="1.0" encoding="UTF-8"?>  
<note>  
  <to>Mary</to>  
  <from>John</from>  
  <heading>Reminder</heading>  
  <body>Call Cindy on Tuesday</body>  
</note>
```

现在，你可以 import 这四种类型的数据(JSON, CSV, TSV, XML)中的任何一种，所导入的 Data 变量，将包含可直接使用的已解析 JSON:

src/index.js

```
import _ from 'lodash';  
import './style.css';  
import Icon from './icon.png';  
+ import Data from './data.xml';  
  
function component() {  
  var element = document.createElement('div');  
  
  // lodash, 现在通过 script 标签导入  
  element.innerHTML = _.join(['Hello', 'webpack'], ' ');  
  element.classList.add('hello');  
  
  // 将图像添加到我们已经存在的 div 中。  
  var myIcon = new Image();  
  myIcon.src = Icon;  
  
  element.appendChild(myIcon);  
  
+  console.log(Data);  
  
  return element;  
}  
  
document.body.appendChild(component());
```

重新执行 npm run build 命令，然后打开 index.html。查看开发者工具中的控制台，你应该能够看到导入的数据会被打印出来！

在使用 d3 等工具实现某些数据可视化时，这个功能极其有用。可以不用在运行时再去发送一个 ajax 请求获取和解析数据，而是在构建过程中将其提前加载到模块中，以便浏览器加载模块后，直接就可以访问解析过的数据。

全局资源

上述所有内容中最出色之处在于，以这种方式加载资源，你可以以更直观的方式将模块和资源组合在一起。无需依赖于含有全部资源的 `/assets` 目录，而是将资源与代码组合在一起使用。例如，类似这样的结构会非常有用：

```
- |- /assets
+ |- /components
+ |   |- /my-component
+ |   |   |- index.jsx
+ |   |   |- index.css
+ |   |   |- icon.svg
+ |   |   |- img.png
```

这种配置方式会使你的代码更具备可移植性，因为现有的集中放置的方式会让所有资源紧密耦合起来。假如你想在另一个项目中使用 `/my-component`，只需将其复制或移动到 `/components` 目录下。只要你已经安装过全部外部依赖，并且已经在配置中定义过相同的 `loader`，那么项目应该能够良好运行。

但是，假如你只能被局限在旧有开发方式，或者你有一些在多个组件（视图、模板、模块等）之间共享的资源。你仍然可以将这些资源存储在一个基本目录(base directory)中，甚至配合使用 `alias` 来使它们更方便 `import` 导入。

回退处理

对于下篇指南，我们无需使用本指南中所有用到的资源，因此我们会进行一些清理工作，以便为下篇指南 管理输出 做好准备：

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
-   |- data.xml
-   |- my-font.woff
-   |- my-font.woff2
-   |- icon.png
-   |- style.css
  |- index.js
|- /node_modules
```

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
```

```

        filename: 'bundle.js',
        path: path.resolve(__dirname, 'dist')
    },
-   module: {
-     rules: [
-       {
-         test: /\.css$/,
-         use: [
-           'style-loader',
-           'css-loader'
-         ]
-       },
-       {
-         test: /\.(png|svg|jpg|gif)$/,
-         use: [
-           'file-loader'
-         ]
-       },
-       {
-         test: /\.(woff|woff2|eot|ttf|otf)$/,
-         use: [
-           'file-loader'
-         ]
-       },
-       {
-         test: /\.csv$/,
-         use: [
-           'csv-loader'
-         ]
-       },
-       {
-         test: /\.xml$/,
-         use: [
-           'xml-loader'
-         ]
-       }
-     ]
-   }
- };

```

src/index.js

```

import _ from 'lodash';
- import './style.css';
- import Icon from './icon.png';
- import Data from './data.xml';

function component() {
  var element = document.createElement('div');

  // lodash, 现在通过 script 标签导入
  element.innerHTML = _.join(['Hello', 'webpack'], ' ');
  element.classList.add('hello');

  // 将图像添加到我们已经存在的 div 中。
  var myIcon = new Image();
  myIcon.src = Icon;

```

```
-     element.appendChild(myIcon);
-
-     console.log(Data);
-
     return element;
}

document.body.appendChild(component());
```

下篇指南

我们继续移步到 [管理输出](#)

延伸阅读

- [Loading Fonts on SurviveJS](#)

管理输出

本指南继续沿用 管理资源 指南中的代码示例。

到目前为止，我们都是在 `index.html` 文件中手动引入所有资源，然而随着应用程序增长，并且一旦开始 在文件名中使用 hash] 并输出 多个 bundle，如果继续手动管理 `index.html` 文件，就会变得困难起来。然而，通过一些插件可以使这个过程更容易管控。

预先准备

首先，调整一下我们的项目：

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
+  |- print.js
|- /node_modules
```

我们在 `src/print.js` 文件中添加一些逻辑：

src/print.js

```
export default function printMe() {
  console.log('I get called from print.js!');
}
```

并且在 `src/index.js` 文件中使用这个函数：

src/index.js

```
import _ from 'lodash';
+ import printMe from './print.js';

function component() {
  var element = document.createElement('div');
+  var btn = document.createElement('button');

  element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+  btn.innerHTML = '点击这里，然后查看 console!';
+  btn.onclick = printMe;
+
+  element.appendChild(btn);
```

```
        return element;
    }

document.body.appendChild(component());
```

还要更新 `dist/index.html` 文件，来为 webpack 分离入口做好准备：

dist/index.html

```
<!doctype html>
<html>
  <head>
-   <title>管理资源</title>
+   <title>管理输出</title>
+   <script src="./print.bundle.js"></script>
  </head>
  <body>
-   <script src="./bundle.js"></script>
+   <script src="./app.bundle.js"></script>
  </body>
</html>
```

现在调整配置。我们将在 `entry` 添加 `src/print.js` 作为新的入口起点 (`print`)，然后修改 `output`，以便根据入口起点定义的名称，动态地产生 `bundle` 名称：

webpack.config.js

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

module.exports = {
-   entry: './src/index.js',
+   entry: {
+     app: './src/index.js',
+     print: './src/print.js'
+   },
+   output: {
-     filename: 'bundle.js',
+     filename: '[name].bundle.js',
       path: path.resolve(__dirname, 'dist')
   }
};
```

执行 `npm run build`，然后看到生成如下：

```
...
      Asset      Size  Chunks      Chunk Names
app.bundle.js  545 kB     0, 1  [emitted]  [big]  app
print.bundle.js  2.74 kB           1  [emitted]          print
...
```

我们可以看到，webpack 生成 `print.bundle.js` 和 `app.bundle.js` 文件，这也和我们在 `index.html` 文件中指定的文件名称相对应。如果你在浏览器中打开

`index.html`, 就可以看到在点击按钮时会发生什么。

但是, 如果我们更改了我们的一个入口起点的名称, 甚至添加了一个新的入口, 会发生什么? 会在构建时重新命名生成的 bundle, 但是我们的 `index.html` 文件仍然引用旧的名称。让我们用 [HtmlWebpackPlugin](#) 来解决这个问题。

设置 HtmlWebpackPlugin

首先安装插件, 并且调整 `webpack.config.js` 文件:

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

webpack.config.js

```
const path = require('path');
+ const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: {
    app: './src/index.js',
    print: './src/print.js'
  },
+  plugins: [
+    new HtmlWebpackPlugin({
+      title: '管理输出'
+    })
+  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

在我们构建之前, 你应该了解, 虽然在 `dist/` 文件夹我们已经有了 `index.html` 这个文件, 然而 `HtmlWebpackPlugin` 还是会默认生成它自己的 `index.html` 文件。也就是说, 它会用新生成的 `index.html` 文件, 替换我们的原有文件。我们看下执行 `npm run build` 后会发生什么:

```
...
          Asset      Size  Chunks      Chunk Names
print.bundle.js    544 kB      0  [emitted]  [big]  print
  app.bundle.js    2.81 kB      1  [emitted]
  index.html    249 bytes      [emitted]
...
...
```

如果在代码编辑器中打开 `index.html`, 你会看到 `HtmlWebpackPlugin` 创建了一个全新的文件, 所有的 bundle 会自动添加到 html 中。

如果你想要了解 `HtmlWebpackPlugin` 插件提供的全部的功能和选项, 你就应该阅读 [HtmlWebpackPlugin](#) 仓库中的源码。

还可以看下 [html-webpack-template](#)，除了提供默认模板之外，还提供了一些额外的功能。

清理 /dist 文件夹

你可能已经注意到，由于遗留了之前的指南和代码示例，我们的 /dist 文件夹显得相当杂乱。webpack 将生成文件并放置在 /dist 文件夹中，但是它不会追踪哪些文件是实际在项目中用到的。

通常比较推荐的做法是，在每次构建前清理 /dist 文件夹，这样只会生成用到的文件。让我们实现这个需求。

[clean-webpack-plugin](#) 是一个流行的清理插件，安装和配置它。

```
npm install --save-dev clean-webpack-plugin
```

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
+ const CleanWebpackPlugin = require('clean-webpack-plugin');

module.exports = {
  entry: {
    app: './src/index.js',
    print: './src/print.js'
  },
  plugins: [
+   new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: '管理输出'
    })
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

现在，执行 `npm run build`，检查 /dist 文件夹。如果一切顺利，现在只会看到构建后生成的文件，而没有旧文件！

manifest

你可能会很感兴趣，webpack 和 webpack 插件似乎“知道”应该哪些文件生成。答案是，webpack 通过 manifest，可以追踪所有模块到输出 bundle 之间的映射。如果你想要知道如何以其他方式来控制 webpack 输出，了解 manifest 是个好的开始。

通过 [WebpackManifestPlugin](#) 插件，可以将 manifest 数据提取为一个容易使用的 json 文件。

我们不会在此展示一个如何在项目中使用此插件的完整示例，你可以在 [manifest](#) 概念页面深入阅读，以及在 [缓存](#) 指南中，了解它与长效缓存有何关系。

结论

现在，你已经了解如何向 HTML 动态添加 bundle，让我们深入 [开发环境](#) 指南。或者如果你想要深入更多相关高级话题，我们推荐你前往 [代码分离](#) 指南。

开发环境

本指南继续沿用 [管理输出](#) 指南中的代码示例。

如果你一直跟随之前的指南，应该对一些 webpack 基础知识有着很扎实的理解。在我们继续之前，先来看看如何设置一个开发环境，使我们的开发体验变得更轻松一些。

本指南中的工具仅用于开发环境，请不要在生产环境中使用它们！

在开始前，我们先将 `mode` 设置为 `'development'`。

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');

module.exports = {
+  mode: 'development',
  entry: {
    app: './src/index.js',
    print: './src/print.js'
  },
  plugins: [
    new CleanWebpackPlugin(['dist']),
    new HtmlWebpackPlugin({
      title: '开发环境'
    })
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

使用 source map

当 webpack 打包源代码时，可能会很难追踪到 error(错误) 和 warning(警告) 在源代码中的原始位置。例如，如果将三个源文件 (`a.js`, `b.js` 和 `c.js`) 打包到一个 bundle (`bundle.js`) 中，而其中一个源文件包含一个错误，那么堆栈跟踪就会直接指向到 `bundle.js`。你可能需要准确地知道错误来自于哪个源文件，所以这种提示这通常不会提供太多帮助。

为了更容易地追踪 error 和 warning，JavaScript 提供了 [source map](#) 功能，可以将编译后的代码映射回原始源代码。如果一个错误来自于 `b.js`，source map 就会明确的告诉你。

source map 有许多 可用选项， 请务必仔细阅读它们，以便可以根据需要进行配置。

对于本指南，我们将使用 `inline-source-map` 选项，这有助于解释说明示例意图（此配置仅用于示例，不要用于生产环境）：

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');

module.exports = {
  mode: 'development',
  entry: {
    app: './src/index.js',
    print: './src/print.js'
  },
+  devtool: 'inline-source-map',
  plugins: [
    new CleanWebpackPlugin(['dist']),
    new HtmlWebpackPlugin({
      title: 'Development'
    })
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

现在，让我们来做一些调试，在 `print.js` 文件中生成一个错误：

src/print.js

```
- export default function printMe() {
-   console.log('I get called from print.js!');
+   cosnole.log('I get called from print.js!');
}
```

运行 `npm run build`，编译如下：

```
...
          Asset      Size  Chunks      Chunk Names
app.bundle.js  1.44 MB  0, 1  [emitted]  [big]  app
print.bundle.js  6.43 kB       1  [emitted]
index.html    248 bytes           [emitted]
...
```

现在，在浏览器中打开生成的 `index.html` 文件，点击按钮，并且在控制台查看显示的错误。错误应该如下：

```
Uncaught ReferenceError: cosnole is not defined
```

```
at HTMLButtonElement.printMe (print.js:2)
```

我们可以看到，此错误包含有发生错误的文件（`print.js`）和行号（2）的引用。这是非常有帮助的，因为现在我们可以确切地知道，所要解决问题的位置。

选择一个开发工具

某些文本编辑器具有 "safe write(安全写入)" 功能，可能会干扰下面一些工具。阅读 [调整文本编辑器](#) 以解决这些问题。

在每次编译代码时，手动运行 `npm run build` 会显得很麻烦。

`webpack` 提供几种可选方式，帮助你在代码发生变化后自动编译代码：

1. `webpack` watch mode(`webpack` 观察模式)
2. `webpack-dev-server`
3. `webpack-dev-middleware`

多数场景中，你可能需要使用 `webpack-dev-server`，但是不妨探讨一下以上的所有选项。

使用 watch mode(观察模式)

你可以指示 `webpack` "watch" 依赖图中所有文件的更改。如果其中一个文件被更新，代码将被重新编译，所以你不必再去手动运行整个构建。

我们添加一个用于启动 `webpack` watch mode 的 `npm scripts`：

package.json

```
{  
  "name": "development",  
  "version": "1.0.0",  
  "description": "",  
  "main": "webpack.config.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1",  
+    "watch": "webpack --watch",  
    "build": "webpack"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "clean-webpack-plugin": "^0.1.16",  
    "css-loader": "^0.28.4",  
    "csv-loader": "^2.1.1",  
    "file-loader": "^0.11.2",  
    "html-webpack-plugin": "^2.29.0",  
  }  
}
```

```
"style-loader": "^0.18.2",
"webpack": "^3.0.0",
"xml-loader": "^1.2.1"
}
}
```

现在，你可以在命令行中运行 `npm run watch`，然后就会看到 webpack 是如何编译代码。然而，你会发现并没有退出命令行。这是因为此 script 当前还在 watch 你的文件。

现在，webpack 观察文件的同时，先移除我们之前加入的错误：

src/print.js

```
- export default function printMe() {
-   cosnole.log('I get called from print.js!');
+   console.log('I get called from print.js!');
}
```

现在，保存文件并检查 terminal(终端) 窗口。应该可以看到 webpack 自动地重新编译修改后的模块！

唯一的缺点是，为了看到修改后的实际效果，你需要刷新浏览器。如果能够自动刷新浏览器就更好了，因此接下来我们会尝试通过 `webpack-dev-server` 实现此功能。

使用 webpack-dev-server

`webpack-dev-server` 为你提供了一个简单的 web server，并且具有 live reloading(实时重新加载) 功能。设置如下：

```
npm install --save-dev webpack-dev-server
```

修改配置文件，告诉 dev server，从什么位置查找文件：

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');

module.exports = {
  mode: 'development',
  entry: {
    app: './src/index.js',
    print: './src/print.js'
  },
  devtool: 'inline-source-map',
+  devServer: {
+   contentBase: './dist'
+ },
}
```

```

plugins: [
  new CleanWebpackPlugin(['dist']),
  new HtmlWebpackPlugin({
    title: 'Development'
  })
],
output: {
  filename: '[name].bundle.js',
  path: path.resolve(__dirname, 'dist')
}
};

```

以上配置告知 webpack-dev-server，将 dist 目录下的文件 serve 到 localhost:8080 下。（译注：serve，将资源作为 server 的可访问文件）

webpack-dev-server 在编译之后不会写入到任何输出文件。而是将 bundle 文件保留在内存中，然后将它们 serve 到 server 中，就好像它们是挂载在 server 根路径上的真实文件一样。如果你的页面希望在其他不同路径中找到 bundle 文件，则可以通过 dev server 配置中的 `publicPath` 选项进行修改。

我们添加一个可以直接运行 dev server 的 script:

package.json

```

{
  "name": "development",
  "version": "1.0.0",
  "description": "",
  "main": "webpack.config.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\" && exit 1",
    "watch": "webpack --watch",
+   "start": "webpack-dev-server --open",
    "build": "webpack"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "clean-webpack-plugin": "^0.1.16",
    "css-loader": "^0.28.4",
    "csv-loader": "^2.1.1",
    "file-loader": "^0.11.2",
    "html-webpack-plugin": "^2.29.0",
    "style-loader": "^0.18.2",
    "webpack": "^3.0.0",
    "xml-loader": "^1.2.1"
  }
}

```

现在，在命令行中运行 `npm start`，我们会看到浏览器自动加载页面。如果你更改任何源文件并保存它们，web server 将在编译代码后自动重新加载。试试看！

webpack-dev-server 具有许多可配置的选项。关于其他更多配置，请查看 [配置文档](#)。

现在，server 正在运行，你可能需要尝试 [模块热替换\(hot module replacement\)](#)！

使用 webpack-dev-middleware

webpack-dev-middleware 是一个封装器(wrapper)，它可以把 webpack 处理过的文件发送到一个 server。webpack-dev-server 在内部使用了它，然而它也可以作为一个单独的 package 来使用，以便根据需求进行更多自定义设置。下面是一个 webpack-dev-middleware 配合 express server 的示例。

首先，安装 express 和 webpack-dev-middleware：

```
npm install --save-dev express webpack-dev-middleware
```

现在，我们需要调整 webpack 配置文件，以确保 middleware(中间件) 功能能够正确启用：

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');

module.exports = {
  mode: 'development',
  entry: {
    app: './src/index.js',
    print: './src/print.js'
  },
  devtool: 'inline-source-map',
  devServer: {
    contentBase: './dist'
  },
  plugins: [
    new CleanWebpackPlugin(['dist']),
    new HtmlWebpackPlugin({
      title: '管理输出'
    })
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist'),
+   publicPath: '/'
  }
};
```

我们将会在 server 脚本使用 `publicPath`，以确保文件资源能够正确地 serve 在 `http://localhost:3000` 下，稍后我们会指定 port number(端口号)。接下来是设

置自定义 express server:

project

```
webpack-demo
|- package.json
|- webpack.config.js
+ |- server.js
|- /dist
|- /src
  |- index.js
  |- print.js
|- /node_modules
```

server.js

```
const express = require('express');
const webpack = require('webpack');
const webpackDevMiddleware = require('webpack-dev-middleware');

const app = express();
const config = require('./webpack.config.js');
const compiler = webpack(config);

// 告诉 express 使用 webpack-dev-middleware,
// 以及将 webpack.config.js 配置文件作为基础配置
app.use(webpackDevMiddleware(compiler, {
  publicPath: config.output.publicPath
}));

// 将文件 serve 到 port 3000。
app.listen(3000, function () {
  console.log('Example app listening on port 3000!\n');
});
```

现在，添加一个 npm script，以使我们更方便地运行服务：

package.json

```
{
  "name": "development",
  "version": "1.0.0",
  "description": "",
  "main": "webpack.config.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "watch": "webpack --watch",
    "start": "webpack-dev-server --open",
+   "server": "node server.js",
    "build": "webpack"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
```

```
"clean-webpack-plugin": "^0.1.16",
"css-loader": "^0.28.4",
"csv-loader": "^2.1.1",
"express": "^4.15.3",
"file-loader": "^0.11.2",
"html-webpack-plugin": "^2.29.0",
"style-loader": "^0.18.2",
"webpack": "^3.0.0",
"webpack-dev-middleware": "^1.12.0",
"xml-loader": "^1.2.1"
}
}
```

现在，在 terminal(终端) 中执行 `npm run server`，将会有类似如下信息输出：

```
Example app listening on port 3000!
```

```
...
Asset          Size  Chunks      Chunk Names
app.bundle.js  1.44 MB  0, 1    [emitted]  [big]  app
print.bundle.js 6.57 kB   1    [emitted]    print
index.html    306 bytes           [emitted]
...
webpack: Compiled successfully.
```

现在，打开浏览器，访问 `http://localhost:3000`。应该看到webpack 应用程序已经运行！

如果想要了解更多关于模块热替换(hot module replacement)的运行机制，我们推荐你查看 [模块热替换\(hot module replacement\)](#) 指南。

调整文本编辑器

使用自动编译代码时，可能会在保存文件时遇到一些问题。某些编辑器具有 "safe write(安全写入)" 功能，会影响重新编译。

在一些常见的编辑器中禁用此功能，查看以下列表：

- **Sublime Text 3:** 在用户首选项(user preferences)中添加 `atomic_save: 'false'`。
- **JetBrains IDEs (e.g. WebStorm):** 在 `Preferences > Appearance & Behavior > System Settings` 中取消选中 "Use safe write"。
- **Vim:** 在设置(settings)中增加 `:set backupcopy=yes`。

结论

现在，你已经学会了如何自动编译代码，并运行一个简单的 development server，查看下一个指南，其中将介绍 [模块热替换\(hot module replacement\)](#)。

模块热替换

本指南继续沿用 [开发环境](#) 指南中的代码示例。

模块热替换(hot module replacement 或 HMR)是 webpack 提供的最有用的功能之一。它允许在运行时更新所有类型的模块，而无需完全刷新。本页面重点介绍其实现，而 [概念](#) 页面提供了更多关于它的工作原理以及为什么它有用 的细节。

HMR 不适用于生产环境，这意味着它应当用于开发环境。更多详细信息，请查看 [生产环境](#) 指南。

启用 HMR

此功能可以很大程度提高生产效率。我们要做的就是更新 `webpack-dev-server` 配置，然后使用 webpack 内置的 HMR 插件。我们还要删除掉 `print.js` 的入口起点，因为现在已经在 `index.js` 模块中引用了它。

如果你在技术选型中使用了 `webpack-dev-middleware` 而没有使用 `webpack-dev-server`，请使用 `webpack-hot-middleware` package，以在你的自定义 server 或应用程序上启用 HMR。

`webpack.config.js`

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');
+ const webpack = require('webpack');

module.exports = {
  entry: {
-   app: './src/index.js',
-   print: './src/print.js'
+   app: './src/index.js'
  },
  devtool: 'inline-source-map',
  devServer: {
    contentBase: './dist',
+   hot: true
  },
  plugins: [
    new CleanWebpackPlugin(['dist']),
    new HtmlWebpackPlugin({
      title: '模块热替换'
    }),
+   new webpack.HotModuleReplacementPlugin()
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
```

```
    }  
};
```

可以通过命令来修改 `webpack-dev-server` 的配置: `webpack-dev-server --hotOnly`。

现在, 修改 `index.js` 文件, 以便在 `print.js` 内部发生变更时, 告诉 webpack 接受 updated module。

index.js

```
import _ from 'lodash';  
import printMe from './print.js';  
  
function component() {  
  var element = document.createElement('div');  
  var btn = document.createElement('button');  
  
  element.innerHTML = _.join(['Hello', 'webpack'], ' ');  
  
  btn.innerHTML = 'Click me and check the console!';  
  btn.onclick = printMe;  
  
  element.appendChild(btn);  
  
  return element;  
}  
  
document.body.appendChild(component());  
+  
+ if (module.hot) {  
+   module.hot.accept('./print.js', function() {  
+     console.log('Accepting the updated printMe module!');  
+     printMe();  
+   })  
+ }
```

修改 `print.js` 中 `console.log` 语句, 你将会在浏览器中看到如下的输出 (暂时不要担心 `button.onclick = printMe()` 输出, 我们稍后也会更新这部分)。

print.js

```
export default function printMe() {  
-   console.log('I get called from print.js!');  
+   console.log('Updating print.js...')  
}
```

console

```
[HMR] Waiting for update signal from WDS...  
main.js:4395 [WDS] Hot Module Replacement enabled.  
+ 2main.js:4395 [WDS] App updated. Recompiling...  
+ main.js:4395 [WDS] App hot update...  
+ main.js:4330 [HMR] Checking for updates on the server...
```

```
+ main.js:10024 Accepting the updated printMe module!
+ 0.4b8ee77....hot-update.js:10 Updating print.js...
+ main.js:4330 [HMR] Updated modules:
+ main.js:4330 [HMR] - 20
```

通过 Node.js API

在 Node.js API 中使用 webpack dev server 时，不要将 dev server 选项放在 webpack 配置对象(webpack config object)中。而是，在创建时，将其作为第二个参数传递。例如：

```
new WebpackDevServer(compiler, options)
```

想要启用 HMR，还需要修改 webpack 配置对象，使其包含 HMR 入口起点。webpack-dev-server package 中具有一个叫做 `addDevServerEntrypoints` 的方法，你可以通过使用这个方法来实现。这是关于如何使用的一个基本示例：

dev-server.js

```
const webpackDevServer = require('webpack-dev-server');
const webpack = require('webpack');

const config = require('./webpack.config.js');
const options = {
  contentBase: './dist',
  hot: true,
  host: 'localhost'
};

webpackDevServer.addDevServerEntrypoints(config, options);
const compiler = webpack(config);
const server = new webpackDevServer(compiler, options);

server.listen(5000, 'localhost', () => {
  console.log('dev server listening on port 5000');
});
```

如果你正在使用 `webpack-dev-middleware`，可以通过 `webpack-hot-middleware` package，在自定义 dev server 中启用 HMR。

问题

模块热替换可能比较难以掌握。为了说明这一点，我们回到刚才的示例中。如果你继续点击示例页面上的按钮，你会发现控制台仍在打印旧的 `printMe` 函数。

这是因为按钮的 `onclick` 事件处理函数仍然绑定在旧的 `printMe` 函数上。

为了让 HMR 正常工作，我们需要更新代码，使用 `module.hot.accept` 将其绑定到新的 `printMe` 函数上：

index.js

```
import _ from 'lodash';
import printMe from './print.js';

function component() {
  var element = document.createElement('div');
  var btn = document.createElement('button');

  element.innerHTML = _.join(['Hello', 'webpack'], ' ');
  btn.innerHTML = 'Click me and check the console!';
  btn.onclick = printMe; // onclick 事件绑定原始的 printMe 函数上

  element.appendChild(btn);

  return element;
}

document.body.appendChild(component());
+ let element = component(); // 存储 element，以便在 print.js 修改时重新渲染
+ document.body.appendChild(element);

if (module.hot) {
  module.hot.accept('./print.js', function() {
    console.log('Accepting the updated printMe module!');
  - printMe();
  + document.body.removeChild(element);
  + element = component(); // Re-render the "component" to update the
  + element = component(); // 重新渲染 "component"，以便更新 click 事件处理
  + document.body.appendChild(element);
  })
}
}
```

这仅仅是一个示例，还有很多让人易于犯错的情况。幸运的是，有很多 loader（下面会提到一些）可以使得模块热替换变得更加容易。

HMR 加载样式

借助于 `style-loader`，使用模块热替换来加载 CSS 实际上极其简单。此 loader 在幕后使用了 `module.hot.accept`，在 CSS 依赖模块更新之后，会将其 patch(修补) 到 `<style>` 标签中。

首先使用以下命令安装两个 loader：

```
npm install --save-dev style-loader css-loader
```

然后更新配置文件，使用这两个 loader。

webpack.config.js

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

```

const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');
const webpack = require('webpack');

module.exports = {
  entry: {
    app: './src/index.js'
  },
  devtool: 'inline-source-map',
  devServer: {
    contentBase: './dist',
    hot: true
  },
+  module: {
+    rules: [
+      {
+        test: /\.css$/,
+        use: ['style-loader', 'css-loader']
+      }
+    ]
+  },
  plugins: [
    new CleanWebpackPlugin(['dist']),
    new HtmlWebpackPlugin({
      title: '模块热替换'
    }),
    new webpack.HotModuleReplacementPlugin()
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};

```

如同 import 模块，热加载样式表同样很简单：

project

```

webpack-demo
| - package.json
| - webpack.config.js
| - /dist
|   - bundle.js
| - /src
|   - index.js
|   - print.js
+   | - styles.css

```

styles.css

```

body {
  background: blue;
}

```

index.js

```

import _ from 'lodash';
import printMe from './print.js';
+ import './styles.css';

function component() {
  var element = document.createElement('div');
  var btn = document.createElement('button');

  element.innerHTML = _.join(['Hello', 'webpack'], ' ');
  btn.innerHTML = 'Click me and check the console!';
  btn.onclick = printMe; // onclick event is bind to the original pr:
  element.appendChild(btn);

  return element;
}

let element = component();
document.body.appendChild(element);

if (module.hot) {
  module.hot.accept('./print.js', function() {
    console.log('Accepting the updated printMe module!');
    document.body.removeChild(element);
    element = component(); // Re-render the "component" to update the
    document.body.appendChild(element);
  })
}

```

将 body 的 style 改为 background: red;，你应该可以立即看到页面的背景颜色随之更改，而无需完全刷新。

styles.css

```

body {
-   background: blue;
+   background: red;
}

```

其他代码和框架

社区还提供许多其他 loader 和示例，可以使 HMR 与各种框架和库平滑地进行交互.....

- [React Hot Loader](#): 实时调整 react 组件。
- [Vue Loader](#): 此 loader 支持 vue 组件的 HMR，提供开箱即用体验。
- [Elm Hot Loader](#): 支持 Elm 编程语言的 HMR。
- [Angular HMR](#): 没有必要使用 loader！直接修改 NgModule 主文件就够了，它可以完全控制 HMR API。

如果你知道任何其他 loader 或 plugin，能够有助于或增强模块热替换(hot module replacement)，请提交一个 pull request 以添加到此列表中！

tree shaking

tree shaking 是一个术语，通常用于描述移除 JavaScript 上下文中的未引用代码 (dead-code)。它依赖于 ES2015 模块语法的 静态结构 特性，例如 `import` 和 `export`。这个术语和概念实际上是由 ES2015 模块打包工具 `rollup` 普及起来的。

webpack 2 正式版本内置支持 ES2015 模块（也叫做 *harmony modules*）和未使用模块检测能力。新的 webpack 4 正式版本扩展了此检测能力，通过 `package.json` 的 `"sideEffects"` 属性作为标记，向 `compiler` 提供提示，表明项目中的哪些文件是 "pure(纯的 ES2015 模块)"，由此可以安全地删除文件中未使用的部分。

本指南的继承自 [起步指南](#)。如果你尚未阅读该指南，请先行阅读。

添加一个通用模块

在我们的项目中添加一个新的通用模块文件 `src/math.js`，并导出两个函数：

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
  |- index.js
+ |- math.js
|- /node_modules
```

src/math.js

```
export function square(x) {
  return x * x;
}

export function cube(x) {
  return x * x * x;
}
```

将 `mode` 配置选项设置为 `development` 以确保 `bundle` 是未压缩版本：

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
```

```

filename: 'bundle.js',
path: path.resolve(__dirname, 'dist')
- }
+ },
+ mode: 'development',
+ optimization: {
+   usedExports: true
+ }
};


```

配置完这些后，更新入口脚本，使用其中一个新方法，并且为了简化示例，我们先将 `lodash` 删除：

src/index.js

```

- import _ from 'lodash';
+ import { cube } from './math.js';

function component() {
-   var element = document.createElement('div');
+   var element = document.createElement('pre');

-   // lodash 是由当前 script 脚本 import 进来的
-   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+   element.innerHTML = [
+     'Hello webpack!',
+     '5 cubed is equal to ' + cube(5)
+   ].join('\n\n');

   return element;
}

document.body.appendChild(component());

```

注意，我们没有从 `src/math.js` 模块中 `import` 另外一个 `square` 方法。这个函数就是所谓的“未引用代码(dead code)”，也就是说，应该删除掉未被引用的 `export`。现在运行 `npm script npm run build`，并查看输出的 `bundle`：

dist/bundle.js (around lines 90 - 100)

```

/* 1 */
/**/(function(module, __webpack_exports__, __webpack_require__) {
'use strict';
/* unused harmony export square */
/* harmony export (immutable) */ __webpack_exports__['a'] = cube;
function square(x) {
  return x * x;
}

function cube(x) {
  return x * x * x;
}
});
```

注意，上面的 `unused harmony export square` 注释。如果你观察它下面的代码，你会注意到虽然我们没有引用 `square`，但它仍然被包含在 bundle 中。我们将在下一节解决这个问题。

将文件标记为 side-effect-free(无副作用)

在一个纯粹的 ESM 模块世界中，很容易识别出哪些文件有 side effect。然而，我们的项目无法达到这种纯度，所以，此时有必要提示 webpack compiler 哪些代码是“纯粹部分”。

通过 `package.json` 的 `"sideEffects"` 属性，来实现这种方式。

```
{  
  "name": "your-project",  
  "sideEffects": false  
}
```

如果所有代码都不包含 side effect，我们就可以简单地将该属性标记为 `false`，来告知 webpack，它可以安全地删除未用到的 `export`。

"side effect(副作用)" 的定义是，在导入时会执行特殊行为的代码，而不是仅仅暴露一个 `export` 或多个 `export`。举例说明，例如 polyfill，它影响全局作用域，并且通常不提供 `export`。

如果你的代码确实有一些副作用，可以改为提供一个数组：

```
{  
  "name": "your-project",  
  "sideEffects": [  
    "./src/some-side-effectful-file.js"  
  ]  
}
```

数组方式支持相对路径、绝对路径和 glob 模式匹配相关文件。它在内部使用 [micromatch](#)。

注意，所有导入文件都会受到 tree shaking 的影响。这意味着，如果在项目中使用类似 `css-loader` 并 `import` 一个 CSS 文件，则需要将其添加到 side effect 列表中，以免在生产模式中无意中将它删除：

```
{  
  "name": "your-project",  
  "sideEffects": [  
    "./src/some-side-effectful-file.js",  
    "*.*css"  
  ]  
}
```

最后，还可以在 `module.rules` 配置选项中设置 `"sideEffects"`。

压缩输出结果

通过 `import` 和 `export` 语法，我们已经找出需要删除的“未引用代码(dead code)”，然而，不仅仅是找出，还要在 bundle 中删除它们。为此，我们需要将 `mode` 配置选项设置为 `production`。

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
- mode: 'development',
- optimization: {
-   usedExports: true
- }
+ mode: 'production'
};
```

注意，也可以在命令行接口中使用 `--optimize-minimize` 标记，来启用 `TerserPlugin`。

准备就绪后，然后运行另一个 `npm script npm run build`，看看输出结果是否发生改变。

你发现 `dist/bundle.js` 中的差异了吗？显然，现在整个 bundle 都已经被 `minify`(压缩) 和 `mangle`(混淆破坏)，但是如果仔细观察，则不会看到引入 `square` 函数，但能看到 `cube` 函数的混淆破坏版本 (`function r(e){return e*e*e}n.a=r`)。现在，随着 `minification`(代码压缩) 和 `tree shaking`，我们的 bundle 减小几个字节！虽然，在这个特定示例中，可能看起来没有减少很多，但是，在有着复杂依赖树的大型应用程序上运行 `tree shaking` 时，会对 bundle 产生显著的体积优化。

运行 `tree shaking` 需要 `ModuleConcatenationPlugin`。通过 `mode: "production"` 可以添加此插件。如果你没有使用 `mode` 设置，记得手动添加 `ModuleConcatenationPlugin`。

结论

我们已经知道，想要使用 `tree shaking` 必须注意以下……

- 使用 ES2015 模块语法（即 `import` 和 `export`）。
- 确保没有 compiler 将 ES2015 模块语法转换为 CommonJS 模块（这也是流行的）

Babel preset 中 `@babel/preset-env` 的默认行为 - 更多详细信息请查看 [文档](#)）。

- 在项目 `package.json` 文件中，添加一个 "sideEffects" 属性。
- 通过将 `mode` 选项设置为 `production`，启用 minification(代码压缩) 和 tree shaking。

你可以将应用程序想象成一棵树。绿色表示实际用到的 source code(源码) 和 library(库)，是树上活的树叶。灰色表示未引用代码，是秋天树上枯萎的树叶。为了除去死去的树叶，你必须摇动这棵树，使它们落下。

如果你对优化输出很感兴趣，请进入到下个指南，来了解 [生产环境 构建](#) 的详细细节。

生产环境

在本指南中，我们将深入一些最佳实践和工具，将站点或应用程序构建到生产环境中。

以下示例来源于 [tree shaking](#) 和 [开发环境](#)。在继续之前，请确保你已经熟悉这些指南中所介绍的概念/配置。

配置

development(开发环境) 和 *production(生产环境)* 这两个环境下的构建目标存在着巨大差异。在开发环境中，我们需要：强大的 source map 和一个有着 live reloading(实时重新加载) 或 hot module replacement(热模块替换) 能力的 localhost server。而生产环境目标则转移至其他方面，关注点在于压缩 bundle、更轻量的 source map、资源优化等，通过这些优化方式改善加载时间。由于要遵循逻辑分离，我们通常建议为每个环境编写彼此独立的 webpack 配置。

虽然，以上我们将生产环境和开发环境做了略微区分，但是，请注意，我们还是会遵循不重复原则(Don't repeat yourself - DRY)，保留一个 "common(通用)" 配置。为了将这些配置合并在一起，我们将使用一个名为 [webpack-merge](#) 的工具。此工具会引用 "common" 配置，因此我们不必再在环境特定(environment-specific)的配置中编写重复代码。

我们先从安装 [webpack-merge](#) 开始，并将之前指南中已经成型的那些代码进行分离：

```
npm install --save-dev webpack-merge
```

project

```
webpack-demo
|- package.json
- |- webpack.config.js
+ |- webpack.common.js
+ |- webpack.dev.js
+ |- webpack.prod.js
|- /dist
|- /src
  |- index.js
  |- math.js
|- /node_modules
```

webpack.common.js

```
+ const path = require('path');
+ const CleanWebpackPlugin = require('clean-webpack-plugin');
+ const HtmlWebpackPlugin = require('html-webpack-plugin');
```

```
+  
+ module.exports = {  
+   entry: {  
+     app: './src/index.js'  
+   },  
+   plugins: [  
+     new CleanWebpackPlugin(['dist']),  
+     new HtmlWebpackPlugin({  
+       title: 'Production'  
+     })  
+   ],  
+   output: {  
+     filename: '[name].bundle.js',  
+     path: path.resolve(__dirname, 'dist')  
+   }  
+};
```

webpack.dev.js

```
+ const merge = require('webpack-merge');  
+ const common = require('./webpack.common.js');  
+  
+ module.exports = merge(common, {  
+   mode: 'development',  
+   devtool: 'inline-source-map',  
+   devServer: {  
+     contentBase: './dist'  
+   }  
+});
```

webpack.prod.js

```
+ const merge = require('webpack-merge');  
+ const common = require('./webpack.common.js');  
+  
+ module.exports = merge(common, {  
+   mode: 'production',  
+ });
```

现在，在 `webpack.common.js` 中，我们设置了 `entry` 和 `output` 配置，并且在其中引入这两个环境公用的全部插件。在 `webpack.dev.js` 中，我们将 `mode` 设置为 `development`，并且为此环境添加了推荐的 `devtool`（强大的 source map）和简单的 `devServer` 配置。最后，在 `webpack.prod.js` 中，我们将 `mode` 设置为 `production`，其中会引入之前在 `tree shaking` 指南中介绍过的 `TerserPlugin`。

注意，在环境特定的配置中使用 `merge()` 功能，可以很方便地引用 `dev` 和 `prod` 中公用的 `common` 配置。`webpack-merge` 工具提供了各种 `merge(合并)` 高级功能，但是在我们的用例中，无需用到这些功能。

npm scripts

现在，我们把 `scripts` 重新指向到新配置。让 `npm start` script 中 `webpack-dev-`

server 使用 *development*(开发环境) 配置文件，而让 `npm run build` script 使用 *production*(生产环境) 配置文件：

package.json

```
{  
  "name": "development",  
  "version": "1.0.0",  
  "description": "",  
  "main": "src/index.js",  
  "scripts": {  
    - "start": "webpack-dev-server --open",  
    + "start": "webpack-dev-server --open --config webpack.dev.js",  
    - "build": "webpack"  
    + "build": "webpack --config webpack.prod.js"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "clean-webpack-plugin": "^0.1.17",  
    "css-loader": "^0.28.4",  
    "csv-loader": "^2.1.1",  
    "express": "^4.15.3",  
    "file-loader": "^0.11.2",  
    "html-webpack-plugin": "^2.29.0",  
    "style-loader": "^0.18.2",  
    "webpack": "^3.0.0",  
    "webpack-dev-middleware": "^1.12.0",  
    "webpack-dev-server": "^2.9.1",  
    "webpack-merge": "^4.1.0",  
    "xml-loader": "^1.2.1"  
  }  
}
```

随便运行下这些脚本，然后查看输出结果的变化，然后我们会继续添加一些生产环境配置。

指定 mode

许多 library 通过与 `process.env.NODE_ENV` 环境变量关联，以决定 library 中应该引用哪些内容。例如，当不处于生产环境中时，某些 library 为了使调试变得容易，可能会添加额外的 log(日志记录) 和 test(测试) 功能。并且，在使用 `process.env.NODE_ENV === 'production'` 时，一些 library 可能针对具体用户的环境，删除或添加一些重要代码，以进行代码执行方面的优化。从 webpack v4 开始，指定 `mode` 会自动地配置 `DefinePlugin`：

webpack.prod.js

```
const merge = require('webpack-merge');  
const common = require('./webpack.common.js');
```

```
module.exports = merge(common, {
  mode: 'production',
});
```

技术上讲，`NODE_ENV` 是一个由 Node.js 暴露给执行脚本的系统环境变量。通常用于决定在开发环境与生产环境(dev-vs-prod)下，server tools(服务期工具)、build scripts(构建脚本) 和 client-side libraries(客户端库) 的行为。然而，与预期相反，无法在构建脚本 `webpack.config.js` 中，将 `process.env.NODE_ENV` 设置为 "production"，请查看 #2537。因此，在 `webpack` 配置文件中，`process.env.NODE_ENV === 'production'` ? '`[name].[hash].bundle.js' : '[name].bundle.js'` 这样的条件语句，无法按照预期运行。

如果你正在使用像 `react` 这样的 library，那么在添加此 `DefinePlugin` 插件后，你应该看到 bundle 大小显著下降。还要注意，任何位于 `/src` 的本地代码都可以关联到 `process.env.NODE_ENV` 环境变量，所以以下检查也是有效的：

src/index.js

```
import { cube } from './math.js';
+
+ if (process.env.NODE_ENV !== 'production') {
+   console.log('Looks like we are in development mode!');
+ }

function component() {
  var element = document.createElement('pre');

  element.innerHTML = [
    'Hello webpack!',
    '5 cubed is equal to ' + cube(5)
  ].join('\n\n');

  return element;
}

document.body.appendChild(component());
```

minification(压缩)

设置 `production mode` 配置后，`webpack v4+` 会默认压缩你的代码。

注意，虽然生产环境下默认使用 `TerserPlugin`，并且也是代码压缩方面比较好的选择，但是还有一些其他可选择项。以下有几个同样很受欢迎的插件：

- [BabelMinifyWebpackPlugin](#)
- [ClosureCompilerPlugin](#)

如果决定尝试一些其他压缩插件，只要确保新插件也会按照 [tree shake](#) 指南中所陈述的具有删除未引用代码(dead code)的能力，以及提供 [optimization.minimizer](#)。

source mapping(源码映射)

我们鼓励你在生产环境中启用 source map，因为它们对 debug(调试源码) 和运行 benchmark tests(基准测试) 很有帮助。虽然有着如此强大的功能，然而还是应该针对生产环境用途，选择一个可以快速构建的推荐配置（更多选项请查看 [devtool](#)）。对于本指南，我们将在生产环境中使用 `source-map` 选项，而不是我们在开发环境中用到的 `inline-source-map`：

webpack.prod.js

```
const merge = require('webpack-merge');
const common = require('./webpack.common.js');

module.exports = merge(common, {
  mode: 'production',
+  devtool: 'source-map'
});
```

避免在生产中使用 `inline-***` 和 `eval-***`，因为它们会增加 bundle 体积大小，并降低整体性能。

最小化 CSS

将生产环境下的 CSS 进行压缩会非常重要，请查看 [在生产环境下压缩](#) 章节。

CLI 替代选项

以上所述也可以通过命令行实现。例如，`--optimize-minimize` 标记将在幕后引用 `TerserPlugin`。和以上描述的 `DefinePlugin` 实例相同，`--define process.env.NODE_ENV="'production'"` 也会做同样的事情。而且，`webpack -p` 将自动地配置上述这两个标记，从而调用需要引入的插件。

虽然这种简短的方式很好，但通常我们建议只使用配置方式，因为在这两种方式中，配置方式能够更准确地理解现在正在做的事情。配置方式还为可以让你更加细微地控制这两个插件中的其他选项。

代码分离

本指南继续沿用 [起步](#) 和 [管理输出](#) 中的代码示例。请确保你已熟悉这些指南中提供的示例。

代码分离是 webpack 中最引人注目的特性之一。此特性能够把代码分离到不同的 bundle 中，然后可以按需加载或并行加载这些文件。代码分离可以用于获取更小的 bundle，以及控制资源加载优先级，如果使用合理，会极大影响加载时间。

常用的代码分离方法有三种：

- 入口起点：使用 `entry` 配置手动地分离代码。
- 防止重复：使用 `SplitChunksPlugin` 去重和分离 chunk。
- 动态导入：通过模块中的内联函数调用来分离代码。

入口起点(entry points)

这是迄今为止最简单、最直观的分离代码的方式。不过，这种方式手动配置较多，并有一些隐患，我们将会解决这些问题。先来看看如何从 main bundle 中分离 another module(另一个模块)：

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
+ |- another-module.js
|- /node_modules
```

another-module.js

```
import _ from 'lodash';

console.log(
  _.join(['Another', 'module', 'loaded!'], ' ')
);
```

webpack.config.js

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

module.exports = {
  mode: 'development',
  entry: {
    index: './src/index.js',
```

```
+   another: './src/another-module.js'
},
output: {
  filename: '[name].bundle.js',
  path: path.resolve(__dirname, 'dist')
}
};
```

这将生成如下构建结果：

```
...
Asset      Size    Chunks          Chunk Names
another.bundle.js  550 KiB  another  [emitted]  another
index.bundle.js   550 KiB  index   [emitted]  index
Entrypoint index = index.bundle.js
Entrypoint another = another.bundle.js
...
```

正如前面提到的，这种方式存在一些隐患：

- 如果入口 chunk 之间包含一些重复的模块，那些重复模块都会被引入到各个 bundle 中。
- 这种方法不够灵活，并且不能动态地将核心应用程序逻辑中的代码拆分出来。

这两点中的第一点，对我们的示例来说毫无疑问是个严重问题，因为我们在 `./src/index.js` 中也引入过 `lodash`，这样就造成在两个 bundle 中重复引用。我们可以通过使用 [SplitChunksPlugin](#) 插件来移除重复模块。

防止重复(prevent duplication)

[SplitChunksPlugin](#) 插件可以将公共的依赖模块提取到已有的 entry chunk 中，或者提取到一个新生成的 chunk。让我们使用这个插件，将前面示例中重复的 `lodash` 模块去除：

`CommonsChunkPlugin` 已经从 webpack v4（代号 legato）中移除。想要了解最新版本是如何处理 chunk，请查看 [SplitChunksPlugin](#)。

webpack.config.js

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

module.exports = {
  mode: 'development',
  entry: {
    index: './src/index.js',
    another: './src/another-module.js'
  },
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
```

```
  },
+ optimization: {
+   splitChunks: {
+     chunks: 'all'
+   }
+ }
};
```

使用 `optimization.splitChunks` 配置选项，现在可以看到已经从 `index.bundle.js` 和 `another.bundle.js` 中删除了重复的依赖项。需要注意的是，此插件将 `lodash` 这个沉重负担从主 bundle 中移除，然后分离到一个单独的 chunk 中。执行 `npm run build` 查看效果：

```
...
          Asset      Size      Chunks
another.bundle.js  5.95 KiB  another [emitt
           index.bundle.js  5.89 KiB  index [emitt
vendors~another~index.bundle.js  547 KiB  vendors~another~index [emitt
Entrypoint index = vendors~another~index.bundle.js index.bundle.js
Entrypoint another = vendors~another~index.bundle.js another.bundle.js
...
```

以下是由社区提供，一些对于代码分离很有帮助的 plugin 和 loader：

- `mini-css-extract-plugin`: 用于将 CSS 从主应用程序中分离。
- `bundle-loader`: 用于分离代码和延迟加载生成的 bundle。
- `promise-loader`: 类似于 `bundle-loader`，但是使用了 promise API。

动态导入(dynamic imports)

当涉及到动态代码拆分时，webpack 提供了两个类似的技术。第一种，也是推荐选择的方式是，使用符合 ECMAScript 提案的 `import()` 语法来实现动态导入。第二种，则是 webpack 的遗留功能，使用 webpack 特定的 `require.ensure`。让我们先尝试使用第一种……

`import()` 调用会在内部用到 `promises`。如果在旧版本浏览器中使用 `import()`，记得使用一个 polyfill 库（例如 `es6-promise` 或 `promise-polyfill`），来 shim `Promise`。

在开始之前，我们先从配置中移除掉多余的 `entry` 和 `optimization.splitChunks`，因为接下来的演示中并不需要它们：

webpack.config.js

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

module.exports = {
  mode: 'development',
  entry: {
```

```

+     index: './src/index.js'
-     index: './src/index.js',
-     another: './src/another-module.js'
},
output: {
  filename: '[name].bundle.js',
+  chunkFilename: '[name].bundle.js',
  path: path.resolve(__dirname, 'dist')
},
optimization: {
  splitChunks: {
    chunks: 'all'
  }
}
};


```

注意，这里使用了 `chunkFilename`，它决定 non-entry chunk(非入口 chunk) 的名称。关于 `chunkFilename` 更多信息，请查看输出文档。更新我们的项目，移除现在不会用到的文件：

project

```

webpack-demo
|- package.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
- |- another-module.js
|- /node_modules


```

现在，我们不再使用 statically import(静态导入) `lodash`，而是通过 dynamic import(动态导入) 来分离出一个 chunk：

src/index.js

```

- import _ from 'lodash';
-
- function component() {
+ function getComponent() {
-   var element = document.createElement('div');
-
-   // Lodash, now imported by this script
-   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+   return import(/* webpackChunkName: "lodash" */ 'lodash').then(({ de:
+     var element = document.createElement('div');
+
+     element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+
+     return element;
+
+   }).catch(error => 'An error occurred while loading the component');
}
-
- document.body.appendChild(component());

```

```
+ getComponent().then(component => {
+   document.body.appendChild(component);
+ })
```

这里我们需要使用 `default` 的原因是，从 webpack v4 开始，在 import CommonJS 模块时，不会再将导入模块解析为 `module.exports` 的值，而是为 CommonJS 模块创建一个 artificial namespace object(人工命名空间对象)，关于其背后原因的更多信息，请阅读 [webpack 4: import\(\)](#) 和 [CommonJs](#)。

注意，在注释中我们提供了 `webpackChunkName`。这样会将拆分出来的 bundle 命名为 `lodash.bundle.js`，而不是 `[id].bundle.js`。想了解更多关于 `webpackChunkName` 和其他可用选项，请查看 [import\(\)](#) 文档。让我们执行 webpack，看到 `lodash` 分离出一个单独的 bundle：

```
...
      Asset      Size      Chunks      Chunk Nar
index.bundle.js  7.88 KiB  index      [emitted]  index
vendors~lodash.bundle.js  547 KiB  vendors~lodash  [emitted]  vendors~:
Entrypoint index = index.bundle.js
...
...
```

由于 `import()` 会返回一个 promise，因此它可以和 `async` 函数一起使用。但是，需要使用像 Babel 这样的预处理器和 [Syntax Dynamic Import Babel Plugin](#)。下面是如何通过 `async` 函数简化代码：

src/index.js

```
- function getComponent() {
+ async function getComponent() {
-   return import(/* webpackChunkName: "lodash" */ 'lodash').then({ default: _ })
-   var element = document.createElement('div');
-
-   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
-
-   return element;
-
- }).catch(error => 'An error occurred while loading the component');
+ var element = document.createElement('div');
+ const { default: _ } = await import(/* webpackChunkName: "lodash" */ 'lodash');
+
+ element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+
+ return element;
}

getComponent().then(component => {
  document.body.appendChild(component);
});
```

预取/预加载模块(prefetch/preload module)

webpack v4.6.0+ 添加了预取和预加载的支持。

在声明 import 时，使用下面这些内置指令，可以让 webpack 输出 "resource hint(资源提示)"，来告知浏览器：

- prefetch(预取)：将来某些导航下可能需要的资源
- preload(预加载)：当前导航下可能需要资源

下面这个 prefetch 的简单示例中，有一个 `HomePage` 组件，其内部渲染一个 `LoginButton` 组件，然后在点击后按需加载 `LoginModal` 组件。

LoginButton.js

```
//...
import /* webpackPrefetch: true */ 'LoginModal';
```

这会生成 `<link rel="prefetch" href="login-modal-chunk.js">` 并追加到页面头部，指示着浏览器在闲置时间预取 `login-modal-chunk.js` 文件。

只要父 chunk 完成加载，webpack 就会添加 prefetch hint(预取提示)。

与 prefetch 指令相比，preload 指令有许多不同之处：

- preload chunk 会在父 chunk 加载时，以并行方式开始加载。prefetch chunk 会在父 chunk 加载结束后开始加载。
- preload chunk 具有中等优先级，并立即下载。prefetch chunk 在浏览器闲置时下载。
- preload chunk 会在父 chunk 中立即请求，用于当下时刻。prefetch chunk 会用于未来的某个时刻。
- 浏览器支持程度不同。

下面这个简单的 preload 示例中，有一个 `Component`，依赖于一个较大的 library，所以应该将其分离到一个独立的 chunk 中。

我们假想这里的图表组件 `ChartComponent` 组件需要依赖体积巨大的 `ChartingLibrary` 库。它会在渲染时显示一个 `LoadingIndicator`(加载进度条) 组件，然后立即按需导入 `ChartingLibrary`：

ChartComponent.js

```
//...
import /* webpackPreload: true */ 'ChartingLibrary';
```

在页面中使用 `ChartComponent` 时，在请求 `ChartComponent.js` 的同时，还会通过 `<link rel="preload">` 请求 `charting-library-chunk`。假定 `page-chunk` 体积很小，很快就被加载好，页面此时就会显示 `LoadingIndicator`(加载进度条)，等到

`charting-library-chunk` 请求完成，`LoadingIndicator` 组件才消失。启动仅需要很少的加载时间，因为只进行单次往返，而不是两次往返。尤其是在高延迟环境下。

不正确地使用 `webpackPreload` 会有损性能，请谨慎使用。

bundle 分析(bundle analysis)

如果我们以分离代码作为开始，那么就应该以检查模块的输出结果作为结束，对其进行分析是很有用处的。官方提供分析工具是一个好的初始选择。下面是一些可选择的社区支持(community-supported)工具：

- [webpack-chart](#): `webpack stats` 可交互饼图。
- [webpack-visualizer](#): 可视化并分析你的 bundle，检查哪些模块占用空间，哪些可能是重复使用的。
- [webpack-bundle-analyzer](#): 一个 plugin 和 CLI 工具，它将 bundle 内容展示为便捷的、交互式、可缩放的树状图形式。
- [webpack bundle optimize helper](#): 此工具会分析你的 bundle，并为你提供可操作的改进措施建议，以减少 bundle 体积大小。

下一步

接下来，查看 [延迟加载](#) 来学习如何在实际一个真实应用程序中使用 `import()` 的具体示例，以及查看 [缓存](#) 来学习如何有效地分离代码。

懒加载

本指南的继承自代码分离。如果你尚未阅读该指南，请先行阅读。

懒加载或者按需加载，是一种很好的优化网页或应用的方式。这种方式实际上是先把你的代码在一些逻辑断点处分离开，然后在一些代码块中完成某些操作后，立即引用或即将引用另外一些新的代码块。这样加快了应用的初始加载速度，减轻了它的总体体积，因为某些代码块可能永远不会被加载。

示例

我们在代码分离中的例子基础上，进一步做些调整来说明这个概念。那里的代码确实会在脚本运行的时候产生一个分离的代码块 `lodash.bundle.js`，在技术概念上“懒加载”它。问题是加载这个包并不需要用户的交互 -- 意思是每次加载页面的时候都会请求它。这样做并没有对我们有很多帮助，还会对性能产生负面影响。

我们试试不同的做法。我们增加一个交互，当用户点击按钮的时候用 `console` 打印一些文字。但是会等到第一次交互的时候再加载那个代码块 (`print.js`)。为此，我们返回到代码分离的例子中，把 `lodash` 放到主代码块中，重新运行代码分离中的代码 [*final Dynamic Imports example*](#)。

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
+ |- print.js
|- /node_modules
```

src/print.js

```
console.log('The print.js module has loaded! See the network tab in dev
export default () => {
  console.log('Button Clicked: Here\'s "some text"!');
};
```

src/index.js

```
+ import _ from 'lodash';
+
- async function getComponent() {
+ function component() {
```

```

var element = document.createElement('div');
- const _ = await import(/* webpackChunkName: "lodash" */ 'lodash');
+ var button = document.createElement('button');
+ var br = document.createElement('br');

+ button.innerHTML = 'Click me and look at the console!';
element.innerHTML = _.join(['Hello', 'webpack'], ' ');
element.appendChild(br);
element.appendChild(button);

+
+ // Note that because a network request is involved, some indication
+ // of loading would need to be shown in a production-level site/app
+ button.onclick = e => import(/* webpackChunkName: "print" */ './pri
+     var print = module.default;
+
+     print();
+ });

return element;
}

- getComponent().then(component => {
-   document.body.appendChild(component);
- });
+ document.body.appendChild(component());

```

注意当调用 ES6 模块的 `import()` 方法（引入模块）时，必须指向模块的 `.default` 值，因为它才是 promise 被处理后返回的实际的 `module` 对象。

现在运行 webpack 来验证一下我们的懒加载功能：

```

...
      Asset      Size  Chunks      Chunk Names
print.bundle.js  417 bytes    0  [emitted]  print
index.bundle.js    548 kB     1  [emitted]  [big]  index
      index.html  189 bytes            [emitted]
...

```

框架

许多框架和类库对于如何用它们自己的方式来实现（懒加载）都有自己的建议。这里有一些例子：

- React: [Code Splitting and Lazy Loading](#)
- Vue: [Lazy Load in Vue using Webpack's code splitting](#)
- AngularJS: [AngularJS + Webpack = lazyLoad by @var_bincom](#)

缓存

本指南继续沿用 [起步](#)、[管理输出](#) 和 [代码分离](#) 中的代码示例。

以上，我们使用 webpack 来打包模块化的应用程序，webpack 会生成一个可部署的 `/dist` 目录，然后把打包后的内容放置在此目录中。只要 `/dist` 目录中的内容部署到 server 上，client（通常是浏览器）就能够访问网站此 server 的网站及其资源。而最后一步获取资源是比较耗费时间的，这就是为什么浏览器使用一种名为缓存 的技术。可以通过命中缓存，以降低网络流量，使网站加载速度更快，然而，如果我们在部署新版本时不更改资源的文件名，浏览器可能会认为它没有被更新，就会使用它的缓存版本。由于缓存的存在，当你需要获取新的代码时，就会显得很棘手。

此指南的重点在于通过必要的配置，以确保 webpack 编译生成的文件能够被客户端缓存，而在文件内容变化后，能够请求到新的文件。

输出文件的文件名(output filename)

我们可以通过替换 `output.filename` 中的 `substitutions` 设置，来定义输出文件的名称。webpack 提供了一种使用称为 **substitution(可替换模板字符串)** 的方式，通过带括号字符串来模板化文件名。其中，`[contenthash]` substitution 将根据资源内容创建出唯一 hash。当资源内容发生变化时，`[contenthash]` 也会发生变化。

这里使用 [起步](#) 中的示例和 [管理输出](#) 中的 `plugins` 插件来作为项目基础，所以我们依然不必手动地维护 `index.html` 文件：

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
|- /node_modules
```

webpack.config.js

```
const path = require('path');
const CleanWebpackPlugin = require('clean-webpack-plugin');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.js',
  plugins: [
    new CleanWebpackPlugin(['dist']),
    new HtmlWebpackPlugin({
```

```

-
    title: 'Output Management'
+
    title: 'Caching'
  })
],
output: {
-
  filename: 'bundle.js',
+
  filename: '[name].[contenthash].js',
  path: path.resolve(__dirname, 'dist')
}
};

```

使用此配置，然后运行我们的 build script `npm run build`，产生以下输出：

```

...
Asset           Size   Chunks      Chun]
main.7e2c49a622975ebd9b7e.js  544 kB     0  [emitted]  [big]  main
index.html     197 bytes
...

```

可以看到，bundle 的名称是它内容（通过 hash）的映射。如果我们不做修改，然后再次运行构建，我们认为文件名会保持不变。然而，如果我们真的运行，可能会发现情况并非如此 - 文件名发生变化：

```

...
Asset           Size   Chunks      Chun]
main.205199ab45963f6a62ec.js  544 kB     0  [emitted]  [big]  main
index.html     197 bytes
...

```

这也是因为 webpack 在入口 chunk 中，包含了某些 boilerplate(引导模板)，特别是 runtime 和 manifest。（译注：boilerplate 指 webpack 运行时的引导代码）

输出可能会因当前的 webpack 版本而稍有差异。与旧版本相比，新版本不一定有完全相同的问题，但我们仍然推荐的以下步骤，确保结果可靠。

提取引导模板(extracting boilerplate)

正如我们在 [代码分离](#) 中所学到的，`SplitChunksPlugin` 可以用于将模块分离到单独的 bundle 中。webpack 还提供了一个优化功能，可使用 `optimization.runtimeChunk` 选项将 runtime 代码拆分为一个单独的 chunk。将其设置为 `single` 来为所有 chunk 创建一个 runtime bundle：

`webpack.config.js`

```

const path = require('path');
const CleanWebpackPlugin = require('clean-webpack-plugin');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.js',
  plugins: [

```

```

        new CleanWebpackPlugin(['dist']),
        new HtmlWebpackPlugin({
          title: 'Caching'
      },
      output: {
        filename: '[name].[contenthash].js',
        path: path.resolve(__dirname, 'dist')
      },
+     optimization: {
+       runtimeChunk: 'single'
+     }
);

```

再次构建，然后查看提取出来的 runtime bundle:

```

Hash: 82c9c385607b2150fab2
Version: webpack 4.12.0
Time: 3027ms
          Asset           Size  Chunks      Chunk Nar
runtime.cc17ae2a94ec771e9221.js   1.42 KiB      0  [emitted]  runtime
  main.e81de2cf758ada72f306.js    69.5 KiB      1  [emitted]  main
            index.html    275 bytes          [emitted]
[1] (webpack)/buildin/module.js 497 bytes {1} [built]
[2] (webpack)/buildin/global.js 489 bytes {1} [built]
[3] ./src/index.js 309 bytes {1} [built]
  + 1 hidden module

```

将第三方库(library)（例如 `lodash` 或 `react`）提取到单独的 vendor chunk 文件中，是比较推荐的做法，这是因为，它们很少像本地的源代码那样频繁修改。因此通过实现以上步骤，利用 client 的长效缓存机制，命中缓存来消除请求，并减少向 server 获取资源，同时还能保证 client 代码和 server 代码版本一致。这可以通过使用 [SplitChunksPlugin](#) [示例 2](#) 中演示的 `SplitChunksPlugin` 插件的 `cacheGroups` 选项来实现。我们在 `optimization.splitChunks` 添加如下 `cacheGroups` 参数并构建：

webpack.config.js

```

var path = require('path');
const CleanWebpackPlugin = require('clean-webpack-plugin');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.js',
  plugins: [
    new CleanWebpackPlugin(['dist']),
    new HtmlWebpackPlugin({
      title: 'Caching'
    }),
  ],
  output: {
    filename: '[name].[contenthash].js',
    path: path.resolve(__dirname, 'dist')
  },
  optimization: {

```

```

-
  runtimeChunk: 'single'
+
  runtimeChunk: 'single',
+
  splitChunks: {
    cacheGroups: {
      vendor: {
        test: /[\\/]node_modules[\\/]/,
        name: 'vendors',
        chunks: 'all'
      }
    }
  }
};


```

再次构建，然后查看新的 vendor bundle:

```

...
          Asset      Size  Chunks      Chunk Nar
runtime.cc17ae2a94ec771e9221.js   1.42 KiB    0  [emitted]  runtime
vendors.a42c3ca0d742766d7a28.js  69.4 KiB   1  [emitted]  vendors
  main.abf44fedb7d11d4312d7.js   240 bytes  2  [emitted]  main
          index.html  353 bytes
...

```

现在，我们可以看到 main 不再含有来自 node_modules 目录的 vendor 代码，并且体积减少到 240 bytes！

模块标识符(module identifier)

在项目中再添加一个模块 print.js:

project

```

webpack-demo
|- package.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
+ |- print.js
|- /node_modules

```

print.js

```

+ export default function print(text) {
+   console.log(text);
+ };

```

src/index.js

```

import _ from 'lodash';
+ import Print from './print';

```

```

function component() {
  var element = document.createElement('div');

  // lodash 是由当前 script 脚本 import 进来的
  element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+  element.onclick = Print.bind(null, 'Hello webpack!');

  return element;
}

document.body.appendChild(component());

```

再次运行构建，然后我们期望的是，只有 `main` bundle 的 hash 发生变化，然而.....

```

...
      Asset      Size  Chunks
runtime.1400d5af64fc1b7b3a45.js    5.85 kB    0  [emitted]
vendor.a7561fb0e9a071baadb9.js    541 kB     1  [emitted]  [big]
  main.b746e3eb72875af2caa9.js    1.22 kB     2  [emitted]
          index.html   352 bytes
...

```

.....我们可以看到这三个文件的 hash 都变化了。这是因为每个 `module.id` 会默认地基于解析顺序(resolve order)进行增量。也就是说，当解析顺序发生变化，ID 也会随之改变。因此，简要概括：

- `main` bundle 会随着自身的新增内容的修改，而发生变化。
- `vendor` bundle 会随着自身的 `module.id` 的变化，而发生变化。
- `manifest` bundle 会因为现在包含一个新模块的引用，而发生变化。

第一个和最后一个都是符合预期的行为 - 而 `vendor` hash 发生变化是我们要修复的。幸运的是，可以使用两个插件来解决这个问题。第一个插件是 `NamedModulesPlugin`，将使用模块的路径，而不是一个数字 identifier。虽然此插件有助于在开发环境下产生更加可读的输出结果，然而其执行时间会有些长。第二个选择是使用 `HashedModuleIdsPlugin`，推荐用于生产环境构建：

webpack.config.js

```

const path = require('path');
+ const webpack = require('webpack');
const CleanWebpackPlugin = require('clean-webpack-plugin');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.js',
  plugins: [
    new CleanWebpackPlugin(['dist']),
    new HtmlWebpackPlugin({
      title: 'Caching'
    }),
+    new webpack.HashedModuleIdsPlugin()

```

```

],
output: {
  filename: '[name].[contenthash].js',
  path: path.resolve(__dirname, 'dist')
},
optimization: {
  runtimeChunk: 'single',
  splitChunks: {
    cacheGroups: {
      vendor: {
        test: /[\\/]node_modules[\\/]/,
        name: 'vendors',
        chunks: 'all'
      }
    }
  }
}
;

```

现在，不论是否添加任何新的本地依赖，对于前后两次构建，`vendor hash`都应该保持一致：

```

...
Asset          Size  Chunks      Chunk Nar
main.216e852f60c8829c2289.js 340 bytes    0 [emitted]  main
vendors.55e79e5927a639d21a1b.js 69.5 KiB   1 [emitted]  vendors
runtime.725a1a51ede5ae0cfde0.js 1.42 KiB   2 [emitted]  runtime
index.html    353 bytes           [emitted]
Entrypoint main = runtime.725a1a51ede5ae0cfde0.js vendors.55e79e5927a63!
...

```

然后，修改 `src/index.js`，临时移除额外的依赖：

src/index.js

```

import _ from 'lodash';
- import Print from './print';
+ // import Print from './print';

function component() {
  var element = document.createElement('div');

  // lodash 是由当前 script 脚本 import 进来的
  element.innerHTML = _.join(['Hello', 'webpack'], ' ');
- element.onclick = Print.bind(null, 'Hello webpack!');
+ // element.onclick = Print.bind(null, 'Hello webpack!');

  return element;
}

document.body.appendChild(component());

```

最后，再次运行我们的构建：

```

...

```

```
Asset           Size   Chunks     Chunk Nar
main.ad717f2466ce655fff5c.js 274 bytes    0 [emitted]  main
vendors.55e79e5927a639d21a1b.js 69.5 KiB    1 [emitted]  vendors
runtime.725a1a51ede5ae0cfde0.js 1.42 KiB    2 [emitted]  runtime
index.html      353 bytes          [emitted]
Entrypoint main = runtime.725a1a51ede5ae0cfde0.js vendors.55e79e5927a63!
...

```

我们可以看到，这两次构建中，`vendor bundle` 文件名称，都是`55e79e5927a639d21a1b`。

结论

缓存可能很复杂，但是从应用程序或站点用户可以获得的收益来看，这值得付出努力。想要了解更多信息，请查看下面进一步阅读部分。

创建 library

除了打包应用程序，webpack 还可以用于打包 JavaScript library。以下指南适用于希望简化打包策略的 library 作者。

创建一个 library

假设你正在编写一个名为 `webpack-numbers` 的小的 library，可以将数字 1 到 5 转换为文本表示，反之亦然，例如将 2 转换为 'two'。

基本的项目结构可能如下所示：

project

```
+  |- webpack.config.js
+  |- package.json
+  |- /src
+    |- index.js
+    |- ref.json
```

初始化 npm，安装 webpack 和 lodash：

```
npm init -y
npm install --save-dev webpack lodash
```

src/ref.json

```
[  
  {  
    "num": 1,  
    "word": "One"  
  },  
  {  
    "num": 2,  
    "word": "Two"  
  },  
  {  
    "num": 3,  
    "word": "Three"  
  },  
  {  
    "num": 4,  
    "word": "Four"  
  },  
  {  
    "num": 5,  
    "word": "Five"  
  },  
  {  
    "num": 0,  
    "word": ""  
  }]
```

```
        "word": "Zero"
    }
]
```

src/index.js

```
import _ from 'lodash';
import numRef from './ref.json';

export function numToWord(num) {
    return _.reduce(numRef, (accum, ref) => {
        return ref.num === num ? ref.word : accum;
    }, '');
}

export function wordToNum(word) {
    return _.reduce(numRef, (accum, ref) => {
        return ref.word === word && word.toLowerCase() ? ref.num : accum;
    }, -1);
}
```

这个 library 的调用规范如下：

- **ES2015 module import:**

```
import * as webpackNumbers from 'webpack-numbers';
// ...
webpackNumbers.wordToNum('Two');
```

- **CommonJS module require:**

```
var webpackNumbers = require('webpack-numbers');
// ...
webpackNumbers.wordToNum('Two');
```

- **AMD module require:**

```
require(['webpackNumbers'], function (webpackNumbers) {
    // ...
    webpackNumbers.wordToNum('Two');
});
```

consumer(使用者) 还可以通过一个 script 标签来加载和使用此 library:

```
<!doctype html>
<html>
    ...
    <script src="https://unpkg.com/webpack-numbers"></script>
    <script>
        // ...
        // 全局变量
        webpackNumbers.wordToNum('Five')
        // window 对象中的属性
        window.webpackNumbers.wordToNum('Five')
```

```
// ...
</script>
</html>
```

注意，我们还可以通过以下配置方式，将 library 暴露为：

- global 对象中的属性，用于 Node.js。
- this 对象中的属性。

完整的 library 配置和代码，请查看 [webpack-library-example](#)。

基本配置

现在，让我们以某种方式打包这个 library，能够实现以下几个目标：

- 使用 `externals` 选项，避免将 `lodash` 打包到应用程序，而使用者会去加载它。
- 将 library 的名称设置为 `webpack-numbers`。
- 将 library 暴露为一个名为 `webpackNumbers` 的变量。
- 能够访问其他 Node.js 中的 library。

此外，consumer(使用者) 应该能够通过以下方式访问 library：

- ES2015 模块。例如 `import webpackNumbers from 'webpack-numbers'`。
- CommonJS 模块。例如 `require('webpack-numbers')`。
- 全局变量，在通过 `script` 标签引入时

我们可以从如下 webpack 基本配置开始：

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'webpack-numbers.js'
  }
};
```

外部化 lodash(externalize lodash)

现在，如果执行 `webpack`，你会发现创建了一个体积相当大的文件。如果你查看这个文件，会看到 `lodash` 也被打包到代码中。在这种场景中，我们更倾向于把 `lodash` 当作 `peerDependency`。也就是说，consumer(使用者) 应该已经安装过 `lodash`。因此，你就可以放弃控制此外部 library，而是将控制权让给使用 library

的 consumer。

这可以使用 `externals` 配置来完成：

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'webpack-numbers.js'
  },
  externals: {
    lodash: {
      commonjs: 'lodash',
      commonjs2: 'lodash',
      amd: 'lodash',
      root: '_'
    }
  }
};
```

这意味着你的 library 需要一个名为 `lodash` 的依赖，这个依赖在 consumer 环境中必须存在且可用。

注意，如果你仅计划将 library 用作另一个 webpack bundle 中的依赖模块，则可以直接将 `externals` 指定为一个数组。

外部化限制(external limitations)

对于想要实现从一个依赖中调用多个文件的那些 library:

```
import A from 'library/one';
import B from 'library/two';

// ...
```

无法通过在 `externals` 中指定整个 `library` 的方式，将它们从 bundle 中排除。而是需要逐个或者使用一个正则表达式，来排除它们。

```
module.exports = {
  //...
  externals: [
    'library/one',
    'library/two',
    // 匹配以 "library/" 开始的所有依赖
    /^library\//.+$/]
};
```

暴露 library

对于用法广泛的 library，我们希望它能够兼容不同的环境，例如 CommonJS，AMD，Node.js 或者作为一个全局变量。为了让你的 library 能够在各种使用环境中可用，需要在 `output` 中添加 `library` 属性：

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
-   filename: 'webpack-numbers.js'
+   filename: 'webpack-numbers.js',
+   library: 'webpackNumbers'
  },
  externals: {
    lodash: {
      commonjs: 'lodash',
      commonjs2: 'lodash',
      amd: 'lodash',
      root: '_'
    }
  }
};
```

注意，`library` 设置绑定到 `entry` 配置。对于大多数 library，指定一个入口起点就足够了。虽然 一次打包暴露多个库 也是可以的，然而，通过 index script(索引脚本)（仅用于访问一个入口起点）暴露部分导出则更为简单。我们不推荐使用数组作为 `library` 的 `entry`。

这会将你的 library bundle 暴露为名为 `webpackNumbers` 的全局变量，consumer 通过此名称来 import。为了让 library 和其他环境兼容，则需要在配置中添加 `libraryTarget` 属性。这个选项可以控制以不同形式暴露 library。

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'webpack-numbers.js',
-   library: 'webpackNumbers'
+   library: 'webpackNumbers',
+   libraryTarget: 'umd'
  },
  externals: {
    lodash: {
  
```

```
commonjs: 'lodash',
commonjs2: 'lodash',
amd: 'lodash',
root: '_'
}
}
};
```

有以下几种方式暴露 library:

- **变量:** 作为一个全局变量, 通过 `script` 标签来访问

`(libraryTarget:'var')`。

- **this:** 通过 `this` 对象访问 `(libraryTarget:'this')`。

- **window:** 在浏览器中通过 `window` 对象访问 `(libraryTarget:'window')`。

- **UMD:** 在 AMD 或 CommonJS `require` 之后可访问

`(libraryTarget:'umd')`。

如果设置了 `library` 但没有设置 `libraryTarget`, 则 `libraryTarget` 默认指定为 `var`, 详细说明请查看 [output](#) 文档。查看 [output.libraryTarget](#) 文档, 以获取所有可用选项的详细列表。

在 webpack v3.5.5 中, 使用 `libraryTarget: { root:'_' }` 将无法正常工作 (参考 [issue 4824](#)) 所述)。然而, 可以设置 `libraryTarget: { var: '_' }` 来将 `library` 作为全局变量。

最终步骤

遵循 [生产环境](#) 指南中的步骤, 来优化生产环境下的输出结果。那么, 我们还需要将生成 bundle 的文件路径, 添加到 `package.json` 中的 `main` 字段中。

package.json

```
{
  ...
  "main": "dist/webpack-numbers.js",
  ...
}
```

或者, 按照这个指南, 将其添加为标准模块:

```
{
  ...
  "module": "src/index.js",
  ...
}
```

这里的 key(键) `main` 是参照 `package.json` 标准, 而 `module` 是参照 [一个提案](#), 此提案允许 JavaScript 生态系统升级使用 ES2015 模块, 而不会破坏向后兼容性。

`module` 属性应指向一个使用 ES2015 模块语法（而不是其他浏览器或 Node.js 尚不支持的模块语法）的脚本。这使得 webpack 本身就可以解析模块语法，如果用户只用到 library 的某些部分，可以通过 `tree shaking` 打包更轻量的包。

现在，你可以将其发布为一个 `npm package`，并且在 [unpkg.com](#) 找到它，并分发给你的用户。

为了暴露和 library 关联着的样式表，你应该使用 `MiniCssExtractPlugin`。然后，用户可以像使用其他样式表一样使用和加载这些样式表。

shim 预置依赖

webpack compiler 能够识别遵循 ES2015 模块语法、CommonJS 或 AMD 规范编写的模块。然而，一些 third party(第三方库) 可能会引用一些全局依赖（例如 `jQuery` 中的 `$`）。因此这些 library 也可能创建一些需要导出的全局变量。这些 "broken modules(不符合规范的模块)" 就是 *shim(预置依赖)* 发挥作用的地方。

我们不推荐使用全局依赖！ webpack 背后的整个理念是使前端开发更加模块化。也就是说，需要编写具有良好的封闭性(well contained)、不依赖于隐含依赖（例如，全局变量）的彼此隔离的模块。请只在必要的时候才使用这些特性。

shim 另外一个极其有用的使用场景就是：当你希望 `polyfill` 扩展浏览器能力，来支持到更多用户时。在这种情况下，你可能只是想要将这些 `polyfills` 提供给需要修补(patch)的浏览器（也就是实现按需加载）。

下面的文章将向我们展示这两种用例。

为了方便，本指南继续沿用 [起步](#) 中的代码示例。在继续之前，请确保你已经熟悉这些配置。

shim 预置全局变量

让我们开始第一个 `shim` 全局变量的用例。在此之前，先看下我们的项目：

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
|- /node_modules
```

还记得我们之前用过的 `lodash` 吗？出于演示目的，例如把这个应用程序中的模块依赖，改为一个全局变量依赖。要实现这些，我们需要使用 `ProvidePlugin` 插件。

使用 `ProvidePlugin` 后，能够在 webpack 编译的每个模块中，通过访问一个变量来获取一个 package。如果 webpack 看到模块中用到这个变量，它将在最终 bundle 中引入给定的 package。让我们先移除 `lodash` 的 `import` 语句，改为通过插件提供它：

src/index.js

```

- import _ from 'lodash';
-
function component() {
  var element = document.createElement('div');

- // Lodash, now imported by this script
  element.innerHTML = _.join(['Hello', 'webpack'], ' ');
-
  return element;
}

document.body.appendChild(component());

```

webpack.config.js

```

const path = require('path');
+ const webpack = require('webpack');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
},
+ plugins: [
+   new webpack.ProvidePlugin({
+     _: 'lodash'
+   })
+ ];
};

```

我们本质上所做的，就是告诉 webpack.....

如果你遇到了至少一处用到`_`变量的模块实例，那请你将`lodash` package 引入进来，并将其提供给需要用到它的模块。

运行我们的构建脚本，将会看到同样的输出：

```

...
Asset      Size  Chunks      Chunk Names
bundle.js  544 kB      0  [emitted]  [big]  main
...

```

还可以使用`ProvidePlugin`暴露出某个模块中单个导出，通过配置一个“数组路径”（例如`[module, child, ...children?]`）实现此功能。所以，我们假想如下，无论`join`方法在何处调用，我们都只会获取到`lodash`中提供的`join`方法。

src/index.js

```

function component() {
  var element = document.createElement('div');

```

```

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

   return element;
}

document.body.appendChild(component());

```

webpack.config.js

```

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  plugins: [
    new webpack.ProvidePlugin({
-      _: 'lodash'
+      join: ['lodash', 'join']
    })
  ]
};

```

这样就能很好的与 tree shaking 配合，将 lodash library 中的其余没有用到的导出去除。

细粒度 shim

一些遗留模块依赖的 `this` 指向的是 `window` 对象。在接下来的用例中，调整我们的 `index.js`:

```

function component() {
  var element = document.createElement('div');

  element.innerHTML = join(['Hello', 'webpack'], ' ');
+
+ // 假设我们处于 `window` 上下文
+ this.alert('Hmmm, this probably isn\'t a great idea...')

  return element;
}

document.body.appendChild(component());

```

当模块运行在 CommonJS 上下文中，这将会变成一个问题，也就是说此时的 `this` 指向的是 `module.exports`。在这种情况下，你可以通过使用 imports-loader 覆盖 `this` 指向:

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
+  module: {
+    rules: [
+      {
+        test: require.resolve('index.js'),
+        use: 'imports-loader?this=>window'
+      }
+    ]
+  },
  plugins: [
    new webpack.ProvidePlugin({
      join: ['lodash', 'join']
    })
  ]
};

};
```

全局 export

让我们假设，某个 library 创建出一个全局变量，它期望 consumer(使用者) 使用这个变量。为此，我们可以在项目配置中，添加一个小模块来演示说明：

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
+  |- globals.js
|- /node_modules
```

src/globals.js

```
var file = 'blah.txt';
var helpers = {
  test: function() { console.log('test something'); },
  parse: function() { console.log('parse something'); }
};
```

你可能从来没有在自己的源码中做过这些事情，但是你也许遇到过一个老旧的 library，和上面所展示的代码类似。在这种情况下，我们可以使用 exports-loader，将一个全局变量作为一个普通的模块来导出。例如，为了将 `file` 导出

为 file 以及将 helpers.parse 导出为 parse，做如下调整：

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: require.resolve('index.js'),
        use: 'imports-loader?this=>window'
      },
      {
        test: require.resolve('globals.js'),
        use: 'exports-loader?file,parse=helpers.parse'
      }
    ]
  },
  plugins: [
    new webpack.ProvidePlugin({
      join: ['lodash', 'join']
    })
  ]
};
```

现在，在我们的 entry 入口文件中（即 src/index.js），我们能 import { file, parse } from './globals.js';，然后一切将顺利运行。

加载 polyfill

目前为止我们所讨论的所有内容都是处理那些遗留的 package，让我们进入到第二个话题： **polyfill**。

有很多方法来加载 polyfill。例如，想要引入 babel-polyfill 我们只需如下操作：

```
npm install --save babel-polyfill
```

然后，使用 import 将其引入到我们的主 bundle 文件：

src/index.js

```
+ import 'babel-polyfill';
+
```

```
function component() {
  var element = document.createElement('div');

  element.innerHTML = join(['Hello', 'webpack'], ' ');
  return element;
}

document.body.appendChild(component());
```

注意，我们没有将 `import` 绑定到某个变量。这是因为 polyfill 直接基于自身执行，并且是在基础代码执行之前，这样通过这些预置，我们就可以假定已经具有某些原生功能。

注意，这种方式优先考虑正确性，而不考虑 bundle 体积大小。为了安全和可靠，polyfill/shim 必须运行于所有其他代码之前，而且需要同步加载，或者说，需要在所有 polyfill/shim 加载之后，再去加载所有应用程序代码。社区中存在许多误解，即现代浏览器“不需要”polyfill，或者 polyfill/shim 仅用于添加缺失功能 - 实际上，它们通常用于修复损坏实现(*repair broken implementation*)，即使是在最现代的浏览器中，也会出现这种情况。因此，最佳实践仍然是，不加选择地和同步地加载所有 polyfill/shim，尽管这会导致额外的 bundle 体积成本。

如果你认为自己已经打消这些顾虑，并且希望承受损坏的风险。那么接下来的这件事情，可能是你应该要做的：我们将会把 `import` 放入一个新文件，并加入 `whatwg-fetch` polyfill：

```
npm install --save whatwg-fetch
```

src/index.js

```
- import 'babel-polyfill';
-
function component() {
  var element = document.createElement('div');

  element.innerHTML = join(['Hello', 'webpack'], ' ');
  return element;
}

document.body.appendChild(component());
```

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
  |- globals.js
+   |- polyfills.js
```

```
| - /node_modules
```

src/polyfills.js

```
import 'babel-polyfill';
import 'whatwg-fetch';
```

webpack.config.js

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

module.exports = {
-   entry: './src/index.js',
+   entry: {
+     polyfills: './src/polyfills.js',
+     index: './src/index.js'
+   },
+   output: {
-     filename: 'bundle.js',
+     filename: '[name].bundle.js',
       path: path.resolve(__dirname, 'dist')
   },
   module: {
     rules: [
       {
         test: require.resolve('index.js'),
         use: 'imports-loader?this=>window'
       },
       {
         test: require.resolve('globals.js'),
         use: 'exports-loader?file,parse=helpers.parse'
       }
     ]
   },
   plugins: [
     new webpack.ProvidePlugin({
       join: ['lodash', 'join']
     })
   ]
};
```

如上配置之后，我们可以在代码中添加一些逻辑，条件地加载新的 polyfills.bundle.js 文件。根据需要支持的技术和浏览器来决定是否加载。我们将做一些简单的试验，来确定是否需要引入这些 polyfill：

dist/index.html

```
<!doctype html>
<html>
  <head>
    <title>Getting Started</title>
+   <script>
+     var modernBrowser = (
+       'fetch' in window &&
```

```

+         'assign' in Object
+
+     if ( !modernBrowser ) {
+         var scriptElement = document.createElement('script');
+
+         scriptElement.async = false;
+         scriptElement.src = '/polyfills.bundle.js';
+         document.head.appendChild(scriptElement);
+
+     }
+     </script>
</head>
<body>
    <script src="index.bundle.js"></script>
</body>
</html>

```

现在，在 entry 入口文件中，可以通过 `fetch` 获取一些数据：

src/index.js

```

function component() {
    var element = document.createElement('div');

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

    return element;
}

document.body.appendChild(component());
+
+ fetch('https://jsonplaceholder.typicode.com/users')
+   .then(response => response.json())
+   .then(json => {
+     console.log('We retrieved some data! AND we\'re confident it will'
+     console.log(json)
+   })
+   .catch(error => console.error('Something went wrong when fetching th

```

执行构建脚本，可以看到，浏览器发送了额外的 `polyfills.bundle.js` 文件请求，然后所有代码顺利执行。注意，以上的这些设定可能还会有所改进，这里我们向你提供一个很棒的想法：将 `polyfill` 提供给需要引入它的用户。

进一步优化

`babel-preset-env` package 通过 `browserslist` 来转译那些你浏览器中不支持的特性。这个 preset 使用 `useBuiltIns` 选项，默认值是 `false`，这种方式可以将全局 `babel-polyfill` 导入，改进为更细粒度的 `import` 格式：

```

import 'core-js/modules/es7.string.pad-start';
import 'core-js/modules/es7.string.pad-end';
import 'core-js/modules/web.timers';
import 'core-js/modules/web.immediate';

```

```
import 'core-js/modules/web.dom iterable';
```

查看仓库以获取更多信息。

Node 内置

像 `process` 这种 Node 内置模块，能直接根据配置文件进行正确的 polyfill，而不需要任何特定的 loader 或者 plugin。查看 node 配置页面获取更多信息。

其他工具

还有一些其他的工具，也能够帮助我们处理这些遗留模块。

`script-loader` 会在 global context(全局上下文) 中对代码进行 eval 取值，这类似于通过一个 `script` 标签引入脚本。在这种模式下，每个正常的 library 都应该能运行。对 `require`, `module` 等取值是 `undefined`。

在使用 `script-loader` 时，模块将转为一个字符串，然后添加到 bundle 中。它不会被 `webpack` 压缩，所以你应该选择一个 min 版本。而且，使用此 loader 添加的 library 也没有 `devtool` 支持。

这些遗留模块如果没有 AMD/CommonJS 版本，但你也想将他们加入 `dist` 文件，则可以使用 `noParse` 来标识出这个模块。这样就能使 `webpack` 将引入这些模块，但是不进行转化(parse)，以及不解析(resolve) `require()` 和 `import` 语句。这种用法还会提高构建性能。

任何需要 AST 的功能（例如 `ProvidePlugin`）都起作用。

最后，一些模块支持多种 模块格式，例如一个混合有 AMD、CommonJS 和 legacy(遗留) 的模块。在大多数这样的模块中，会首先检查 `define`，然后使用一些怪异代码导出一些属性。在这些情况下，可以通过 `imports-loader` 设置 `define=>false` 来强制 CommonJS 路径。

译者注：shim 是一个库(library)，它将一个新的 API 引入到一个旧的环境中，而且仅靠旧的环境中已有的手段实现。polyfill 就是一个用在浏览器 API 上的 shim。我们通常的做法是先检查当前浏览器是否支持某个 API，如果不支持的话就按需加载对应的 polyfill。然后新旧浏览器就都可以使用这个 API 了。

渐进式网络应用程序

本指南继续沿用 [管理输出](#) 中的代码示例。

渐进式网络应用程序(progressive web application - PWA)，是一种可以提供类似于 native app(原生应用程序) 体验的 web app(网络应用程序)。PWA 可以用来做很多事。其中最重要的是，在离线([offline](#))时应用程序能够继续运行功能。这是通过使用名为 [Service Workers](#) 的 web 技术来实现的。

本章将重点介绍，如何为我们的应用程序添加离线体验。我们将使用名为 [Workbox](#) 的 Google 项目来实现此目的，该项目提供的工具可帮助我们更简单地为 web app 提供离线支持。

现在，我们并没有运行在离线环境下

到目前为止，我们一直是直接查看本地文件系统的输出结果。通常情况下，真正的用户是通过网络访问 web app；用户的浏览器会与一个提供所需资源（例如，`.html`, `.js` 和 `.css` 文件）的 **server** 通讯。

我们通过搭建一个简易 server 下，测试下这种离线体验。这里使用 [http-server](#) package: `npm install http-server --save-dev`。还要修改 `package.json` 的 `scripts` 部分，来添加一个 `start` script:

package.json

```
{  
  ...  
  "scripts": {  
    "build": "webpack"  
    "build": "webpack",  
    "start": "http-server dist"  
  },  
  ...  
}
```

如果你之前没有操作过，先得运行命令 `npm run build` 来构建你的项目。然后运行命令 `npm start`。应该产生以下输出：

```
> http-server dist  
  
启动 http-server, serve 整个 dist 文件夹  
到:  
  http://xx.x.x.x:8080  
  http://127.0.0.1:8080  
  http://xxx.xxx.x.x:8080  
按下 CTRL-C 停止服务
```

如果你打开浏览器访问 `http://localhost:8080` (即 `http://127.0.0.1`)，你应该会看到 webpack 应用程序被 serve 到 `dist` 目录。如果停止 server 然后刷新，则 webpack 应用程序不再可访问。

这就是我们为实现离线体验所需要的改变。在本章结束时，我们应该要实现的是，停止 server 然后刷新，仍然可以看到应用程序正常运行。

添加 Workbox

添加 `workbox-webpack-plugin` 插件，然后调整 `webpack.config.js` 文件：

```
npm install workbox-webpack-plugin --save-dev
```

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');
+ const WorkboxPlugin = require('workbox-webpack-plugin');

module.exports = {
  entry: {
    app: './src/index.js',
    print: './src/print.js'
  },
  plugins: [
    new CleanWebpackPlugin(['dist']),
    new HtmlWebpackPlugin({
-      title: '管理输出',
+      title: '渐进式网络应用程序',
-    })
+  },
+    new WorkboxPlugin.GenerateSW({
+      // 这些选项帮助快速启用 ServiceWorkers
+      // 不允许遗留任何“旧的” ServiceWorkers
+      clientsClaim: true,
+      skipWaiting: true
+    })
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

完成这些设置，再次执行 `npm run build`，看下会发生什么：

```
...
          Asset        Size  Chunks      Chunk Name
app.bundle.js     545 kB     0, 1  [emitted]  [big]  app
print.bundle.js   2.74 kB      1  [emitted]
index.html      254 bytes
precache-manifest.b5ca1c555e832d6fbf9462efd29d27eb.js  268 bytes
```

```
service-worker.js          1 kB          [emitted]
```

...

现在你可以看到，生成了两个额外的文件：`service-worker.js` 和名称冗长的 `precache-manifest.b5ca1c555e832d6fbf9462efd29d27eb.js`。`service-worker.js` 是 Service Worker 文件，`precache-manifest.b5ca1c555e832d6fbf9462efd29d27eb.js` 是 `service-worker.js` 引用的文件，所以它也可以运行。你本地生成的文件可能会有所不同；但是应该会有一个 `service-worker.js` 文件。

所以，值得高兴的是，我们现在已经创建出一个 Service Worker。接下来该做什么？

注册 Service Worker

接下来我们注册 Service Worker，使其出场并开始表演。通过添加以下注册代码来完成此操作：

index.js

```
import _ from 'lodash';
import printMe from './print.js';

+ if ('serviceWorker' in navigator) {
+   window.addEventListener('load', () => {
+     navigator.serviceWorker.register('/service-worker.js').then(registration => {
+       console.log('SW registered: ', registration);
+     }).catch(registrationError => {
+       console.log('SW registration failed: ', registrationError);
+     });
+   });
+ }
```

再次运行 `npm build build` 来构建包含注册代码版本的应用程序。然后用 `npm start` 将构建结果 `serve` 到服务下。导航至 `http://localhost:8080` 并查看 `console` 控制台。应该看到：

```
SW registered
```

现在来进行测试。停止 `server` 并刷新页面。如果浏览器能够支持 Service Worker，应该可以看到你的应用程序还在正常运行。然而，`server` 已经停止 `serve` 整个 `dist` 文件夹，此刻是 Service Worker 在进行 `serve`。

结论

你已经使用 Workbox 项目构建了一个离线应用程序。开始进入将 web app 改造为 PWA 的旅程。你现在可能想要考虑下一步做什么。这里谷歌提供的 PWA 文档中

可以找到一些不错的资源。

TypeScript

本指南继续沿用 [起步](#) 中的代码示例。

TypeScript 是 JavaScript 的超集，为其增加了类型系统，可以编译为普通 JavaScript 代码。这篇指南里我们将会学习是如何将 webpack 和 TypeScript 进行集成。

基础配置

首先，执行以下命令安装 TypeScript compiler 和 loader:

```
npm install --save-dev typescript ts-loader
```

现在，我们将修改目录结构和配置文件:

project

```
webpack-demo
|- package.json
+ |- tsconfig.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
  |- index.js
+   |- index.ts
|- /node_modules
```

tsconfig.json

这里我们设置一个基本的配置，来支持 JSX，并将 TypeScript 编译到 ES5.....

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "noImplicitAny": true,
    "module": "es6",
    "target": "es5",
    "jsx": "react",
    "allowJs": true
  }
}
```

查看 [TypeScript 官方文档](#) 了解更多关于 `tsconfig.json` 的配置选项。

想要了解 webpack 配置的更多信息，请查看 [配置概念](#)。

现在，配置 webpack 处理 TypeScript:

webpack.config.js

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

module.exports = {
  entry: './src/index.ts',
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/
      }
    ]
  },
  resolve: {
    extensions: [ '.tsx', '.ts', '.js' ]
  },
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

这会让 webpack 直接从 `./index.ts` 进入，然后通过 `ts-loader` 加载所有的 `.ts` 和 `.tsx` 文件，并且在当前目录输出一个 `bundle.js` 文件。

loader

ts-loader

在本指南中，我们使用 `ts-loader`，因为它能够很方便地启用额外的 webpack 功能，例如将其他 web 资源导入到项目中。

source map

想要了解 source map 的更多信息，请查看 [开发指南](#)。

想要启用 source map，我们必须配置 TypeScript，以将内联的 source map 输出到编译后的 JavaScript 文件中。必须在 TypeScript 配置中添加下面这行：

tsconfig.json

```
{
  "compilerOptions": {
    "outDir": "./dist/",
+    "sourceMap": true,
    "noImplicitAny": true,
```

```
        "module": "commonjs",
        "target": "es5",
        "jsx": "react",
        "allowJs": true
    }
}
```

现在，我们需要告诉 webpack 提取这些 source map，并内联到最终的 bundle 中。

webpack.config.js

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

module.exports = {
  entry: './src/index.ts',
+  devtool: 'inline-source-map',
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/
      }
    ]
  },
  resolve: {
    extensions: [ '.tsx', '.ts', '.js' ]
  },
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

查看 [devtool](#) 文档以了解更多信息。

使用 third party library

在从 npm 安装 third party library(第三方库)时，一定要记得同时安装此 library 的类型声明文件(typing definition)。你可以从 [TypeSearch](#) 中找到并安装这些第三方库的类型声明文件。

举个例子，如果想安装 lodash 类型声明文件，我们可以运行下面的命令：

```
npm install --save-dev @types/lodash
```

想了解更多，可以查看 [这篇文章](#)。

导入其他资源

想要在 TypeScript 中使用非代码资源(non-code asset)，我们需要告诉 TypeScript 推

断导入资源的类型。在项目里创建一个 `custom.d.ts` 文件，这个文件用来表示项目中 TypeScript 的自定义类型声明。我们为 `.svg` 文件设置一个声明：

custom.d.ts

```
declare module "*svg" {
  const content: any;
  export default content;
}
```

这里，我们通过指定任何以 `.svg` 结尾的导入(`import`)，将 SVG 声明(`declare`)为一个新的模块(`module`)，并将模块的 `content` 定义为 `any`。我们可以通过将类型定义为字符串，来更加显式地将它声明为一个 url。同样的概念适用于其他资源，包括 CSS, SCSS, JSON 等。

构建性能

这可能会降低构建性能。

关于构建工具，请查看 [构建性能指南](#)。

环境变量

想要消除 开发环境 和 生产环境 之间的 `webpack.config.js` 差异，你可能需要环境变量(environment variable)。

`webpack` 命令行 环境配置 的 `--env` 参数，可以允许你传入任意数量的环境变量。而在 `webpack.config.js` 中可以访问到这些环境变量。例如，`--env.production` 或 `--env.NODE_ENV=local` (`NODE_ENV` 通常约定用于定义环境类型，查看 [这里](#))。

```
webpack --env.NODE_ENV=local --env.production --progress
```

如果设置 `env` 变量，却没有赋值，`--env.production` 默认表示将 `--env.production` 设置为 `true`。还有许多其他可以使用的语法。更多详细信息，请查看 [webpack CLI](#) 文档。

对于我们的 `webpack` 配置，有一个必须要修改之处。通常，`module.exports` 指向配置对象。要使用 `env` 变量，你必须将 `module.exports` 转换成一个函数：

webpack.config.js

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

module.exports = env => {
  // Use env.<YOUR VARIABLE> here:
  console.log('NODE_ENV: ', env.NODE_ENV); // 'local'
  console.log('Production: ', env.production); // true

  return {
    entry: './src/index.js',
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist')
    }
  };
};
```

构建性能

本指南包含一些改进构建/编译性能的实用技巧。

通用环境

无论你是在 [开发环境](#) 还是在 [生产环境](#) 下运行构建脚本，以下最佳实践都应该有所帮助。

更新到最新版本

使用最新的 webpack 版本。我们会经常进行性能优化。webpack 的最新稳定版本是：

webpack v4.39.1

将 [Node.js](#) 更新到最新版本，也有助于提高性能。除此之外，将你的 package 管理工具（例如 `npm` 或者 `yarn`）更新到最新版本，也有助于提高性能。较新的版本能够建立更高效的模块树以及提高解析速度。

loader

对最少数量的必要模块使用 loader。不如下：

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.js$/,
        loader: 'babel-loader'
      }
    ]
  }
};
```

而是使用 `include` 字段仅将 loader 应用在实际需要将其转换的模块所处路径：

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.js$/,
        include: path.resolve(__dirname, 'src'),
        loader: 'babel-loader'
      }
    ]
  }
};
```

```
        }
    ]
}
};
```

引导时间(bootstrap)

每个额外的 loader/plugin 都有其启动时间。尽量少使用工具。

解析

以下步骤可以提高解析速度:

- 减少 `resolve.modules`, `resolve.extensions`, `resolve.mainFiles`, `resolve.descriptionFiles` 中 items 数量, 因为他们会增加文件系统调用的次数。
- 如果你不使用 symlinks (例如 `npm link` 或者 `yarn link`) , 可以设置 `resolve.symlinks: false`。
- 如果你使用自定义 `resolve plugin` 规则, 并且没有指定 context 上下文, 可以设置 `resolve.cacheWithContext: false`。

dll

使用 `DllPlugin` 为更改不频繁的代码生成单独编译结果。这可以提高应用程序的编译速度, 尽管它确实增加了构建过程的复杂度。

小即是快(smaller = faster)

减少编译结果的整体大小, 以提高构建性能。尽量保持 chunk 体积小。

- 使用数量更少/体积更小的 library。
- 在多页面应用程序中使用 `CommonsChunkPlugin`。
- 在多页面应用程序中使用 `CommonsChunkPlugin`, 并开启 `async` 模式。
- 移除未引用代码。
- 只编译你当前正在开发的那些代码。

worker 池(worker pool)

`thread-loader` 可以将非常消耗资源的 loader 分流给一个 worker pool。

不要使用太多的 worker, 因为 Node.js 的 runtime 和 loader 都有启动开销。最小化 worker 和 main process(主进程)之间的模块传输。进程间通讯(IPC, inter process communication)是非常消耗资源的。

持久化缓存

使用 `cache-loader` 启用持久化缓存。使用 `package.json` 中的 "postinstall" 清除缓存目录。

自定义 plugin/loader

这里不对它们的性能问题作过多赘述。

开发环境

以下步骤对于开发环境特别有帮助。

增量编译

使用 webpack 的 watch mode(监听模式)。而不使用其他工具来 watch 文件和调用 webpack。内置的 watch mode 会记录时间戳并将此信息传递给 compilation 以使缓存失效。

在某些配置环境中，watch mode 会回退到 poll mode(轮询模式)。监听许多文件会导致 CPU 大量负载。在这些情况下，可以使用 `watchOptions.poll` 来增加轮询的间隔。

在内存中编译

下面几个工具通过在内存中（而不是写入磁盘）编译和 serve 资源来提高性能：

- `webpack-dev-server`
- `webpack-hot-middleware`
- `webpack-dev-middleware`

stats.toJson 加速

webpack 4 默认使用 `stats.toJson()` 输出大量数据。除非在增量步骤中做必要的统计，否则请避免获取 `stats` 对象的部分内容。`webpack-dev-server` 在 v3.1.3 以后的版本，包含一个重要的性能修复，即最小化每个增量构建步骤中，从 `stats` 对象获取的数据量。

devtool

需要注意的是不同的 `devtool` 设置，会导致不同的性能差异。

- "eval" 具有最好的性能，但并不能帮助你转译代码。
- 如果你能接受稍差一些的 map 质量，可以使用 `cheap-source-map` 变体配置来提高性能
- 使用 `eval-source-map` 变体配置进行增量编译。

=> 在大多数情况下，最佳选择是 `cheap-module-eval-source-map`。

避免在生产环境下才会用到的工具

某些 utility, plugin 和 loader 都只用于生产环境。例如，在开发环境下使用 `TerserPlugin` 来 minify(压缩) 和 mangle(混淆破坏) 代码是没有意义的。通常在开发环境下，应该排除以下这些工具：

- `TerserPlugin`
- `ExtractTextPlugin`
- `[hash]/[chunkhash]`
- `AggressiveSplittingPlugin`
- `AggressiveMergingPlugin`
- `ModuleConcatenationPlugin`

最小化 entry chunk

webpack 只会在文件系统中生成已经更新的 chunk。某些配置选项（HMR, `output.chunkFilename` 的 `[name]/[chunkhash]`, `[hash]`）来说，除了对更新的 chunk 无效之外，对于 entry chunk 也不会生效。

确保在生成 entry chunk 时，尽量减少其体积以提高性能。下面的代码块将只提取包含 runtime 的 chunk，其他 chunk 都作为其子 chunk:

```
new CommonsChunkPlugin({  
  name: 'manifest',  
  minChunks: Infinity  
});
```

避免额外的优化步骤

webpack 通过执行额外的算法任务，来优化输出结果的体积和加载性能。这些优化适用于小型代码库，但是在大型代码库中却非常耗费性能：

```
module.exports = {  
  // ...  
  optimization: {  
    removeAvailableModules: false,  
    removeEmptyChunks: false,  
    splitChunks: false,  
  }  
};
```

输出结果不携带路径信息

webpack 会在输出的 bundle 中生成路径信息。然而，在打包数千个模块的项目中，这会导致造成垃圾回收性能压力。在 `options.output.pathinfo` 设置中关闭：

```
module.exports = {
  // ...
  output: {
    pathinfo: false
  }
};
```

Node.js 版本

最新稳定版本的 Node.js 及其 ES2015 `Map` 和 `Set` 实现，出现一些 性能回退。其修复版本已经合并到 master 分支，但是有些已经发布的正式版本无法应用到这些修复内容。同时，为了充分利用增量构建速度，请尝试使用 8.9.x 版本（8.9.10 - 9.11.1 之间的版本存在性能问题）。webpack 已经开始大量使用这些 ES2015 数据结构，因此选择这些版本也将改善初始构建时间。

TypeScript loader

现在，`ts-loader` 已经开始使用 TypeScript 内置 watch mode API，可以明显减少每次迭代时重新构建的模块数量。`experimentalWatchApi` 与普通 TypeScript watch mode 共享同样的逻辑，并且在开发环境使用时非常稳定。此外开启 `transpileOnly`，用于真正快速增量构建。

```
module.exports = {
  // ...
  test: /\.tsx?$/,
  use: [
    {
      loader: 'ts-loader',
      options: {
        transpileOnly: true,
        experimentalWatchApi: true,
      },
    },
  ],
};
```

注意：`ts-loader` 文档建议使用 `cache-loader`，但是这实际上会由于使用硬盘写入而减缓增量构建速度。

为了重新获得类型检查，请使用 [ForkTsCheckerWebpackPlugin](#)。

`ts-loader` 的 github 仓库中有一个 完整示例。

生产环境

以下步骤对于生产环境特别有帮助。

不要为了很小的性能收益，牺牲应用程序的质量！注意，在大多数情况下，优化代码质量比构建性能更重要。

多个 compilation(编译时)

在进行多个 compilation 时，以下工具可以帮助到你：

- `parallel-webpack`: 它允许在 worker 池中运行 compilation。
- `cache-loader`: 可以在多个 compilation 之间共享缓存。

source map

source map 相当消耗资源。你真的需要它们？

工具相关问题

下列工具存在某些可能会降低构建性能的问题：

Babel

- 最小化项目中的 preset/plugins 数量。

TypeScript

- 在单独的进程中使用 `fork-ts-checker-webpack-plugin` 进行类型检查。
- 配置 loader 跳过类型检查。
- 使用 `ts-loader` 时，设置 `happyPackMode: true / transpileOnly: true`。

Sass

- `node-sass` 中有个来自 Node.js 线程池的阻塞线程的 bug。当使用 `thread-loader` 时，需要设置 `workerParallelJobs: 2`。

内容安全策略

webpack 能够为其加载的所有脚本添加 `nonce`。要启用此功能，需要在引入的入口脚本中设置一个 `_webpack_nonce_` 变量。应该为每个唯一的页面视图生成和提供一个唯一的基于 hash 的 nonce，这就是为什么 `_webpack_nonce_` 要在入口文件中指定，而不是在配置中指定的原因。注意，`nonce` 应该是一个 base64 编码的字符串。

示例

在 entry 文件中：

```
// ...
__webpack_nonce__ = 'c29tZSBjb29sIHN0cmluZyB3aWxsIHBvcCB1cCAxMjM=';
// ...
```

启用 CSP

注意，默认情况下不启用 CSP。需要与文档(`document`)一起发送相应的 `CSP` header 或 `meta` 标签 `<meta http-equiv="Content-Security-Policy" ...>`，以告知浏览器启用 CSP。以下是一个包含 CDN 白名单 URL 的 `CSP` header 的示例：

```
Content-Security-Policy: default-src 'self'; script-src 'self' https://
```

有关 `CSP` 和 `nonce` 属性的更多信息，请查看页面底部的进一步阅读部分。

开发 - Vagrant

如果你在开发一个更加高级的项目，并且使用 `Vagrant` 来实现在虚拟机(Virtual Machine)上运行你的开发环境，那你可能会需要在虚拟机中运行 `webpack`。

配置项目

首先，确保 `Vagrantfile` 拥有一个静态 IP。

```
Vagrant.configure("2") do |config|
  config.vm.network :private_network, ip: "10.10.10.61"
end
```

然后，在项目中安装 `webpack` 和 `webpack-dev-server`。

```
npm install --save-dev webpack webpack-dev-server
```

确保提供一个 `webpack.config.js` 配置文件。如果还没有准备，下面的示例代码可以作为起步的最简配置：

```
module.exports = {
  context: __dirname,
  entry: './app.js'
};
```

然后，创建一个 `index.html` 文件。其中的 `script` 标签应当指向你的 `bundle`。如果没有在配置中指定 `output.filename`，其默认值是 `bundle.js`。

```
<!doctype html>
<html>
  <head>
    <script src="/bundle.js" charset="utf-8"></script>
  </head>
  <body>
    <h2>Heey!</h2>
  </body>
</html>
```

注意，你还需要创建一个 `app.js` 文件。

启动 server

现在，我们启动 server:

```
webpack-dev-server --host 0.0.0.0 --public 10.10.10.61:8080 --watch-pol:
```

默认只允许从 `localhost` 访问 server。所以我们需要修改 `--host` 参数，以允许在我

们的宿主 PC 上访问。

webpack-dev-server 会在 bundle 中引入一个脚本，此脚本连接到 WebSocket，这样可以在任何文件修改时，触发重新加载应用程序。--public 标记可以确保脚本知道从哪里查找 WebSocket。默认情况下，server 会使用 8080 端口，因此也需要在这里指定。

--watch-poll 可以确保 webpack 能够检测到文件更改。webpack 默认会监听文件系统触发的相关事件，但是 VirtualBox 使用默认配置会有许多问题。

现在 server 应该能够通过 `http://10.10.10.61:8080` 访问了。修改 `app.js`，应用程序就会实时重新加载。

配合 nginx 的高级用法

为了更好的模拟类生产环境(production-like environment)，还可以用 nginx 来代理 webpack-dev-server。

在你的 nginx 配置文件中，加入下面代码：

```
server {
  location / {
    proxy_pass http://127.0.0.1:8080;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    error_page 502 @start-webpack-dev-server;
  }

  location @start-webpack-dev-server {
    default_type text/plain;
    return 502 "Please start the webpack-dev-server first.";
  }
}
```

`proxy_set_header` 这几行配置很重要，因为它们能够使 WebSocket 正常运行。

然后 webpack-dev-server 启动命令可以修改为：

```
webpack-dev-server --public 10.10.10.61 --watch-poll
```

现在只能通过 `127.0.0.1` 访问 server，这点关系不大，因为 nginx 能够使得你的 PC 能访问到 server。

结论

我们能够从静态 IP 访问 Vagrant box，然后使 webpack-dev-server 可以公开访问，

以便浏览器可以访问到它。然后，我们解决了 VirtualBox 不发送到文件系统事件的常见问题，此问题会导致 server 无法重新加载文件更改。

管理依赖

es6 modules

commonjs

amd

带表达式的 require 语句

如果你的 request 含有表达式(expressions)，就会创建一个上下文(context)，因为在编译时(compile time)并不清楚具体导入哪个模块。

示例：

```
require('./template/' + name + '.ejs');
```

webpack 解析 require() 调用，然后提取出如下一些信息：

```
Directory: ./template
Regular expression: /^.+\.\.ejs$/
```

context module

生成一个 context module(上下文模块)。它包含目录下的所有模块的引用，是通过一个 request 解析出来的正则表达式，去匹配目录下所有符合的模块，然后都 require 进来。此 context module 包含一个 map 对象，会把 request 中所有模块翻译成对应的模块 id。(译者注：request 参考 概念术语 文档)

示例：

```
{
  "./table.ejs": 42,
  "./table-row.ejs": 43,
  "./directory/folder.ejs": 44
}
```

此 context module 还包含一些访问这个 map 对象的 runtime 逻辑。

这意味着 webpack 能够支持动态地 require，但会导致所有可能用到的模块都包含在 bundle 中。

require.context

你还可以通过 require.context() 函数来创建自己的 context。

可以给这个函数传入三个参数：一个要搜索的目录，一个标记表示是否还搜索其子目录，以及一个匹配文件的正则表达式。

webpack 会在构建中解析代码中的 `require.context()`。

语法如下：

```
require.context(directory, useSubdirectories = false, regExp = /^\.\/\//),
```

示例：

```
require.context('./test', false, /\.test\.js$/);
// (创建出) 一个 context，其中文件来自 test 目录，request 以 `test.js` 结尾。

require.context('../', true, /\.stories\.js$/);
// (创建出) 一个 context，其中所有文件都来自父文件夹及其所有子级文件夹，request 以
```

传递给 `require.context` 的参数必须是字面量(literal)！

context module API

一个 context module 会导出一个 (require) 函数，此函数可以接收一个参数：`request`。

此导出函数有三个属性：`resolve`, `keys`, `id`。

- `resolve` 是一个函数，它返回 `request` 被解析后得到的模块 `id`。
- `keys` 也是一个函数，它返回一个数组，由所有可能被此 context module 处理的请求（译者注：参考下面第二段代码中的 `key`）组成。

如果想引入一个文件夹下面的所有文件，或者引入能匹配一个正则表达式的所有文件，这个功能就会很有帮助，例如：

```
function importAll (r) {
  r.keys().forEach(r);
}

importAll(require.context('../components/', true, /\.js$/));

var cache = {};

function importAll (r) {
  r.keys().forEach(key => cache[key] = r(key));
}

importAll(require.context('../components/', true, /\.js$/));
// 在构建时(build-time)，所有被 require 的模块都会被填充到 cache 对象中。
```

- `id` 是 context module 里面所包含的模块 `id`。它可能在你使用 `module.hot.accept` 时会用到。

公共路径

`publicPath` 配置选项在各种场景中都非常有用。你可以通过它来指定应用程序中所有资源的基础路径。

示例

下面提供一些用于实际应用程序的示例，通过这些示例，此功能显得极其简单。实质上，发送到 `output.path` 目录的每个文件，都将从 `output.publicPath` 位置引用。这也包括（通过 [代码分离](#) 创建的）子 chunk 和作为依赖图一部分的所有其他资源（例如 `image`, `font` 等）。

基于环境设置

在开发环境中，我们通常有一个 `assets/` 文件夹，它与索引页面位于同一级别。这没太大问题，但是，如果我们将所有静态资源托管至 CDN，然后想在生产环境中使用呢？

想要解决这个问题，可以直接使用一个有着悠久历史的 environment variable(环境变量)。假设我们有一个变量 `ASSET_PATH`:

```
import webpack from 'webpack';

// 尝试使用环境变量，否则使用根路径
const ASSET_PATH = process.env.ASSET_PATH || '/';

export default {
  output: {
    publicPath: ASSET_PATH
  },
  plugins: [
    // 这可以帮助我们在代码中安全地使用环境变量
    new webpack.DefinePlugin({
      'process.env.ASSET_PATH': JSON.stringify(ASSET_PATH)
    })
  ]
};
```

在运行时设置

另一个可能出现的情况是，需要在运行时设置 `publicPath`。`webpack` 暴露了一个名为 `__webpack_public_path__` 的全局变量。所以在应用程序的 entry point 中，可以直接如下设置：

```
__webpack_public_path__ = process.env.ASSET_PATH;
```

这些内容就是你所需要的。由于我们已经在配置中使用了 `DefinePlugin`, `process.env.ASSET_PATH` 将始终都被定义，因此我们可以安全地使用。

注意，如果在 entry 文件中使用 ES2015 module import，则会在 import 之后进行 `__webpack_public_path__` 赋值。在这种情况下，你必须将 public path 赋值移至一个专用模块中，然后将它的 import 语句放置到 entry.js 最上面：

```
// entry.js
import './public-path';
import './app';
```

集成

首先，我们要消除一个常见的误解。webpack 是一个模块打包工具(module bundler)（例如，[Browserify](#) 或 [Brunch](#)）。而不是一个任务执行工具(task runner)（例如，[Make](#), [Grunt](#) 或者 [Gulp](#)）。任务执行工具用来自动化处理常见的开发任务，例如，lint(代码检测)、build(构建)、test(测试)。相比模块打包工具，任务执行工具则聚焦在偏重上层的问题上面。你仍然可以得益于这种用法：使用上层的工具，而将打包部分的问题留给 webpack。

打包工具帮助你取得准备用于部署的 JavaScript 和 stylesheet，将它们转换为适合浏览器的可用格式。例如，可以通过压缩、分离 chunk 和惰性加载我们的 JavaScript 来提高性能。打包是 web 开发中最重要的挑战之一，解决此问题可以消除开发过程中的大部分痛点。

好的消息是，虽然有一些功能重叠，但是如果使用方式正确，任务运行工具和模块打包工具还是能够一起协同工作。本指南提供了关于如何将 webpack 与一些流行的任务运行工具集成在一起的高度概述。

npm scripts

通常 webpack 用户使用 `npm scripts` 来作为任务执行工具。这是比较好的开始。然而跨平台支持可能是个问题，但是有几种解决方案。许多用户（但不是大多数用户）直接使用 `npm scripts` 和各种级别的 webpack 配置和工具。

因此，虽然 webpack 核心重点是打包，但是可以通过各种扩展，将它用于任务运行工具的常见工作。集成一个单独的工具会增加复杂度，因此在开始前一定要权衡利弊。

Grunt

对于那些使用 Grunt 的人，我们推荐使用 [`grunt-webpack`](#) package。使用 `grunt-webpack` 你可以将 webpack 或 `webpack-dev-server` 作为一项任务(task)执行，访问 `grunt template tags` 中的统计信息，拆分开发和生产配置等等。如果还没有安装 `grunt-webpack` 和 `webpack`，请先安装它们：

```
npm install --save-dev grunt-webpack webpack
```

然后，注册一个配置并加载任务：

Gruntfile.js

```
const webpackConfig = require('./webpack.config.js');
```

```
module.exports = function(grunt) {
  grunt.initConfig({
    webpack: {
      options: {
        stats: !process.env.NODE_ENV || process.env.NODE_ENV === 'development',
      },
      prod: webpackConfig,
      dev: Object.assign({ watch: true }, webpackConfig)
    }
  });
  grunt.loadNpmTasks('grunt-webpack');
};
```

获取更多信息，请查看[仓库](#)。

Gulp

在 `webpack-stream` package (也称作 `gulp-webpack`) 的帮助下，可以相当直接地将 Gulp 与 webpack 集成。在这种情况下，不需要单独安装 webpack，因为它是 `webpack-stream` 直接依赖：

```
npm install --save-dev webpack-stream
```

只要将 `webpack` 替换为 `require('webpack-stream')`，并传递一个配置：

gulpfile.js

```
var gulp = require('gulp');
var webpack = require('webpack-stream');
gulp.task('default', function() {
  return gulp.src('src/entry.js')
    .pipe(webpack({
      // 所有配置选项.....
    }))
    .pipe(gulp.dest('dist/'));
});
```

获取更多信息，请查看[仓库](#)。

Mocha

`mocha-webpack` 可以将 Mocha 与 webpack 完全集成。这个仓库提供了很多关于其优势和劣势的细节，基本上 `mocha-webpack` 只是一个简单封装，提供与 Mocha 几乎相同的 CLI，并提供各种 webpack 功能，例如改进了 watch mode 和改进了路径分析。这里是一个如何安装并使用它来运行测试套件的示例（在 `./test` 中找到）：

```
npm install --save-dev webpack mocha mocha-webpack
mocha-webpack 'test/**/*.{js,jsx}'
```

获取更多信息，请查看 [仓库](#)。

Karma

`karma-webpack` package 允许你使用 webpack 预处理 Karma 中的文件。它也可以使用 `webpack-dev-middleware`，并允许传递两者的配置。下面是一个简单的示例：

```
npm install --save-dev webpack karma karma-webpack
```

`karma.conf.js`

```
module.exports = function(config) {
  config.set({
    files: [
      { pattern: 'test/*_test.js', watched: false },
      { pattern: 'test/**/*_test.js', watched: false }
    ],
    preprocessors: {
      'test/*_test.js': [ 'webpack' ],
      'test/**/*_test.js': [ 'webpack' ]
    },
    webpack: {
      // 所有自定义的 webpack 配置.....
    },
    webpackMiddleware: {
      // 所有自定义的 webpack-dev-middleware 配置.....
    }
  });
};
```

获取更多信息，请查看 [仓库](#)。

loader

webpack 可以使用 `loader` 来预处理文件。这允许你打包除 JavaScript 之外的任何静态资源。你可以使用 Node.js 来很简单地编写自己的 loader。

loader 通过在 `require()` 语句中使用 `loadername!` 前缀来激活，或者通过 webpack 配置中的正则表达式来自动应用 - 查看配置。

文件

- `raw-loader` 加载文件原始内容 (utf-8)
- `val-loader` 将代码作为模块执行，并将 `exports` 转为 JS 代码
- `url-loader` 像 file loader 一样工作，但如果文件小于限制，可以返回 `data URL`
- `file-loader` 将文件发送到输出文件夹，并返回（相对）URL
- `ref-loader` 手动创建所有文件之间的依赖关系

JSON

- `json-loader` 加载 JSON 文件（默认包含）
- `json5-loader` 加载和转译 JSON 5 文件
- `cson-loader` 加载和转译 CSON 文件

转译(transpiling)

- `script-loader` 在全局上下文中执行一次 JavaScript 文件（如在 `script` 标签），不需要解析
- `babel-loader` 加载 ES2015+ 代码，然后使用 `Babel` 转译为 ES5
- `buble-loader` 使用 `Bublé` 加载 ES2015+ 代码，并且将代码转译为 ES5
- `traceur-loader` 加载 ES2015+ 代码，然后使用 `Traceur` 转译为 ES5
- `ts-loader` 或 `awesome-typescript-loader` 像 JavaScript 一样加载 `TypeScript` 2.0+
- `coffee-loader` 像 JavaScript 一样加载 `CoffeeScript`
- `fengari-loader` 使用 `fengari` 加载 Lua 代码

模板(templateing)

- `html-loader` 导出 HTML 为字符串，需要引用静态资源
- `pug-loader` 加载 Pug 模板并返回一个函数
- `markdown-loader` 将 Markdown 转译为 HTML
- `react-markdown-loader` 使用 `markdown-parse parser`(解析器) 将 Markdown 编译

为 React 组件

- `posthtml-loader` 使用 `PostHTML` 加载并转换 HTML 文件
- `handlebars-loader` 将 Handlebars 转移为 HTML
- `markup-inline-loader` 将内联的 SVG/MathML 文件转换为 HTML。在应用于图标字体，或将 CSS 动画应用于 SVG 时非常有用。
- `twig-loader` 编译 Twig 模板，然后返回一个函数

样式

- `style-loader` 将模块的导出作为样式添加到 DOM 中
- `css-loader` 解析 CSS 文件后，使用 import 加载，并且返回 CSS 代码
- `less-loader` 加载和转译 LESS 文件
- `sass-loader` 加载和转译 SASS/SCSS 文件
- `postcss-loader` 使用 `PostCSS` 加载和转译 CSS/SSS 文件
- `stylus-loader` 加载和转译 Stylus 文件

代码检查和测试(linting && testing)

- `mocha-loader` 使用 `mocha` 测试（浏览器/NodeJS）
- `eslint-loader` PreLoader，使用 `ESLint` 清理代码
- `jshint-loader` PreLoader，使用 `JSHint` 清理代码
- `jscs-loader` PreLoader，使用 `JSCS` 检查代码样式
- `coverjs-loader` PreLoader，使用 `CoverJS` 确定测试覆盖率

框架(frameworks)

- `vue-loader` 加载和转译 `Vue` 组件
- `polymer-loader` 使用选择预处理器(preprocessor)处理，并且 `require()` 类似一等模块(first-class)的 Web 组件
- `angular2-template-loader` 加载和转译 `Angular` 组件

更多第三方 loader，查看 [awesome-webpack](#) 列表。

babel-loader

此 package 允许你使用 [Babel](#) 和 [webpack](#) 转译 JavaScript 文件。

注意：请在 [Babel Issues](#) tracker 上报告输出时遇到的问题。

中文文档

[Babel 中文文档](#)

安装

webpack 4.x | babel-loader 8.x | babel 7.x

```
npm install -D babel-loader @babel/core @babel/preset-env webpack
```

用法

webpack 文档: [loaders](#)

在 webpack 配置对象中，需要将 babel-loader 添加到 module 列表中，就像下面这样：

```
module: {
  rules: [
    {
      test: /\.m?js$/,
      exclude: /(node_modules|bower_components)/,
      use: [
        {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        }
      ]
    }
  ]
}
```

选项

查看 [babel 选项](#)。

你可以使用 `options` 属性，来向 loader 传递 options 选项：

```
module: {
```

```

rules: [
  {
    test: /\.m?js$/,
    exclude: /(node_modules|bower_components)/,
    use: [
      {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env'],
          plugins: ['@babel/plugin-proposal-object-rest-spread']
        }
      }
    ]
  }
]
}

```

此 loader 也支持下面这些 loader 特有的选项：

- `cacheDirectory`: 默认值为 `false`。当有设置时，指定的目录将用来缓存 loader 的执行结果。之后的 webpack 构建，将会尝试读取缓存，来避免在每次执行时，可能产生的、高性能消耗的 Babel 重新编译过程(recompilation process)。如果设置了一个空值 (`loader: 'babel-loader?cacheDirectory'`) 或者 `true` (`loader: 'babel-loader?cacheDirectory=true'`)，loader 将使用默认的缓存目录 `node_modules/.cache/babel-loader`，如果在任何根目录下都没有找到 `node_modules` 目录，将会降级回退到操作系统默认的临时文件目录。
- `cacheIdentifier`: 默认是由 `@babel/core` 版本号，`babel-loader` 版本号，`.babelrc` 文件内容（存在的情况下），环境变量 `BABEL_ENV` 的值（没有时降级到 `NODE_ENV`）组成的一个字符串。可以设置为一个自定义的值，在 `identifier` 改变后，来强制缓存失效。
- `cacheCompression`: 默认值为 `true`。当设置此值时，会使用 Gzip 压缩每个 Babel transform 输出。如果你想要退出缓存压缩，将它设置为 `false` -- 如果你的项目中有数千个文件需要压缩转译，那么设置此选项可能会从中收益。
- `customize`: 默认值为 `null`。导出 `custom` 回调函数的模块路径，例如传入 `.custom()` 的 `callback` 函数。由于你必须创建一个新文件才能使用它，建议改为使用 `.custom` 来创建一个包装 loader。只有在你必须继续直接使用 `babel-loader` 但又想自定义的情况下，才使用这项配置。

疑难解答

babel-loader 很慢！

确保转译尽可能少的文件。你可能使用 `/\.m?js$/` 来匹配，这样也许会去转译 `node_modules` 目录或者其他不需要的源代码。

要排除 `node_modules`，参考文档中的 `loaders` 配置的 `exclude` 选项。

你也可以通过使用 `cacheDirectory` 选项，将 babel-loader 提速至少两倍。这会将转译的结果缓存到文件系统中。

Babel 在每个文件都插入了辅助代码，使代码体积过大！

Babel 对一些公共方法使用了非常小的辅助代码，比如 `_extend`。默认情况下会被添加到每一个需要它的文件中

你可以引入 Babel runtime 作为一个独立模块，来避免重复引入。

下面的配置禁用了 Babel 自动对每个文件的 runtime 注入，而是引入 `@babel/plugin-transform-runtime` 并且使所有辅助代码从这里引用。

更多信息请查看 [文档](#)。

注意：你必须执行 `npm install -D @babel/plugin-transform-runtime` 来把它包含到你的项目中，然后使用 `npm install @babel/runtime` 把 `@babel/runtime` 安装为一个依赖。

```
rules: [
  // 'transform-runtime' 插件告诉 Babel
  // 要引用 runtime 来代替注入。
  {
    test: /\.m?js$/,
    exclude: /(node_modules|bower_components)/,
    use: {
      loader: 'babel-loader',
      options: {
        presets: ['@babel/preset-env'],
        plugins: ['@babel/plugin-transform-runtime']
      }
    }
  }
]
```

注意： transform-runtime 和自定义 polyfills (例如 Promise library)

由于 `@babel/plugin-transform-runtime` 包含了一个 polyfill，含有自定义的 `regenerator-runtime` 和 `core-js`，下面使用 `webpack.ProvidePlugin` 来配置 shimming 的常用方法将没有作用：

```
// ...
  new webpack.ProvidePlugin({
    'Promise': 'bluebird'
  }),
// ...
```

下面这样的写法也没有作用：

```
require('@babel/runtime/core-js/promise').default = require('bluebird'),  
var promise = new Promise;
```

它其实会生成下面这样 (使用了 `runtime` 后):

```
'use strict';  
  
var _Promise = require('@babel/runtime/core-js/promise')['default'];  
require('@babel/runtime/core-js/promise')['default'] = require('bluebird');  
  
var promise = new _Promise();
```

前面的 `Promise` library 在被覆盖前已经被引用和使用了。

一种可行的办法是，在你的应用程序中加入一个“引导(bootstrap)”步骤，在应用程序开始前先覆盖默认的全局变量。

```
// bootstrap.js  
  
require('@babel/runtime/core-js/promise').default = require('bluebird'),  
// ...  
  
require('./app');
```

babel 的 Node.js API 已经被移到 babel-core 中。 (原文: The Node.js API for babel has been moved to babel-core.)

如果你收到这个信息，这说明你有一个已经安装的 `babel` npm package，并且在 webpack 配置中使用 loader 简写方式（在 webpack 2.x 版本中将不再支持这种方式）。

```
{  
  test: /\.m?js$/,  
  loader: 'babel',  
}
```

webpack 将尝试读取 `babel` package 而不是 `babel-loader`。

想要修复这个问题，你需要卸载 `babel` npm package，因为它在 Babel v6 中已经被废除。（安装 `@babel/cli` 或者 `@babel/core` 来替代它）在另一种场景中，如果你的依赖于 `babel` 而无法删除它，可以在 webpack 配置中使用完整的 loader 名称来解决：

```
{  
  test: /\.m?js$/,  
  loader: 'babel-loader',  
}
```

自定义 loader

`babel-loader` 提供了一个 `loader-builder` 工具函数，允许用户为 Babel 处理过的每个文件添加自定义处理选项。

`.custom` 接收一个 `callback` 函数，它将被调用，并传入 `loader` 中的 `babel` 实例，因此，此工具函数才能够完全确保它使用与 `loader` 的 `@babel/core` 相同的实例。

如果你想自定义，但实际上某个文件又不想调用 `.custom`，可以向 `customize` 选项传入一个字符串，此字符串指向一个导出 `custom` 回调函数的文件。

示例

```
// 从 "./my-custom-loader.js" 中导出，或者任何你想要的文件中导出。
module.exports = require("babel-loader").custom(babel => {
  function myPlugin() {
    return {
      visitor: {},
    };
  }

  return {
    // 传给 loader 的选项。
    customOptions({ opt1, opt2, ...loader }) {
      return {
        // 获取 loader 可能会有的自定义选项
        custom: { opt1, opt2 },

        // 传入"移除了两个自定义选项"后的选项
        loader,
      };
    },
  },
  // 提供 Babel 的 'PartialConfig' 对象
  config(cfg) {
    if (cfg.hasFilesystemConfig()) {
      // 使用正常的配置
      return cfg.options;
    }

    return {
      ...cfg.options,
      plugins: [
        ...(cfg.options.plugins || []),
        // 在选项中包含自定义 plugin
        myPlugin,
      ],
    };
  },
  result(result) {
    return {

```

```
    ...result,
    code: result.code + "\n// Generated by some custom loader",
  );
},
);
});

// 然后, 在你的 webpack config 文件中
module.exports = {
  // ..
  module: {
    rules: [
      // ...
      loader: path.join(__dirname, 'my-custom-loader.js'),
      // ...
    ]
  }
};
}
```

customOptions(options: Object): { custom: Object, loader: Object }

指定的 loader 的选项, 从 `babel-loader` 选项中分离出自定义选项。

config(cfg: PartialConfig): Object

指定的 Babel 的 `PartialConfig` 对象, 返回应该被传递给 `babel.transform` 的 `option` 对象。

result(result: Result): Result

指定的 Babel 结果对象, 允许 loaders 对它进行额外的调整。

License

[MIT](#)

yaml-frontmatter-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

YAML frontmatter loader for webpack. Converts YAML in files to JSON.

Requirements

This module requires a minimum of Node v6.9.0 and Webpack v4.0.0.

Getting Started

To begin, you'll need to install `yaml-frontmatter-loader`:

```
$ npm install yaml-frontmatter-loader --save-dev
```

Then add the loader to your `webpack` config. For example:

```
const json = require('yaml-frontmatter-loader!./file.md');
// => returns file.md as javascript object

// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.md$/,
        use: [ 'json-loader', 'yaml-frontmatter-loader' ]
      }
    ]
  }
}
```

And run `webpack` via your preferred method.

License

[MIT](#) [MIT](#)

cache-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![coverage][cover]][cover-url] [![chat][chat]][chat-url] [![size][size]][size-url]

The `cache-loader` allow to Caches the result of following loaders on disk (default) or in the database.

起步

To begin, you'll need to install `cache-loader`:

```
npm install --save-dev cache-loader
```

在一些性能开销较大的 loader 之前添加此 loader，以将结果缓存到磁盘里。

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.ext$/,
        use: ['cache-loader', ...loaders],
        include: path.resolve('src'),
      },
    ],
  },
};
```

□□ 请注意，保存和读取这些缓存文件会有一些时间开销，所以请只对性能开销较大的 loader 使用此 loader。

选项

Name	cacheContext
Type	{String}
Default	undefined
Description	Allows you to override the default cache context in order to generate the cache relatively to a path. By default it will use absolute paths
Name	cacheKey
Type	{Function(options, request) -> {String}}

Default	undefined
Description	Allows you to override default cache key generator
Name	cacheDirectory
Type	{String}
Default	path.resolve('.cache-loader')
Description	Provide a cache directory where cache items should be stored (used for default read/write implementation)
Name	cacheIdentifier
Type	{String}
Default	cache-loader:{version} {process.env.NODE_ENV}
Description	Provide an invalidation identifier which is used to generate the hashes. You can use it for extra dependencies of loaders (used for default read/write implementation)
Name	write
Type	{Function(cacheKey, data, callback) -> {void}}
Default	undefined
Description	Allows you to override default write cache data to file (e.g. Redis, memcached)
Name	read
Type	{Function(cacheKey, callback) -> {void}}
Default	undefined
Description	Allows you to override default read cache data from file
Name	readOnly
Type	{Boolean}
Default	false
Description	Allows you to override default value and make the cache read only (useful for some environments where you don't want the cache to be updated, only read from it)

示例

Basic

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        use: ['cache-loader', 'babel-loader'],
        include: path.resolve('src'),
      },
    ],
  },
};
```

Database Integration

webpack.config.js

```
// Or different database client - memcached, mongodb, ...
const redis = require('redis');
const crypto = require('crypto');

// ...
// connect to client
// ...

const BUILD_CACHE_TIMEOUT = 24 * 3600; // 1 day

function digest(str) {
  return crypto
    .createHash('md5')
    .update(str)
    .digest('hex');
}

// Generate own cache key
function cacheKey(options, request) {
  return `build:cache:${digest(request)}`;
}

// Read data from database and parse them
function read(key, callback) {
  client.get(key, (err, result) => {
    if (err) {
      return callback(err);
    }

    if (!result) {
      return callback(new Error(`Key ${key} not found`));
    }

    try {
      let data = JSON.parse(result);
    }
  });
}
```

```
        callback(null, data);
    } catch (e) {
        callback(e);
    }
}

// Write data to database under cacheKey
function write(key, data, callback) {
    client.set(key, JSON.stringify(data), 'EX', BUILD_CACHE_TIMEOUT, callback);
}

module.exports = {
    module: {
        rules: [
            {
                test: /\.js$/,
                use: [
                    {
                        loader: 'cache-loader',
                        options: {
                            cacheKey,
                            read,
                            write,
                        },
                    },
                    'babel-loader',
                ],
                include: path.resolve('src'),
            },
        ],
    },
};
```

贡献

Please take a moment to read our contributing guidelines if you haven't yet done so.

CONTRIBUTING

License

MIT

coffee-loader

就像加载 JavaScript 那样，加载 CoffeeScript

安装

```
npm install --save-dev coffee-loader
```

用法

```
import coffee from 'coffee-loader!./file.coffee';  
##  
import coffee from 'file.coffee';
```

webpack.config.js

```
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.coffee$/,  
        use: [ 'coffee-loader' ]  
      }  
    ]  
  }  
}
```

选项

名称	literate
默认	false
描述	在 markdown (代码块) 中启用 CoffeeScript, 例如 file.coffee.md
名称	transpile
默认	false
描述	提供 Babel 预设(preset)和插件(plugin)

[**Literate** Literate](#)" class="icon-link" href="#literate">>

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.coffee\.md$/,
        use: [
          {
            loader: 'coffee-loader',
            options: { literate: true }
          }
        ]
      }
    ]
  }
}
```

Sourcemaps

source maps 总是产生。

[Transpile](#) Transpile" class="icon-link" href="#transpile">>

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.coffee$/,
        use: [
          {
            loader: 'coffee-loader',
            options: {
              transpile: {
                presets: ['env']
              }
            }
          }
        ]
      }
    ]
  }
}
```

coffee-redux-loader

Coffee Script Redux loader for Webpack.

安装

```
npm i -D coffee-redux-loader
```

用法

```
var exportsOfFile = require("coffee-redux-loader!./file.coffee");  
// => return exports of executed and compiled file.coffee
```

如果你想要在 node 运行环境中使用，不要忘了 polyfill `require`。请查看 `webpack` 文档。

Maintainers



[Juho Vepsäläinen](#) [Joshua Wiens](#) [Kees Kluskens](#) [Sean Larkin](#)

config-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

DEPRECATED. `webpack-command` is also deprecated. Please use `webpack-cli`. If any features were not implemented in `webpack-cli` feel free to create issue.

Why deprecated `webpack-command`?

- `webpack-cli` is very stable and have more features.
- Two CLIs are misleading for developers.
- Hard to maintain two package with same purpose.
- The author stopped developing the package.
- Most of the features are already implemented in `webpack-cli`.

Thanks for using `webpack`! We apologize for the inconvenience. In the future, we will avoid such situations.

A webpack configuration loader.

This module utilizes `cosmiconfig` which supports declaring a webpack configuration in a number of different file formats including; `.webpackrc`, `webpack.config.js`, and a `webpack` property in a `package.json`.

`config-loader` supports configuration modules which export an `Object`, `Array`, `Function`, `Promise`, and `Function` which returns a `Promise`.

The module also validates found configurations against webpack's options schema to ensure that the configuration is correct before webpack attempts to use it.

Requirements

This module requires a minimum of Node v6.9.0 and Webpack v4.0.0.

Getting Started

To begin, you'll need to install `config-loader`:

```
$ npm install @webpack-contrib/config-loader --save-dev
```

And get straight to loading a config:

```
const loader = require('@webpack-contrib/config-loader');
const options = { ... };

loader(options).then((result) => {
  // ...
  // result = { config: Object, configPath: String }
}) ;
```

Extending Configuration Files

This module supports extending webpack configuration files with [ESLint-style extends](#) functionality. This feature allows users to create a "base" config and in essence, "inherit" from that base config in a separate config. A bare-bones example:

```
// base.config.js
module.exports = {
  name: 'base',
  mode: 'development',
  plugins: [...]
}

// webpack.config.js
module.exports = {
  extends: path.join(..., 'base-config.js'),
  name: 'dev'
```

The resulting configuration object would resemble:

```
// result
{
  name: 'dev',
  mode: 'development',
  plugins: [...]
}
```

The `webpack.config.js` file will be intelligently extended with properties from `base.config.js`.

The `extends` property also supports naming installed NPM modules which export webpack configurations. Various configuration properties can also be filtered in different ways based on need.

[Read More about Extending Configuration Files](#)

Gotchas

Function-Config Parameters

When using a configuration file that exports a `Function`, users of `webpack-cli` have become accustom to the function signature:

```
function config (env, argv)
```

`webpack-cli` provides any CLI flags prefixed with `--env` as a single object in the `env` parameter, which is an unnecessary feature. Environment Variables have long served the same purpose, and are easily accessible within a Node environment.

As such, `config-loader` does not call `Function` configs with the `env` parameter. Rather, it makes calls with only the `argv` parameter.

Extending Configuration Files in Symlinked Modules

When using `extends` to extend a configuration which exists in a different package, care must be taken to ensure you don't hit module resolution issues if you are developing with these packages with symlinks (i.e. with `npm link` or `yarn link`).

By default, Node.js does not search for modules through symlinks, and so you may experience errors such as:

```
module not found: Error: Can't resolve 'webpack-hot-client/client'
```

This can be fixed by using Node's `--preserve-symlinks` flag which will allow you to develop cross-module, without experiencing inconsistencies when comparing against a normal, non-linked install:

For `webpack-command`:

```
node --preserve-symlinks ./node_modules/.bin/wp
```

For `webpack-serve`:

```
node --preserve-symlinks ./node_modules/.bin/webpack-serve
```

Supported Compilers

This module can support non-standard JavaScript file formats when a compatible compiler is registered via the `require` option. If the option is defined, `config-loader` will attempt to require the specified module(s) before the target config is found and loaded.

As such, `config-loader` will also search for the following file extensions; `.js`, `.es6`, `.flow`, `.mjs`, and `.ts`.

The module is also tested with the following compilers:

- [babel-register](#)
- [flow-remove-types/register](#)
- [ts-node/register](#)

Note: Compilers are not part of or built-into this module. To use a specific compiler, you must install it and specify its use by using the --require CLI flag.

API

loader([options])

Returns a `Promise`, which resolves with an `Object` containing:

config

Type: `Object`

Contains the actual configuration object.

configPath

Type: `String`

Contains the full, absolute filesystem path to the configuration file.

Options

allowMissing

Type: `Boolean`

Default: `false`

Instructs the module to allow a missing config file, and returns an `Object` with empty `config` and `configPath` properties in the event a config file was not found.

configPath

Type: `String` Default: `undefined`

Specifies an absolute path to a valid configuration file on the filesystem.

cwd

Type: `String` Default: `process.cwd()`

Specifies an filesystem path from which point `config-loader` will begin looking for a configuration file.

require

Type: `String | Array[String]` Default: `undefined`

Specifies compiler(s) to use when loading modules from files containing the configuration. For example:

```
const loader = require('@webpack-contrib/config-loader');
const options = { require: 'ts-node/register' };

loader(options).then((result) => { ... });
```

See [Supported Compilers](#) for more information.

schema

Type: `Object` Default: `undefined`

An object containing a valid [JSON Schema Definition](#).

By default, `config-loader` validates your webpack config against the [webpack config schema](#). However, it can be useful to append additional schema data to allow configs, which contain properties not present in the webpack schema, to pass validation.

Contributing

Please take a moment to read our contributing guidelines if you haven't yet done so.

[**CONTRIBUTING**](#)

License

[**MIT**](#)

coverjs-loader

用法

```
webpack-dev-server "mocha!./cover-my-client-tests.js" --options webpack{
  // webpackOptions.js
  module.exports = {
    // 你的 webpack options
    output: "bundle.js",
    publicPrefix: "/",
    debug: true,
    includeFilenames: true,
    watch: true,

    // 绑定 coverjs loader
    postLoaders: [
      {
        test: "", // 所有文件
        exclude: [
          "node_modules.chai",
          "node_modules.coverjs-loader",
          "node_modules.webpack.buildin"
        ],
        loader: "coverjs-loader"
      }
    ]
  }

  // cover-my-client-tests.js
  require("./my-client-tests");

  after(function() {
    require("cover-loader").reportHtml();
  });
}
```

参考示例 [the-big-test](#)。

这是一个独立的 loader，你不必一定把它和 mocha loader 结合一起使用。如果你想 cover 一个普通的项目，也可以直接使用它。`reportHtml` 方法会把输出内容添加到 body 中。

License

MIT (<http://www.opensource.org/licenses/mit-license.php>)

css-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![coverage][cover]][cover-url] [![chat][chat]][chat-url] [![size][size]][size-url]

The `css-loader` interprets `@import` and `url()` like `import/require()` and will resolve them.

起步

To begin, you'll need to install `css-loader`:

```
npm install --save-dev css-loader
```

Then add the plugin to your `webpack` config. For example:

file.js

```
import css from 'file.css';
```

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader'],
      },
    ],
  },
};
```

对于这些引用资源，你应该在配置中指定的，比较合适的 loader 是 `file-loader` 和 `url-loader`（查看如下设置）。

And run `webpack` via your preferred method.

toString

你也可以直接将 `css-loader` 的结果作为字符串使用，例如 Angular 的组件样式。

webpack.config.js

```
module.exports = {
  module: {
```

```

rules: [
  {
    test: /\.css$/,
    use: ['to-string-loader', 'css-loader'],
  },
],
},
];

```

或者

```

const css = require('./test.css').toString();

console.log(css); // {String}

```

如果有 SourceMap，它们也将包含在字符串结果中。

如果由于某种原因，你需要将 CSS 提取为纯粹的字符串资源（即不包含在 JS 模块中），则可能需要查看 [extract-loader](#)。例如，当你需要将 CSS 作为字符串进行后处理时，这很有用。

webpack.config.js

```

module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'handlebars-loader', // handlebars loader expects raw resource
          'extract-loader',
          'css-loader',
        ],
      },
    ],
  },
};

```

选项

名称	<u>url</u>
类型	{Boolean Function}
默认值	true
描述	启用/禁用 url() 处理
名称	<u>import</u>
类型	{Boolean \ /Function}
默认值	true
描述	启用/禁用 @import 处理

名称	<u>modules</u>
类型	{Boolean\Function}
默认值	false
描述	启用/禁用 CSS 模块和设置模式
名称	<u>localIdentName</u>
类型	{String}
默认值	[hash:base64]
描述	配置生成资源的标识符名称
名称	<u>context</u>
类型	{String}
默认值	undefined
描述	Allow to redefine basic loader context for local ident name
名称	<u>hashPrefix</u>
类型	{String}
默认值	undefined
描述	Allow to add custom hash to generate more unique classes
名称	<u>getLocalIdent</u>
类型	{Function}
默认值	undefined
描述	Configure the function to generate classname based on a different schema
名称	<u>sourceMap</u>
类型	{Boolean}
默认值	false
描述	启用/禁用 sourcemap
名称	<u>camelCase</u>
类型	{Boolean String}
默认值	false
描述	以驼峰式命名导出类名
名称	<u>importLoaders</u>
类型	{Number}
默认值	0
描述	在 css-loader 前应用的 loader 的数量

名称	<u>exportOnlyLocals</u>
类型	{Boolean}
默认值	false
描述	Export only locals

url

类型: Boolean | Function 默认: true

Control url() resolving. Absolute URLs and root-relative URLs are not resolving.

Examples resolutions:

```
url('image.png') => require('./image.png')
url('image.png') => require('./image.png')
url('./image.png') => require('./image.png')
url('./image.png') => require('./image.png')
url('http://dontwritehorriblecode.com/2112.png') => require('http://don't
image-set(url('image2x.png') 1x, url('image1x.png') 2x) => require('./ir
```

To import assets from a node_modules path (include resolve.modules) and for alias, prefix it with a ~:

```
url('~module/image.png') => require('module/image.png')
url('~/module/image.png') => require('module/image.png')
url(~aliasDirectory/image.png) => require('otherDirectory/image.png')
```

Boolean

Enable/disable url() resolving.

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        loader: 'css-loader',
        options: {
          url: true,
        },
      },
    ],
  },
};
```

Function

Allow to filter `url()`. All filtered `url()` will not be resolved (left in the code as they were written).

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        loader: 'css-loader',
        options: {
          url: (url, resourcePath) => {
            // resourcePath - path to css file

            // `url()` with `img.png` stay untouched
            return url.includes('img.png');
          },
        },
      },
    ],
  },
};
```

import

类型: Boolean **默认:** true

Control `@import` resolving. Absolute urls in `@import` will be moved in runtime code.

Examples resolutions:

```
@import 'style.css' => require('./style.css')
@import url(style.css) => require('./style.css')
@import url('style.css') => require('./style.css')
@import './style.css' => require('./style.css')
@import url(./style.css) => require('./style.css')
@import url('./style.css') => require('./style.css')
@import url('http://dontwritehorriblecode.com/style.css') => @import ur:
```

To import styles from a `node_modules` path (include `resolve.modules`) and for alias, prefix it with a ~:

```
@import url(~module/style.css) => require('module/style.css')
@import url('~module/style.css') => require('module/style.css')
@import url(~aliasDirectory/style.css) => require('otherDirectory/style
```

Boolean

Enable/disable `@import` resolving.

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        loader: 'css-loader',
        options: {
          import: true,
        },
      },
    ],
  },
};
```

Function

Allow to filter `@import`. All filtered `@import` will not be resolved (left in the code as they were written).

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        loader: 'css-loader',
        options: {
          import: (parsedImport, resourcePath) => {
            // parsedImport.url - url of `@import`
            // parsedImport.media - media query of `@import`
            // resourcePath - path to css file

            // `@import` with `style.css` stay untouched
            return parsedImport.url.includes('style.css');
          },
        },
      ],
    },
  },
};
```

modules `modules` " class="icon-link" href="#modules">

类型: Boolean | String **默认:** false

The `modules` option enables/disables the **CSS Modules** spec and setup basic behaviour.

Name	true
Type	{ Boolean }

Description	Enables local scoped CSS by default (use local mode by default)
Name	false
Type	{Boolean}
Description	Disable the CSS Modules spec, all CSS Modules features (like <code>@value</code> , <code>:local</code> , <code>:global</code> and <code>composes</code>) will not work
Name	' local '
Type	{String}
Description	Enables local scoped CSS by default (same as <code>true</code> value)
Name	' global '
Type	{String}
Description	Enables global scoped CSS by default

Using `false` value increase performance because we avoid parsing **CSS Modules** features, it will be useful for developers who use vanilla css or use other technologies.

You can read about **modes** below.

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        loader: 'css-loader',
        options: {
          modules: true,
        },
      },
    ],
  },
};
```

Scope

Using `local` value requires you to specify `:global` classes. Using `global` value requires you to specify `:local` classes.

You can find more information [here](#).

Styles can be locally scoped to avoid globally scoping styles.

语法 :local(.className) 可以被用来在局部作用域中声明 className。局部的作用域标识符会以模块形式暴露出去。

使用 :local (无括号) 可以为此选择器启用局部模式。:global(.className) 可以用来声明一个明确的全局选择器。使用 :global (无括号) 可以将此选择器切换至全局模式。

loader 会用唯一的标识符(identifier)来替换局部选择器。所选择的唯一标识符以模块形式暴露出去。

```
:local(.className) {  
  background: red;  
}  
:local .className {  
  color: green;  
}  
:local(.className .subClass) {  
  color: green;  
}  
:local .className .subClass :global(.global-class-name) {  
  color: blue;  
}  
  
. _23_aKvs-b8bW2Vg3fwHozO {  
  background: red;  
}  
. _23_aKvs-b8bW2Vg3fwHozO {  
  color: green;  
}  
. _23_aKvs-b8bW2Vg3fwHozO . _13LGdX8RMStbBE9w-t0gZ1 {  
  color: green;  
}  
. _23_aKvs-b8bW2Vg3fwHozO . _13LGdX8RMStbBE9w-t0gZ1 .global-class-name {  
  color: blue;  
}
```

i□ 主要信息: 标识符被导出

```
exports.locals = {  
  className: '_23_aKvs-b8bW2Vg3fwHozO',  
  subClass: '_13LGdX8RMStbBE9w-t0gZ1',  
};
```

建议局部选择器使用驼峰式。它们在导入 JS 模块中更容易使用。

你可以使用 :local(#someId)，但不推荐这种用法。推荐使用 class 代替 id。

Composing

当声明一个局部类名时，你可以与另一个局部类名组合为一个局部类。

```
:local(.className) {
```

```
background: red;
color: yellow;
}

:local(.subClass) {
  composes: className;
  background: blue;
}
```

这不会导致 CSS 本身有任何更改，而是导出多个类名。

```
exports.locals = {
  className: '_23_aKvs-b8bW2Vg3fwHozO',
  subClass: '_13LGdX8RMStbBE9w-t0gZ1 _23_aKvs-b8bW2Vg3fwHozO',
};

._23_aKvs-b8bW2Vg3fwHozO {
  background: red;
  color: yellow;
}

._13LGdX8RMStbBE9w-t0gZ1 {
  background: blue;
}
```

Importing

从其他模块导入局部类名。

```
:local(.continueButton) {
  composes: button from 'library/button.css';
  background: red;
}

:local(.nameEdit) {
  composes: edit highlight from './edit.css';
  background: red;
}
```

要从多个模块导入，请使用多个 `composes:` 规则。

```
:local(.className) {
  composes: edit hightlight from './edit.css';
  composes: button from 'module/button.css';
  composes: classFromThisModule;
  background: red;
}
```

localIdentName

类型: `String` 默认: `[hash:base64]`

You can configure the generated ident with the `localIdentName` query parameter. See

[loader-utils's documentation](#) for more information on options.

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        loader: 'css-loader',
        options: {
          modules: true,
          localIdentName: '[path] [name]__[local]--[hash:base64:5]',
        },
      },
    ],
  },
};
```

context

类型: String 默认: undefined

Allow to redefine basic loader context for local ident name. By default we use `rootContext` of loader.

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        loader: 'css-loader',
        options: {
          modules: true,
          context: path.resolve(__dirname, 'context'),
        },
      },
    ],
  },
};
```

hashPrefix

类型: String 默认: undefined

Allow to add custom hash to generate more unique classes.

webpack.config.js

```
module.exports = {
```

```
module: {
  rules: [
    {
      test: /\.css$/,
      loader: 'css-loader',
      options: {
        modules: true,
        hashPrefix: 'hash',
      },
    },
  ],
},
};
```

getLocalIdent

类型: Function 默认: undefined

You can also specify the absolute path to your custom `getLocalIdent` function to generate classname based on a different schema. By default we use built-in function to generate a classname.

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        loader: 'css-loader',
        options: {
          modules: true,
          getLocalIdent: (context, localIdentName, localName, options) =
            return 'whatever_random_class_name';
        },
      },
    ],
  },
};
```

sourceMap

类型: Boolean 默认: false

设置 `sourceMap` 选项查询参数来引入 source map。

例如, `mini-css-extract-plugin` 能够处理它们。

默认情况下不启用它们, 因为它们会导致运行时的额外开销, 并增加了 bundle 大小 (JS source map 不会)。此外, 相对路径是错误的, 你需要使用包含服务器 URL

的绝对公用路径。

webpack.config.js

```
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.css$/,  
        loader: 'css-loader',  
        options: {  
          sourceMap: true,  
        },  
      },  
    ],  
  },  
};
```

camelCase

类型: Boolean|String 默认: false

默认情况下，导出 JSON 键值对形式的类名。如果想要驼峰化(camelize)类名（有助于在 JS 中使用），通过设置 css-loader 的查询参数 camelCase 即可实现。

名称	false
类型	{Boolean}
描述	Class names will be camelized, the original class name will not be removed from the locals
名称	true
类型	{Boolean}
描述	类名将被骆驼化
名称	'dashes'
类型	{String}
描述	只有类名中的破折号将被骆驼化
名称	'only'
类型	{String}
描述	在 0.27.1 中加入。类名将被骆驼化，初始类名将从局部移除
名称	'dashesOnly'
类型	{String}
描述	在 0.27.1 中加入。类名中的破折号将被骆驼化，初始类名将从局部移除

file.css

```
.class-name {  
}
```

file.js

```
import { className } from 'file.css';
```

webpack.config.js

```
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.css$/,  
        loader: 'css-loader',  
        options: {  
          camelCase: true,  
        },  
      },  
    ],  
  },  
};
```

importLoaders

类型: Number 默认: 0

查询参数 `importLoaders`, 用于配置「`css-loader` 作用于 `@import` 的资源之前」有多少个 loader。

webpack.config.js

```
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.css$/,  
        use: [  
          'style-loader',  
          {  
            loader: 'css-loader',  
            options: {  
              importLoaders: 2, // 0 => no loaders (default); 1 => postcss  
            },  
          },  
          'postcss-loader',  
          'sass-loader',  
        ],  
      },  
    ],  
  },  
};
```

```
},  
};
```

在模块系统（即 webpack）支持原始 loader 匹配后，此功能可能在将来会发生变化。

exportOnlyLocals

类型: Boolean 默认: false

Export only locals (**useful** when you use **css modules**). For pre-rendering with `mini-css-extract-plugin` you should use this option instead of `style-loader!css-loader` **in the pre-rendering bundle**. It doesn't embed CSS but only exports the identifier mappings.

webpack.config.js

```
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.css$/,  
        loader: 'css-loader',  
        options: {  
          exportOnlyLocals: true,  
        },  
      },  
    ],  
  },  
};
```

示例

资源

以下 `webpack.config.js` 可以加载 CSS 文件，将小体积 PNG/JPG/GIF/SVG 图像转为像字体那样的 Data URL 嵌入，并复制较大的文件到输出目录。

webpack.config.js

```
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.css$/,  
        use: ['style-loader', 'css-loader'],  
      },  
      {  
        test: /\.(png|jpg|gif|svg|eot|ttf|woff|woff2)$/,  
        loader: 'url-loader',  
      },  
    ],  
  },  
};
```

```
        options: {
          limit: 10000,
        },
      ],
    },
  };
}
```

提取

对于生产环境构建，建议从 bundle 中提取 CSS，以便之后可以并行加载 CSS/JS 资源。

- 可以通过使用 [mini-css-extract-plugin](#) 来实现，在生产环境模式运行中提取 CSS。
- As an alternative, if seeking better development performance and css outputs that mimic production. [extract-css-chunks-webpack-plugin](#) offers a hot module reload friendly, extended version of mini-css-extract-plugin. HMR real CSS files in dev, works like mini-css in non-dev

贡献

Please take a moment to read our contributing guidelines if you haven't yet done so.

[贡献指南](#)

License

[MIT](#)

eslint-loader

eslint loader for webpack

Install

```
$ npm install eslint-loader --save-dev
```

NOTE: You also need to install `eslint` from npm, if you haven't already:

```
$ npm install eslint --save-dev
```

Usage

In your webpack configuration

```
module.exports = {
  // ...
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "eslint-loader",
        options: {
          // eslint options (if necessary)
        }
      }
    ]
  }
  // ...
};
```

When using with transpiling loaders (like `babel-loader`), make sure they are in correct order (bottom to top). Otherwise files will be checked after being processed by `babel-loader`

```
module.exports = {
  // ...
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: ["babel-loader", "eslint-loader"]
      }
    ]
  }
  // ...
};
```

To be safe, you can use `enforce: "pre"` section to check source files, not modified by other loaders (like `babel-loader`)

```
module.exports = {
  // ...
  module: {
    rules: [
      {
        enforce: "pre",
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "eslint-loader"
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "babel-loader"
      }
    ]
  }
  // ...
};
```

Options

You can pass [eslint options](#) using standard webpack [loader options](#).

Note that the config option you provide will be passed to the `CLIEngine`. This is a different set of options than what you'd specify in `package.json` or `.eslintrc`. See the [eslint docs](#) for more detail.

fix (default: false)

This option will enable [ESLint autofix feature](#).

Be careful: this option will change source files.

cache (default: false)

This option will enable caching of the linting results into a file. This is particularly useful in reducing linting time when doing a full build.

This can either be a boolean value or the cache directory path(ex: `'./.eslint-loader-cache'`).

If `cache: true` is used, the cache file is written to the `./node_modules/.cache` directory. This is the recommended usage.

formatter (default: eslint stylish formatter)

Loader accepts a function that will have one argument: an array of eslint messages (object). The function must return the output as a string. You can use official eslint formatters.

```
module.exports = {
  entry: "...",
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "eslint-loader",
        options: {
          // several examples !

          // default value
          formatter: require("eslint/lib/formatters/stylish"),

          // community formatter
          formatter: require("eslint-friendly-formatter"),

          // custom formatter
          formatter: function(results) {
            // `results` format is available here
            // http://eslint.org/docs/developer-guide/nodejs-api.html#eslintrun

            // you should return a string
            // DO NOT USE console.*() directly !
            return "OUTPUT";
          }
        }
      ]
    }
  };
};
```

eslintPath (default: "eslint")

Path to eslint instance that will be used for linting.

If the eslintPath is a folder like a official eslint, or specify a formatter option. now you dont have to install eslint .

```
module.exports = {
  entry: "...",
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "eslint-loader",
        options: {
```

```
        eslintPath: path.join(__dirname, "reusable-eslint")
    }
}
]
}
} ;
```

Errors and Warning

By default the loader will auto adjust error reporting depending on eslint errors/warnings counts. You can still force this behavior by using `emitError` or `emitWarning` options:

`emitError (default: false)`

Loader will always return errors if this option is set to `true`.

```
module.exports = {
  entry: "...",
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "eslint-loader",
        options: {
          emitError: true
        }
      }
    ]
  }
};
```

`emitWarning (default: false)`

Loader will always return warnings if option is set to `true`. If you're using hot module replacement, you may wish to enable this in development, or else updates will be skipped when there's an eslint error.

`quiet (default: false)`

Loader will process and report errors only and ignore warnings if this option is set to true

```
module.exports = {
  entry: "...",
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "eslint-loader"
      }
    ]
  }
};
```

```
        loader: "eslint-loader",
        options: {
          quiet: true
        }
      ]
    }
};


```

failOnWarning (default: false)

Loader will cause the module build to fail if there are any eslint warnings.

```
module.exports = {
  entry: "...",
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "eslint-loader",
        options: {
          failOnWarning: true
        }
      }
    ]
  }
};


```

failOnError (default: false)

Loader will cause the module build to fail if there are any eslint errors.

```
module.exports = {
  entry: "...",
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "eslint-loader",
        options: {
          failOnError: true
        }
      }
    ]
  }
};


```

outputReport (default: false)

Write the output of the errors to a file, for example a checkstyle xml file for use for reporting on Jenkins CI

The `filePath` is relative to the webpack config: `output.path`. You can pass in a different formatter for the output file, if none is passed in the default/configured formatter will be used.

```
module.exports = {
  entry: "...",
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: "eslint-loader",
        options: {
          outputReport: {
            filePath: "checkstyle.xml",
            formatter: require("eslint/lib/formatters/checkstyle")
          }
        }
      }
    ]
  }
};
```

Gotchas

NoErrorsPlugin

`NoErrorsPlugin` prevents webpack from outputting anything into a bundle. So even ESLint warnings will fail the build. No matter what error settings are used for `eslint-loader`.

So if you want to see ESLint warnings in console during development using `WebpackDevServer` remove `NoErrorsPlugin` from webpack config.

Defining `configFile` or using `eslint -c path/.eslintrc`

Bear in mind that when you define `configFile`, `eslint` doesn't automatically look for `.eslintrc` files in the directory of the file to be linted. More information is available in official `eslint` documentation in section [*Using Configuration Files*](#). See #129.

Changelog [Changelog](#changelog)

License [License](#license)

exports-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

exports loader module for webpack

Requirements

This module requires a minimum of Node v6.9.0 and Webpack v4.0.0.

Getting Started

To begin, you'll need to install `exports-loader`:

```
$ npm install exports-loader --save-dev
```

Then add the loader to the desired `require` calls. For example:

```
require('exports-loader?file,parse=helpers.parse!./file.js');
// adds the following code to the file's source:
//   exports['file'] = file;
//   exports['parse'] = helpers.parse;

require('exports-loader?file!./file.js');
// adds the following code to the file's source:
//   module.exports = file;

require('exports-loader?[name]!./file.js');
// adds the following code to the file's source:
//   module.exports = file;
```

And run `webpack` via your preferred method.

Contributing

Please take a moment to read our contributing guidelines if you haven't yet done so.

[**CONTRIBUTING**](#)

License

[**MIT**](#)

expose-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

expose loader module for webpack

Requirements

This module requires a minimum of Node v6.9.0 and Webpack v4.0.0.

Getting Started

To begin, you'll need to install `expose-loader`:

```
$ npm install expose-loader --save-dev
```

Then add the loader to your `webpack` config. For example:

```
// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        use: [
          {
            loader: `expose-loader`,
            options: {...options}
          }
        ]
      }
    ]
  }
}
```

And then require the target file in your bundle's code:

```
// src/entry.js
require("expose-loader?libraryName!./thing.js");
```

And run `webpack` via your preferred method.

Examples

例如，假设你要将 jQuery 暴露至全局并称为 `$`:

```
require("expose-loader?$.jquery");
```

然后，`window.$` 就可以在浏览器控制台中使用。

或者，你可以通过配置文件来设置：

```
// webpack.config.js
module: {
  rules: [
    {
      test: require.resolve('jquery'),
      use: [
        {
          loader: 'expose-loader',
          options: '$'
        }
      ]
    }
}
```

除了暴露为 `window.$` 之外，假设你还想把它暴露为 `window.jQuery`。对于多个暴露，你可以在 loader 字符串中使用 `!:`

```
// webpack.config.js
module: {
  rules: [
    {
      test: require.resolve('jquery'),
      use: [
        {
          loader: 'expose-loader',
          options: 'jQuery'
        },
        {
          loader: 'expose-loader',
          options: '$'
        }
      ]
    }
}
```

`require.resolve` 调用是一个 Node.js 函数（与 webpack 处理流程中的 `require.resolve` 无关）。`require.resolve` 用来获取模块的绝对路径（`"./.../app/node_modules/react/react.js"`）。所以这里的暴露只会作用于 React 模块。并且只在 bundle 中使用到它时，才进行暴露。

Contributing

Please take a moment to read our contributing guidelines if you haven't yet done so.

[**CONTRIBUTING**](#)

License

[**MIT**](#)

extract-loader

extract-loader

webpack loader to extract HTML and CSS from the bundle.



The extract-loader evaluates the given source code on the fly and returns the result as string. Its main use-case is to resolve urls within HTML and CSS coming from their respective loaders. Use the [file-loader](#) to emit the extract-loader's result as separate file.

```
import stylesheetUrl from "file-loader!extract-loader!css-loader!main.css"
// stylesheetUrl will now be the hashed url to the final stylesheet
```

The extract-loader works similar to the [extract-text-webpack-plugin](#) and the [mini-css-extract-plugin](#) and is meant as a lean alternative to it. When evaluating the source code, it provides a fake context which was especially designed to cope with the code generated by the [html-](#) or the [css-loader](#). Thus it might not work in other situations.

Installation

```
npm install extract-loader
```

Examples

```
##
```

Bundling CSS with webpack has some nice advantages like referencing images and fonts with hashed urls or [hot module replacement](#) in development. In production, on the other hand, it's not a good idea to apply your stylesheets depending on JS execution. Rendering may be delayed or even a [FOUC](#) might be visible. Thus it's still better to have them as separate files in your final production build.

With the extract-loader, you are able to reference your `main.css` as regular entry. The following `webpack.config.js` shows how to load your styles with the [style-loader](#) in development and as separate file in production.

```

module.exports = ({ mode }) => {
  const pathToMainCss = require.resolve("./app/main.css");
  const loaders = [
    {
      loader: "css-loader",
      options: {
        sourceMap: true
      }
    }
  ];

  if (mode === "production") {
    loaders.unshift(
      "file-loader",
      "extract-loader"
    );
  } else {
    loaders.unshift("style-loader");
  }

  return {
    mode,
    entry: pathToMainCss,
    module: {
      rules: [
        {
          test: pathToMainCss,
          loaders: loaders
        },
      ]
    }
  };
};

```

Extracting the index.html Extracting the index.html" class="icon-link" href="#extracting-the-index-html">

You can even add your `index.html` as `entry` and just reference your stylesheets from there. You just need to tell the `html-loader` to also pick up `link:href:`

```

module.exports = ({ mode }) => {
  const pathToMainJs = require.resolve("./app/main.js");
  const pathToIndexHtml = require.resolve("./app/index.html");

  return {
    mode,
    entry: [
      pathToMainJs,
      pathToIndexHtml
    ],
    module: {
      rules: [
        {
          test: pathToIndexHtml,
          use: [
            "file-loader",
            "extract-loader",

```

```

        {
            loader: "html-loader",
            options: {
                attrs: ["img:src", "link:href"]
            }
        }
    ]
},
{
    test: /\.css$/,
    use: [
        "file-loader",
        "extract-loader",
        {
            loader: "css-loader",
            options: {
                sourceMap: true
            }
        }
    ]
},
{
    test: /\.jpg$/,
    use: "file-loader"
}
]
}
);
}
}

```

turns

```

<html>
<head>
    <link href="main.css" type="text/css" rel="stylesheet">
</head>
<body>
    
</body>
</html>

```

into

```

<html>
<head>
    <link href="7c57758b88216530ef48069c2a4c685a.css" type="text/css" re
</head>
<body>
    
</body>
</html>

```

Source Maps

If you want source maps in your extracted CSS files, you need to set the `sourceMap` option of the `css-loader`:

```
{  
  loader: "css-loader",  
  options: {  
    sourceMap: true  
  }  
}  
}
```

Options

There is currently exactly one option: `publicPath`. If you are using a relative `publicPath` in webpack's `output options` and extracting to a file with the `file-loader`, you might need this to account for the location of your extracted file. `publicPath` may be defined as a string or a function that accepts current `loader context` as single argument.

Example with `publicPath` option as a string:

```
module.exports = {  
  output: {  
    path: path.resolve("./dist"),  
    publicPath: "dist/"  
  },  
  module: {  
    rules: [  
      {  
        test: /\.css$/,  
        use: [  
          {  
            loader: "file-loader",  
            options: {  
              name: "assets/[name].[ext]",  
            },  
          },  
          {  
            loader: "extract-loader",  
            options: {  
              publicPath: "../",  
            },  
          },  
          {  
            loader: "css-loader",  
          },  
        ],  
      }  
    ]  
  }  
};
```

Example with publicPath option as a function:

```
module.exports = {
  output: {
    path: path.resolve("./dist"),
    publicPath: "dist/"
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          {
            loader: "file-loader",
            options: {
              name: "assets/[name].[ext]",
            },
          },
          {
            loader: "extract-loader",
            options: {
              // dynamically return a relative publicPath
              publicPath: (context) => `.../.repeat(path.:${path.dirname(context).split('/').length - 1})`,
            },
          },
          {
            loader: "css-loader",
          },
        ],
      },
    ],
  },
};
```

You need another option? Then you should think about:

Contributing

From opening a bug report to creating a pull request: **every contribution is appreciated and welcome**. If you're planning to implement a new feature or change the api please create an issue first. This way we can ensure that your precious work is not in vain.

All pull requests should have 100% test coverage (with notable exceptions) and need to pass all tests.

- Call `npm test` to run the unit tests
- Call `npm run coverage` to check the test coverage (using `istanbul`)

License

Unlicense

Sponsors



peerigon

file-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![coverage][cover]][cover-url] [![chat][chat]][chat-url] [![size][size]][size-url]

The `file-loader` resolves `import/require()` on a file into a url and emits the file into the output directory.

起步

你需要预先安装 `file-loader`:

```
$ npm install file-loader --save-dev
```

在一个 `bundle` 文件中 `import` (或 `require`) 目标文件:

file.js

```
import img from './file.png';
```

然后，在 `webpack` 配置中添加 loader。例如:

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(\png|jpg|gif)$/,
        use: [
          {
            loader: 'file-loader',
            options: {},
          },
        ],
      },
    ],
  },
};
```

然后，通过你偏爱的方式去运行 `webpack`。将 `file.png` 作为一个文件，生成到输出目录，（如果指定了选项，则使用指定的命名约定）并返回文件的 public URI。

i 默认情况下，生成文件的文件名，是文件内容的 MD5 哈希值，并会保留所引用资源的原始扩展名。

选项

name

类型: String|Function **默认:** '[hash].[ext]'

Specifies a custom filename template for the target file(s) using the query parameter `name`. For example, to emit a file from your `context` directory into the output directory retaining the full directory structure, you might use:

String

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(\png|jpg|gif)$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              name: '[path][name].[ext]',
            },
          },
        ],
      },
    ],
  },
};
```

Function

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(\png|jpg|gif)$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              name(file) {
                if (process.env.NODE_ENV === 'development') {
                  return '[path][name].[ext]';
                }

                return '[hash].[ext]';
              }
            }
          }
        ]
      }
    ]
  }
};
```

```
        },
        ],
        ],
        ],
        },
    };
};
```

i 默认情况下，文件会按照你指定的路径和名称输出同一目录中，且会使用相同的 URI 路径来访问文件。

outputPath

类型: String|Function **默认:** undefined

Specify a filesystem path where the target file(s) will be placed.

String

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(\png|jpg|gif)$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              outputPath: 'images',
            },
          },
        ],
      },
    ],
  },
};
```

Function

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(\png|jpg|gif)$/,
        use: [
          {
            loader: 'file-loader',
```

```
options: {
  outputPath: (url, resourcePath, context) => {
    // `resourcePath` is original absolute path to asset
    // `context` is directory where stored asset (`rootContext`)

    // To get relative path you can use
    // const relativePath = path.relative(context, resourcePath)

    if (/my-custom-image/.test(resourcePath)) {
      return `other_output_path/${url}`;
    }

    if (/images/.test(context)) {
      return `image_output_path/${url}`;
    }

    return `output_path/${url}`;
  },
},
],
},
],
},
],
},
};

};
```

publicPath

类型: String|Function 默认: webpack_public_path

Specifies a custom public path for the target file(s).

String

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(\png|jpg|gif)$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              publicPath: 'assets',
            },
          },
        ],
      },
    ],
  },
};
```

Function

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.png|jpg|gif$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              publicPath: (url, resourcePath, context) => {
                // `resourcePath` is original absolute path to asset
                // `context` is directory where stored asset (`rootContext`)

                // To get relative path you can use
                // const relativePath = path.relative(context, resourcePath);

                if (/my-custom-image\.png/.test(resourcePath)) {
                  return `other_public_path/${url}`;
                }

                if (/images/.test(context)) {
                  return `image_output_path/${url}`;
                }

                return `public_path/${url}`;
              },
            },
          },
        ],
      },
    ],
  },
};
```

context

类型: String 默认: context

Specifies a custom file context.

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.png|jpg|gif$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              context: 'project',
            }
          }
        ],
      },
    ],
  },
};
```

```
        },
        ],
        },
        ],
        },
    };
```

emitFile

类型: Boolean **默认:** true

如果是 true，生成一个文件（向文件系统写入一个文件）。如果是 false，loader 会返回 public URI，但不会生成文件。对于服务器端 package，禁用此选项通常很有用。

file.js

```
// bundle file
import img from './file.png';
```

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              emitFile: false,
            },
          },
        ],
      },
    ],
  },
};
```

regExp

类型: RegExp **默认:** undefined

Specifies a Regular Expression to one or many parts of the target file path. The capture groups can be reused in the `name` property using [N] placeholder.

file.js

```
import img from './customer01/file.png';
```

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(\png|jpg|gif)$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              regExp: /\(([a-z0-9]+)\)/[a-z0-9]+\.\png$/,
              name: '[1]-[name].[ext]',
            },
          },
        ],
      },
    ],
  },
};
```

如果 [0] 是使用，它将被替换成整个测试字符串，而 [1] 将包含您 regex 中的第一个捕获括号的值等等...

placeholders

Full information about placeholders you can find [here](#).

[ext]

类型: String 默认: file.extname

目标文件/资源的文件扩展名。

[name]

类型: String 默认: file.basename

文件/资源的基本名称。

[path]

类型: String 默认: file.directory

The path of the resource relative to the webpack/config context.

[folder]

类型: String **默认:** file.folder

The folder of the resource is in.

[emoji]

类型: String **默认:** undefined

A random emoji representation of content.

[emoji:<length>]

类型: String **默认:** undefined

Same as above, but with a customizable number of emojis

[hash]

类型: String **默认:** md5

指定生成文件内容哈希值的哈希方法。

[<hashType>:hash:<digestType>:<length>]

类型: String

The hash of options.content (Buffer) (by default it's the hex digest of the hash).

digestType

类型: String **默认:** 'hex'

The digest that the hash function should use. Valid values include: base26, base32, base36, base49, base52, base58, base62, base64, and hex.

hashType

类型: String **默认:** 'md5'

The type of hash that the has function should use. Valid values include: md5, sha1, sha256, and sha512.

length

类型: Number **默认:** undefined

Users may also specify a length for the computed hash.

[N]

类型: String **默认:** undefined

The n-th match obtained from matching the current file name against the `RegExp`.

示例

The following examples show how one might use `file-loader` and what the result would be.

file.js

```
import png from './image.png';
```

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(\png|jpg|gif)$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              name: 'dirname/[hash].[ext]',
            },
          },
        ],
      },
    ],
  },
};
```

结果:

```
# result
dirname/0dcbbfa701328ae351f.png
```

file.js

```
import png from './image.png';
```

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(\png|jpg|gif)$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              name: '[sha512:hash:base64:7].[ext]',
            },
          },
        ],
      },
    ],
  },
};
```

结果:

```
# result
gdyb21L.png
```

file.js

```
import png from './path/to/file.png';
```

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(\png|jpg|gif)$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              name: '[path][name].[ext]?[hash]',
            },
          },
        ],
      },
    ],
  },
};
```

结果:

```
# result
path/to/file.png?e43b20c069c4a01867c31e98cbce33c9
```

贡献

如果你从未阅读过我们的贡献指南，请在上面花点时间。

[贡献指南](#)

License

[MIT](#)

gzip-loader

gzip loader module for webpack

Enables loading gzipped resources.

安装

```
npm install --save-dev gzip-loader
```

用法

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.gz$/,
        enforce: 'pre',
        use: 'gzip-loader'
      }
    ]
  }
}
```

bundle.js

```
require("gzip-loader!./file.js.gz");
```

html-loader

Exports HTML as string. HTML is minimized when the compiler demands.

安装

```
npm i -D html-loader
```

用法

默认情况下，每个本地的 `` 都需要通过 `require('require('./image.png'))` 来进行加载。你可能需要在配置中为图片指定 loader（推荐 `file-loader` 或 `url-loader`）

你可以通过查询参数 `attrs`，来指定哪个标签属性组合(tag-attribute combination)应该被此 loader 处理。传递数组或以空格分隔的 `<tag>:<attribute>` 组合的列表。
(默认值: `attrs=img:src`)

If you use `<custom-elements>`, and lots of them make use of a `custom-src` attribute, you don't have to specify each combination `<tag>:<attribute>`: just specify an empty tag like `attrs=:custom-src` and it will match every element.

```
{  
  test: /\.html$/,
  use: [
    loader: 'html-loader',
    options: {
      attrs: [':data-src']
    }
  ]
}
```

要完全禁用对标签属性的处理（例如，如果你在客户端处理图片加载），你可以传入 `attrs=false`。

示例

使用此配置：

```
{
  module: {
    rules: [
      { test: /\.jpg$/, use: [ "file-loader" ] },
      { test: /\.png$/, use: [ "url-loader?mimetype=image/png" ] }
    ],
    output: {
      publicPath: "http://cdn.example.com/[hash]/"
    }
  }
}

<!-- file.html -->


require("html-loader!./file.html");

// => ''

require("html-loader?attrs=img:data-src!./file.html");

// => ''

require("html-loader?attrs=img:src img:data-src!./file.html");
require("html-loader?attrs[]=img:src&attrs[]=img:data-src!./file.html"),
// => '
require("html-loader?-attrs!./file.html");
// => ''
```

通过运行 `webpack --optimize-minimize` 来最小化

```
'<img src=http://cdn.example.com/49eba9f/a9f92ca.jpg
      data-src=data:image/png;base64,...>'
```

或者在 `webpack.conf.js` 的 `rule` 选项中指定 `minimize` 属性

```
module: {
  rules: [ {
    test: /\.html$/,
    use: [ {
      loader: 'html-loader',
      options: {
        minimize: true
      }
    }],
  }]
}
```

See [html-minifier's documentation](#) for more information on the available options.

The enabled rules for minimizing by default are the following ones:

- `removeComments`
- `removeCommentsFromCDATA`
- `removeCDATASectionsFromCDATA`
- `collapseWhitespace`
- `conservativeCollapse`
- `removeAttributeQuotes`
- `useShortDoctype`
- `keepClosingSlash`
- `minifyJS`
- `minifyCSS`
- `removeScriptTypeAttributes`
- `removeStyleTypeAttributes`

The rules can be disabled using the following options in your `webpack.conf.js`

```
module: {
  rules: [ {
    test: /\.html$/,
    use: [ {
      loader: 'html-loader',
      options: {

```

```
        minimize: true,
        removeComments: false,
        collapseWhitespace: false
    }
},
],
}
}

##
```

对于以 / 开头的 url， 默认行为是不转换它们。 如果设置了 root 查询参数， 它将被添加到 URL 之前， 然后进行转换。

和上面配置相同：

```


require("html-loader!./file.html");

// => ''

require("html-loader?root=!.!/file.html");

// => ''
```

插值

你可以使用 `interpolate` 标记， 为 ES6 模板字符串启用插值语法， 就像这样：

```
require("html-loader?interpolate!./file.html");



<div>${require('./components/gallery.html')}</div>
```

如果你只想在模板中使用 `require`， 任何其它的 `{}$` 不被转换， 你可以设置 `interpolate` 标记为 `require`， 就像这样：

```
require("html-loader?interpolate=require!./file.ftl");

<#list list as list>
    <a href="${list.href!}" />${list.name}</a>
</#list>



<div>${require('./components/gallery.html')}</div>
```

导出格式

这里有几种不同的可用导出格式：

- `module.exports` (默认配置, cjs 格式)。"Hello world" 转为 `module.exports = "Hello world";`
- `exports.default` (当设置了 `exportAsDefault` 参数, es6to5 格式)。"Hello world" 转为 `exports.default = "Hello world";`
- `export default` (当设置了 `exportAsEs6Default` 参数, es6 格式)。"Hello world" 转为 `export default "Hello world";`

高级选项

如果你需要传递更多高级选项, 特别是那些不能被字符串化, 你还可以在 `webpack.config.js` 中定义一个 `htmlLoader` 属性:

```
var path = require('path')

module.exports = {
  ...
  module: {
    rules: [
      {
        test: /\.html$/,
        use: [ "html-loader" ]
      }
    ]
  },
  htmlLoader: {
    ignoreCustomFragments: [/^\{\{.*?\}\}/],
    root: path.resolve(__dirname, 'assets'),
    attrs: ['img:src', 'link:href']
  }
};
```

如果你需要定义两个不同的 loader 配置, 你也可以通过 `html-loader?config=otherHtmlLoaderConfig` 改变配置的属性名:

```
module.exports = {
  ...
  module: {
    rules: [
      {
        test: /\.html$/,
        use: [ "html-loader?config=otherHtmlLoaderConfig" ]
      }
    ]
  },
  otherHtmlLoaderConfig: {
    ...
  }
};
```

导出到 HTML 文件

一个很常见的场景, 将 HTML 导出到 `.html` 文件中, 直接访问它们, 而不是使用

javascript 注入。这可以通过3个 loader 的组合来实现：

- [file-loader](#)
- [extract-loader](#)
- html-loader

html-loader 将解析 URL，并请求图片和你所期望的一切资源。extract-loader 会将 javascript 解析为合适的 html 文件，确保引用的图片指向正确的路径，file-loader 将结果写入 .html 文件。示例：

```
{  
  test: /\.html$/,
  use: ['file-loader?name=[name].[ext]', 'extract-loader', 'html-loader']
}
```

i18n-loader

用法

./colors.json

```
{  
  "red": "red",  
  "green": "green",  
  "blue": "blue"  
}
```

./de-de.colors.json

```
{  
  "red": "rot",  
  "green": "gr n"  
}
```

调用

```
// 假如我们的所在区域是 "de-de-berlin"
var locale = require("i18n!./colors.json");

// 等待准备就绪，在一个 web 项目中所有地区只需要一次
// 因为所有地区的语言被合并到一个块中
locale(function() {
  console.log(locale.red); // 输出 rot
  console.log(locale.blue); // 输出 blue
});
```

配置

如果想要一次加载然后可以同步地使用，你应该告诉 loader 所有要使用的地区。

```
{  
  "i18n": {  
    "locales": [  
      "de",  
      "de-de",  
      "fr"  
    ],  
    // "bundleTogether": false  
    // 可以禁止所有地区打包到一起  
  }  
}
```

可选的调用方法

```
require("i18n/choose!./file.js"); // 根据地区选择正确的文件,  
                                  // 但是不会合并到对象中  
require("i18n/concat!./file.js"); // 拼接所有合适的地区  
require("i18n/merge!./file.js"); // 合并到对象中  
                                // ./file.js 在编译时会被排除掉  
require("i18n!./file.json") == require("i18n/merge!json!./file.json")
```

如果需要在 node 中使用，不要忘记填补 (polyfill) require。可以参考 webpack 文档。

License

MIT (<http://www.opensource.org/licenses/mit-license.php>)

imports-loader

The imports loader allows you to use modules that depend on specific global variables.

This is useful for third-party modules that rely on global variables like `$` or `this` being the `window` object. The imports loader can add the necessary `require('whatever')` calls, so those modules work with webpack.

安装

```
npm install imports-loader
```

用法

假设你有 `example.js` 这个文件

```
$( "img" ).doSomeAwesomeJqueryPluginStuff();
```

然后你可以像下面这样通过配置 `imports-loader` 插入 `$` 变量到模块中：

```
require("imports-loader?$=jquery!./example.js");
```

这将简单的把 `var $ = require("jquery");` 前置插入到 `example.js` 中。

```
##
```

loader 查询值 含义	angular <code>var angular = require("angular");</code>
loader 查询值 含义	<code>\$=jquery</code> <code>var \$ = require("jquery");</code>
loader 查询值 含义	<code>define=>false</code> <code>var define = false;</code>
loader 查询值 含义	<code>config=>{size:50}</code> <code>var config = {size:50};</code>
loader 查询值 含义	<code>this=>window</code> <code>(function () { ... }).call(window);</code>

多个值

使用逗号，来分隔和使用多个值：

```
require("imports-loader?$/=jquery,angular,config=>{size:50}!./file.js");
```

webpack.config.js

同样的，在你的 `webpack.config.js` 配置文件中进行配置会更好：

```
// ./webpack.config.js

module.exports = {
  ...
  module: {
    rules: [
      {
        test: require.resolve("some-module"),
        use: "imports-loader?this=>window"
      }
    ]
  }
};
```

典型的使用场景

jQuery 插件

```
imports-loader?$/=jquery
```

自定义的 Angular 模块

```
imports-loader?angular
```

禁用 AMD

有很多模块在使用 CommonJS 前会进行 `define` 函数的检查。自从 webpack 两种格式都可以使用后，在这种场景下默认使用了 AMD 可能会造成某些问题（如果接口的实现比较古怪）。

你可以像下面这样轻松的禁用 AMD

```
imports-loader?define=>false
```

关于兼容性问题的更多提示，可以参考官方的文档 [Shimming Modules](#)。

istanbul-instrumenter-loader

Instrument JS files with [istanbul-lib-instrument](#) for subsequent code coverage reporting

安装

```
npm i -D istanbul-instrumenter-loader
```

用法

##

- [karma-webpack](#)
- [karma-coverage-istanbul-reporter](#)

结构

```
|- src
  |- components
    |- bar
      |- index.js
    |- foo/
      |- index.js
  |- test
    |- src
      |- components
        |- foo
          |- index.js
```

为生成所有组件（包括你没写测试的那些）的代码覆盖率报告，你需要 require 所有业务和测试的代码。相关内容在 [karma-webpack](#) 其他用法中有涉及

test/index.js

```
// requires 所有在 `project/test/src/components/**/index.js` 中的测试
const tests = require.context('../src/components/', true, /index\.js$/);

tests.keys().forEach(tests);

// requires 所有在 `project/src/components/**/index.js` 中的组件
const components = require.context('../src/components/', true, /index\.js$/);

components.keys().forEach(components);
```

i 以下为 karma 的唯一入口起点文件

karma.conf.js

```
config.set({
  ...
  files: [
```

```

        'test/index.js'
    ],
    preprocessors: {
        'test/index.js': 'webpack'
    },
    webpack: {
        ...
        module: {
            rules: [
                // 用 Istanbul 只监测业务代码
                {
                    test: /\.js$/,
                    use: { loader: 'istanbul-instrumenter-loader' },
                    include: path.resolve('src/components/')
                }
            ]
        }
    }
    ...
},
reporters: [ 'progress', 'coverage-istanbul' ],
coverageIstanbulReporter: {
    reports: [ 'text-summary' ],
    fixWebpackSourcePaths: true
}
...
}) ;

```

使用 Babel

You must run the instrumentation as a post step

webpack.config.js

```
{
    test: /\.js$|\.jsx$/,
    use: {
        loader: 'istanbul-instrumenter-loader',
        options: { esModules: true }
    },
    enforce: 'post',
    exclude: /node_modules|\.spec\.js$/,
}
```

Options Options" [class="icon-link" href="#options">>](#options)

此 loader 支持 `istanbul-lib-instrument` 的所有配置选项

Name	debug
Type	{Boolean}
Default	false
Description	Turn on debugging mode

Name	compact
Type	{Boolean}
Default	true
Description	Generate compact code
Name	autoWrap
Type	{Boolean}
Default	false
Description	Set to <code>true</code> to allow return statements outside of functions
Name	esModules
Type	{Boolean}
Default	false
Description	Set to <code>true</code> to instrument ES2015 Modules
Name	coverageVariable
Type	{String}
Default	<code>__coverage__</code>
Description	Name of global coverage variable
Name	preserveComments
Type	{Boolean}
Default	false
Description	Preserve comments in <code>output</code>
Name	produceSourceMap
Type	{Boolean}
Default	false
Description	Set to <code>true</code> to produce a source map for the instrumented code
Name	sourceMapUrlCallback
Type	{Function}
Default	null
Description	A callback function that is called when a source map URL is found in the original code. This function is called with the source filename and the source map URL

webpack.config.js

```
{  
  test: /\.js$/,
  use: [
    loader: 'istanbul-instrumenter-loader',
    options: {...options}
  ]
}
```

Maintainers



Kir
Belevich

Juho Vepsäläinen

Joshua Wiens

Michael Ciniawsky

jshint-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

处理 JSHint 模块的 webpack loader。在构建时(build-time)，对 bundle 中的 JavaScript 文件，执行 JSHint 检查。

要求

此模块需要 Node v6.9.0+ 和 webpack v4.0.0+。

起步

你需要预先安装 jshint-loader：

```
$ npm install jshint-loader --save-dev
```

然后，在 webpack 配置中添加 loader。例如：

```
// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        enforce: 'pre',
        exclude: /node_modules/,
        use: [
          {
            loader: `jshint-loader`,
            options: {...options}
          }
        ]
      }
    ]
  }
}
```

然后，通过你偏爱的方式去运行 webpack。

选项

除了下面列出的自定义 loader 选项外，所有有效的 JSHint 选项在此对象中都有效：

delete options.; delete options.; delete options.;

emitErrors

类型: Boolean 默认值: undefined

命令 loader, 将所有 JSHint 警告和错误, 都作为 webpack 错误触发。

failOnHint

类型: Boolean 默认值: undefined

命令 loader, 在所有 JSHint 发生警告和错误时, 都产生 webpack 构建失败。

reporter

类型: Function 默认值: undefined

此函数用于对 JSHint 输出进行格式化, 也可以发出警告和错误。

自定义报告函数(custom reporter)

默认情况下, `jshint-loader` 自带一个默认报告函数。

然而, 如果你想设置自定义的报告函数, 向 `jshint` 选项的 `reporter` 属性传递一个函数 (查看[上面用法](#))

报告函数执行时, 会传入一个 JSHint 产生的, 由错误/警告构成的数组, 结构如下:

```
[  
{  
  id:      [字符串, 通常是 '(error)'],  
  code:    [字符串, 错误/警告编码(error/warning code)],  
  reason:  [字符串, 错误/警告消息(error/warning message)],  
  evidence: [字符串, 产生此错误的那段代码]  
  line:    [数字]  
  character: [数字]  
  scope:   [字符串, 消息作用域;  
            通常是 '(main)' 除非代码被解析过(eval)]  
  
  [+ 还有一些旧有字段, 不必关心。]  
,  
// ...  
// 更多的错误/警告  
]
```

报告函数执行时, loader context 会作为函数中的 `this`。你可以使用

`this.emitWarning(...)` 或 `this.emitError(...)` 来发出消息。查看 [webpack](#) 文档中关于 `loader context` 的部分。

注意: *JSHint reporter* 并不兼容 *JSHint-loader*! 这是因为,事实上 *reporter* 的输入,只能处理一个文件,而不能处理多个文件。以这种方式报告的错误,不同于用于 *JSHint* 的传统 *reporter* 报告的错误,这是因为,会对每个资源文件执行 *loader plugin* (也就是 *JSHint-loader*) ,因此 *reporter* 函数也会被每个文件执行。

webpack CLI 中的输出通常是:

```
...
WARNING in ./path/to/file.js
<reporter output>
...
`
```

贡献

如果你从未阅读过我们的贡献指南,请在上面花点时间。

[贡献指南](#) [贡献指南](#)" class="icon-link" href="#贡献指南">>

License

[MIT](#) [MIT](#)" class="icon-link" href="#mit">>

json-loader

安装

```
npm install --save-dev json-loader
```

□□ 注意：由于 `webpack >= v2.0.0` 默认支持导入 JSON 文件。如果你使用自定义文件扩展名，你可能仍然需要使用此 loader。See the [v1.0.0 -> v2.0.0 Migration Guide for more information](#)

用法

```
##  
  
const json = require('json-loader!./file.json');
```

通过配置（推荐）

```
const json = require('./file.json');
```

webpack.config.js

```
module.exports = {  
  module: {  
    loaders: [  
      {  
        test: /\.json$/,  
        loader: 'json-loader'  
      }  
    ]  
  }  
}
```

json5-loader

```
[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![cover][cover]][cover-url] [![chat][chat]][chat-url] [![size][size]][size-url]
```

A webpack loader for parsing `json5` files into JavaScript objects.

Getting Started

To begin, you'll need to install json5-loader:

```
$ npm install json5-loader --save-dev
```

你可以通过以下两种用法使用此 loader:

- 在 webpack 配置里的 module.loaders 对象中配置 json5-loader;
- 或者, 直接在 require 语句中使用 json5-loader! 前缀。

假设我们有如下 json5 文件:

file.json5

```
// file.json5
{
  env: 'production',
  passwordStrength: 'strong',
}
```

Usage with preconfigured loader

webpack.config.js

```
// webpack.config.js
module.exports = {
  entry: './index.js',
  output: {
    /* ... */
  },
  module: {
    loaders: [
      {
        // 使所有以 .json5 结尾的文件使用 `json5-loader`
        test: /\.json5$/,
        loader: 'json5-loader',
      },
    ],
  },
};

// index.js
var appConfig = require('./appData.json5');
// 或者 ES6 语法
// import appConfig from './appData.json5'

console.log(appConfig.env); // 'production'
```

require 语句使用 loader 前缀的用法

```
var appConfig = require('json5-loader!./appData.json5');
// 返回的是 json 解析过的对象

console.log(appConfig.env); // 'production'
```

如果需要在 Node.js 中使用，不要忘记兼容(polyfill) require。更多参考 webpack 文档。

贡献

Please take a moment to read our contributing guidelines if you haven't yet done so.

[贡献指南](#)

License

[MIT](#)

bundle-loader

webpack 的 bundle loader

安装

```
npm i bundle-loader --save
```

用法

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.bundle\.js$/,
        use: 'bundle-loader'
      }
    ]
  }
}
```

当你引用 bundle-loader 时， chunk 会被浏览器请求(request)。

file.js

```
import bundle from './file.bundle.js';
```

为了 chunk 在浏览器加载（以及在获取其导出）时可用时， 你需要异步等待。

```
bundle((file) => {
  // use the file like it was required
  const file = require('./file.js')
});
```

上述代码会将 `require('file.js')` 包裹在一段 `require.ensure` 代码块中

可以添加多个回调函数。它们会按照添加的顺序依次执行。

```
bundle(callbackTwo)
bundle(callbackThree)
```

当依赖模块都加载完毕时，如果此时添加一个回调函数，它将会立即执行。

选项(options)

名称	lazy
类型	{Boolean}
默认值	false
描述	异步加载导入的 bundle
名称	name
类型	{String}
默认值	[id].[name]
描述	为导入的 bundle 配置自定义文件名

##

当你使用 bundle-loader 时，文件会被请求(request)。如果想让它按需加载(request it lazy)，请使用：

webpack.config.js

```
{
  loader: 'bundle-loader',
  options: {
    lazy: true
  }
}

import bundle from './file.bundle.js'

bundle((file) => {...})
```

i 只有调用 load 函数时，chunk 才会被请求(request)

name

可以通过配置中 name 选项参数，来设置 bundle 的名称。查看 [文档](#)。

webpack.config.js

```
{
  loader: 'bundle-loader',
  options: {
    name: '[name]'
  }
}
```

:warning: 一旦 loader 创建了 chunk，它们将遵循以下命名规则

`output.chunkFilename` 规则，默认是 [id].[name]。这里 [name] 对应着配置中 name 选项参数设置的 chunk 名称。

示例

```
import bundle from './file.bundle.js'
```

webpack.config.js

```
module.exports = {
  entry: {
    index: './App.js'
  },
  output: {
    path: path.resolve(__dirname, 'dest'),
    filename: '[name].js',
    // 此处可以自定义其他格式
    chunkFilename: '[name].[id].js',
  },
  module: {
    rules: [
      {
        test: /\.bundle\.js$/,
        use: {
          loader: 'bundle-loader',
          options: {
            name: 'my-chunk'
          }
        }
      }
    ]
  }
}
```

一般情况下，chunk 会使用上面的 `filename` 规则，并根据其对应的 `[chunkname]` 命名。

然而，来自 `bundle-loader` 中的 chunk 会使用 `chunkFilename` 规则命名。因此，打包后的示例文件最终将生成为 `my-chunk.1.js` 和 `file-2.js`。

当然，你也可以在 `chunkFilename` 添加哈希值作为文件名的一部分，这是因为在 `bundle` 的配置选项中放置 `[hash]` 不会生效。

mocha-loader

Allows Mocha tests to be loaded and run via webpack

安装

```
npm i -D mocha-loader
```

用法

##

```
webpack --module-bind 'mocha-loader!./test'
```

要求

```
import test from 'mocha-loader!./test'
```

配置 (推荐)

```
import test from './test'
```

webpack.config.js

```
module.exports = {
  entry: './entry.js',
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  module: {
    rules: [
      {
        test: /test\.js$/,
        use: 'mocha-loader',
        exclude: /node_modules/
      }
    ]
  }
}
```

选项

None

示例

基本

module.js

```
module.exports = true
```

test.js

```
describe('Test', () => {
```

```
it('should succeed', (done) => {
  setTimeout(done, 1000)
})

it('should fail', () => {
  setTimeout(() => {
    throw new Error('Failed')
  }, 1000)
})

it('should randomly fail', () => {
  if (require('./module')) {
    throw new Error('Randomly failed')
  }
})
```

multi-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

用于分离模块和组合使用多个 loader 的 `webpack` loader。这个 loader 会多次引用一个模块，每次不同的 loader 加载这个模块，就像下面配置中定义的那样。

注意：在多入口，最后一项的 `exports` 会被导出。

要求

此模块需要 Node v6.9.0+ 和 webpack v4.0.0+。

起步

你需要预先安装 `multi-loader`:

```
$ npm install multi-loader --save-dev
```

然后，在 `webpack` 配置中添加 loader。例如：

```
// webpack.config.js
const multi = require('multi-loader');
{
  module: {
    loaders: [
      {
        test: /\.css$/,
        // 向 DOM 中添加 CSS，然后返回原始内容
        loader: multi(
          'style-loader!css-loader!autoprefixer-loader',
          'raw-loader'
        )
      }
    ]
  }
}
```

然后，通过你偏爱的方式去运行 `webpack`。

License

[MIT](#) [MIT](#)

node-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

A [Node.js add-ons](#) loader module for enhanced-require. Executes add-ons in [enhanced-require](#).

Requirements

This module requires a minimum of Node v6.9.0 and Webpack v4.0.0.

Getting Started

To begin, you'll need to install `node-loader`:

```
$ npm install node-loader --save-dev
```

Then add the loader to your `webpack` config. For example:

```
import node from 'file.node';

// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.node$/,
        use: 'node-loader'
      }
    ]
  }
}
```

Or on the command-line:

```
$ webpack --module-bind 'node=node-loader'
```

Inline

In your application

```
import node from 'node-loader!./file.node';
```

And run `webpack` via your preferred method.

License

[MIT](#) [MIT](#)" class="icon-link" href="#mit">>

null-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

返回一个空模块的 webpack loader。

此 loader 的一个用途是，使依赖项导入的模块静音。例如，项目依赖于一个 ES6 库，会导入不需要的 polyfill，因此删除它将不会导致功能损失。

要求

此模块需要 Node v6.9.0+ 和 webpack v4.0.0+。

起步

你需要预先安装 null-loader：

```
$ npm install null-loader --save-dev
```

然后，在 webpack 配置中添加 loader。例如：

```
// webpack.config.js
const path = require('path');

module.exports = {
  module: {
    rules: [
      {
        // 匹配一个 polyfill (或任何文件) ,
        // 然后在 bundle 中不会引入这个 polyfill
        test: path.resolve(__dirname, 'node_modules/library/polyfill.js'),
        use: 'null-loader'
      }
    ]
  }
}
```

然后，通过你偏爱的方式去运行 webpack。

贡献人员

如果你从未阅读过我们的贡献指南，请在上面花点时间。

[贡献指南](#)

License

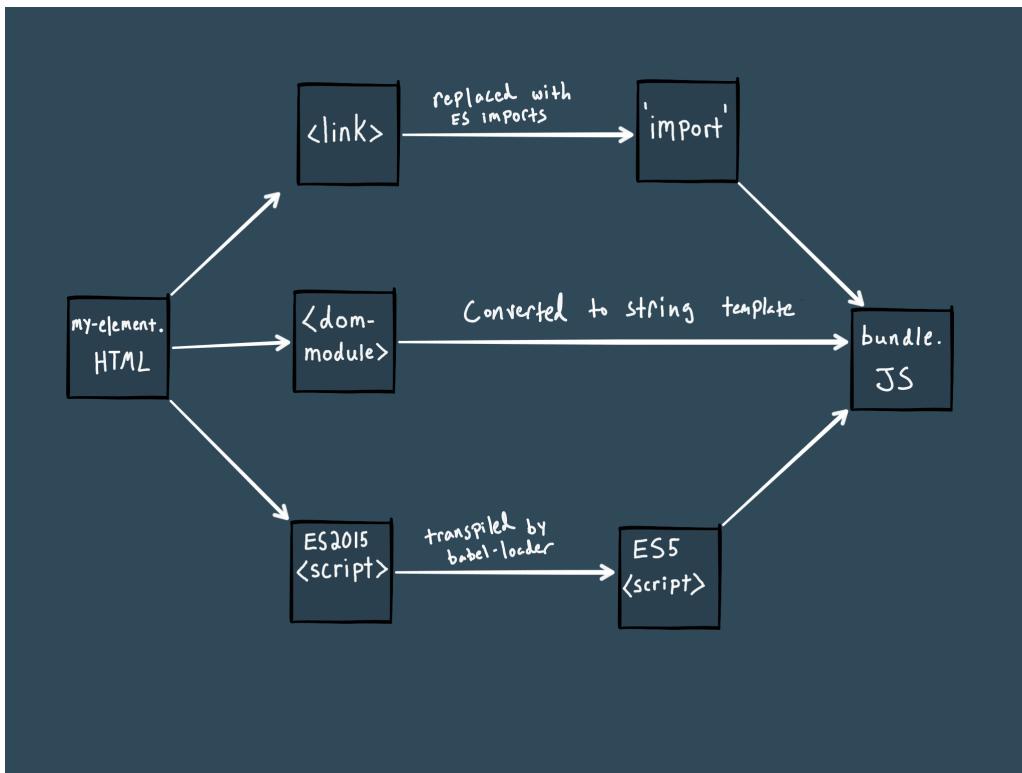
[MIT](#) [MIT](#)" class="icon-link" href="#mit">>

polymer-webpack-loader

[npm package 2.0.3](#) [build passing](#)

Polymer component loader for [webpack](#).

The loader allows you to write mixed HTML, CSS and JavaScript Polymer elements and still use the full webpack ecosystem including module bundling and code splitting.



The loader transforms your components:

- `<link rel="import" href="./my-other-element.html">` -> `import './my-other-element.html';`
- `<dom-module>` becomes a string which is registered at runtime
- `<script src="./other-script.js"></script>` -> `import './other-script.js';`
- `<script>/* contents */</script>` -> `/* contents */`

What does that mean?

Any `<link>` "href" or `<script>` "src" that is **not an external link** and does not start with `~, /, ./` or a series of `../` will have `./` appended to the beginning of the value. To prevent this change use options `ignoreLinks` below.

Path Translations

tag	<link rel="import" href="path/to/some-element.html">
import	import "./path/to/some-element.html"
tag	<link rel="import" href="/path/to/some-element.html">
import	import "/path/to/some-element.html"
tag	<link rel="import" href="../../path/to/some-element.html">
import	import "../../path/to/some-element.html"
tag	<link rel="import" href=".//path/to/some-element.html">
import	import ".//path/to/some-element.html"
tag	<link rel="import" href="~path/to/some-element.html">
import	import "~path/to/some-element.html"

Configuring the Loader

```
{
  test: /\.html$/,
  include: Condition(s) (optional),
  exclude: Condition(s) (optional),
  options: {
    ignoreLinks: Condition(s) (optional),
    ignorePathReWrite: Condition(s) (optional),
    processStyleLinks: Boolean (optional),
    htmlLoader: Object (optional)
  },
  loader: 'polymer-webpack-loader'
},
```

include: Condition(s)

See [Rule.include] and [Condition] in the webpack documentation. Paths matching this option will be processed by polymer-webpack-loader. **WARNING:** If this property exists the loader will only process files matching the given conditions. If your component has a `<link>` pointing to a component e.g. in another directory, the `include` condition(s) **MUST** also match that directory.

exclude: Condition(s)

See [Rule.exclude] and [Condition] in the webpack documentation. Paths matching this

option will be excluded from processing by polymer-webpack-loader. NOTE: Files imported through a `<link>` will not be excluded by this property. See `Options.ignoreLinks`.

Options

`ignoreLinks: Condition(s)`

`<link>`s pointing to paths matching these conditions (see [Condition] in the webpack documentation) will not be transformed into `imports`.

`ignorePathReWrite: Condition(s)`

`<link>` paths matching these conditions (see [Condition] in the webpack documentation) will not be changed when transformed into `imports`. This can be useful for respecting aliases, loader syntax (e.g. `markup-inline-loader!./my-element.html`), or module paths.

`processStyleLinks Boolean`

If set to true the loader will rewrite `<link import="css" href="...">` or `<link rel="stylesheet" href="...">` that are inside the dom-module with `<style>require('...')</style>`. This will allow for the file to be processed by loaders that are set up in the webpack config to handle their file type.

1. Any `<link>` that is inside the `<dom-module>` but not in the `<template>` will be added to the `<template>` in the order the tags appear in the file.

```
<dom-module>
  <link rel="stylesheet" href="file1.css">
  <template>
    <link rel="stylesheet" href="file2.css">
  </template>
</dom-module>
```

would produce

```
<dom-module>
  <template>
    <style>require('file1.css')</style>
    <style>require('file2.css')</style>
  </template>
</dom-module>
```

1. The loader will only replace a `<link>` if the href is a relative path. Any link attempting to access an external link i.e. `http`, `https` or `//` will not be replaced.

htmlLoader: Object

Options to pass to the html-loader. See [html-loader](#).

Use with Babel (or other JS transpilers)

If you'd like to transpile the contents of your element's `<script>` block you can [chain an additional loader](#).

```
module: {
  loaders: [
    {
      test: /\.html$/,
      use: [
        // Chained loaders are applied last to first
        { loader: 'babel-loader' },
        { loader: 'polymer-webpack-loader' }
      ]
    }
  ]
}

// alternative syntax

module: {
  loaders: [
    {
      test: /\.html$/,
      // Chained loaders are applied right to left
      loader: 'babel-loader!polymer-webpack-loader'
    }
  ]
}
```

Use of HtmlWebpackPlugin

Depending on how you configure the HtmlWebpackPlugin you may encounter conflicts with the polymer-webpack-loader.

Example:

```
{
  test: /\.html$/,
  loader: 'html-loader',
  include: [
    path.resolve(__dirname, './index.html'),
  ],
},
{
  test: /\.html$/,
  loader: 'polymer-webpack-loader'
}
```

This would expose your index.html file to the polymer-webpack-loader based on the process used by the html-loader. In this case you would need to exclude your html file from the polymer-webpack-loader or look for other ways to avoid this conflict. See: [html-webpack-plugin template options](#)

Shimming

Not all Polymer Elements have been written to execute as a module and will require changes to work with webpack. The most common issue encountered is because modules do not execute in the global scope. Variables, functions and classes will no longer be global unless they are declared as properties on the global object (window).

```
class MyElement {} // I'm not global anymore
window.myElement = MyElement; // Now I'm global again
```

For external library code, webpack provides [shimming options](#).

- Use the [exports-loader](#) to add a module export to components which expect a symbol to be global.
- Use the [imports-loader](#) when a script expects the `this` keyword to reference `window`.
- Use the [ProvidePlugin](#) to add a module import statement when a script expects a variable to be globally defined (but is now a module export).
- Use the [NormalModuleReplacementPlugin](#) to have webpack swap a module-compliant version for a script.

You may need to apply multiple shimming techniques to the same component.

Boostrapping Your Application

The webcomponent polyfills must be added in a specific order. If you do not delay loading the main bundle with your components, you will see the following exceptions in the browser console:

```
Uncaught TypeError: Failed to construct 'HTMLElement': Please use the 'i
```

Reference the [demo html file](#) for the proper loading sequence.

Maintainers



Bryan Coulter



Chad Killingsworth



Rob Dodson

postcss-loader

Loader for [webpack](#) to process CSS with [PostCSS](#)

raw-loader

```
[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![coverage][cover]][cover-url] [![chat][chat]][chat-url] [![size][size]][size-url]
```

可以将文件作为字符串导入的 webpack loader。

起步

你需要预先安装 raw-loader:

```
$ npm install raw-loader --save-dev
```

在然后 webpack 配置中添加 loader:

file.js

```
import txt from './file.txt';
```

webpack.config.js

```
// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.txt$/i,
        use: 'raw-loader',
      },
    ],
  },
};
```

或者，通过命令行使用：

```
$ webpack --module-bind 'txt=raw-loader'
```

然后，运行 webpack。

示例

Inline

```
import txt from 'raw-loader!./file.txt';
```

Beware, if you already define loader(s) for extension(s) in webpack.config.js you

should use:

```
import css from '!!raw-loader!./file.css'; // Adding `!!` to a request ↴
```

Contributing

Please take a moment to read our contributing guidelines if you haven't yet done so.

[CONTRIBUTING](#)

License

[MIT](#)

react-proxy-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url]

[![chat][chat]][chat-url]

通过将 react 组件包裹在一个代理组件中，来启用代码分离(code splitting)功能，可以按需加载 react 组件和它的依赖。

要求

此模块需要 Node v6.9.0+ 和 webpack v4.0.0+。

起步

你需要预先安装 react-proxy-loader:

```
$ npm install react-proxy-loader --save-dev
```

然后，在 webpack 配置中添加 loader。例如：

```
// 返回代理组件，并按需加载
// webpack 会为此组件和它的依赖创建一个额外的 chunk
const Component = require('react-proxy-loader!./Component');

// 返回代理组件的 mixin
// 可以在这里设置代理组件的加载中状态
const ComponentProxyMixin = require('react-proxy-loader!./Component').M:

const ComponentProxy = React.createClass({
  mixins: [ComponentProxyMixin],
  renderUnavailable: function() {
    return <p>Loading...</p>;
  }
});
```

或者，在配置中指定需要代理的组件：

```
// webpack.config.js
module.exports = {
  module: {
    loaders: [
      /* ... */
      {
        test: [
          /component\.\jsx$/, // 通过正则表达式(RegExp)匹配选择组件
          /\.async\.\jsx$/, // 通过扩展名(extension)匹配选择组件
          "/abs/path/to/component.jsx" // 通过绝对路径(absolute
        ],
      }
    ]
  }
};
```

```
        loader: "react-proxy-loader"
    }
]
}
};
```

或者，在`name`查询参数中提供一个chunk名称：

```
var Component = require("react-proxy-loader?name=chunkName!./Component")
```

然后，通过你偏爱的方式去运行`webpack`。

License

[MIT](#) [MIT](#)

restyle-loader

Updates style <link>

href value with a hash to trigger a style reload

Loader new home: [restyle-loader](#)

安装

```
npm install --save-dev restyle-loader
```

用法

[文档：使用 loader](#)

示例

webpack.config.js

```
{  
  test: /\.css?$/,  
  use: [  
    {  
      loader: "restyle-loader"  
    },  
    {  
      loader: "file-loader",  
      options: {  
        name: "[name].css?[hash:8]"  
      }  
    }  
  ]  
}
```

hash 需要启用热模块替换(Hot Module Replacement)

bundle.js

```
require("./index.css");  
// 在这里打包代码...
```

index.html

```
<head>
```

```
<link rel="stylesheet" type="text/css" href="css/index.css">
</head>
```

loader 运行后就变成

```
<head>
  <link rel="stylesheet" type="text/css" href="css/index.css?531fdfd0">
</head>
```

sass-loader

Loads a Sass/SCSS file and compiles it to CSS.

Use the [css-loader](#) or the [raw-loader](#) to turn it into a JS module and the [mini-css-extract-plugin](#) to extract it into a separate file. Looking for the webpack 1 loader? Check out the [archive/webpack-1 branch](#).

安装

```
npm install sass-loader node-sass webpack --save-dev
```

[webpack](#) 是 sass-loader 的 [peerDependency](#)， 并且还需要你预先安装 [Node Sass](#) 或 [Dart Sass](#)。 这可以控制所有依赖的版本， 并选择要使用的 Sass 实现。

[Node Sass]: <https://github.com/sass/node-sass> [Dart Sass]: <http://sass-lang.com/dart-sass>

示例

通过将 [style-loader](#) 和 [css-loader](#) 与 sass-loader 链式调用， 可以立刻将样式作用在 DOM 元素。

```
npm install style-loader css-loader --save-dev

// webpack.config.js
module.exports = {
  ...
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: [
          "style-loader", // 将 JS 字符串生成为 style 节点
          "css-loader", // 将 CSS 转化成 CommonJS 模块
          "sass-loader" // 将 Sass 编译成 CSS， 默认使用 Node Sass
        ]
      }
    ]
  }
}
```

```
        }
    }
};
```

也可以直接将选项传递给 [Node Sass][] 或 [Dart Sass][]:

```
// webpack.config.js
module.exports = {
  ...
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: [
          {
            loader: "style-loader"
          },
          {
            loader: "css-loader"
          },
          {
            loader: "sass-loader",
            options: {
              includePaths: ["absolute/path/a", "absolute/path/b"]
            }
          }
        ]
      }
    ]
  }
};
```

更多 Sass 可用选项，查看 [Node Sass 文档](#) for all available Sass options.

By default the loader resolve the implementation based on your dependencies. Just add required implementation to `package.json` (`node-sass` or `sass` package) and install dependencies.

Example where the `sass-loader` loader uses the `sass` (`dart-sass`) implementation:

package.json

```
{
  "devDependencies": {
    "sass-loader": "*",
    "sass": "*"
  }
}
```

Example where the `sass-loader` loader uses the `node-sass` implementation:

package.json

```
{
  "devDependencies": {
    "sass-loader": "*",
    "node-sass": "*"
  }
}
```

Beware the situation when `node-sass` and `sass` was installed, by default the `sass-loader` prefers `node-sass`, to avoid this situation use the `implementation` option.

The special `implementation` option determines which implementation of Sass to use. It takes either a [Node Sass][] or a [Dart Sass][] module. For example, to use Dart Sass, you'd pass:

```
// ...
{
  loader: "sass-loader",
  options: {
    implementation: require("sass")
  }
}
// ...
```

Note that when using Dart Sass, **synchronous compilation is twice as fast as asynchronous compilation** by default, due to the overhead of asynchronous callbacks. To avoid this overhead, you can use the `fibers` package to call asynchronous importers from the synchronous code path. To enable this, pass the `Fiber` class to the `fiber` option:

```
// webpack.config.js
const Fiber = require('fibers');

module.exports = {
  ...
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: [
          {
            loader: "style-loader"
          },
          {
            loader: "css-loader"
          },
          {
            loader: "sass-loader",
            options: {
              implementation: require("sass"),
              fiber: Fiber
            }
          }
        ]
      }
    ]
  }
};

##
```

通常，生产环境下比较推荐的做法是，使用 `mini-css-extract-plugin` 将样式表抽离成专门的单独文件。这样，样式表将不再依赖于 JavaScript：

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
```

```

...
module: {
  rules: [
    test: /\.scss$/,
    use: [
      // fallback to style-loader in development
      process.env.NODE_ENV !== 'production' ? 'style-loader'
      "css-loader",
      "sass-loader"
    ]
  ]
},
plugins: [
  new MiniCssExtractPlugin({
    // Options similar to the same options in webpackOptions.output
    // both options are optional
    filename: "[name].css",
    chunkFilename: "[id].css"
  })
]
};

```

用法

导入(import)

webpack 提供一种解析文件的高级的机制。sass-loader 使用 Sass 的 custom importer 特性，将所有的 query 传递给 webpack 的解析引擎(resolving engine)。只要它们前面加上 ~，告诉 webpack 它不是一个相对路径，这样就可以 import 导入 node_modules 目录里面的 sass 模块：

```
@import "~bootstrap/dist/css/bootstrap";
```

重要的是，只在前面加上 ~，因为 ~/ 会解析到主目录(home directory)。webpack 需要区分 bootstrap 和 ~bootstrap，因为 CSS 和 Sass 文件没有用于导入相关文件的特殊语法。@import "file" 与 @import "./file"；这两种写法是相同的

url(...) 的问题

由于 Sass/libsass 并没有提供 url rewriting 的功能，所以所有的链接资源都是相对输出文件(output)而言。

- 如果生成的 CSS 没有传递给 css-loader，它相对于网站的根目录。
- 如果生成的 CSS 传递给了 css-loader，则所有的 url 都相对于入口文件（例如： main.scss）。

第二种情况可能会带来一些问题。正常情况下我们期望相对路径的引用是相对于 .scss 去解析（如同在 .css 文件一样）。幸运的是，有2个方法可以解决这个问题：

- 将 `resolve-url-loader` 设置于 loader 链中的 `sass-loader` 之前，就可以重写 url。
- Library 作者一般都会提供变量，用来设置资源路径，如 `bootstrap-sass` 可以通过 `$icon-font-path` 来设置。参见[this working bootstrap example](#)。

提取样式表

使用 webpack 打包 CSS 有许多优点，在开发环境，可以通过 hashed urls 或 模块热替换(HMR) 引用图片和字体资源。而在线上环境，使样式依赖 JS 执行环境并不是一个好的实践。渲染会被推迟，甚至会出现 FOUIC，因此在最终线上环境构建时，最好还是能够将 CSS 放在单独的文件中。

从 bundle 中提取样式表，有2种可用的方法：

- `extract-loader` (简单，专门针对 `css-loader` 的输出)
- `mini-css-extract-plugin` (use this, when using webpack 4 configuration. Works in all use-cases)

Source maps

要启用 CSS source map，需要将 `sourceMap` 选项作为参数，传递给 `sass-loader` 和 `css-loader`。此时 `webpack.config.js` 如下：

```
module.exports = {
  ...
  devtool: "source-map", // any "source-map"-like devtool is possible
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: [
          {
            loader: "style-loader", options: {
              sourceMap: true
            }
          },
          {
            loader: "css-loader", options: {
              sourceMap: true
            }
          },
          {
            loader: "sass-loader", options: {
              sourceMap: true
            }
          }
        ]
      }
    ]
  }
};
```

如果你要在 Chrome 中编辑原始的 Sass 文件，建议阅读这篇不错的博客文章。具体示例参考 [test/sourceMap](#)。

环境变量

如果你要将 Sass 代码放在实际的入口文件(entry file)之前，可以设置 `data` 选项。此时 `sass-loader` 不会覆盖 `data` 选项，只会将它拼接在入口文件的内容之前。当 Sass 变量依赖于环境时，这一点尤其有用。

```
{  
  loader: "sass-loader",  
  options: {  
    data: "$env: " + process.env.NODE_ENV + ";"  
  }  
}
```

The `data` option supports Function notation:

```
{  
  loader: "sass-loader",  
  options: {  
    data: (loaderContext) => {  
      // More information about available options https://webpack.js.org/guides/asset-modules/#css  
      const { resourcePath, rootContext } = loaderContext;  
      const relativePath = path.relative(rootContext, resourcePath);  
  
      if (relativePath === "styles/foo.scss") {  
        return "$value: 100px;"  
      }  
  
      return "$value: 200px;"  
    }  
  }  
}
```

注意：由于代码注入，会破坏整个入口文件的 source map。通常一个简单的解决方案是，多个 Sass 文件入口。

script-loader

安装

```
npm install --save-dev script-loader
```

用法

在全局上下文(global context)执行一次 JS 脚本。

:警告: 在 node.js 中不会运行

```
##
```

```
import exec from 'script.exec.js';
```

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.exec\.js$/,
        use: [ 'script-loader' ]
      }
    ]
  }
}
```

内联

```
import exec from 'script-loader!./script.js';
```

Options

Name	<u>sourceMap</u>
Type	{Boolean}
Default	false
Description	Enable/Disable Sourcemaps

sourceMap

Type: Boolean Default: false

To include source maps set the `sourceMap` option.

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.script\.js$/,
        use: [
          {
            loader: 'script-loader',
            options: {
              sourceMap: true,
            },
          },
        ],
      }
    ]
  }
}
```

source-map-loader

Extracts source maps from existing source files (from their `sourceMappingURL`).

安装

```
npm i -D source-map-loader
```

用法

[文档：使用 loader](#)

```
##
```

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        use: ["source-map-loader"],
        enforce: "pre"
      }
    ]
  }
};
```

`source-map-loader` extracts existing source maps from all JavaScript entries. This includes both inline source maps as well as those linked via URL. All source map data is passed to webpack for processing as per a chosen [source map style](#) specified by the `devtool` option in [webpack.config.js](#).

This loader is especially useful when using 3rd-party libraries having their own source maps. If not extracted and processed into the source map of the webpack bundle, browsers may misinterpret source map data. `source-map-loader` allows webpack to maintain source map data continuity across libraries so ease of debugging is preserved.

`source-map-loader` will extract from any JavaScript file, including those in the `node_modules` directory. Be mindful in setting `include` and `exclude` rule conditions to maximize bundling performance.

style-loader

Adds CSS to the DOM by injecting a `<style>` tag

安装

```
npm install style-loader --save-dev
```

用法

建议将 `style-loader` 与 `css-loader` 结合使用

component.js

```
import style from './file.css';
```

webpack.config.js

```
{
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [{ loader: 'style-loader' }, { loader: 'css-loader' }],
      },
    ],
  }
}

###
```

在使用局部作用域 CSS 时，模块导出生成的（局部）标识符(identifier)。

component.js

```
import style from './file.css';

style.className === 'z849f98ca812';
```

Url

也可以添加一个URL `<link href="path/to/file.css" rel="stylesheet">`，而不是用带有 `<style></style>` 标签的内联 CSS `{String}`。

```
import url from 'file.css';
```

webpack.config.js

```
{
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [{ loader: 'style-loader' }, { loader: 'css-loader' }],
      },
    ],
  }
}
```

```

        use: [{ loader: 'style-loader/url' }, { loader: 'file-loader' }],
      },
    ],
  }
}

<link rel="stylesheet" href="path/to/file.css" />

```

i 使用 `url` 引用的 Source map 和资源：当 `style-loader` 与 `{ options: { sourceMap: true } }` 选项一起使用时，CSS 模块将生成为 `Blob`，因此相对路径无法正常工作（他们将相对于 `chrome:blob` 或 `chrome:devtools`）。为了使资源保持正确的路径，必须设置 `webpack` 配置中的 `output.publicPath` 属性，以便生成绝对路径。或者，你可以启用上面提到的 `convertToAbsoluteUrls` 选项。

可选的 (Useable)

The `style-loader` injects the styles lazily making them useable on-demand via `style.use()` / `style.unuse()`

按照惯例，引用计数器 API (Reference Counter API) 应关联到 `.useable.css`，而 `.css` 的载入，应该使用基本的 `style-loader` 用法（类似于其他文件类型，例如 `.useable.less` 和 `.less`）。

webpack.config.js

```
{
  module: {
    rules: [
      {
        test: /\.css$/,
        exclude: /\useable\.css$/,
        use: [
          { loader: "style-loader" },
          { loader: "css-loader" },
        ],
      },
      {
        test: /\useable\.css$/,
        use: [
          {
            loader: "style-loader/useable"
          },
          { loader: "css-loader" },
        ],
      },
    ],
  },
}
```

引用计数器 API (reference counter API)

component.js

```
import style from './file.css';

style.use(); // = style.ref();
style.unuse(); // = style.unref();
```

样式不会添加在 `import/require()` 上，而是在调用 `use/ref` 时添加。如果 `unuse/unref` 与 `use/ref` 一样频繁地被调用，那么样式将从页面中删除。

□□ 当 `unuse/unref` 比 `use/ref` 调用频繁的时候，产生的行为是不确定的。所以不要这样做。

选项

名称	hmr
类型	{Boolean}
默认值	true
描述	Enable/disable Hot Module Replacement (HMR), if disabled no HMR Code will be added (good for non local development/production)
名称	base
类型	{Number}
默认值	true
描述	设置模块 ID 基础 (DLLPlugin)
名称	attrs
类型	{Object}
默认值	{ }
描述	添加自定义 attrs 到 <style></style>
名称	transform
类型	{Function}
默认值	false
描述	转换/条件加载 CSS，通过传递转换/条件函数
名称	insertAt
类型	{String Object}
默认值	bottom
描述	在给定位置处插入 <style></style>
名称	insertInto
类型	{String Object}

八王	<code>insertIntoHead</code>
默认值	<code><head></code>
描述	给定位置中插入 <code><style></style></code>
名称	singleton
类型	{Boolean}
默认值	<code>undefined</code>
描述	Reuses a single <code><style></style></code> element, instead of adding/removing individual elements for each required module.
名称	sourceMap
类型	{Boolean}
默认值	<code>false</code>
描述	启用/禁用 Sourcemap
名称	convertToAbsoluteUrls
类型	{Boolean}
默认值	<code>false</code>
描述	启用 source map 后，将相对 URL 转换为绝对 URL

hmr

Enable/disable Hot Module Replacement (HMR), if disabled no HMR Code will be added. This could be used for non local development and production.

webpack.config.js

```
{
  loader: 'style-loader',
  options: {
    hmr: false
  }
}
```

base

当使用一个或多个 `DllPlugin` 时，此设置主要用作 css 冲突的修补方案。`base` 可以防止 `app` 的 css (或 `DllPlugin2` 的 css) 覆盖 `DllPlugin1` 的 css，方法是指定一个 css 模块的 id 大于 `DllPlugin1` 的范围，例如：

webpack.dll1.config.js

```
{
```

```
test: /\.css$/,
use: [
  'style-loader',
  'css-loader'
]
}
```

webpack.dll2.config.js

```
{
  test: /\.css$/,
  use: [
    { loader: 'style-loader', options: { base: 1000 } },
    'css-loader'
  ]
}
```

webpack.app.config.js

```
{
  test: /\.css$/,
  use: [
    { loader: 'style-loader', options: { base: 2000 } },
    'css-loader'
  ]
}
```

attrs

如果已定义， style-loader 将把属性值附加在 `<style>/<link>` 元素上。

component.js

```
import style from './file.css';
```

webpack.config.js

```
{
  test: /\.css$/,
  use: [
    { loader: 'style-loader', options: { attrs: { id: 'id' } } }
    { loader: 'css-loader' }
  ]
}
```

```
<style id="id"></style>
```

Url

component.js

```
import link from './file.css';
```

webpack.config.js

```
{  
  test: /\.css$/,
  use: [  
    { loader: 'style-loader/url', options: { attrs: { id: 'id' } } }
    { loader: 'file-loader' }
  ]
}
```

transform

`transform` 是一个函数，可以在通过 `style-loader` 加载到页面之前修改 css。该函数将在即将加载的 css 上调用，函数的返回值将被加载到页面，而不是原始的 css 中。如果 `transform` 函数的返回值是 falsy 值，那么 css 根本就不会加载到页面中。

□□ In case you are using ES Module syntax in `transform.js` then, you **need to transpile** it or otherwise it will throw an `{Error}`.

webpack.config.js

```
{  
  loader: 'style-loader',
  options: {
    transform: 'path/to/transform.js'
  }
}
```

transform.js

```
module.exports = function(css) {
  // Here we can change the original css
  const transformed = css.replace('.classNameA', '.classNameB');

  return transformed;
};
```

Conditional

webpack.config.js

```
{  
  loader: 'style-loader',
  options: {
    transform: 'path/to/conditional.js'
  }
}
```

conditional.js

```
module.exports = function(css) {
  // 如果条件匹配则加载[和转换] CSS
  if (css.includes('something I want to check')) {
    return css;
  }
  // 如果返回 falsy 值，则不会加载 CSS
  return false;
};
```

insertAt

默认情况下，style-loader 将 `<style>` 元素附加到样式目标(style target)的末尾，即页面的 `<head>` 标签，也可以是由 `insertInto` 指定其他标签。这将导致 loader 创建的 CSS 优先于目标中已经存在的 CSS。要在目标的起始处插入 style 元素，请将此查询参数(query parameter)设置为 'top'，例如

webpack.config.js

```
{
  loader: 'style-loader',
  options: {
    insertAt: 'top'
  }
}
```

A new `<style>` element can be inserted before a specific element by passing an object, e.g.

webpack.config.js

```
{
  loader: 'style-loader',
  options: {
    insertAt: {
      before: '#id'
    }
  }
}
```

insertInto

默认情况下，样式加载器将 `<style>` 元素插入到页面的 `<head>` 标签中。如果要将标签插入到其他位置，可以在这里为该元素指定 CSS 选择器。如果你想要插入到 `Iframe` 中，请确保你具有足够的访问权限，样式将被注入到内容文档的 `head` 中。

You can also pass function to override default behavior and insert styles in your container, e.g

webpack.config.js

```
{  
  loader: 'style-loader',  
  options: {  
    insertInto: () => document.querySelector("#root"),  
  }  
}
```

通过使用函数，可以将样式插入到 `ShadowRoot` 中，例如

webpack.config.js

```
{  
  loader: 'style-loader',  
  options: {  
    insertInto: () => document.querySelector("#root").shadowRoot,  
  }  
}
```

singleton

如果已定义，则 `style-loader` 将重用单个 `<style></style>` 元素，而不是为每个所需的模块添加/删除单个元素。

i 默认情况下启用此选项，IE9 对页面上允许的 `style` 标签数量有严格的限制。你可以使用 `singleton` 选项来启用或禁用它。

webpack.config.js

```
{  
  loader: 'style-loader',  
  options: {  
    singleton: true  
  }  
}
```

sourceMap

启用/禁用 source map 加载

webpack.config.js

```
{  
  loader: 'style-loader',  
  options: {  
    sourceMap: true  
  }  
}
```

convertToAbsoluteUrls

如果 convertToAbsoluteUrls 和 sourceMaps 都启用，那么相对 url 将在 css 注入页面之前，被转换为绝对 url。这解决了在启用 source map 时，相对资源无法加载的问题。你可以使用 convertToAbsoluteUrls 选项启用它。

webpack.config.js

```
{  
  loader: 'style-loader',  
  options: {  
    sourceMap: true,  
    convertToAbsoluteUrls: true  
  }  
}
```

svg-inline-loader

This Webpack loader inlines SVG as module. If you use Adobe suite or Sketch to export SVGs, you will get auto-generated, unneeded crusts. This loader removes it for you, too.

安装

```
npm install svg-inline-loader --save-dev
```

配置

只需加载配置对象到 `module.loaders` 像下面这样。

```
{  
  test: /\.svg$/,  
  loader: 'svg-inline-loader'  
}
```

警告: 这个loader你应只配置一次，通过 `module.loaders` 或者 `require('!...')` 配置。更多细节参考 [#15](#)。

Query 选项

###

删除指定的标签和它的子元素，你可以指定标签通过设置 `removingTags` 查询多个。

默认值: `removeTags: false`

removingTags: [...string]

警告: 你指定 `removeTags: true` 时, 它才会执行。

默认值: `removingTags: ['title', 'desc', 'defs', 'style']`

warnTags: [...string]

警告标签,例: `['desc', 'defs', 'style']`

默认值: `warnTags: []`

removeSVGTagAttrs: boolean

删除 `<svg />` 的 `width` 和 `height` 属性。

默认值: `removeSVGTagAttrs: true`

removingTagAttrs: [...string]

删除内部的 `<svg />`的属性。

默认值: `removingTagAttrs: []`

warnTagAttrs: [...string]

在console发出关于内部 `<svg />` 属性的警告

默认值: `warnTagAttrs: []`

classPrefix: boolean || string

添加一个前缀到svg文件的class, 以避免碰撞。

默认值: `classPrefix: false`

idPrefix: boolean || string

添加一个前缀到svg文件的id, 以避免碰撞。

默认值: `idPrefix: false`

使用示例

```
// 使用默认 hashed prefix (__[hash:base64:7]__)
var logoTwo = require('svg-inline-loader?classPrefix!./logo_two.svg');

// 使用自定义字符串
var logoOne = require('svg-inline-loader?classPrefix=my-prefix-!./logo_');

// 使用自定义字符串和hash
var logoThree = require('svg-inline-loader?classPrefix=__prefix-[sha512
```

hash 操作请参照 [loader-utils](#)。

通过 `module.loaders` 优先使用:

```
{
  test: /\.svg$/,
  loader: 'svg-inline-loader?classPrefix'
}
```

thread-loader

Runs the following loaders in a worker pool.

安装

```
npm install --save-dev thread-loader
```

用法

把这个 loader 放置在其他 loader 之前， 放置在这个 loader 之后的 loader 就会在一个单独的 worker 池(worker pool)中运行

在 worker 池(worker pool)中运行的 loader 是受到限制的。例如：

- 这些 loader 不能产生新的文件。
- 这些 loader 不能使用定制的 loader API (也就是说，通过插件)。
- 这些 loader 无法获取 webpack 的选项设置。

每个 worker 都是一个单独的有 600ms 限制的 node.js 进程。同时跨进程的数据交换也会被限制。

请仅在耗时的 loader 上使用

示例

webpack.config.js

```

module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        include: path.resolve("src"),
        use: [
          "thread-loader",
          // your expensive loader (e.g babel-loader)
        ]
      }
    ]
  }
}

```

可配选项

```

use: [
  {
    loader: "thread-loader",
    // 有同样配置的 loader 会共享一个 worker 池(worker pool)
    options: {
      // 产生的 worker 的数量, 默认是 (cpu 核心数 - 1)
      // 或者, 在 require('os').cpus() 是 undefined 时回退至 1
      workers: 2,
      // 一个 worker 进程中并行执行工作的数量
      // 默认为 20
      workerParallelJobs: 50,
      // 额外的 Node.js 参数
      workerNodeArgs: ['--max-old-space-size=1024'],
      // Allow to respawn a dead worker pool
      // respawning slows down the entire compilation
      // and should be set to false for development
      poolRespawn: false,
      // 闲置时定时删除 worker 进程
      // 默认为 500ms
      // 可以设置为无穷大, 这样在监视模式(--watch)下可以保持 worker 持续存在
      poolTimeout: 2000,
      // 池(pool)分配给 worker 的工作数量
      // 默认为 200
      // 降低这个数值会降低总体的效率, 但是会提升工作分布更均一
      poolParallelJobs: 50,
      // 池(pool)的名称
      // 可以修改名称来创建其余选项都一样的池(pool)
      name: "my-pool"
    }
  },
  // your expensive loader (e.g babel-loader)
]

```

预热

可以通过预热 worker 池(worker pool)来防止启动 worker 时的高延时。

这会启动池(pool)内最大数量的 worker 并把指定的模块载入 node.js 的模块缓存中。

```
const threadLoader = require('thread-loader');

threadLoader.warmup({
  // pool options, like passed to loader options
  // must match loader options to boot the correct pool
}, [
  // modules to load
  // can be any module, i. e.
  'babel-loader',
  'babel-preset-es2015',
  'sass-loader',
]);

```

transform-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

A browserify transformation loader for webpack.

This loader allows use of [browserify transforms](#) via a webpack loader.query parameter

Requirements

This module requires a minimum of Node v6.9.0 and Webpack v4.0.0.

Getting Started

To begin, you'll need to install `transform-loader`:

```
$ npm install transform-loader --save-dev
```

Note: We're using the `coffeeify` transform for these examples.

Then invoke the loader through a require like so:

```
const thing = require('!transform-loader?coffeeify!widget/thing');
```

Or add the loader to your `webpack` config. For example:

```
// entry.js
import thing from 'widget/thing';
```

```
// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.coffee?$/,
        loader: `transform-loader?coffeeify`,
        // options: {...}
      },
    ],
  },
}
```

And run `webpack` via your preferred method.

QueryString Options

When using the loader via a `require` query string you may specify one of two types; a loader name, or a function index.

Type: `String`

The name of the `browserify` transform you wish to use.

Note: You must install the correct transform manually. Webpack nor this loader will do that for you.

Type: `Number`

The index of a function contained within `options.transforms` which to use to transform the target file(s).

Options

`transforms`

Type: `Array[Function]` Default: `undefined`

An array of `functions` that can be used to transform a given file matching the configured loader `test`. For example:

```
// entry.js
const thing = require('widget/thing');

// webpack.config.js
const through = require('through2');

module.exports = {
```

```
module: {
  rules: [
    {
      test: /\.ext$/,
      // NOTE: we've specified an index of 0, which will use the `transform` function in `transforms` below.
      loader: 'transform-loader?0',
      options: {
        transforms: [
          function transform() {
            return through(
              (buffer) => {
                const result = buffer
                  .split('')
                  .map((chunk) => String.fromCharCode(127 - chunk.charCodeAt(0)))
                return this.queue(result).join('');
              },
              () => this.queue(null)
            );
          }
        ]
      }
    ]
  }
}
```

License

[MIT](#) [MIT](#)

url-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url] [![size][size]][size-url]

A loader for webpack which transforms files into base64 URIs.

Getting Started

To begin, you'll need to install `url-loader`:

```
$ npm install url-loader --save-dev
```

`url-loader` works like `file-loader`, but can return a DataURL if the file is smaller than a byte limit.

```
import img from './image.png'

// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.(\png|jpg|gif)$/i,
        use: [
          {
            loader: 'url-loader',
            options: {
              limit: 8192
            }
          }
        ]
      }
    ]
  }
}
```

And run `webpack` via your preferred method.

Options

`fallback`

Type: `String` Default: '`file-loader`'

Specifies an alternative loader to use when a target file's size exceeds the limit set in the `limit` option.

```
// webpack.config.js
{
  loader: 'url-loader',
  options: {
    fallback: 'responsive-loader'
  }
}
```

The fallback loader will receive the same configuration options as url-loader.

For example, to set the quality option of a responsive-loader above use:

```
{
  loader: 'url-loader',
  options: {
    fallback: 'responsive-loader',
    quality: 85
  }
}
```

limit

Type: Number Default: undefined

A Number specifying the maximum size of a file in bytes. If the file is greater than the limit, `file-loader` is used by default and all query parameters are passed to it. Using an alternative to `file-loader` is enabled via the `fallback` option.

The limit can be specified via loader options and defaults to no limit.

```
// webpack.config.js
{
  loader: 'url-loader',
  options: {
    limit: 8192
  }
}
```

mimetype

Type: String Default: (file extension)

Sets the MIME type for the file to be transformed. If unspecified the file extensions will be used to lookup the MIME type.

```
// webpack.config.js
{
  loader: 'url-loader',
  options: {
    mimetype: 'image/png'
  }
}
```

}

Contributing

Please take a moment to read our contributing guidelines if you haven't yet done so.

[CONTRIBUTING](#)

License

[MIT](#)

val-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

A webpack loader which executes a given module, and returns the result of the execution at build-time, when the module is required in the bundle. In this way, the loader changes a module from code to a result.

Another way to view `val-loader`, is that it allows a user a way to make their own custom loader logic, without having to write a custom loader.

Requirements

This module requires a minimum of Node v6.9.0 and Webpack v4.0.0.

Getting Started

To begin, you'll need to install `val-loader`:

```
$ npm install val-loader --save-dev
```

Then add the loader to your `webpack config`. For example:

```
// target-file.js
module.exports = () => {
  return { code: 'module.exports = 42;' }
};
```

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /target-file.js$/,
        use: [
          {
            loader: `val-loader`
          }
        ]
      }
    ]
  }
}

// src/entry.js
const answer = require('test-file');
```

And run `webpack` via your preferred method.

Return Object Properties

Targeted modules of this loader must export either a `Function` or `Promise` that returns an object containing a `code` property at a minimum, but can contain any number of additional properties.

`code`

Type: `String | Buffer` Default: `undefined` *Required*

Code passed along to webpack or the next loader that will replace the module.

`sourceMap`

Type: `Object` Default: `undefined`

A source map passed along to webpack or the next loader.

`ast`

Type: `Array[Object]` Default: `undefined`

An Abstract Syntax Tree that will be passed to the next loader. Useful to speed up the build time if the next loader uses the same AST.

`dependencies`

Type: `Array[String]` Default: `[]`

An array of absolute, native paths to file dependencies that should be watched by webpack for changes.

`contextDependencies`

Type: `Array[String]` Default: `[]`

An array of absolute, native paths to directory dependencies that should be watched by webpack for changes.

`cacheable`

Type: Boolean Default: false

If `true`, specifies that the code can be re-used in watch mode if none of the dependencies have changed.

Examples

In this example the loader is configured to operator on a file name of `years-in-ms.js`, execute the code, and store the result in the bundle as the result of the execution. This example passes `years` as an option, which corresponds to the `years` parameter in the target module exported function:

```
// years-in-ms.js
module.exports = function yearsInMs({ years }) {
  const value = years * 365 * 24 * 60 * 60 * 1000;
  // NOTE: this return value will replace the module in the bundle
  return { code: 'module.exports = ' + value };
}

// webpack.config.js
module.exports = {
  ...
  module: {
    rules: [
      {
        test: require.resolve('src/years-in-ms.js'),
        use: [
          {
            loader: 'val-loader',
            options: {
              years: 10
            }
          }
        ]
      }
    ]
  }
};
```

In the bundle, requiring the module then returns:

```
// ... bundle code ...

const tenYearsMs = require('years-in-ms'); // 315360000000

// ... bundle code ...

require("val-loader!tenyearsinms") == 315360000000
```

License

MIT MIT" class="icon-link" href="#mit">>

worker-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url] [![size][size]][size-url]

将脚本注册为 Web Worker 的 webpack loader

要求

此模块需要 Node v6.9.0+ 和 webpack v4.0.0+。

起步

你需要预先安装 worker-loader:

```
$ npm install worker-loader --save-dev
```

内联

```
// App.js
import Worker from 'worker-loader!./Worker.js';
```

配置

```
// webpack.config.js
{
  module: {
    rules: [
      {
        test: /\.worker\.js$/,
        use: { loader: 'worker-loader' }
      }
    ]
  }
}

// App.js
import Worker from './file.worker.js';

const worker = new Worker();

worker.postMessage({ a: 1 });
worker.onmessage = function (event) {};

worker.addEventListener("message", function (event) {});
```

然后，通过你偏爱的方式去运行 webpack。

选项

fallback

类型: Boolean 默认值: false

是否需要支持非 worker 环境的回退

```
// webpack.config.js
{
  loader: 'worker-loader',
  options: { fallback: false }
}
```

inline

类型: Boolean 默认值: false

你也可以使用 inline 参数，将 worker 内联为一个 BLOB

```
// webpack.config.js
{
  loader: 'worker-loader',
  options: { inline: true }
}
```

注意：内联模式还会为不支持内联 worker 的浏览器创建 chunk，要禁用此行为，只需将 fallback 参数设置为 false。

```
// webpack.config.js
{
  loader: 'worker-loader',
  options: { inline: true, fallback: false }
}
```

name

类型: String 默认值: [hash].worker.js

使用 name 参数，为每个输出的脚本，设置一个自定义名称。名称可能含有 [hash] 字符串，会被替换为依赖内容哈希值(content dependent hash)，用于缓存目的。只使用 name 时，可以省略 [hash]。

```
// webpack.config.js
{
  loader: 'worker-loader',
  options: { name: 'WorkerName.[hash].js' }
}
```

publicPath

类型: String 默认值: null

重写 worker 脚本的下载路径。如果未指定，则使用与其他 webpack 资源相同的公共路径(public path)。

```
// webpack.config.js
{
  loader: 'worker-loader',
  options: { publicPath: '/scripts/workers/' }
}
```

示例

worker 文件可以像其他脚本文件导入依赖那样，来导入依赖：

```
// Worker.js
const _ = require('lodash')

const obj = { foo: 'foo' }

_.has(obj, 'foo')

// 发送数据到父线程
self.postMessage({ foo: 'foo' })

// 响应父线程的消息
self.addEventListener('message', (event) => console.log(event))
```

集成 ES2015 模块

注意：如果配置好 `babel-loader`，你甚至可以使用 ES2015 模块。

```
// Worker.js
import _ from 'lodash'

const obj = { foo: 'foo' }

_.has(obj, 'foo')

// 发送数据到父线程
self.postMessage({ foo: 'foo' })

// 响应父线程的消息
self.addEventListener('message', (event) => console.log(event))
```

集成 TypeScript

集成 TypeScript，在导出 worker 时，你需要声明一个自定义模块

```

// typings/custom.d.ts
declare module "worker-loader!*" {
  class WebpackWorker extends Worker {
    constructor();
  }

  export default WebpackWorker;
}

// Worker.ts
const ctx: Worker = self as any;

// 发送数据到父线程
ctx.postMessage({ foo: "foo" });

// 响应父线程的消息
ctx.addEventListener("message", (event) => console.log(event));

// App.ts
import Worker from "worker-loader!./Worker";

const worker = new Worker();

worker.postMessage({ a: 1 });
worker.onmessage = (event) => {};

worker.addEventListener("message", (event) => {});

```

跨域策略

WebWorkers 受到 同源策略 的限制，如果 webpack 资源的访问服务，和应用程序不是同源，就会在浏览器中拦截其下载。如果在 CDN 域下托管资源，通常就会出现这种情况。甚至从 webpack-dev-server 下载时都会被拦截。有两种解决方法：

第一种，通过配置 inline 参数，将 worker 作为 blob 内联，而不是将其作为外部脚本下载

```

// App.js
import Worker from './file.worker.js';

// webpack.config.js
{
  loader: 'worker-loader'
  options: { inline: true }
}

```

第二种，通过配置 publicPath 选项，重写 worker 脚本的基础下载 URL

```

// App.js
// 会从 `'/workers/file.worker.js` 下载 worker 脚本
import Worker from './file.worker.js';

// webpack.config.js

```

```
{  
  loader: 'worker-loader'  
  options: { publicPath: '/workers/' }  
}
```

贡献

如果你从未阅读过我们的贡献指南，请在上面花点时间。

[贡献指南](#)

License

[MIT](#)

workerize-loader

workerize-loader

A webpack loader that moves a module and its dependencies into a Web Worker, automatically reflecting exported functions as asynchronous proxies.

- Bundles a tiny, purpose-built RPC implementation into your app
- If exported module methods are already async, signature is unchanged
- Supports synchronous and asynchronous worker functions
- Works beautifully with `async/await`
- Imported value is instantiable, just a decorated `Worker`

Install

```
npm install -D workerize-loader
```

Usage

worker.js:

```
// block for `time` ms, then return the number of loops we could run in
export function expensive(time) {
  let start = Date.now(),
      count = 0
  while (Date.now() - start < time) count++
  return count
}
```

index.js: (our demo)

```
import worker from 'workerize-loader!./worker'

let instance = worker() // `new` is optional

instance.expensive(1000).then( count => {
  console.log(`Ran ${count} loops`)
})
```

Credit

The inner workings here are heavily inspired by [worker-loader](#). It's worth a read!

License

[MIT License](#) © Jason Miller

less-loader

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

处理 less 的 webpack loader。将 Less 编译为 CSS。

要求

此模块需要 Node v6.9.0+ 和 webpack v4.0.0+。

起步

你需要预先安装 less-loader:

```
$ npm install less-loader --save-dev
```

然后，修改 webpack.config.js:

```
// webpack.config.js
module.exports = {
  ...
  module: {
    rules: [
      {
        test: /\.less$/,
        loader: 'less-loader' // 将 Less 编译为 CSS
      }
    ]
  }
};
```

然后，通过你偏爱的方式去运行 webpack。

The less-loader requires `less` as `peerDependency`. Thus you are able to control the versions accurately.

示例

将 `css-loader`、`style-loader` 和 `less-loader` 链式调用，可以把所有样式立即应用于 DOM。

```
// webpack.config.js
module.exports = {
  ...
  module: {
    rules: [
      {
        test: /\.less$/,
        loader: 'less-loader'
      }
    ]
  }
};
```

```

use: [
  loader: 'style-loader' // creates style nodes from JS strings
], {
  loader: 'css-loader' // translates CSS into CommonJS
}, {
  loader: 'less-loader' // compiles Less to CSS
}]
}
};


```

You can pass any Less specific options to the `less-loader` via [loader options](#). See the [Less documentation](#) for all available options in dash-case. Since we're passing these options to Less programmatically, you need to pass them in camelCase here:

```

// webpack.config.js
module.exports = {
  ...
  module: {
    rules: [
      {
        test: /\.less$/,
        use: [
          {
            loader: 'style-loader'
          },
          {
            loader: 'css-loader'
          },
          {
            loader: 'less-loader',
            options: {
              strictMath: true,
              noIeCompat: true
            }
          }
        ]
      }
    ]
  }
};

```

Unfortunately, Less doesn't map all options 1-by-1 to camelCase. When in doubt, [check their executable](#) and search for the dash-case option.

In production

Usually, it's recommended to extract the style sheets into a dedicated file in production using the [MiniCssExtractPlugin](#). This way your styles are not dependent on JavaScript.

Imports

Starting with `less-loader` 4, you can now choose between Less' builtin resolver and webpack's resolver. By default, webpack's resolver is used.

webpack resolver

webpack 提供了一种 [解析文件的高级机制](#)。`less-loader` 应用一个 Less 插件，并

且将所有查询参数传递给 webpack resolver。所以，你可以从 `node_modules` 导入你的 less 模块。只要添加一个 ~ 前缀，告诉 webpack 去查询模块。

```
@import '~bootstrap/less/bootstrap';
```

重要的是只使用 ~ 前缀，因为 ~/ 会解析为主目录。webpack 需要区分 `bootstrap` 和 `~bootstrap`，因为 CSS 和 Less 文件没有用于导入相对文件的特殊语法。

`@import "file"` 与 `@import "./file";` 写法相同

Non-Less imports

使用 webpack resolver，你可以引入任何文件类型。你只需要一个导出有效的 Less 代码的 loader。通常，你还需要设置 `issuer` 条件，以确保此规则仅适用于 Less 文件的导入：

```
// webpack.config.js
module.exports = {
  ...
  module: {
    rules: [
      {
        test: /\.js$/,
        issuer: /\.less$/,
        use: [
          {
            loader: 'js-to-less-loader'
          }
        ]
      }
    ]
  }
};
```

Less resolver

If you specify the `paths` option, the `less-loader` will not use webpack's resolver. Modules, that can't be resolved in the local folder, will be searched in the given `paths`. This is Less' default behavior. `paths` should be an array with absolute paths:

```
// webpack.config.js
module.exports = {
  ...
  module: {
    rules: [
      {
        test: /\.less$/,
        use: [
          {
            loader: 'style-loader'
          },
          {
            loader: 'css-loader'
          },
          {
            loader: 'less-loader', options: {
              paths: [
                path.resolve(__dirname, 'node_modules')
              ]
            }
          }
        ]
      }
    ]
  }
};
```

```
        } ]
    } ]
}
};
```

In this case, all webpack features like importing non-Less files or aliasing won't work of course.

插件

想要使用 插件， 只需像下面这样简单设置 `plugins` 选项：

```
// webpack.config.js
const CleanCSSPlugin = require('less-plugin-clean-css');

module.exports = {
    ...
    {
        loader: 'less-loader', options: {
            plugins: [
                new CleanCSSPlugin({ advanced: true })
            ]
        }
    }
}
...
};
```

Extracting style sheets

Bundling CSS with webpack has some nice advantages like referencing images and fonts with hashed urls or hot module replacement in development. In production, on the other hand, it's not a good idea to apply your style sheets depending on JS execution. Rendering may be delayed or even a FOUC might be visible. Thus it's often still better to have them as separate files in your final production build.

There are two possibilities to extract a style sheet from the bundle:

- `extract-loader` (simpler, but specialized on the css-loader's output)
- `MiniCssExtractPlugin` (more complex, but works in all use-cases)

source map

想要启用 CSS 的 source map， 你需要将 `sourceMap` 选项传递给 `less-loader` 和 `css-loader`。 所以你的 '`webpack.config.js`' 应该是这样：

```
module.exports = {
    ...
    module: {
        rules: [ {
```

```
test: /\.less$/,
use: [
  {
    loader: 'style-loader'
  },
  {
    loader: 'css-loader', options: {
      sourceMap: true
    }
  },
  {
    loader: 'less-loader', options: {
      sourceMap: true
    }
  }
]
}
};
```

Also checkout the [sourceMaps example](#).

如果你要在 Chrome 中编辑原始 Less 文件， 这里有一个很好的博客文章。 此博客文章是关于 Sass 的， 但它也适用于 Less。

CSS modules gotcha

There is a known problem with Less and [CSS modules](#) regarding relative file paths in `url(...)` statements. See [this issue for an explanation](#).

贡献

如果你从未阅读过我们的贡献指南， 请在上面花点时间。

[贡献指南](#) [贡献指南](#)" class="icon-link" href="#贡献指南">>

License

[MIT](#) [MIT](#)" class="icon-link" href="#mit">>

plugin

webpack 有着丰富的插件接口(rich plugin interface)。 webpack 自身的多数功能都使用这个插件接口。这个插件接口使 webpack 变得极其灵活。

Name	BabelMinifyWebpackPlugin
Description	使用 <code>babel-minify</code> 进行压缩
Name	BannerPlugin
Description	在每个生成的 chunk 顶部添加 banner
Name	CommonsChunkPlugin
Description	提取 chunks 之间共享的通用模块
Name	CompressionWebpackPlugin
Description	预先准备的资源压缩版本，使用 Content-Encoding 提供访问服务
Name	ContextReplacementPlugin
Description	重写 <code>require</code> 表达式的推断上下文
Name	CopyWebpackPlugin
Description	将单个文件或整个目录复制到构建目录
Name	DefinePlugin
Description	允许在编译时(compile time)配置的全局常量
Name	DllPlugin
Description	为了极大减少构建时间，进行分离打包
Name	EnvironmentPlugin
Description	<code>DefinePlugin</code> 中 <code>process.env</code> 键的简写方式。
Name	ExtractTextWebpackPlugin
Description	从 bundle 中提取文本 (CSS) 到单独的文件
Name	HotModuleReplacementPlugin
Description	启用模块热替换(Enable Hot Module Replacement - HMR)
Name	HtmlWebpackPlugin
Description	简单创建 HTML 文件，用于服务器访问
Name	I18nWebpackPlugin
Description	为 bundle 增加国际化支持

Name	IgnorePlugin
Description	从 bundle 中排除某些模块
Name	LimitChunkCountPlugin
Description	设置 chunk 的最小/最大限制，以微调和控制 chunk
Name	LoaderOptionsPlugin
Description	用于从 webpack 1 迁移到 webpack 2
Name	MinChunkSizePlugin
Description	确保 chunk 大小超过指定限制
Name	MiniCssExtractPlugin
Description	为每个引入 CSS 的 JS 文件创建一个 CSS 文件
Name	NoEmitOnErrorsPlugin
Description	在输出阶段时，遇到编译错误跳过
Name	NormalModuleReplacementPlugin
Description	替换与正则表达式匹配的资源
Name	NpmInstallWebpackPlugin
Description	在开发环境下自动安装缺少的依赖
Name	ProgressPlugin
Description	报告编译进度
Name	ProvidePlugin
Description	不必通过 import/require 使用模块
Name	SourceMapDevToolPlugin
Description	对 source map 进行更细粒度的控制
Name	EvalSourceMapDevToolPlugin
Description	对 eval source map 进行更细粒度的控制
Name	UglifyjsWebpackPlugin
Description	可以控制项目中 UglifyJS 的版本
Name	TerserPlugin
Description	允许控制项目中 Terser 的版本
Name	ZopfliWebpackPlugin
Description	通过 node-zopfli 将资源预先压缩的版本

更多第三方插件，请查看 [awesome-webpack](#) 列表。



AutomaticPrefetchPlugin

The `AutomaticPrefetchPlugin` discovers **all modules** from the previous compilation upfront while watching for changes, trying to improve the incremental build times. Compared to `PrefetchPlugin` which discovers a **single module** upfront.

May or may not have a performance benefit since the incremental build times are pretty fast.

webpack.config.js

```
module.exports = {
  // ...
  plugins: [
    new webpack.AutomaticPrefetchPlugin()
  ]
};
```

ZopfliWebpackPlugin

Node-Zopfli plugin for Webpack.

安装

```
npm i -D zopfli-webpack-plugin
```

Usage

```
var ZopfliPlugin = require("zopfli-webpack-plugin");
module.exports = {
  plugins: [
    new ZopfliPlugin({
      asset: "[path].gz[query]",
      algorithm: "zopfli",
      test: /\.js|html$/,
      threshold: 10240,
      minRatio: 0.8
    })
  ]
}
```

Arguments

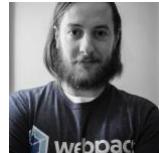
- **asset:** The target asset name. `[file]` is replaced with the original asset. `[path]` is replaced with the path of the original asset and `[query]` with the query. Defaults to "`[path].gz[query]`".
- **filename:** A `function(asset)` which receives the asset name (after processing `asset` option) and returns the new asset name. Defaults to `false`.
- **algorithm:** Can be a `function(buf, callback)` or a string. For a string the algorithm is taken from `zopfli`.
- **test:** All assets matching this RegExp are processed. Defaults to every asset.
- **threshold:** Only assets bigger than this size are processed. In bytes. Defaults to 0.
- **minRatio:** Only assets that compress better than this ratio are processed. Defaults to 0.8.
- **deleteOriginalAssets:** Whether to delete the original assets or not. Defaults to `false`.

Option Arguments

- **verbose:** Default: `false`,
- **verbose_more:** Default: `false`,

- numiterations: Default: 15,
- blocksplitting: Default: true,
- blocksplittinglast: Default: false,
- blocksplittingmax: Default: 15

Maintainers



Juho Vepsäläinen Joshua Wiens Kees Kluskens Sean Larkin

BannerPlugin

为每个 chunk 文件头部添加 banner。

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

new webpack.BannerPlugin(banner);
// or
new webpack.BannerPlugin(options);
```

选项

```
{
  banner: string | function, // 其值为字符串或函数，将作为注释存在
  raw: boolean, // 如果值为 true，将直出，不会被作为注释
  entryOnly: boolean, // 如果值为 true，将只在入口 chunks 文件中添加
  test: string | RegExp | Array,
  include: string | RegExp | Array,
  exclude: string | RegExp | Array,
}
```

Usage

```
import webpack from 'webpack';

// string
new webpack.BannerPlugin({
  banner: 'hello world'
});

// function
new webpack.BannerPlugin({
  banner: (yourVariable) => { return `yourVariable: ${yourVariable}`; }
});
```

占位符(placeholder)

从 webpack 2.5.0 开始，会对 `banner` 字符串中的占位符取值：

```
import webpack from 'webpack';

new webpack.BannerPlugin({
  banner: 'hash:[hash], chunkhash:[chunkhash], name:[name], filebase:[f:]'
});
```

ClosureWebpackPlugin

npm package 2.0.1

This plugin supports the use of Google's Closure Tools with webpack.

Note: This is the webpack 4 branch.

Closure-Compiler is a full optimizing compiler and transpiler. It offers unmatched optimizations, provides type checking and can easily target transpilation to different versions of ECMASCIPT.

Closure-Library is a utility library designed for full compatibility with Closure-Compiler.

Older Versions

For webpack 3 support, see <https://github.com/webpack-contrib/closure-webpack-plugin/tree/webpack-3>

Install

You must install both the google-closure-compiler package as well as the closure-webpack-plugin.

```
npm install --save-dev closure-webpack-plugin google-closure-compiler
```

Usage example

```
const ClosurePlugin = require('closure-webpack-plugin');

module.exports = {
  optimization: {
    minimizer: [
      new ClosurePlugin({mode: 'STANDARD'}, {
        // compiler flags here
        //
        // for debugging help, try these:
        //
        // formatting: 'PRETTY_PRINT'
        // debug: true,
        // renaming: false
      })
    ]
  }
};
```

Options

- **platform** - `native`, `java` or `javascript`. Controls which version to use of closure-compiler. By default the plugin will attempt to automatically choose the fastest option available.
 - `JAVASCRIPT` does not require the JVM to be installed. Not all flags are supported.
 - `JAVA` utilizes the jvm. Utilizes multiple threads for parsing and results in faster compilation for large builds.
 - `NATIVE` only available on linux or MacOS. Faster compilation times without requiring a JVM.
- **mode** - `STANDARD` (default) or `AGGRESSIVE_BUNDLE`. Controls how the plugin utilizes the compiler.
 - `STANDARD` mode, closure-compiler is used as a direct replacement for other minifiers as well as most Babel transformations.
 - `AGGRESSIVE_BUNDLE` mode, the compiler performs additional optimizations of modules to produce a much smaller file
- **childCompilations** - boolean or function. Defaults to `false`. In order to decrease build times, this plugin by default only operates on the main compilation. Plugins such as `extract-text-plugin` and `html-webpack-plugin` run as child compilations and usually do not need transpilation or minification. You can enable this for all child compilations by setting this option to `true`. For specific control, the option can be set to a function which will be passed a `compilation` object.
Example: `function(compilation) { return /html-webpack/.test(compilation.name); }`
- **output** - An object with either `filename` or `chunkfilename` properties. Used to override the output file naming for a particular compilation. See <https://webpack.js.org/configuration/output/> for details.

Compiler Flags

The plugin controls several compiler flags. The following flags should not be used in any mode:

- `module_resolution`
- `output_wrapper`
- `dependency_mode`
- `create_source_map`
- `module`
- `entry_point`

Aggressive Bundle Mode

In this mode, the compiler rewrites CommonJS modules and hoists require calls. Some modules are not compatible with this type of rewriting. In particular, hoisting will cause the following code to execute out of order:

```
const foo = require('foo');
addPolyfillToFoo(foo);
const bar = require('bar');
```

Aggressive Bundle Mode utilizes a custom runtime in which modules within a chunk file are all included in the same scope. This avoids the cost of small modules.

In Aggressive Bundle Mode, a file can only appear in a single output chunk. Use the [Split Chunks Plugin](#) to split duplicated files into a single output chunk. If a module is utilized by more than one chunk, the plugin will move it up to the first common parent to prevent code duplication.

The [concatenatedModules optimization](#) is not compatible with this mode since Closure-Compiler performs an equivalent optimization). The plugin will emit a warning if this optimization is not disabled.

Multiple Output Languages

You can add the plugin multiple times. This easily allows you to target multiple output languages. Use `ECMASCRIPT_2015` for modern browsers and `ECMASCRIPT5` for older browsers.

Use the `output` option to change the filenames of specific plugin instances.

Use `<script type="module" src="es6_out_path.js">` to target modern browsers and `<script nomodule src="es5_out_path.js">` for older browsers.

See the [es5 and es6 output demo](#) for an example.

Other tips for Use

- Don't use babel at the same time - closure-compiler is also a transpiler. If you need features not yet supported by closure-compiler, have babel only target those features. Closure Compiler can transpile async/await - you don't need babel for that functionality either.

Closure Library Plugin

In order for webpack to recognize `goog.require`, `goog.provide`, `goog.module` and

related primitives, a separate plugin is shipped.

```
const ClosurePlugin = require('closure-webpack-plugin');

module.exports = {
  plugins: [
    new ClosurePlugin.LibraryPlugin({
      closureLibraryBase: require.resolve(
        'google-closure-library/closure/goog/base'
      ),
      deps: [
        require.resolve('google-closure-library/closure/goog/deps'),
        './public/deps.js',
      ],
    }),
  ],
};
```

The plugin adds extra functionality to support using Closure Library without Closure Compiler. This is typically used during development mode. When the webpack mode is production, only dependency information is provided to webpack as Closure Compiler will natively recognize the Closure Library primitives.

The Closure Library Plugin is only compatible with the `AGGRESSIVE_BUNDLE` mode of the Closure-Compiler webpack plugin.

Options

- **closureLibraryBase** - (optional) string. Path to the base.js file in Closure-Library.
- **deps** - (optional) string or Array. Closures style dependency mappings. Typically generated by the [depswriter.py](#) script included with Closure-Library.
- **extraDeps** - (optional) Object. Mapping of namespace to file path for closure-library provided namespaces.

Maintainers



Chad Killingsworth

Joshua Wiens

CommonsChunkPlugin

CommonsChunkPlugin 插件，是一个可选的用于建立一个独立文件(又称作 chunk)的功能，这个文件包括多个入口 chunk 的公共模块。

The CommonsChunkPlugin 已经从 webpack v4 legato 中移除。想要了解在最新版本中如何处理 chunk，请查看 [SplitChunksPlugin](#)。

通过将公共模块拆出来，最终合成的文件能够在最开始的时候加载一次，便存到缓存中供后续使用。这个带来页面速度上的提升，因为浏览器会迅速将公共的代码从缓存中取出来，而不是每次访问一个新页面时，再去加载一个更大的文件。

```
new webpack.optimize.CommonsChunkPlugin(options);
```

配置

```
{
  name: string, // or
  names: string[],
  // 这是 common chunk 的名称。已经存在的 chunk 可以通过传入一个已存在的 chunk :
  // 如果一个字符串数组被传入，这相当于插件针对每个 chunk 名被多次调用
  // 如果该选项被忽略，同时 `options.async` 或者 `options.children` 被设置，所
  // 否则 `options.filename` 会用于作为 chunk 名。
  // When using `options.async` to create common chunks from other async
  // chunk name here instead of omitting the `option.name`.

  filename: string,
  // common chunk 的文件名模板。可以包含与 `output.filename` 相同的占位符。
  // 如果被忽略，原本的文件名不会被修改(通常是 `output.filename` 或者 `output.c
  // This option is not permitted if you're using `options.async` as we

  minChunks: number|Infinity|function(module, count) => boolean,
  // 在传入 公共chunk(common chunk) 之前所需要包含的最少数量的 chunks 。
  // 数量必须大于等于2，或者少于等于 chunks的数量
  // 传入 `Infinity` 会马上生成 公共chunk，但里面没有模块。
  // 你可以传入一个 `function`，以添加定制的逻辑（默认是 chunk 的数量）

  chunks: string[],
  // 通过 chunk name 去选择 chunks 的来源。chunk 必须是 公共chunk 的子模块。
  // 如果被忽略，所有的，所有的 入口chunk (entry chunk) 都会被选择。

  children: boolean,
  // 如果设置为 `true`，所有公共 chunk 的子模块都会被选择

  deepChildren: boolean,
  // 如果设置为 `true`，所有公共 chunk 的后代模块都会被选择

  async: boolean|string,
  // 如果设置为 `true`，一个异步的 公共chunk 会作为 `options.name` 的子模块，和
  // 它会与 `options.chunks` 并行被加载。
  // Instead of using `option.filename`，it is possible to change the na
```

```
// the desired string here instead of `true`.  
  
minSize: number,  
// 在公共chunk 被创建之前，所有 公共模块 (common module) 的最少大小。  
}
```

webpack1 构造函数 new webpack.optimize.CommonsChunkPlugin(options, filenameTemplate, selectedChunks, minChunks) 不再被支持。请使用相应的选项对象。

例子

公共chunk 用于 入口chunk (entry chunk)

生成一个额外的 chunk 包含入口chunk 的公共模块。

```
new webpack.optimize.CommonsChunkPlugin({  
  name: 'commons',  
  // (公共 chunk(common chunk) 的名称)  
  
  filename: 'commons.js',  
  // (公共chunk 的文件名)  
  
  // minChunks: 3,  
  // (模块必须被3个 入口chunk 共享)  
  
  // chunks: ["pageA", "pageB"],  
  // (只使用这些 入口chunk)  
});
```

你必须在 入口chunk 之前加载生成的这个 公共chunk:

```
<script src="commons.js" charset="utf-8"></script>  
<script src="entry.bundle.js" charset="utf-8"></script>
```

明确第三方库 chunk

将你的代码拆分成公共代码和应用代码。

```
module.exports = {  
  //...  
  entry: {  
    vendor: ['jquery', 'other-lib'],  
    app: './entry'  
  },  
  plugins: [  
    new webpack.optimize.CommonsChunkPlugin({  
      name: 'vendor',  
      // filename: "vendor.js"  
      // (给 chunk 一个不同的名字)  
  
      minChunks: Infinity,
```

```

    // (随着 entry chunk 越来越多,
    // 这个配置保证没其它的模块会打包进 vendor chunk)
  })
]

};

<script src="vendor.js" charset="utf-8"></script>
<script src="app.js" charset="utf-8"></script>

```

结合长期缓存，你可能需要使用这个插件去避免公共chunk改变。你也需要使用 records 去保持稳定的模块 id，例如，使用 [NamedModulesPlugin](#) 或 [HashedModuleIdsPlugin](#)。

将公共模块打包进父 chunk

使用代码拆分功能，一个 chunk 的多个子 chunk 会有公共的依赖。为了防止重复，可以将这些公共模块移入父 chunk。这会减少总体的大小，但会对首次加载时间产生不良影响。如果预期到用户需要下载许多兄弟 chunks（例如，入口 trunk 的子 chunk），那这对改善加载时间将非常有用。

```

new webpack.optimize.CommonsChunkPlugin({
  // names: ["app", "subPageA"]
  // (选择 chunks，或者忽略该项设置以选择全部 chunks)

  children: true,
  // (选择所有被选 chunks 的子 chunks)

  // minChunks: 3,
  // (在提取之前需要至少三个子 chunk 共享这个模块)
});

```

额外的异步 公共chunk

与上面的类似，但是并非将公共模块移动到父 chunk（增加初始加载时间），而是使用新的异步加载的额外公共chunk。当下载额外的 chunk 时，它将自动并行下载。

```

new webpack.optimize.CommonsChunkPlugin({
  name: 'app',
  // or
  names: ['app', 'subPageA'],
  // the name or list of names must match the name or names
  // of the entry points that create the async chunks

  children: true,
  // (选择所有被选 chunks 的子 chunks)

  async: true,
  // (创建一个异步 公共chunk)

  minChunks: 3,
  // (在提取之前需要至少三个子 chunk 共享这个模块)
}

```

```
});
```

给 minChunks 配置传入函数

你也可以给 `minChunks` 传入一个函数。这个函数会被 `CommonsChunkPlugin` 插件回调，并且调用函数时会传入 `module` 和 `count` 参数。

`module` 参数代表每个 `chunks` 里的模块，这些 `chunks` 是你通过 `name/names` 参数传入的。`module` has the shape of a `NormalModule`, which has two particularly useful properties for this use case:

- `module.context`: The directory that stores the file. For example:
`'/my_project/node_modules/example-dependency'`
- `module.resource`: The name of the file being processed. For example:
`'/my_project/node_modules/example-dependency/index.js'`

`count` 参数表示 `module` 被使用的 `chunk` 数量。

当你想要对 `CommonsChunk` 如何决定模块被打包到哪里的算法有更为细致的控制，这个配置就会非常有用。

```
new webpack.optimize.CommonsChunkPlugin({
  name: 'my-single-lib-chunk',
  filename: 'my-single-lib-chunk.js',
  minChunks: function(module, count) {
    // 如果模块是一个路径，而且在路径中有 "somelib" 这个名字出现，
    // 而且它还被三个不同的 chunks/入口chunk 所使用，那请将它拆分到
    // 另一个分开的 chunk 中，chunk 的 keyname 是 "my-single-lib-chunk"，而
    return module.resource && (/somelib/).test(module.resource) && count > 1
  }
});
```

正如上面看到的，这个例子允许你只将其中一个库移到一个分开的文件当中，当而仅当函数中的所有条件都被满足了。

This concept may be used to obtain implicit common vendor chunks:

```
new webpack.optimize.CommonsChunkPlugin({
  name: 'vendor',
  minChunks: function (module) {
    // this assumes your vendor imports exist in the node_modules directory
    return module.context && module.context.includes('node_modules');
  }
});
```

In order to obtain a single CSS file containing your application and vendor CSS, use the following `minChunks` function together with `ExtractTextPlugin`:

```
new webpack.optimize.CommonsChunkPlugin({
  name: 'vendor',
```

```

minChunks: function (module) {
  // This prevents stylesheet resources with the .css or .scss extens:
  // from being moved from their original chunk to the vendor chunk
  if(module.resource && (/^.*\.(css|scss)$/).test(module.resource)) {
    return false;
  }
  return module.context && module.context.includes('node_modules');
}
);

```

Manifest file

To extract the webpack bootstrap logic into a separate file, use the `CommonsChunkPlugin` on a `name` which is not defined as `entry`. Commonly the name `manifest` is used. See the [caching guide](#) for details.

```

new webpack.optimize.CommonsChunkPlugin({
  name: 'manifest',
  minChunks: Infinity
});

```

Combining implicit common vendor chunks and manifest file

Since the `vendor` and `manifest` chunk use a different definition for `minChunks`, you need to invoke the plugin twice:

```

[
  new webpack.optimize.CommonsChunkPlugin({
    name: 'vendor',
    minChunks: function(module) {
      return module.context && module.context.includes('node_modules');
    }
  },
  new webpack.optimize.CommonsChunkPlugin({
    name: 'manifest',
    minChunks: Infinity
  }),
];

```

More Examples

- [Common and Vendor Chunks](#)
- [Multiple Common Chunks](#)
- [Multiple Entry Points with Commons Chunk](#)

CompressionWebpackPlugin

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![cover][cover]][cover-url] [![chat][chat]][chat-url] [![size][size]][size-url]

预先提供带 Content-Encoding 编码的压缩版本的资源。

需求

This module requires a minimum of Node v6.9.0 and Webpack v4.0.0.

起步

To begin, you'll need to install `compression-webpack-plugin`:

```
$ npm install compression-webpack-plugin --save-dev
```

Then add the plugin to your `webpack` config. For example:

webpack.config.js

```
const CompressionPlugin = require('compression-webpack-plugin');

module.exports = {
  plugins: [
    new CompressionPlugin()
  ]
}
```

And run `webpack` via your preferred method.

选项

test

类型: `String|RegExp|Array<String|RegExp>` **Default:** `undefined`

匹配所有对应的文件。

```
// 在 webpack.config.js 中
new CompressionPlugin({
  test: /\.js(\?.*)?$/i
})
```

include

类型: String|RegExp|Array<String|RegExp> **默认:** undefined

所有包含(include)的文件。

```
// 在 webpack.config.js 中
new CompressionPlugin({
  include: /\includes/
})
```

exclude

类型: String|RegExp|Array<String|RegExp> **默认:** undefined

所有排除(exclude)的文件。

```
// 在 webpack.config.js 中
new CompressionPlugin({
  exclude: /\excludes/
})
```

cache

类型: Boolean|String **默认:** false

Enable file caching. The default path to cache directory:

`node_modules/.cache/compression-webpack-plugin.`

Boolean

Enable/disable file caching.

```
// 在 webpack.config.js 中
new CompressionPlugin({
  cache: true
})
```

String

Enable file caching and set path to cache directory.

```
// 在 webpack.config.js 中
new CompressionPlugin({
  cache: 'path/to/cache'
})
```

filename

类型: String|Function **默认:** [path].gz[query]

目标资源文件名称。

String

[file] is replaced with the original asset filename. [path] is replaced with the path of the original asset. [query] is replaced with the query.

```
// 在 webpack.config.js 中
new CompressionPlugin({
  filename: '[path].gz[query]'
})
```

Function

```
// 在 webpack.config.js 中
new CompressionPlugin({
  filename(info) {
    // info.file is the original asset filename
    // info.path is the path of the original asset
    // info.query is the query
    return `${info.path}.gz${info.query}`
  }
})
```

algorithm

类型: String|Function **默认:** gzip

The compression algorithm/function.

String

The algorithm is taken from [zlib](#).

```
// 在 webpack.config.js 中
new CompressionPlugin({
  algorithm: 'gzip'
})
```

Function

Allow to specify a custom compression function.

```
// 在 webpack.config.js 中
new CompressionPlugin({
  algorithm(input, compressionOptions, callback) {
```

```
        return compressionFunction(input, compressionOptions, callback);
    }
})
```

compressionOptions

类型: Object 默认: { level: 9 }

If you use custom function for the algorithm option, the default value is {}.

Compression options. You can find all options here [zlib](#).

```
// 在 webpack.config.js 中
new CompressionPlugin({
  compressionOptions: { level: 1 }
})
```

threshold

类型: Number 默认: 0

只处理比这个值大的资源。按字节计算。

```
// 在 webpack.config.js 中
new CompressionPlugin({
  threshold: 8192
})
```

minRatio

类型: Number 默认: 0.8

只有压缩率比这个值小的资源才会被处理 ($\text{minRatio} = \text{压缩大小} / \text{原始大小}$)。
Example: you have `image.png` file with 1024b size, compressed version of file has 768b size, so `minRatio` equal 0.75. In other words assets will be processed when the Compressed Size / Original Size value less `minRatio` value. You can use 1 value to process all assets.

```
// 在 webpack.config.js 中
new CompressionPlugin({
  minRatio: 0.8
})
```

deleteOriginalAssets

类型: Boolean 默认: false

是否删除原始资源。

```
// 在 webpack.config.js 中
new CompressionPlugin({
  deleteOriginalAssets: true
})
```

示例

使用 Zopfli

Prepare compressed versions of assets using `zopfli` library.

i `@gfx/zopfli` require minimum 8 version of node.

To begin, you'll need to install `@gfx/zopfli`:

```
$ npm install @gfx/zopfli --save-dev
```

webpack.config.js

```
const zopfli = require('@gfx/zopfli');

module.exports = {
  plugins: [
    new CompressionPlugin({
      compressionOptions: {
        numiterations: 15
      },
      algorithm(input, compressionOptions, callback) {
        return zopfli.gzip(input, compressionOptions, callback);
      }
    })
  ]
}
```

使用 Brotli

Brotli is a compression algorithm originally developed by Google, and offers compression superior to gzip.

Node 11.7.0 and later has native support for Brotli compression in its zlib module.

We can take advantage of this built-in support for Brotli in Node 11.7.0 and later by just passing in the appropriate `algorithm` to the `CompressionPlugin`:

```
// 在 webpack.config.js 中
module.exports = {
  plugins: [
    new CompressionPlugin({
      filename: '[path].br[query]',
      algorithm: 'brotliCompress',
    })
  ]
}
```

```
    test: /\.js|css|html|svg$/,
    compressionOptions: { level: 11 },
    threshold: 10240,
    minRatio: 0.8,
    deleteOriginalAssets: false
  )
]
}
```

N.B.: The `level` option matches `BROTLI_PARAM_QUALITY` for Brotli-based streams

Contributing

Please take a moment to read our contributing guidelines if you haven't yet done so.

[CONTRIBUTING](#)

License

[MIT](#)

ContextReplacementPlugin

上下文(*context*) 与一个 含有表达式的 require 语句 相关，例如

`require('./locale/' + name + '.json')`。遇见此类表达式时，webpack 查找目录`('./locale/')`下符合正则表达式`(/^.*\.json$/)`的文件。由于 `name` 在编译时 (compile time)还是未知的，webpack 会将每个文件都作为模块引入到 bundle 中。

上下文替换插件 (ContextReplacementPlugin) 允许你覆盖查找规则，该插件有许多配置方式：

用法

```
new webpack.ContextReplacementPlugin(  
  resourceRegExp: RegExp,  
  newContentResource?: string,  
  newContentRecursive?: boolean,  
  newContentRegExp?: RegExp  
)
```

如果资源（或目录）符合 `resourceRegExp` 正则表达式，插件会替换默认资源为 `newContentResource`，布尔值 `newContentRecursive` 表明是否使用递归查找，`newContentRegExp` 用于筛选新上下文里的资源。如果 `newContentResource` 为相对路径，会相对于前一匹配资源路径去解析。

这是一个限制模块使用的小例子：

```
new webpack.ContextReplacementPlugin(  
  /moment[/\\"locale$/,  
  /de|fr|hu/  
)
```

限定查找 `moment/locale` 上下文里符合 `/de|fr|hu/` 表达式的文件，因此也只会打包这几种本地化内容（更多详细信息，请查看[这个 issue](#)）。

内容回调函数

```
new webpack.ContextReplacementPlugin(  
  resourceRegExp: RegExp,  
  newContentCallback: (data) => void  
)
```

`newContentCallback` 函数的第一形参为上下文模块工厂 (`ContextModuleFactory`) 的 `data` 对象，你需要覆写该对象的 `request` 属性。

使用这个回调函数，我们可以动态地将请求重定向到一个新的位置：

```
new webpack.ContextReplacementPlugin(/^\.\/locale$/, (context) => {
  if ( !/\moment\//.test(context.context) ) return;

  Object.assign(context, {
    regExp: /^\.\/\w+/,  
    request: '../..../locale' // 相对路径解析
  });
});
```

其他选项

`newContentResource` 和 `newContentCreateContextMap` 参数也可用：

```
new webpack.ContextReplacementPlugin(  
  resourceRegExp: RegExp,  
  newContentResource: string,  
  newContentCreateContextMap: object // 将运行时请求(runtime-request)映射到  
);
```

这两个参数可以一起使用，来更加有针对性的重定向请求。

`newContentCreateContextMap` 允许你将运行时的请求，映射为形式为对象的编译请求：

```
new ContextReplacementPlugin(/selector/, './folder', {  
  './request': './request',  
  './other-request': './new-request'  
});
```

CopyWebpackPlugin

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![cover][cover]][cover-url] [![chat][chat]][chat-url] [![size][size]][size-url]

Copies individual files or entire directories to the build directory.

Getting Started

To begin, you'll need to install `copy-webpack-plugin`:

```
$ npm install copy-webpack-plugin --save-dev
```

Then add the loader to your `webpack` config. For example:

webpack.config.js

```
const CopyPlugin = require('copy-webpack-plugin');

module.exports = {
  plugins: [
    new CopyPlugin([
      { from: 'source', to: 'dest' },
      { from: 'other', to: 'public' },
    ]),
  ],
};

};
```

i If you want `webpack-dev-server` to write files to the output directory during development, you can force it with the `writeToDisk` option or the `write-file-webpack-plugin`.

Options

The plugin's signature:

webpack.config.js

```
module.exports = {
  plugins: [new CopyPlugin(patterns, options)],
};
```

Patterns

Name _____ from _____

Type	{String Object}
Default	undefined
Description	Glob or path from where we copy files.
Name	<u>to</u>
Type	{String}
Default	undefined
Description	Output path.
Name	<u>context</u>
Type	{String}
Default	options.context compiler.options.context
Description	A path that determines how to interpret the <code>from</code> path.
Name	<u>toType</u>
Type	{String}
Default	undefined
Description	Determinate what is <code>to</code> option - directory, file or template.
Name	<u>test</u>
Type	{RegExp}
Default	undefined
Description	Pattern for extracting elements to be used in <code>to</code> templates.
Name	<u>force</u>
Type	{Boolean}
Default	false
Description	Overwrites files already in <code>compilation.assets</code> (usually added by other plugins/loaders).
Name	<u>ignore</u>
Type	{Array}
Default	[]
Description	Globs to ignore files.
Name	<u>flatten</u>
Type	{Boolean}
Default	false

Description	Removes all directory references and only copies file names.
Name	<u>transform</u>
Type	{Function Promise}
Default	undefined
Description	Allows to modify the file contents.
Name	<u>cache</u>
Type	{Boolean Object}
Default	false
Description	Enable <code>transform</code> caching. You can use <code>{ cache: { key: 'my-cache-key' } }</code> to invalidate the cache.
Name	<u>transformPath</u>
Type	{Function Promise}
Default	undefined
Description	Allows to modify the writing path.

from

Type: String\|Object Default: undefined

Glob or path from where we copy files. Globs accept [minimatch options](#).

You can defined `from` as Object and use the [node-glob options](#).

□□ Don't use directly \\ in `from` (i.e `path\to\file.ext`) option because on UNIX the backslash is a valid character inside a path component, i.e., it's not a separator. On Windows, the forward slash and the backward slash are both separators. Instead please use / or `path` methods.

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      'relative/path/to/file.ext',
      '/absolute/path/to/file.ext',
      'relative/path/to/dir',
      '/absolute/path/to/dir',
      '**/*',
      { glob: '**/*', dot: false },
    ]),
  ],
};
```

to

Type: String Default: undefined

Output path.

□□ Don't use directly \\ in `to` (i.e `path\to\dest`) option because on UNIX the backslash is a valid character inside a path component, i.e., it's not a separator. On Windows, the forward slash and the backward slash are both separators. Instead please use / or `path` methods.

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      { from: '**/*', to: 'relative/path/to/dest/' },
      { from: '**/*', to: '/absolute/path/to/dest/' },
    ]),
  ],
};
```

context

Type: String Default: `options.context|compiler.options.context`

A path that determines how to interpret the `from` path.

□□ Don't use directly \\ in `context` (i.e `path\to\context`) option because on UNIX the backslash is a valid character inside a path component, i.e., it's not a separator. On Windows, the forward slash and the backward slash are both separators. Instead please use / or `path` methods.

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      {
        from: 'src/*.txt',
        to: 'dest/',
        context: 'app/',
      },
    ]),
  ],
};
```

toType

Type: String Default: undefined

Determinate what is `to` option - directory, file or template. Sometimes it is hard to say what is `to`, example `path/to/dir-with.ext`. If you want to copy files in directory you need use `dir` option. We try to automatically determine the `type` so you most likely do not need this option.

Name	<code>'dir'</code>
Type	{String}
Default	undefined
Description	If <code>from</code> is directory, <code>to</code> has no extension or ends in <code>'/'</code>
Name	<code>'file'</code>
Type	{String}
Default	undefined
Description	If <code>to</code> has extension or <code>from</code> is file
Name	<code>'template'</code>
Type	{String}
Default	undefined
Description	If <code>to</code> contains a <u>template pattern</u>

`'dir'`

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      {
        from: 'path/to/file.txt',
        to: 'directory/with/extension.ext',
        toType: 'dir',
      },
    ]),
  ],
};
```

`'file'`

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
```

```
{
  from: 'path/to/file.txt',
  to: 'file/without/extension',
  toType: 'file',
},
]),
],
},
};
```

'template'

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      {
        from: 'src/',
        to: 'dest/[name].[hash].[ext]',
        toType: 'template',
      },
    ]),
  ],
};
```

test

Type: `RegExp` Default: `undefined`

Pattern for extracting elements to be used in `to` templates.

Defines a `{RegExp}` to match some parts of the file path. These capture groups can be reused in the `name` property using `[N]` placeholder. Note that `[0]` will be replaced by the entire path of the file, whereas `[1]` will contain the first capturing parenthesis of your `{RegExp}` and so on...

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      {
        from: '**/*',
        to: '[1]-[2].[hash].[ext]',
        test: /(([^\-]+)\-([^\-]+)\.(\w+)\.png$/,
      },
    ]),
  ],
};
```

force

Type: Boolean Default: false

Overwrites files already in `compilation.assets` (usually added by other plugins/loaders).

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      {
        from: 'src/**/*',
        to: 'dest/',
        force: true,
      },
    ]),
  ],
};
```

ignore

Type: Array Default: []

Globs to ignore files.

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      {
        from: 'src/**/*',
        to: 'dest/',
        ignore: ['*.js'],
      },
    ]),
  ],
};
```

flatten

Type: Boolean Default: false

Removes all directory references and only copies file names.

- □ If files have the same name, the result is non-deterministic.

webpack.config.js

```
module.exports = {
```

```
plugins: [
  new CopyPlugin([
    {
      from: 'src/**/*',
      to: 'dest/',
      flatten: true,
    },
  ]),
],
};
```

transform

Type: Function | Promise **Default:** undefined

Allows to modify the file contents.

{Function}

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      {
        from: 'src/*.png',
        to: 'dest/',
        transform(content, path) {
          return optimize(content);
        },
      },
    ]),
  ],
};
```

{Promise}

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      {
        from: 'src/*.png',
        to: 'dest/',
        transform(content, path) {
          return Promise.resolve(optimize(content));
        },
      },
    ]),
  ],
};
```

cache

Type: Boolean | Object **Default:** false

Enable/disable transform caching. You can use { cache: { key: 'my-cache-key' } } to invalidate the cache. Default path to cache directory: node_modules/.cache/copy-webpack-plugin.

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      {
        from: 'src/*.png',
        to: 'dest/',
        transform(content, path) {
          return optimize(content);
        },
        cache: true,
      },
    ]),
  ],
};
```

transformPath

Type: Function | Promise **Default:** undefined

Allows to modify the writing path.

□□ Don't return directly \\ in transformPath (i.e path\to\newFile) option because on UNIX the backslash is a valid character inside a path component, i.e., it's not a separator. On Windows, the forward slash and the backward slash are both separators. Instead please use / or path methods.

{Function}

webpack.config.js

```
module.exports = {
  plugins: [
    new CopyPlugin([
      {
        from: 'src/*.png',
        to: 'dest/',
        transformPath(targetPath, absolutePath) {
          return 'newPath';
        },
      },
    ]),
  ],
};
```

```

        ],
    ],
};

{Promise}

```

webpack.config.js

```

module.exports = {
  plugins: [
    new CopyPlugin([
      {
        from: 'src/*.png',
        to: 'dest/',
        transformPath(targetPath, absolutePath) {
          return Promise.resolve('newPath');
        },
      },
    ]),
  ],
};

```

Options

Name	<u>logLevel</u>
Type	{String}
Default	'warn'
Description	Level of messages that the module will log
Name	<u>ignore</u>
Type	{Array}
Default	[]
Description	Array of globs to ignore (applied to <code>from</code>)
Name	<u>context</u>
Type	{String}
Default	<code>compiler.options.context</code>
Description	A path that determines how to interpret the <code>from</code> path, shared for all patterns
Name	<u>copyUnmodified</u>
Type	{Boolean}
Default	false
Description	Copies files, regardless of modification when using <code>watch</code> or <code>webpack-dev-server</code> . All files are copied on first build, regardless of this option

logLevel

This property defines the level of messages that the module will log. Valid levels include:

- trace
- debug
- info
- warn (default)
- error
- silent

Setting a log level means that all other levels below it will be visible in the console.

Setting `logLevel: 'silent'` will hide all console output. The module leverages [`webpack-log`](#) for logging management, and more information can be found on its page.

webpack.config.js

```
module.exports = {
  plugins: [new CopyPlugin([...patterns], { LogLevel: 'debug' })],
};
```

ignore

Array of globs to ignore (applied to `from`).

webpack.config.js

```
module.exports = {
  plugins: [new CopyPlugin([...patterns], { ignore: ['*.js', '*.css'] })],
};
```

context

A path that determines how to interpret the `from` path, shared for all patterns.

webpack.config.js

```
module.exports = {
  plugins: [new CopyPlugin([...patterns], { context: '/app' })],
};
```

copyUnmodified

Copies files, regardless of modification when using `watch` or `webpack-dev-server`. All files are copied on first build, regardless of this option.

i By default, we only copy **modified** files during a `webpack --watch` or `webpack-dev-server` build. Setting this option to `true` will copy all files.

webpack.config.js

```
module.exports = {  
  plugins: [new CopyPlugin([...patterns], { copyUnmodified: true })],  
};
```

Contributing

Please take a moment to read our contributing guidelines if you haven't yet done so.

CONTRIBUTING

License

MIT

CssWebpackPlugin

Nothing to see here yet

DefinePlugin

DefinePlugin 允许创建一个在编译时可以配置的全局常量。这可能会对开发模式和生产模式的构建允许不同的行为非常有用。如果在开发构建中，而在发布构建中执行日志记录，则可以使用全局常量来决定是否记录日志。这就是 DefinePlugin 的用处，设置它，就可以忘记开发环境和生产环境构建的规则。

```
new webpack.DefinePlugin({  
    // Definitions...  
});
```

用法

每个传进 DefinePlugin 的键值都是一个标志符或者多个用 . 连接起来的标志符。

- 如果这个值是一个字符串，它会被当作一个代码片段来使用。
- 如果这个值不是字符串，它会被转化为字符串(包括函数)。
- 如果这个值是一个对象，它所有的 key 会被同样的方式定义。
- 如果在一个 key 前面加了 `typeof`, 它会被定义为 `typeof` 调用。

这些值会被内联进那些允许传一个代码压缩参数的代码中，从而减少冗余的条件判断。

```
new webpack.DefinePlugin({  
    PRODUCTION: JSON.stringify(true),  
    VERSION: JSON.stringify('5fa3b9'),  
    BROWSER_SUPPORTS_HTML5: true,  
    TWO: '1+1',  
    'typeof window': JSON.stringify('object'),  
    'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV)  
});  
  
console.log('Running App version ' + VERSION);  
if(!BROWSER_SUPPORTS_HTML5) require('html5shiv');
```

When defining values for `process` prefer '`process.env.NODE_ENV`':
`JSON.stringify('production')` over `process: { env: { NODE_ENV: JSON.stringify('production') } }`. Using the latter will overwrite the `process` object which can break compatibility with some modules that expect other values on the `process` object to be defined.

注意，因为这个插件直接执行文本替换，给定的值必须包含字符串本身内的实际引号。通常，有两种方式来达到这个效果，使用 `"production"`，或者使用 `JSON.stringify('production')`。

index.js

```
if (!PRODUCTION) {  
  console.log('Debug info');  
}  
  
if (PRODUCTION) {  
  console.log('Production log');  
}
```

通过没有使用压缩的 webpack 的结果:

```
if (!true) {  
  console.log('Debug info');  
}  
if (true) {  
  console.log('Production log');  
}
```

通过使用压缩的 webpack 的结果:

```
console.log('Production log');
```

功能标记(Feature Flags)

使用功能标记来「启用/禁用」「生产/开发」构建中的功能。

```
new webpack.DefinePlugin({  
  'NICE_FEATURE': JSON.stringify(true),  
  'EXPERIMENTAL_FEATURE': JSON.stringify(false)  
});
```

服务 URL(Service URL)

在生产/开发构建中使用不同的服务 URL(Service URL):

```
new webpack.DefinePlugin({  
  'SERVICE_URL': JSON.stringify('http://dev.example.com')  
});
```

DllPlugin

DLLPlugin 和 DLLReferencePlugin 用某种方法实现了拆分 bundles，同时还大大提升了构建的速度。

DllPlugin

这个插件是在一个额外的独立的 webpack 设置中创建一个只有 dll 的 bundle(dll-only-bundle)。这个插件会生成一个名为 `manifest.json` 的文件，这个文件是用来让 `DLLReferencePlugin` 映射到相关的依赖上去的。

- `context` (optional): manifest 文件中请求的上下文(context)(默认值为 webpack 的上下文(context))
- `name`: 暴露出的 DLL 的函数名 (`TemplatePaths: [hash] & [name]`)
- `path`: manifest json 文件的绝对路径 (输出文件)

```
new webpack.DllPlugin(options);
```

在给定的 `path` 路径下创建一个名为 `manifest.json` 的文件。这个文件包含了从 `require` 和 `import` 的 request 到模块 id 的映射。`DLLReferencePlugin` 也会用到这个文件。

这个插件与 `output.library` 的选项相结合可以暴露出 (也叫做放入全局域) dll 函数。

DllReferencePlugin

这个插件是在 webpack 主配置文件中设置的，这个插件把只有 dll 的 bundle(们) (dll-only-bundle(s)) 引用到需要的预编译的依赖。

- `context`: (绝对路径) manifest (或者是内容属性) 中请求的上下文
- `manifest`: 包含 `content` 和 `name` 的对象，或者在编译时(compilation)的一个用于加载的 JSON manifest 绝对路径
- `content` (optional): 请求到模块 id 的映射 (默认值为 `manifest.content`)
- `name` (optional): dll 暴露的地方的名称 (默认值为 `manifest.name`) (可参考 `externals`)
- `scope` (optional): dll 中内容的前缀
- `sourceType` (optional): dll 是如何暴露的 (`libraryTarget`)

```
new webpack.DllReferencePlugin(options);
```

通过引用 dll 的 manifest 文件来把依赖的名称映射到模块的 id 上，之后再在需要的时候通过内置的 `__webpack_require__` 函数来 `require` 他们

与 `output.library` 保持 `name` 的一致性。

模式(Modes)

这个插件支持两种模式，分别是 *作用域(scoped)* 和 *映射(mapped)*。

作用域模式(Scoped Mode)

dll 中的内容可以在模块前缀下才能被引用，举例来说，令 `scope = "xyz"` 的话，这个 dll 中的名为 `abc` 的文件可以通过 `require("xyz/abc")` 来获取

作用域的用例

映射模式(Mapped Mode)

dll 中的内容被映射到了当前目录下。如果一个被 `require` 的文件符合 dll 中的某个文件(解析之后)，那么这个 dll 中的这个文件就会被使用。

由于这是在解析了 dll 中每个文件之后才发生的，相同的路径必须能够确保这个 dll bundle 的使用者(不一定是人，可指某些代码)有权限访问。举例来说，假如一个 dll bundle 中含有 `lodash` 库 以及 文件 `abc`，那么 `require("lodash")` 和 `require("./abc")` 都不会被编译进主要的 bundle 文件，而是会被 dll 所使用。

用法(Usage)

`DllReferencePlugin` 和 `DLL插件DllPlugin` 都是在 另外 的 webpack 设置中使用的。

webpack.vendor.config.js

```
new webpack.DllPlugin({
  context: __dirname,
  name: '[name]_[hash]',
  path: path.join(__dirname, 'manifest.json'),
});
```

webpack.app.config.js

```
new webpack.DllReferencePlugin({
  context: __dirname,
  manifest: require('./manifest.json'),
  name: './my-dll.js',
  scope: 'xyz',
  sourceType: 'commonjs2'
});
```

示例(Examples)

Vendor and User

两个单独的用例，用来分别演示作用域(scope)和上下文(context)。

多个 DllPlugins 和 DllReferencePlugins.

引用参考(References)

Source

- [DllPlugin source](#)
- [DllReferencePlugin source](#)
- [DllEntryPlugin source](#)
- [DllModuleFactory source](#)
- [ManifestPlugin source](#)

Tests

- [DllPlugin creation test](#)
- [DllPlugin without scope test](#)
- [DllReferencePlugin use Dll test](#)

EnvironmentPlugin

EnvironmentPlugin 是一个通过 [DefinePlugin](#) 来设置 `process.env` 环境变量的快捷方式。

用法

EnvironmentPlugin 可以接收键数组或将键映射到其默认值的对象。 (译者注：键是指要设定的环境变量名)

```
new webpack.EnvironmentPlugin(['NODE_ENV', 'DEBUG']);
```

上面的写法和下面这样使用 [DefinePlugin](#) 的效果相同：

```
new webpack.DefinePlugin({
  'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV),
  'process.env.DEBUG': JSON.stringify(process.env.DEBUG)
});
```

使用不存在的环境变量会导致一个 "EnvironmentPlugin - \${key} environment variable is undefined" 错误。

带默认值使用

或者， EnvironmentPlugin 也可以接收一个指定相应默认值的对象，如果在 `process.env` 中对应的环境变量不存在时将使用指定的默认值。

```
new webpack.EnvironmentPlugin({
  NODE_ENV: 'development', // 除非有定义 process.env.NODE_ENV，否则就使用 ''
  DEBUG: false
});
```

从 `process.env` 中取到的值类型均为字符串。

不同于 [DefinePlugin](#)， 默认值将被 EnvironmentPlugin 执行 `JSON.stringify`。

如果要指定一个未设定的默认值， 使用 `null` 来代替 `undefined`。

示例：

让我们看一下对下面这个用来试验的文件 `entry.js` 执行前面配置的 EnvironmentPlugin 的结果：

```
if (process.env.NODE_ENV === 'production') {
  console.log('Welcome to production');
```

```
}
```

```
if (process.env.DEBUG) {
```

```
  console.log('Debugging output');
```

```
}
```

当在终端执行 `NODE_ENV=production webpack` 来构建时，`entry.js` 变成了这样：

```
if ('production' === 'production') { // <-- NODE_ENV 的 'production' 被带
```

```
  console.log('Welcome to production');
```

```
}
```

```
if (false) { // <-- 使用了默认值
```

```
  console.log('Debugging output');
```

```
}
```

执行 `DEBUG=false webpack` 则会生成：

```
if ('development' === 'production') { // <-- 使用了默认值
```

```
  console.log('Welcome to production');
```

```
}
```

```
if ('false') { // <-- DEBUG 的 'false' 被带过来了
```

```
  console.log('Debugging output');
```

```
}
```

DotenvPlugin

The third-party [DotenvPlugin](#) (`dotenv-webpack`) allows you to expose (a subset of) [dotenv variables](#):

```
// .env
```

```
DB_HOST=127.0.0.1
```

```
DB_PASS=foobar
```

```
S3_API=mysecretkey
```



```
new Dotenv({
```

```
  path: './.env', // Path to .env file (this is the default)
```

```
  safe: true // load .env.example (defaults to "false" which does not use)
```

EvalSourceMapDevToolPlugin

This plugin enables more fine grained control of source map generation. It is also enabled automatically by certain settings of the `devtool` configuration option.

```
new webpack.EvalSourceMapDevToolPlugin(options);
```

Options

The following options are supported:

- `test (string|regex|array)`: Include source maps for modules based on their extension (defaults to `.js` and `.css`).
- `include (string|regex|array)`: Include source maps for module paths that match the given value.
- `exclude (string|regex|array)`: Exclude modules that match the given value from source map generation.
- `filename (string)`: Defines the output filename of the SourceMap (will be inlined if no value is provided).
- `append (string)`: Appends the given value to the original asset. Usually the `# sourceMappingURL` comment. `[url]` is replaced with a URL to the source map file. `false` disables the appending.
- `moduleFilenameTemplate (string)`: See [`output.devtoolModuleFilenameTemplate`](#).
- `sourceURLTemplate`: Define the sourceURL default: `webpack-internal:///${module.identifier}`
- `module (boolean)`: Indicates whether loaders should generate source maps (defaults to `true`).
- `columns (boolean)`: Indicates whether column mappings should be used (defaults to `true`).
- `protocol (string)`: Allows user to override default protocol (`webpack-internal://`)

Setting `module` and/or `columns` to `false` will yield less accurate source maps but will also improve compilation performance significantly.

If you want to use a custom configuration for this plugin in [`development mode`](#), make sure to disable the default one. I.e. set `devtool: false`.

Examples

The following examples demonstrate some common use cases for this plugin.

Basic Use Case

You can use the following code to replace the configuration option `devtool: eval-source-map` with an equivalent custom plugin configuration:

```
module.exports = {
  // ...
  devtool: false,
  plugins: [
    new webpack.EvalSourceMapDevToolPlugin({})
  ]
};
```

Exclude Vendor Maps

The following code would exclude source maps for any modules in the `vendor.js` bundle:

```
new webpack.EvalSourceMapDevToolPlugin({
  filename: '[name].js.map',
  exclude: ['vendor.js']
});
```

Setting sourceURL

Set a URL for source maps. Useful for avoiding cross-origin issues such as:

A cross-origin error was thrown. React doesn't have access to the actual

The option can be set to a function:

```
new webpack.EvalSourceMapDevToolPlugin({
  sourceURLOTemplate: module => `/${module.identifier}`
});
```

Or a substitution string:

```
new webpack.EvalSourceMapDevToolPlugin({
  sourceURLOTemplate: '[all-loaders][resource]'
});
```

ExtractTextWebpackPlugin

Extract text from a bundle, or bundles, into a separate file.

安装

```
npm install --save-dev extract-text-webpack-plugin
# 对于 webpack 2
npm install --save-dev extract-text-webpack-plugin@2.1.2
# 对于 webpack 1
npm install --save-dev extract-text-webpack-plugin@1.0.1
```

用法

:warning: Since webpack v4 the `extract-text-webpack-plugin` should not be used for css. Use [mini-css-extract-plugin](#) instead.

:warning: 对于 webpack v1, 请看 [分支为 webpack-1 的 README 文档](#)。

```
const ExtractTextPlugin = require("extract-text-webpack-plugin");

module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ExtractTextPlugin.extract({
          fallback: "style-loader",
          use: "css-loader"
        })
      }
    ]
  },
  plugins: [
    new ExtractTextPlugin("styles.css"),
  ]
}
```

它会将所有的入口 chunk(entry chunks)中引用的 `*.css`, 移动到独立分离的 CSS 文件。因此, 你的样式将不再内嵌到 JS bundle 中, 而是会放到一个单独的 CSS 文件 (即 `styles.css`) 当中。如果你的样式文件大小较大, 这会做更快提前加载, 因为 CSS bundle 会跟 JS bundle 并行加载。

优点	更少 style 标签 (旧版本的 IE 浏览器有限制)
缺点	额外的 HTTP 请求
优点	CSS SourceMap (使用 <code>devtool: "source-map"</code>)

	和 extract-text-webpack-plugin?sourceMap 配置)
缺点	更长的编译时间
优点	CSS 请求并行
缺点	没有运行时(runtime)的公共路径修改
优点	CSS 单独缓存
缺点	没有热替换
优点	更快的浏览器运行时(runtime) (更少代码和 DOM 操作)
缺点	...

选项

```
new ExtractTextPlugin(options: filename | object)
```

名称	id
类型	{String}
描述	此插件实例的唯一 ident。 (仅限高级用途， 默认情况下自动生成)
名称	filename
类型	{String Function}
描述	生成文件的文件名。可能包含 [name], [id] and [contenthash]
名称	allChunks
类型	{Boolean}
描述	从所有额外的 chunk(additional chunk) 提取 (默认情况下，它仅从初始chunk(initial chunk) 中提取) 当使用 CommonsChunkPlugin 并且在公共 chunk 中有提取的 chunk (来自 ExtractTextPlugin.extract) 时， allChunks **必须设置为 true
名称	disable
类型	{Boolean}
描述	禁用插件
名称	ignoreOrder
类型	{Boolean}

名称
描述

{boolean}

禁用顺序检查(这对 CSS 模块很有用!), 默认

false

- [name] chunk 的名称
- [id] chunk 的数量
- [contenthash] 根据提取文件的内容生成的 hash
- [<hashType>:contenthash:<digestType>:<length>] optionally you can configure
 - other hashTypes, e.g. sha1, md5, sha256, sha512
 - other digestTypes, e.g. hex, base26, base32, base36, base49, base52, base58, base62, base64
 - and length, the length of the hash in chars

:警告: ExtractTextPlugin 对每个入口 chunk 都生成一个对应的文件, 所以当你配置多个入口 chunk 的时候, 你必须使用 [name], [id] 或 [contenthash],

#extract

ExtractTextPlugin.extract(options: loader | object)

从一个已存在的 loader 中, 创建一个提取(extract) loader。支持的 loader 类型 { loader: [name]-loader -> {String}, options: {} -> {Object} }。

名称
类型
描述

options.use

{String}/{Array}/{Object}

loader 被用于将资源转换成一个 CSS 导出模块
(必填)

名称
类型
描述

options.fallback

{String}/{Array}/{Object}

loader (例如 'style-loader') 应用于当 CSS
没有被提取(也就是一个额外的 chunk, 当
allChunks: false)

名称
类型
描述

options.publicPath

{String}

重写此 loader 的 publicPath 配置

多个实例

如果有多个 ExtractTextPlugin 实例的情形, 请使用此方法每个实例上的

`extract` 方法。

```
const ExtractTextPlugin = require('extract-text-webpack-plugin');

// 创建多个实例
const extractCSS = new ExtractTextPlugin('stylesheets/[name]-one.css');
const extractLESS = new ExtractTextPlugin('stylesheets/[name]-two.css');

module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: extractCSS.extract(['css-loader', 'postcss-loader'])
      },
      {
        test: /\.less$/i,
        use: extractLESS.extract(['css-loader', 'less-loader'])
      },
    ]
  },
  plugins: [
    extractCSS,
    extractLESS
  ]
};
```

提取 Sass 或 LESS

配置和上面是相同的，需要时可以将 `sass-loader` 切换为 `less-loader`。

```
const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: ExtractTextPlugin.extract({
          fallback: 'style-loader',
          use: ['css-loader', 'sass-loader']
        })
      }
    ]
  },
  plugins: [
    new ExtractTextPlugin('style.css')
    //如果想要传入选项，你可以这样做：
    //new ExtractTextPlugin({
    //  filename: 'style.css'
    //})
  ]
};
```

url() Resolving

If you are finding that urls are not resolving properly when you run webpack. You can expand your loader functionality with options. The `url: false` property allows your paths resolved without any changes.

```
const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: ExtractTextPlugin.extract({
          fallback: 'style-loader',
          use: [
            {
              loader: 'css-loader',
              options: {
                // If you are having trouble with urls not resolving
                // See https://github.com/webpack-contrib/css-loader
                url: false,
                minimize: true,
                sourceMap: true
              }
            },
            {
              loader: 'sass-loader',
              options: {
                sourceMap: true
              }
            }
          ]
        })
      }
    ]
  }
}
```

修改文件名

`filename` 参数可以是 `Function`。它通过 `getPath` 来处理格式，如 `css/[name].css`，并返回真实的文件名，你可以用 `css` 替换 `css/js`，你会得到新的路径 `css/a.css`。

```
entry: {
  'js/a': './a'
},
plugins: [
  new ExtractTextPlugin({
    filename: (getPath) => {
      return getPath('css/[name].css').replace('css/js', 'css');
    },
    allChunks: true
  })
]
```

HashedModuleIdsPlugin

该插件会根据模块的相对路径生成一个四位数的hash作为模块id, 建议用于生产环境。

```
new webpack.HashedModuleIdsPlugin({  
  // 选项.....  
});
```

参数

该插件支持以下参数:

- hashFunction: 散列算法, 默认为 'md4'。支持 Node.JS `crypto.createHash` 的所有功能。
- hashDigest: 在生成 hash 时使用的编码方式, 默认为 'base64'。支持 Node.js `hash.digest` 的所有编码。
- hashDigestLength: 散列摘要的前缀长度, 默认为 4。Note that some generated ids might be longer than specified here, to avoid module id collisions.

用法

下面是使用该插件的例子:

```
new webpack.HashedModuleIdsPlugin({  
  hashFunction: 'sha256',  
  hashDigest: 'hex',  
  hashDigestLength: 20  
});
```

HotModuleReplacementPlugin

启用热替换模块(Hot Module Replacement)，也被称为 HMR。

永远不要在生产环境(production)下启用 HMR

基本用法(Basic Usage)

启用 HMR 非常简单，在大多数情况下也不需要设置选项。

```
new webpack.HotModuleReplacementPlugin({  
    // Options...  
});
```

选项(Options)

包含如下选项：

- `multiStep (boolean)`: 设置为 `true` 时，插件会分成两步构建文件。首先编译热加载 chunks，之后再编译剩余的通常的资源。
- `fullBuildTimeout (number)`: 当 `multiStep` 启用时，表示两步构建之间的延时。
- `requestTimeout (number)`: 下载 manifest 的延时（webpack 3.0.0 后的版本支持）。

这些选项属于实验性内容，因此以后可能会被弃用。就如同上文所说的那样，这些选项通常情况下都是没有必要设置的，仅仅是设置一下 `new webpack.HotModuleReplacementPlugin()` 在大部分情况下就足够了。

HtmlWebpackPlugin

`HtmlWebpackPlugin`简化了HTML文件的创建，以便为你的webpack包提供服务。这对于在文件名中包含每次会随着编译而发生变化哈希的 webpack bundle 尤其有用。你可以让插件为你生成一个HTML文件，使用`lodash`模板提供你自己的模板，或使用你自己的loader。

安装

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

基本用法

该插件将为你生成一个HTML5文件，其中包括使用`script`标签的body中的所有webpack包。只需添加插件到你的webpack配置如下：

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

module.exports = {
  entry: 'index.js',
  output: {
    path: path.resolve(__dirname, './dist'),
    filename: 'index_bundle.js'
  },
  plugins: [new HtmlWebpackPlugin()]
};
```

这将产生一个包含以下内容的文件`dist/index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>webpack App</title>
  </head>
  <body>
    <script src="index_bundle.js"></script>
  </body>
</html>
```

如果你有多个webpack入口点，他们都会在生成的HTML文件中的`script`标签内。

如果你有任何CSS assets在webpack的输出中（例如，利用`MiniCssExtractPlugin`提取CSS），那么这些将被包含在HTML head中的`<link>`标签内。

配置

获取所有的配置选项，请浏览插件文档。

第三方插件

这个插件支持第三方插件。详细列表参阅文档。

I18nWebpackPlugin

i18n (localization) plugin for Webpack.

安装

```
npm i -D i18n-webpack-plugin
```

用法

此插件会创建包含译文的 bundle。所以你可以将翻译后的 bundle 提供给客户端。

参考 [webpack/webpack/examples/i18n](#)。

配置

```
plugins: [
  ...
  new I18nPlugin(languageConfig, optionsObj)
],
```

- `optionsObj.functionName`: 默认值为 `_`, 你可以更改为其他函数名。
- `optionsObj.failOnMissing`: 默认值为 `false`, 找不到映射文本(mapping text)时会给出一个警告信息, 如果设置为 `true`, 则会给出一个错误信息。
- `optionsObj.hideMessage`: 默认值为 `false`, 将会显示警告/错误信息。如果设置为 `true`, 警告/错误信息将会被隐藏。
- `optionsObj.nested`: the default value is `false`. If set to `true`, the keys in `languageConfig` can be nested. This option is interpreted only if `languageConfig` isn't a function.

Maintainers



Juho Vepsäläinen Joshua Wiens Kees Kluskens Sean Larkin

IgnorePlugin

IgnorePlugin 防止在 `import` 或 `require` 调用时，生成以下正则表达式匹配的模块：

Using regular expressions

- `resourceRegExp`: 匹配(test)资源请求路径的正则表达式。
- `contextRegExp`: (可选) 匹配(test)资源上下文(目录)的正则表达式。

```
new webpack.IgnorePlugin({resourceRegExp, contextRegExp});  
// old way, deprecated in webpack v5  
new webpack.IgnorePlugin(resourceRegExp, [contextRegExp]);
```

Using filter functions

- `checkContext(context)` A Filter function that receives context as the argument, must return boolean.
- `checkResource(resource)` A Filter function that receives resource as the argument, must return boolean.

```
new webpack.IgnorePlugin({  
  checkContext (context) {  
    // do something with context  
    return true|false;  
  },  
  checkResource (resource) {  
    // do something with resource  
    return true|false;  
  }  
});
```

忽略 moment 本地化内容的示例

moment 2.18 会将所有本地化内容和核心功能一起打包（见该 [GitHub issue](#)）。

The `resourceRegExp` parameter passed to `IgnorePlugin` is not tested against the resolved file names or absolute module names being imported or required, but rather against the *string* passed to `require` or `import` *within the source code where the import is taking place*. For example, if you're trying to exclude `node_modules/moment/locale/*.js`, this won't work:

```
-new webpack.IgnorePlugin(/moment\/locale\//);
```

Rather, because `moment` imports with this code:

```
require('./locale/' + name);
```

...your first regexp must match that './locale/' string. The second `contextRegExp` parameter is then used to select specific directories from where the import took place. The following will cause those locale files to be ignored:

```
new webpack.IgnorePlugin({
  resourceRegExp: /^\.\/locale$/,
  contextRegExp: /moment$/
});
```

...which means "any require statement matching './locale' from any directories ending with 'moment' will be ignored.

BabelMinifyWebpackPlugin

一个用于`babel-minify`的 webpack 插件 - 基于 babel 的 minifier

安装

```
npm install babel-minify-webpack-plugin --save-dev
```

用法

```
// webpack.config.js
const MinifyPlugin = require("babel-minify-webpack-plugin");
module.exports = {
  entry: //....,
  output: //....,
  plugins: [
    new MinifyPlugin(minifyOpts, pluginOpts)
  ]
}
```

选项

###

`minifyOpts` 被传递给 `babel-preset-minify`。 你可以在包目录中找到所有可用的选项。

Default: {}

pluginOpts

- `test`: Test to match files against. Default: `/\\.js\\($|\\?)\\/i`
- `include`: Files to include. Default: `undefined`
- `exclude`: Files to exclude. Default: `undefined`
- `comments`: 保留注释。默认: `/^**!|@preserve|@license|@cc_on/`, falsy 值将移除所有注释。可以接受函数, 带有属性匹配(正则)的对象和值。
- `sourceMap`: Configure a sourcemap style. 默认: `webpackConfig.devtool`
- `parserOpts`: 配置具有特殊解析器选项的 babel。
- `babel`: 传入一个自定义的 `babel-core`。默认: `require("babel-core")`
- `minifyPreset`: 传入一个自定义 `babel-minify preset` 代替原来的。默认: `require("babel-preset-minify")`

为什么

你也可以在webpack中使用babel-loader，引入`minify`作为一个预设并且应该运行的更快 - 因为`babel-minify`将运行在更小的文件。但是，这个插件为什么还存在呢？

- webpack loader 对单个文件进行操作，并且 minify preset 作为一个 webpack loader 将会把每个文件视为在浏览器全局范围内直接执行（默认情况下），并且不会优化顶级作用域内的某些内容。要在文件的顶级作用域内进行优化，请在`minifyOptions` 中设置`mangle: { topLevel: true }`。
- 当你排除`node_modules` 不通过 babel-loader 运行时，babel-minify 优化不会应用于被排除的文件，因为它们不会通过 minifier。
- 当你使用带有 webpack 的 babel-loader 时，由 webpack 为模块系统生成的代码不会通过 loader，并且不会通过 babel-minify 进行优化。
- 一个 webpack 插件可以在整个 chunk/bundle 输出上运行，并且可以优化整个 bundle，你可以看到一些细微的输出差异。但是，由于文件大小通常非常大，所以会慢很多。所以这里有一个想法 - 我们可以将一些优化作为 loader 的一部分，并在插件中进行一些优化。

LimitChunkCountPlugin

当你在编写代码时，可能已经添加了许多代码分离点(code split point)来实现按需加载(load stuff on demand)。在编译完之后，你可能会注意到有一些很小的 chunk - 这产生了大量 HTTP 请求开销。`LimitChunkCountPlugin` 插件可以通过合并的方式，后处理你的 chunk，以减少请求数。

```
new webpack.optimize.LimitChunkCountPlugin({  
    // 选项.....  
});
```

选项

可以支持以下选项：

`maxChunks`

`number`

使用大于或等于`1` 的值，来限制 chunk 的最大数量。使用`1` 防止添加任何其他额外的 chunk，这是因为 entry/main chunk 也会包含在计数之中。

`webpack.config.js`

```
const webpack = require('webpack');  
module.exports = {  
    // ...
```

```
plugins: [
  new webpack.optimize.LimitChunkCountPlugin({
    maxChunks: 5
  })
]
};
```

minChunkSize

number

设置 chunk 的最小大小。

webpack.config.js

```
const webpack = require('webpack');
module.exports = {
  // ...
  plugins: [
    new webpack.optimize.LimitChunkCountPlugin({
      minChunkSize: 1000
    })
  ]
};
```

命令行接口(CLI)用法

此插件和其选项还可以通过 命令行接口(CLI) 执行：

```
webpack --optimize-max-chunks 15
```

或

```
webpack --optimize-min-chunk-size 10000
```

LoaderOptionsPlugin

`loader-options-plugin` 和其他插件不同，它用于将 webpack 1 迁移至 webpack 2。在 webpack 2 中，对 `webpack.config.js` 的结构要求变得更加严格；不再开放扩展给其他的 loader/插件。webpack 2 推荐的使用方式是直接传递 `options` 给 loader/插件（换句话说，配置选项将不是全局/共享的）。

不过，在某个 loader 升级为依靠直接传递给它的配置选项运行之前，可以使用 `loader-options-plugin` 来抹平差异。你可以通过这个插件配置全局/共享的 loader 配置，使所有的 loader 都能收到这些配置。

```
new webpack.LoaderOptionsPlugin({  
    // Options...  
});
```

将来这个插件可能会被移除，因为它只是用于迁移。

选项

此插件支持以下选项：

- `options.debug (boolean)`: loader 是否为 `debug` 模式。`debug` 在 webpack 3 中将被移除。
- `options.minimize (boolean)`: loader 是否要切换到优化模式。
- `options.options (object)`: 一个配置对象，用来配置旧的 loader - 将使用和 `webpack.config.js` 相同的结构。
- `options.options.context (string)`: 配置 loader 时使用的上下文。
- 任何其他选项和在 `webpack.config.js` 中一样.....

用法

关于此插件可能的用法，这里有个示例：

```
new webpack.LoaderOptionsPlugin({  
    minimize: true,  
    debug: false,  
    options: {  
        context: __dirname  
    }  
});
```

MinChunkSizePlugin

通过合并小于 `minChunkSize` 大小的 chunk，将 chunk 体积保持在指定大小限制以上。

```
new webpack.optimize.MinChunkSizePlugin({  
    minChunkSize: 10000 // Minimum number of characters  
});
```

MiniCssExtractPlugin

[![npm][npm]][npm-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![coverage][cover]][cover-url] [![chat][chat]][chat-url]

This plugin extracts CSS into separate files. It creates a CSS file per JS file which contains CSS. It supports On-Demand-Loading of CSS and SourceMaps.

It builds on top of a new webpack v4 feature (module types) and requires webpack 4 to work.

Compared to the extract-text-webpack-plugin:

- Async loading
- No duplicate compilation (performance)
- Easier to use
- Specific to CSS

TODO:

- HMR support

Install

```
npm install --save-dev mini-css-extract-plugin
```

Usage

##

Minimal example

webpack.config.js

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
module.exports = {
  plugins: [
    new MiniCssExtractPlugin({
      // Options similar to the same options in webpackOptions.output
      // both options are optional
      filename: "[name].css",
      chunkFilename: "[id].css"
    })
  ],
  module: {
```

```

rules: [
  {
    test: /\.css$/,
    use: [
      {
        loader: MiniCssExtractPlugin.loader,
        options: {
          // you can specify a publicPath here
          // by default it use publicPath in webpackOptions.output
          publicPath: '../'
        }
      },
      "css-loader"
    ]
  }
]
}

```

Advanced configuration example

This plugin should be used only on `production` builds without `style-loader` in the loaders chain, especially if you want to have HMR in `development`.

Here is an example to have both HMR in `development` and your styles extracted in a file for `production` builds.

(Loaders options left out for clarity, adapt accordingly to your needs.)

`webpack.config.js`

```

const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const devMode = process.env.NODE_ENV !== 'production'

module.exports = {
  plugins: [
    new MiniCssExtractPlugin({
      // Options similar to the same options in webpackOptions.output
      // both options are optional
      filename: devMode ? '[name].css' : '[name].[hash].css',
      chunkFilename: devMode ? '[id].css' : '[id].[hash].css',
    })
  ],
  module: {
    rules: [
      {
        test: /\.(sa|sc|c)ss$/,
        use: [
          devMode ? 'style-loader' : MiniCssExtractPlugin.loader,
          'css-loader',
          'postcss-loader',
          'sass-loader',
        ],
      },
    ]
  }
}

```

```
        ]
    }
}
```

Minimizing For Production

While webpack 5 is likely to come with a CSS minimizer built-in, with webpack 4 you need to bring your own. To minify the output, use a plugin like [optimize-css-assets-webpack-plugin](#). Setting `optimization.minimizer` overrides the defaults provided by webpack, so make sure to also specify a JS minimizer:

webpack.config.js

```
const TerserJSPlugin = require("terser-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const OptimizeCSSAssetsPlugin = require("optimize-css-assets-webpack-plugin");
module.exports = {
  optimization: {
    minimizer: [
      new TerserJSPlugin({}),
      new OptimizeCSSAssetsPlugin({})
    ],
  },
  plugins: [
    new MiniCssExtractPlugin({
      filename: "[name].css",
      chunkFilename: "[id].css"
    })
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          MiniCssExtractPlugin.loader,
          "css-loader"
        ]
      }
    ]
  }
}
```

Features

Using preloaded or inlined CSS

The runtime code detects already added CSS via `<link>` or `<style>` tag. This can be useful when injecting CSS on server-side for Server-Side-Rendering. The `href` of the `<link>` tag has to match the URL that will be used for loading the CSS chunk. The `data-href` attribute can be used for `<link>` and `<style>` too. When inlining CSS `data-href` must be used.

Extracting all CSS in a single file

Similar to what [extract-text-webpack-plugin](#) does, the CSS can be extracted in one CSS file using `optimization.splitChunks.cacheGroups`.

webpack.config.js

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
module.exports = {
  optimization: {
    splitChunks: {
      cacheGroups: {
        styles: {
          name: 'styles',
          test: /\.css$/,
          chunks: 'all',
          enforce: true
        }
      }
    }
  },
  plugins: [
    new MiniCssExtractPlugin({
      filename: "[name].css",
    })
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          MiniCssExtractPlugin.loader,
          "css-loader"
        ]
      }
    ]
  }
}
```

Extracting CSS based on entry

You may also extract the CSS based on the webpack entry name. This is especially useful if you import routes dynamically but want to keep your CSS bundled according to entry. This also prevents the CSS duplication issue one had with the ExtractTextPlugin.

```
const path = require('path');
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

function recursiveIssuer(m) {
  if (m.issuer) {
    return recursiveIssuer(m.issuer);
  } else if (m.name) {
    return m.name;
  }
}
```

```

    } else {
      return false;
    }
}

module.exports = {
  entry: {
    foo: path.resolve(__dirname, 'src/foo'),
    bar: path.resolve(__dirname, 'src/bar')
  },
  optimization: {
    splitChunks: {
      cacheGroups: {
        fooStyles: {
          name: 'foo',
          test: (m,c,entry = 'foo') => m.constructor.name === 'CssModule',
          chunks: 'all',
          enforce: true
        },
        barStyles: {
          name: 'bar',
          test: (m,c,entry = 'bar') => m.constructor.name === 'CssModule',
          chunks: 'all',
          enforce: true
        }
      }
    }
  },
  plugins: [
    new MiniCssExtractPlugin({
      filename: "[name].css",
    })
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          MiniCssExtractPlugin.loader,
          "css-loader"
        ]
      }
    ]
  }
}

```

Long Term Caching

For long term caching use `filename: "[contenthash].css"`. Optionally add `[name]`.

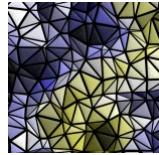
Media Query Plugin

If you'd like to extract the media queries from the extracted CSS (so mobile users don't need to load desktop or tablet specific CSS anymore) you should use one of the following

plugins:

- [Media Query Plugin](#)
- [Media Query Splitting Plugin](#)

Maintainers



[Tobias Koppers](#)

License

[MIT](#)

ModuleConcatenationPlugin

过去 webpack 打包时的一个取舍是将 bundle 中各个模块单独打包成闭包。这些打包函数使你的 JavaScript 在浏览器中处理的更慢。相比之下，一些工具像 Closure Compiler 和 RollupJS 可以提升(hoist)或者预编译所有模块到一个闭包中，提升你的代码在浏览器中的执行速度。

这个插件会在 webpack 中实现以上的预编译功能。By default this plugin is already enabled in `production mode` and disabled otherwise. If you need it in other modes, you can add it manually:

```
new webpack.optimize.ModuleConcatenationPlugin();
```

这种连结行为被称为“作用域提升(scope hoisting)”。

由于实现 ECMAScript 模块语法，作用域提升(scope hoisting)这个特定于此语法的功能才成为可能。`webpack` 可能会根据你正在使用的模块类型和其他的情况，回退到普通打包。

记住，此插件仅适用于由 `webpack` 直接处理的 `ES6` 模块。在使用转译器(transpiler)时，你需要禁用对模块的处理（例如 `Babel` 中的 `modules` 选项）。

绑定失败的优化[Optimization Bailouts]

像文章中解释的，`webpack` 试图达到分批的作用域提升(scope hoisting)。它会将一些模块绑定到一个作用域内，但并不是任何情况下都会这么做。如果 `webpack` 不能绑定模块，将会有两个选择 Prevent 和 Root，Prevent 意思是模块必须在自己的作用域内。Root 意味着将创建一个新的模块组。以下条件决定了输出结果：

Condition	Non ES6 Module
Outcome	Prevent
Condition	Imported By Non Import
Outcome	Root
Condition	Imported From Other Chunk
Outcome	Root
Condition	Imported By Multiple Other Module Groups
Outcome	Root
Condition	Imported With <code>import()</code>
Outcome	Root

Condition	Affected By ProvidePlugin Or Using module
Outcome	Prevent
Condition	HMR Accepted
Outcome	Root
Condition	Using eval()
Outcome	Prevent
Condition	In Multiple Chunks
Outcome	Prevent
Condition	export * from "cjs-module"
Outcome	Prevent

模块分组算法[Module Grouping Algorithm]

以下 JavaScript 伪代码解释了算法：

```

modules.forEach(module => {
  const group = new ModuleGroup({
    root: module
  });
  module.dependencies.forEach(dependency => {
    tryToAdd(group, dependency);
  });
  if (group.modules.length > 1) {
    orderedModules = topologicalSort(group.modules);
    concatenatedModule = new ConcatenatedModule(orderedModules);
    chunk.add(concatenatedModule);
    orderedModules.forEach(groupModule => {
      chunk.remove(groupModule);
    });
  }
});
}

function tryToAdd(group, module) {
  if (group.has(module)) {
    return true;
  }
  if (!hasPreconditions(module)) {
    return false;
  }
  const nextGroup = group;
  const result = module.dependents.reduce((check, dependent) => {
    return check && tryToAdd(nextGroup, dependent);
  }, true);
  if (!result) {
    return false;
  }
  module.dependencies.forEach(dependency => {
    tryToAdd(group, dependency);
  });
}

```

```
});  
group.merge(nextGroup);  
return true;  
}
```

优化绑定失败的调试[Debugging Optimization Bailouts]

当我们使用 webpack CLI 时，加上参数 `--display-optimization-bailout` 将显示绑定失败的原因。在 webpack 配置里，只需将以下内容添加到 stats 对象中：

```
module.exports = {  
  //...  
  stats: {  
    // Examine all modules  
    maxModules: Infinity,  
    // Display bailout reasons  
    optimizationBailout: true  
  }  
};
```

NormalModuleReplacementPlugin

NormalModuleReplacementPlugin 允许你用 `newResource` 替换与 `resourceRegExp` 匹配的资源。如果 `newResource` 是相对路径，它会相对于先前的资源被解析。如果 `newResource` 是函数，它将会覆盖之前被提供资源的请求。

这对于允许在构建中的不同行为是有用的。

```
new webpack.NormalModuleReplacementPlugin(  
  resourceRegExp,  
  newResource  
) ;
```

基本示例

在构建开发环境时替换特定的模块。

假设你有一个配置文件 `some/path/config.development.module.js` 并且在生产环境有一个特殊的版本 `some/path/config.production.module.js`

只需在生产构建时添加以下插件：

```
new webpack.NormalModuleReplacementPlugin(  
  '/some\\path\\config\\.development\\.js/',  
  './config.production.js'  
) ;
```

高级示例

根据指定环境的条件构建。

假设你想要一个为了不同构建目标的特定值的配置。

```
module.exports = function(env) {  
  var appTarget = env.APP_TARGET || 'VERSION_A';  
  return {  
    plugins: [  
      new webpack.NormalModuleReplacementPlugin(/(.*)-APP_TARGET(\.*)/,  
        resource.request = resource.request.replace(/-APP_TARGET/, `-${$  
      })  
    ]  
  };  
};
```

创建两个配置文件：

app/config-VERSION_A.js

```
export default {  
    title : 'I am version A'  
};
```

app/config-VERSION_B.js

```
export default {  
    title : 'I am version B'  
};
```

然后使用在正则中查找的关键字来引入配置：

```
import config from 'app/config-APP_TARGET';  
console.log(config.title);
```

根据你的构建目标，现在你引入了正确的配置。

```
webpack --env.APP_TARGET VERSION_A  
=> 'I am version A'
```

```
webpack --env.APP_TARGET VERSION_B  
=> 'I am version B'
```

NpmInstallWebpackPlugin

Speed up development by **automatically installing & saving dependencies** with Webpack.

It is inefficient to `Ctrl-C` your build script & server just to install a dependency you didn't know you needed until now.

Instead, use `require` or `import` how you normally would and `npm install` will happen **automatically to install & save missing dependencies** while you work!

安装

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

用法

在 `webpack.config.js` 中:

```
plugins: [
  new NpmInstallPlugin()
],
```

相当于:

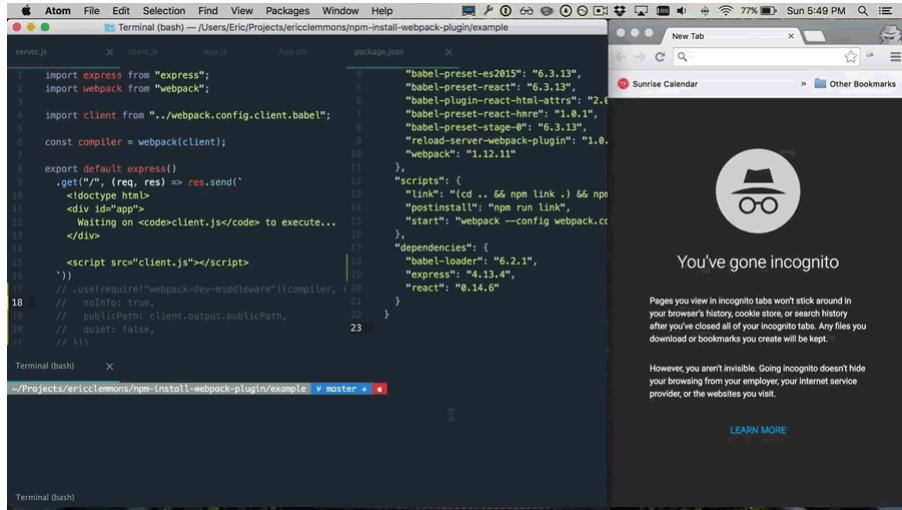
```
plugins: [
  new NpmInstallPlugin({
    // 使用 --save 或者 --save-dev
    dev: false,
    // 安装缺少的 peerDependencies
    peerDependencies: true,
    // 减少控制台日志记录的数量
    quiet: false,
    // npm command used inside company, yarn is not supported yet
    npm: 'tnpm'
  });
],
```

可以提供一个 Function 来动态设置 `dev`:

```
plugins: [
  new NpmInstallPlugin({
    dev: function(module, path) {
      return [
        "babel-preset-react-hmre",
        "webpack-dev-middleware",
        "webpack-hot-middleware",
      ].indexOf(module) !== -1;
    }
  });
],
```

```
    },
  },
]
```

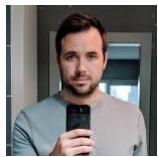
Demo



Features

- [x] Works with both Webpack ^v1.12.0 and ^2.1.0-beta.0.
- [x] Auto-installs .babelrc plugins & presets.
- [x] Supports both ES5 & ES6 Modules. (e.g. require, import)
- [x] Supports Namespaced packages. (e.g. @cycle/dom)
- [x] Supports Dot-delimited packages. (e.g. lodash.capitalize)
- [x] Supports CSS imports. (e.g. @import "~bootstrap")
- [x] Supports Webpack loaders. (e.g. babel-loader, file-loader, etc.)
- [x] Supports inline Webpack loaders. (e.g. require("bundle?lazy!./App"))
- [x] Auto-installs missing peerDependencies. (e.g. @cycle/core will automatically install rx@*)
- [x] Supports Webpack's resolve.alias & resolve.root configuration. (e.g. require("react") can alias to react-lite)

Maintainers



Eric Clemons Jonny Buchanan

PrefetchPlugin

预取出普通的模块请求(module request)，可以让这些模块在他们被 `import` 或者是 `require` 之前就解析并且编译。使用这个预取插件可以提升性能。可以多试试在编译前记录时间(profile)来决定最佳的预取的节点。

```
new webpack.PrefetchPlugin([context], request);
```

选项

- `context`: 文件夹的绝对路径
- `request`: 普通模块的 `request` 字符串

ProfilingPlugin

Generate Chrome profile file which includes timings of plugins execution. Outputs `events.json` file by default. It is possible to provide custom file path using `outputPath` option.

Options

- `outputPath`: A relative path to a custom output file (json)

Usage: default

```
new webpack.debug.ProfilingPlugin();
```

Usage: custom outputPath

```
new webpack.debug.ProfilingPlugin({  
  outputPath: 'profiling/profileEvents.json'  
});
```

In order to view the profile file:

1. Run webpack with `ProfilingPlugin`.
2. Go to Chrome, open DevTools, and go to the `Performance` tab (formerly `Timeline`).
3. Drag and drop generated file (`events.json` by default) into the profiler.

It will then display timeline stats and calls per plugin!

ProgressPlugin

The `ProgressPlugin` provides a way to customize how progress is reported during a compilation.

Usage

Create an instance of `ProgressPlugin` with a handler function which will be called when hooks report progress:

```
const handler = (percentage, message, ...args) => {
  // e.g. Output each progress message directly to the console:
  console.info(percentage, message, ...args);
};

new webpack.ProgressPlugin(handler);
```

- `handler` is a function which takes these arguments:
- `percentage`: a number between 0 and 1 indicating the completion percentage of the compilation.
- `message`: a short description of the currently-executing hook.
- `...args`: zero or more additional strings describing the current progress.

Supported Hooks

The following hooks report progress information to `ProgressPlugin`.

*Hooks marked with * allow plugins to report progress information using `reportProgress`. For more, see [Plugin API: Reporting Progress](#)*

Compiler

- `compilation`
- `emit*`
- `afterEmit*`
- `done`

Compilation

- `buildModule`
- `failedModule`
- `succeedModule`
- `finishModules*`
- `seal*`

- optimizeDependenciesBasic*
- optimizeDependencies*
- optimizeDependenciesAdvanced*
- afterOptimizeDependencies*
- optimize*
- optimizeModulesBasic*
- optimizeModules*
- optimizeModulesAdvanced*
- afterOptimizeModules*
- optimizeChunksBasic*
- optimizeChunks*
- optimizeChunksAdvanced*
- afterOptimizeChunks*
- optimizeTree*
- afterOptimizeTree*
- optimizeChunkModulesBasic*
- optimizeChunkModules*
- optimizeChunkModulesAdvanced*
- afterOptimizeChunkModules*
- reviveModules*
- optimizeModuleOrder*
- advancedOptimizeModuleOrder*
- beforeModuleIds*
- moduleIds*
- optimizeModuleIds*
- afterOptimizeModuleIds*
- reviveChunks*
- optimizeChunkOrder*
- beforeChunkIds*
- optimizeChunkIds*
- afterOptimizeChunkIds*
- recordModules*
- recordChunks*
- beforeHash*
- afterHash*
- recordHash*
- beforeModuleAssets*
- beforeChunkAssets*
- additionalChunkAssets*
- record*
- additionalAssets*
- optimizeChunkAssets*
- afterOptimizeChunkAssets*

- optimizeAssets*
- afterOptimizeAssets*
- afterSeal*

Source

- [ProgressPlugin source](#)

ProvidePlugin

自动加载模块，而不必到处 `import` 或 `require`。

```
new webpack.ProvidePlugin({
  identifier: 'module1',
  // ...
});
```

or

```
new webpack.ProvidePlugin({
  identifier: ['module1', 'property1'],
  // ...
});
```

任何时候，当 `identifier` 被当作未赋值的变量时，`module` 就会自动被加载，并且 `identifier` 会被这个 `module` 导出的内容所赋值。（或者被模块的 `property` 导出的内容所赋值，以支持命名导出(named export)）。

对于 ES2015 模块的 `default export`，你必须指定模块的 `default` 属性。

使用：jQuery

要自动加载 `jquery`，我们可以将两个变量都指向对应的 node 模块：

```
new webpack.ProvidePlugin({
  $: 'jquery',
  jQuery: 'jquery'
});
```

然后在我们任意源码中：

```
// in a module
$('#item'); // <= 起作用
jQuery('#item'); // <= 起作用
// $ 自动被设置为 "jquery" 输出的内容
```

使用：jQuery 和 Angular 1

Angular 会寻找 `window.jQuery` 来决定 jQuery 是否存在，查看源码。

```
new webpack.ProvidePlugin({
  'window.jQuery': 'jquery'
});
```

使用：Lodash Map

```
new webpack.ProvidePlugin({
  _map: ['lodash', 'map']
});
```

使用：Vue.js

```
new webpack.ProvidePlugin({
  Vue: ['vue/dist/vue.esm.js', 'default']
});
```

SourceMapDevToolPlugin

本插件实现了对 source map 生成内容进行更细粒度的控制。它也可以通过 `devtool` 配置选项的某些设置自动启用。

```
new webpack.SourceMapDevToolPlugin(options);
```

选项

支持以下选项：

- `test (string|regex|array)`: 包含基于扩展名的模块的 source map (默认是 `.js`, `.mjs` 和 `.css`)。
- `include (string|regex|array)`: 使路径与该值匹配的模块生成 source map。
- `exclude (string|regex|array)`: 使匹配该值的模块不生成 source map。
- `filename (string)`: 定义生成的 source map 的名称 (如果没有值将会变成 `inlined`)。
- `append (string)`: 在原始资源后追加给定值。通常是 `# sourceMappingURL` 注释。`[url]` 被替换成 source map 文件的 URL。`false` 将禁用追加。
- `moduleFilenameTemplate (string)`: 查看 `output.devtoolModuleFilenameTemplate`。
- `fallbackModuleFilenameTemplate (string)`: 同上。
- `module (boolean)`: 表示 loader 是否生成 source map (默认为 `true`)。
- `columns (boolean)`: 表示是否应该使用 column mapping (默认为 `true`)。
- `lineToLine (boolean 或 object)`: 通过行到行源代码映射(line to line source mappings)简化和提升匹配模块的源代码映射速度。
- `noSources (boolean)`: 防止源文件的内容被包含在 source map 里 (默认为 `false`)。
- `publicPath (string)`: 生成带 public path 前缀的绝对 URL, 例如: `https://example.com/project/`。
- `fileContext (string)`: 使得 `[file]` 参数作为本目录的相对路径。

`lineToLine` 对象允许的值和上面 `test`, `include`, `exclude` 选项一样。

`fileContext` 选项在你想要将 source maps 存储到上层目录, 避免 `..../` 出现在绝对路径 `[url]` 里面时有用。

设置 `module` 和/或 `columns` 为 `false` 将会生成不太精确的 source map, 但同时会显著地提升编译性能。

If you want to use a custom configuration for this plugin in `development mode`, make sure to disable the default one. I.e. set `devtool: false`.

记得在使用 `TerserPlugin` 时，必须使用 `sourceMap` 选项。

用法

下面的示例展示了本插件的一些常见用例。

Basic Use Case

You can use the following code to replace the configuration option `devtool: inline-source-map` with an equivalent custom plugin configuration:

```
module.exports = {
  // ...
  devtool: false,
  plugins: [
    new webpack.SourceMapDevToolPlugin({})
  ]
};
```

排除 vendor 的 map

以下代码会排除 `vendor.js` 内模块的 source map。

```
new webpack.SourceMapDevToolPlugin({
  filename: '[name].js.map',
  exclude: ['vendor.js']
});
```

在宿主环境外部化 source map

设置 source map 的 URL。在宿主环境需要授权的情况下很有用。

```
new webpack.SourceMapDevToolPlugin({
  append: '\n//# sourceMappingURL=http://example.com/sourcemap/[url]',
  filename: '[name].map'
});
```

还有一种场景，source map 存储在上层目录中时：

```
project
|- dist
  |- public
    |- bundle-[hash].js
  |- sourcemaps
    |- bundle-[hash].js.map
```

如下设置：

```
new webpack.SourceMapDevToolPlugin({
```

```
filename: 'sourcemaps/[file].map',
publicPath: 'https://example.com/project/',
fileContext: 'public'
});
```

将会生成以下 URL:

[https://example.com/project/sourcemaps/bundle-\[hash\].js.map](https://example.com/project/sourcemaps/bundle-[hash].js.map)

SplitChunksPlugin

Originally, chunks (and modules imported inside them) were connected by a parent-child relationship in the internal webpack graph. The `CommonsChunkPlugin` was used to avoid duplicated dependencies across them, but further optimizations were not possible.

Since webpack v4, the `CommonsChunkPlugin` was removed in favor of `optimization.splitChunks`.

Defaults

Out of the box `SplitChunksPlugin` should work well for most users.

By default it only affects on-demand chunks, because changing initial chunks would affect the script tags the HTML file should include to run the project.

webpack will automatically split chunks based on these conditions:

- New chunk can be shared OR modules are from the `node_modules` folder
- New chunk would be bigger than 30kb (before min+gz)
- Maximum number of parallel requests when loading chunks on demand would be lower or equal to 5
- Maximum number of parallel requests at initial page load would be lower or equal to 3

When trying to fulfill the last two conditions, bigger chunks are preferred.

Configuration

webpack provides a set of options for developers that want more control over this functionality.

The default configuration was chosen to fit web performance best practices, but the optimal strategy for your project might differ. If you're changing the configuration, you should measure the impact of your changes to ensure there's a real benefit.

`optimization.splitChunks`

This configuration object represents the default behavior of the `SplitChunksPlugin`.

`webpack.config.js`

```
module.exports = {
```

```
//...
optimization: {
  splitChunks: {
    chunks: 'async',
    minSize: 30000,
    maxSize: 0,
    minChunks: 1,
    maxAsyncRequests: 5,
    maxInitialRequests: 3,
    automaticNameDelimiter: '~',
    name: true,
    cacheGroups: {
      vendors: {
        test: /[\\/]node_modules[\\/]/,
        priority: -10
      },
      default: {
        minChunks: 2,
        priority: -20,
        reuseExistingChunk: true
      }
    }
  }
};
```

splitChunks.automaticNameDelimiter

string

By default webpack will generate names using origin and name of the chunk (e.g. `vendors~main.js`). This option lets you specify the delimiter to use for the generated names.

splitChunks.chunks

function (chunk) | string

This indicates which chunks will be selected for optimization. When a string is provided, valid values are `all`, `async`, and `initial`. Providing `all` can be particularly powerful, because it means that chunks can be shared even between `async` and non-`async` chunks.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    splitChunks: {
      // include all types of chunks
      chunks: 'all'
    }
}
```

```
};
```

Alternatively, you may provide a function for more control. The return value will indicate whether to include each chunk.

```
module.exports = {
  //...
  optimization: {
    splitChunks: {
      chunks (chunk) {
        // exclude `my-excluded-chunk`
        return chunk.name !== 'my-excluded-chunk';
      }
    }
  }
};
```

You can combine this configuration with the [HtmlWebpackPlugin](#). It will inject all the generated vendor chunks for you.

splitChunks.maxAsyncRequests

number

Maximum number of parallel requests when on-demand loading.

splitChunks.maxInitialRequests

number

Maximum number of parallel requests at an entry point.

splitChunks.minChunks

number

Minimum number of chunks that must share a module before splitting.

splitChunks.minSize

number

Minimum size, in bytes, for a chunk to be generated.

splitChunks maxSize

number

Using `maxSize` (either globally `optimization.splitChunks.maxSize` per cache group `optimization.splitChunks.cacheGroups[x].maxSize` or for the fallback cache group `optimization.splitChunks.fallbackCacheGroup.maxSize`) tells webpack to try to split chunks bigger than `maxSize` into smaller parts. Parts will be at least `minSize` (next to `maxSize`) in size. The algorithm is deterministic and changes to the modules will only have local impact. So that it is usable when using long term caching and doesn't require records. `maxSize` is only a hint and could be violated when modules are bigger than `maxSize` or splitting would violate `minSize`.

When the chunk has a name already, each part will get a new name derived from that name. Depending on the value of `optimization.splitChunks.hidePathInfo` it will add a key derived from the first module name or a hash of it.

`maxSize` options is intended to be used with HTTP/2 and long term caching. It increase the request count for better caching. It could also be used to decrease the file size for faster rebuilding.

`maxSize` takes higher priority than `maxInitialRequest/maxAsyncRequests`. Actual priority is `maxInitialRequest/maxAsyncRequests < maxSize < minSize`.

splitChunks.name

`boolean: true | function (module, chunks, cacheGroupKey) | string`

The name of the split chunk. Providing `true` will automatically generate a name based on chunks and cache group key. Providing a string or function will allow you to use a custom name. If the name matches an entry point name, the entry point will be removed.

It is recommended to set `splitChunks.name` to `false` for production builds so that it doesn't change names unnecessarily.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    splitChunks: {
      name (module, chunks, cacheGroupKey) {
        // generate a chunk name...
        return; //...
      }
    }
  }
};
```

When assigning equal names to different split chunks, all vendor modules are

placed into a single shared chunk, though it's not recommended since it can result in more code downloaded.

splitChunks.cacheGroups

Cache groups can inherit and/or override any options from `splitChunks.*`; but `test`, `priority` and `reuseExistingChunk` can only be configured on cache group level. To disable any of the default cache groups, set them to `false`.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    splitChunks: {
      cacheGroups: {
        default: false
      }
    }
  }
};
```

splitChunks.cacheGroups.priority

number

A module can belong to multiple cache groups. The optimization will prefer the cache group with a higher `priority`. The default groups have a negative priority to allow custom groups to take higher priority (default value is `0` for custom groups).

splitChunks.cacheGroups.{cacheGroup}.reuseExistingChunk

boolean

If the current chunk contains modules already split out from the main bundle, it will be reused instead of a new one being generated. This can impact the resulting file name of the chunk.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    splitChunks: {
      cacheGroups: {
        vendors: {
          reuseExistingChunk: true
        }
      }
    }
  }
};
```

```
        }
    }
};

splitChunks.cacheGroups.{cacheGroup}.test

function (module, chunk) | RegExp | string
```

Controls which modules are selected by this cache group. Omitting it selects all modules. It can match the absolute module resource path or chunk names. When a chunk name is matched, all modules in the chunk are selected.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    splitChunks: {
      cacheGroups: {
        vendors: {
          test(module, chunks) {
            //...
            return module.type === 'javascript/auto';
          }
        }
      }
    }
};
};
```

splitChunks.cacheGroups.{cacheGroup}.filename

```
string
```

Allows to override the filename when and only when it's an initial chunk. All placeholders available in [output.filename](#) are also available here.

This option can also be set globally in `splitChunks.filename`, but this isn't recommended and will likely lead to an error if `splitchunks.chunks` is not set to '`initial`'. Avoid setting it globally.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    splitChunks: {
      cacheGroups: {
        vendors: {
```

```

        filename: '[name].bundle.js'
    }
}
}
};

splitChunks.cacheGroups.{cacheGroup}.enforce

boolean: false

```

Tells webpack to ignore `splitChunks.minSize`, `splitChunks.minChunks`, `splitChunks.maxAsyncRequests` and `splitChunks.maxInitialRequests` options and always create chunks for this cache group.

webpack.config.js

```

module.exports = {
//...
optimization: {
  splitChunks: {
    cacheGroups: {
      vendors: {
        enforce: true
      }
    }
  }
};

```

Examples

Defaults: Example 1

```

// index.js

import('./a'); // dynamic import

// a.js
import 'react';

//...

```

Result: A separate chunk would be created containing `react`. At the import call this chunk is loaded in parallel to the original chunk containing `./a`.

Why:

- Condition 1: The chunk contains modules from `node_modules`
- Condition 2: `react` is bigger than 30kb

- Condition 3: Number of parallel requests at the import call is 2
- Condition 4: Doesn't affect request at initial page load

What's the reasoning behind this? `react` probably won't change as often as your application code. By moving it into a separate chunk this chunk can be cached separately from your app code (assuming you are using chunkhash, records, Cache-Control or other long term cache approach).

Defaults: Example 2

```
// entry.js

// dynamic imports
import './a';
import './b';

// a.js
import './helpers'; // helpers is 40kb in size

//...
// b.js
import './helpers';
import './more-helpers'; // more-helpers is also 40kb in size

//...
```

Result: A separate chunk would be created containing `./helpers` and all dependencies of it. At the import calls this chunk is loaded in parallel to the original chunks.

Why:

- Condition 1: The chunk is shared between both import calls
- Condition 2: `helpers` is bigger than 30kb
- Condition 3: Number of parallel requests at the import calls is 2
- Condition 4: Doesn't affect request at initial page load

Putting the content of `helpers` into each chunk will result into its code being downloaded twice. By using a separate chunk this will only happen once. We pay the cost of an additional request, which could be considered a tradeoff. That's why there is a minimum size of 30kb.

Split Chunks: Example 1

Create a `commons` chunk, which includes all code shared between entry points.

webpack.config.js

```
module.exports = {
```

```
//...
optimization: {
  splitChunks: {
    cacheGroups: {
      commons: {
        name: 'commons',
        chunks: 'initial',
        minChunks: 2
      }
    }
  }
}
};
```

This configuration can enlarge your initial bundles, it is recommended to use dynamic imports when a module is not immediately needed.

Split Chunks: Example 2

Create a `vendors` chunk, which includes all code from `node_modules` in the whole application.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    splitChunks: {
      cacheGroups: {
        commons: {
          test: /[\\/]node_modules[\\/]/,
          name: 'vendors',
          chunks: 'all'
        }
      }
    }
  }
};
```

This might result in a large chunk containing all external packages. It is recommended to only include your core frameworks and utilities and dynamically load the rest of the dependencies.

Split Chunks: Example 3

Create a `custom vendor` chunk, which contains certain `node_modules` packages matched by `RegExp`.

webpack.config.js

```
module.exports = {
```

```
//...
optimization: {
  splitChunks: {
    cacheGroups: {
      vendor: {
        test: /[\\/]node_modules[\\/](react|react-dom)[\\/]/,
        name: 'vendor',
        chunks: 'all',
      }
    }
  }
};
```

This will result in splitting `react` and `react-dom` into a separate chunk. If you're not sure what packages have been included in a chunk you may refer to [Bundle Analysis](#) section for details.

StylelintWebpackPlugin

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![chat][chat]][chat-url]

A Stylelint plugin for webpack

Requirements

This module requires a minimum of Node v6.9.0 and webpack v4.0.0.

Differences With stylelint-loader

Both `stylelint-loader` and this module have their uses. `stylelint-loader` lints the files you `require` (or the ones you define as an `entry` in your `webpack` config). However, `@imports` in files are not followed, meaning only the main file for each `require/entry` is linted.

`stylelint-webpack-plugin` allows defining a `glob pattern`) matching the configuration and use of `stylelint`.

Getting Started

To begin, you'll need to install `stylelint-webpack-plugin`:

```
$ npm install stylelint-webpack-plugin --save-dev
```

Then add the plugin to your `webpack` config. For example:

file.ext

```
import file from 'file.ext';

// webpack.config.js
const StyleLintPlugin = require('stylelint-webpack-plugin');

module.exports = {
  // ...
  plugins: [
    new StyleLintPlugin(options),
  ],
  // ...
}
```

And run `webpack` via your preferred method.

Options

See stylelint's [options](#) for the complete list of options available. These options are passed through to the `stylelint` directly.

configFile

Type: `String` Default: `undefined`

Specify the config file location to be used by `stylelint`.

Note: By default this is handled by `stylelint` via `cosmiconfig`.

context

Type: `String` Default: `compiler.context`

A `String` indicating the root of your SCSS files.

emitErrors

Type: `Boolean` Default: `true`

If true, pipes `stylelint` error severity messages to the `webpack` compiler's error message handler.

Note: When this property is disabled all `stylelint` messages are piped to the `webpack` compiler's warning message handler.

failOn Error

Type: `Boolean` Default: `false`

If true, throws a fatal error in the global build process. This will end the build process on any `stylelint` error.

files

Type: `String|Array[String]` Default: `'**/*.{s?(a|c)ss}'`

Specify the glob pattern for finding files. Must be relative to `options.context`.

formatter

Type: Object Default: require('stylelint').formatters.string

Specify a custom formatter to format errors printed to the console.

lintDirtyModulesOnly

Type: Boolean Default: false

Lint only changed files, skip lint on start.

syntax

Type: String Default: undefined

See the `styelint` [user guide](#) for more info. e.g. use '`scss`' to lint `.scss` files.

Error Reporting

By default the plugin will dump full reporting of errors. Set `failOnError` to true if you want `webpack` build process breaking with any stylelint error. You can use the `quiet` option to avoid error output to the console.

Acknowledgement

This project was inspired by, and is a heavily modified version of `sasslint-webpack-plugin`.

Thanks to Javier ([@vieron](#)) for authoring this plugin.

License

[MIT](#) [MIT](#)" class="icon-link" href="#mit">>

Terser Webpack Plugin

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![cover][cover]][cover-url] [![chat][chat]][chat-url] [![size][size]][size-url]

This plugin uses [terser](#) to minify your JavaScript.

i For `webpack@3` use [terser-webpack-plugin-legacy](#) package

Getting Started

To begin, you'll need to install `terser-webpack-plugin`:

```
$ npm install terser-webpack-plugin --save-dev
```

Then add the plugin to your `webpack` config. For example:

webpack.config.js

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  optimization: {
    minimizer: [new TerserPlugin()],
  },
};
```

And run `webpack` via your preferred method.

Options

test

Type: `String | RegExp | Array<String | RegExp>` Default: `/\.\m?js(\?.*)?$/i`

Test to match files against.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        test: /\.js(\?.*)?$/i,
    }),
  ],
};
```

```
  ] ,  
} ,  
};
```

include

Type: String|RegExp|Array<String|RegExp> Default: undefined

Files to include.

webpack.config.js

```
module.exports = {  
  optimization: {  
    minimizer: [  
      new TerserPlugin({  
        include: /\u002fincludes/,  
      }),  
    ],  
  },  
};
```

exclude

Type: String|RegExp|Array<String|RegExp> Default: undefined

Files to exclude.

webpack.config.js

```
module.exports = {  
  optimization: {  
    minimizer: [  
      new TerserPlugin({  
        exclude: /\u002fexcludes/,  
      }),  
    ],  
  },  
};
```

chunkFilter

Type: Function<(chunk) -> boolean> Default: () => true

Allowing to filter which chunks should be uglified (by default all chunks are uglified). Return `true` to uglify the chunk, `false` otherwise.

webpack.config.js

```
module.exports = {
```

```
optimization: {
  minimizer: [
    new TerserPlugin({
      chunkFilter: (chunk) => {
        // Exclude uglification for the `vendor` chunk
        if (chunk.name === 'vendor') {
          return false;
        }

        return true;
      },
    }),
  ],
},
};
```

cache

Type: Boolean|String Default: false

Enable file caching. Default path to cache directory: `node_modules/.cache/terser-webpack-plugin`.

i If you use your own `minify` function please read the `minify` section for cache invalidation correctly.

Boolean

Enable/disable file caching.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        cache: true,
      }),
    ],
  },
};
```

String

Enable file caching and set path to cache directory.

webpack.config.js

```
module.exports = {
  optimization: {
```

```

minimizer: [
  new TerserPlugin({
    cache: 'path/to/cache',
  }),
],
},
};

```

cacheKeys

Type: Function<(defaultCacheKeys, file) -> Object> **Default:** defaultCacheKeys => defaultCacheKeys

Allows you to override default cache keys.

Default cache keys:

```
{
  terser: require('terser/package.json').version, // terser version
  'terser-webpack-plugin': require('../package.json').version, // plugin
  'terser-webpack-plugin-options': this.options, // plugin options
  path: compiler.outputPath ? `${compiler.outputPath}/${file}` : file,
  hash: crypto
    .createHash('md4')
    .update(input)
    .digest('hex'), // source file hash
});

```

webpack.config.js

```

module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        cache: true,
        cacheKeys: (defaultCacheKeys, file) => {
          defaultCacheKeys.myCacheKey = 'myCacheKeyValue';

          return defaultCacheKeys;
        },
      }),
    ],
  },
};

```

parallel

Type: Boolean|Number **Default:** false

Use multi-process parallel running to improve the build speed. Default number of concurrent runs: os.cpus().length - 1.

i □ Parallelization can speedup your build significantly and is therefore **highly recommended**.

Boolean

Enable/disable multi-process parallel running.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        parallel: true,
      }),
    ],
  },
};
```

Number

Enable multi-process parallel running and set number of concurrent runs.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        parallel: 4,
      }),
    ],
  },
};
```

sourceMap

Type: Boolean Default: false

Use source maps to map error message locations to modules (this slows down the compilation). If you use your own `minify` function please read the `minify` section for handling source maps correctly.

□ □ **cheap-source-map** options don't work with this plugin.

webpack.config.js

```
module.exports = {
  optimization: {
```

```
    minimizer: [
      new TerserPlugin({
        sourceMap: true,
      }),
    ],
  },
};
```

minify

Type: Function Default: undefined

Allows you to override default minify function. By default plugin uses [terser](#) package. Useful for using and testing unpublished versions or forks.

Always use **require** inside **minify** function when **parallel** option enabled.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        minify: (file, sourceMap) => {
          const extractedComments = [];

          // Custom logic for extract comments

          const { error, map, code, warnings } = require('uglify-module')
            .minify(file, {
              /* Your options for minification */
            });

          return { error, map, code, warnings, extractedComments };
        },
      }),
    ],
  },
};
```

terserOptions

Type: Object Default: default

Terser minify options.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
```

```
terserOptions: {
  ecma: undefined,
  warnings: false,
  parse: {},
  compress: {},
  mangle: true, // Note `mangle.properties` is `false` by default
  module: false,
  output: null,
  toplevel: false,
  nameCache: null,
  ie8: false,
  keep_classnames: undefined,
  keep_fnames: false,
  safari10: false,
},
),
],
},
},
};
```

extractComments

Type: Boolean|String|RegExp|Function<(node, comment) -> Boolean|Object>|Object **Default:** false

Whether comments shall be extracted to a separate file, (see [details](#)). By default extract only comments using `/^**\!|@preserve|@license|@cc_on/i` regexp condition and remove remaining comments. If the original file is named `foo.js`, then the comments will be stored to `foo.js.LICENSE`. The `terserOptions.output.comments` option specifies whether the comment will be preserved, i.e. it is possible to preserve some comments (e.g. annotations) while extracting others or even preserving comments that have been extracted.

Boolean

Enable/disable extracting comments.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        extractComments: true,
      }),
    ],
  },
};
```

String

Extract all or some (use `/^**!|@preserve|@license|@cc_on/i` RegExp) comments.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        extractComments: 'all',
      }),
    ],
  },
};
```

RegExp

All comments that match the given expression will be extracted to the separate file.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        extractComments: /@extract/i,
      }),
    ],
  },
};
```

Function<(node, comment) -> Boolean>

All comments that match the given expression will be extracted to the separate file.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        extractComments: (astNode, comment) => {
          if (/@extract/i.test(comment.value)) {
            return true;
          }

          return false;
        },
      }),
    ],
  },
};
```

Object

Allow to customize condition for extract comments, specify extracted file name and banner.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        extractComments: {
          condition: /^\\**!|@preserve|@license|@cc_on/i,
          filename: (file) => {
            return `${file}.LICENSE`;
          },
          banner: (licenseFile) => {
            return `License information can be found in ${licenseFile}`;
          },
        },
      }),
    ],
  },
};
```

condition

Type: Boolean|String|RegExp|Function<(node, comment) -> Boolean|Object>

Condition what comments you need extract.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        extractComments: {
          condition: 'some',
          filename: (file) => {
            return `${file}.LICENSE`;
          },
          banner: (licenseFile) => {
            return `License information can be found in ${licenseFile}`;
          },
        },
      }),
    ],
  },
};
```

filename

Type: String|Function<(string) -> String **Default:** \${file}.LICENSE

The file where the extracted comments will be stored. Default is to append the suffix .LICENSE to the original filename.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        extractComments: {
          condition: /^ !***!|@preserve|@license|@cc_on/i,
          filename: 'extracted-comments.js',
          banner: (licenseFile) => {
            return `License information can be found in ${licenseFile}`,
          },
        },
      }),
    ],
  },
};
```

banner

Type: Boolean|String|Function<(string) -> String **Default:** /*! For license information please see \${commentsFile} */

The banner text that points to the extracted file and will be added on top of the original file. Can be `false` (no banner), a `String`, or a `Function<(string) -> String` that will be called with the filename where extracted comments have been stored. Will be wrapped into comment.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        extractComments: {
          condition: true,
          filename: (file) => {
            return `${file}.LICENSE`;
          },
          banner: (commentsFile) => {
            return `My custom banner about license information ${commentsFile}`;
          },
        },
      }),
    ],
  },
};
```

warningsFilter

Type: Function<(warning, source) -> Boolean> Default: () => true

Allow to filter terser warnings. Return `true` to keep the warning, `false` otherwise.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        warningsFilter: (warning, source) => {
          if (/Dropping unreachable code/i.test(warning)) {
            return true;
          }

          if (/filename\.js/i.test(source)) {
            return true;
          }

          return false;
        },
      }),
    ],
  },
};
```

Examples

Cache And Parallel

Enable cache and multi-process parallel running.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        cache: true,
        parallel: true,
      }),
    ],
  },
};
```

Preserve Comments

Extract all legal comments (i.e. `/^**!|@preserve|@license|@cc_on/i`) and preserve

```
/@license/i comments.
```

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        terserOptions: {
          output: {
            comments: /@license/i,
          },
        },
        extractComments: true,
      }),
    ],
  },
};
```

Remove Comments

If you avoid building with comments, set **terserOptions.output.comments** to **false** as in this config:

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        terserOptions: {
          output: {
            comments: false,
          },
        },
      }),
    ],
  },
};
```

Custom Minify Function

Override default minify function - use `uglify-js` for minification.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        // Uncomment lines below for cache invalidation correctly
        // cache: true,
        // cacheKeys: (defaultCacheKeys) => {
```

```
//      delete defaultCacheKeys.terser;
//
//      return Object.assign(
//          {},
//          defaultCacheKeys,
//          { 'uglify-js': require('uglify-js/package.json').version
//          });
// },
minify: (file, sourceMap) => {
    // https://github.com/mishoo/UglifyJS2#minify-options
    const uglifyJsOptions = {
        /* your `uglify-js` package options */
    };

    if (sourceMap) {
        uglifyJsOptions.sourceMap = {
            content: sourceMap,
        };
    }

    return require('uglify-js').minify(file, uglifyJsOptions);
},
),
],
},
);
};
```

Contributing

Please take a moment to read our contributing guidelines if you haven't yet done so.

CONTRIBUTING

License

MIT

UglifyjsWebpackPlugin

[![npm][npm]][npm-url] [![node][node]][node-url] [![deps][deps]][deps-url] [![tests][tests]][tests-url] [![cover][cover]][cover-url] [![chat][chat]][chat-url] [![size][size]][size-url]

This plugin uses [uglify-js](#) to minify your JavaScript.

Requirements

This module requires a minimum of Node v6.9.0 and Webpack v4.0.0.

Getting Started

To begin, you'll need to install `uglifyjs-webpack-plugin`:

```
$ npm install uglifyjs-webpack-plugin --save-dev
```

Then add the plugin to your `webpack` config. For example:

webpack.config.js

```
const UglifyJsPlugin = require('uglifyjs-webpack-plugin');

module.exports = {
  optimization: {
    minimizer: [new UglifyJsPlugin()],
  },
};
```

And run `webpack` via your preferred method.

选项

test

Type: String|RegExp|Array<String|RegExp> Default: /\.\js(\?.*)?\$/i

Test to match files against.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
```

```
        new UglifyJsPlugin({
          test: /\.js(\?.*)?$/i,
        }) ,
      ] ,
    } ,
} ;
```

include

Type: String|RegExp|Array<String|RegExp> Default: undefined

Files to include.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        include: /\includes/,
      }) ,
    ] ,
  } ,
};
```

exclude

Type: String|RegExp|Array<String|RegExp> Default: undefined

Files to exclude.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        exclude: /\excludes/,
      }) ,
    ] ,
  } ,
};
```

chunkFilter

Type: Function<(chunk) -> boolean> Default: () => true

Allowing to filter which chunks should be uglified (by default all chunks are uglified).
Return `true` to uglify the chunk, `false` otherwise.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        chunkFilter: (chunk) => {
          // Exclude uglification for the `vendor` chunk
          if (chunk.name === 'vendor') {
            return false;
          }

          return true;
        }
      }),
    ],
  },
};
```

cache

Type: Boolean|String Default: false

Enable file caching. Default path to cache directory: `node_modules/.cache/uglifyjs-webpack-plugin`.

i If you use your own `minify` function please read the `minify` section for cache invalidation correctly.

Boolean

Enable/disable file caching.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        cache: true,
      }),
    ],
  },
};
```

String

Enable file caching and set path to cache directory.

webpack.config.js

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        cache: 'path/to/cache',
      }),
    ],
  },
};
```

cacheKeys

Type: Function<(defaultCacheKeys, file) -> Object> **Default:** defaultCacheKeys => defaultCacheKey

Allows you to override default cache keys.

Default cache keys:

```
({
  'uglify-js': require('uglify-js/package.json').version, // uglify version
  'uglifyjs-webpack-plugin': require('../package.json').version, // plugin version
  'uglifyjs-webpack-plugin-options': this.options, // plugin options
  path: compiler.outputPath ? `${compiler.outputPath}/${file}` : file,
  hash: crypto
    .createHash('md4')
    .update(input)
    .digest('hex'), // source file hash
});
```

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        cache: true,
        cacheKeys: (defaultCacheKeys, file) => {
          defaultCacheKeys.myCacheKey = 'myCacheKeyValue';

          return defaultCacheKeys;
        },
      }),
    ],
  },
};
```

parallel

Type: Boolean|Number **Default:** false

Use multi-process parallel running to improve the build speed. Default number of concurrent runs: `os.cpus().length - 1`.

- Parallelization can speedup your build significantly and is therefore **highly recommended**.

Boolean

Enable/disable multi-process parallel running.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        parallel: true,
      }),
    ],
  },
};
```

Number

Enable multi-process parallel running and set number of concurrent runs.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        parallel: 4,
      }),
    ],
  },
};
```

sourceMap

Type: Boolean Default: false

Use source maps to map error message locations to modules (this slows down the compilation). If you use your own `minify` function please read the `minify` section for handling source maps correctly.

- `cheap-source-map` options don't work with this plugin.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        sourceMap: true,
      }),
    ],
  },
};
```

minify

Type: Function Default: undefined

Allows you to override default minify function. By default plugin uses [uglify-js](#) package. Useful for using and testing unpublished versions or forks.

Always use `require` inside `minify` function when `parallel` option enabled.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        minify(file, sourceMap) {
          const extractedComments = [];

          // Custom logic for extract comments

          const { error, map, code, warnings } = require('uglify-module')
            .minify(file, {
              /* Your options for minification */
            });

          return { error, map, code, warnings, extractedComments };
        },
      ]),
    ],
  },
};
```

uglifyOptions

Type: Object Default: default

UglifyJS minify options.

webpack.config.js

```
module.exports = {
  optimization: {
```

```
minimizer: [
  new UglifyJsPlugin({
    uglifyOptions: {
      warnings: false,
      parse: {},
      compress: {},
      mangle: true, // Note `mangle.properties` is `false` by default
      output: null,
      toplevel: false,
      nameCache: null,
      ie8: false,
      keep_fnames: false,
    },
  }),
],
},
},
};
```

extractComments

Type: Boolean|String|RegExp|Function<(node, comment) -> Boolean|Object>
Default: false

Whether comments shall be extracted to a separate file, (see [details](#)). By default extract only comments using `/^**!|@preserve|@license|@cc_on/i` regexp condition and remove remaining comments. If the original file is named `foo.js`, then the comments will be stored to `foo.js.LICENSE`. The `uglifyOptions.output.comments` option specifies whether the comment will be preserved, i.e. it is possible to preserve some comments (e.g. annotations) while extracting others or even preserving comments that have been extracted.

Boolean

Enable/disable extracting comments.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        extractComments: true,
      }),
    ],
  },
};
```

String

Extract all or some (use `/^**!|@preserve|@license|@cc_on/i` RegExp) comments.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        extractComments: 'all',
      }),
    ],
  },
};
```

RegExp

All comments that match the given expression will be extracted to the separate file.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        extractComments: /@extract/i,
      }),
    ],
  },
};
```

Function<(node, comment) -> Boolean>

All comments that match the given expression will be extracted to the separate file.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        extractComments: function(astNode, comment) {
          if (/@extract/i.test(comment.value)) {
            return true;
          }

          return false;
        },
      }),
    ],
  },
};
```

Object

Allow to customize condition for extract comments, specify extracted file name and banner.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        extractComments: {
          condition: /^\\**!|@preserve|@license|@cc_on/i,
          filename(file) {
            return `${file}.LICENSE`;
          },
          banner(licenseFile) {
            return `License information can be found in ${licenseFile}`;
          },
        },
      }),
    ],
  },
};
```

condition

Type: Boolean|String|RegExp|Function<(node, comment) -> Boolean|Object>

Condition what comments you need extract.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        extractComments: {
          condition: 'some',
          filename(file) {
            return `${file}.LICENSE`;
          },
          banner(licenseFile) {
            return `License information can be found in ${licenseFile}`;
          },
        },
      }),
    ],
  },
};
```

filename

Type: Regex|Function<(string) -> String> Default: \${file}.LICENSE

The file where the extracted comments will be stored. Default is to append the suffix `.LICENSE` to the original filename.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        extractComments: {
          condition: /^\\**!|@preserve|@license|@cc_on/i,
          filename: 'extracted-comments.js',
          banner(licenseFile) {
            return `License information can be found in ${licenseFile}`,
          },
        },
      }),
    ],
  },
};
```

banner

Type: Boolean|String|Function<(string) -> String> **Default:** /*! For license information please see \${commentsFile} */

The banner text that points to the extracted file and will be added on top of the original file. Can be `false` (no banner), a `String`, or a `Function<(string) -> String>` that will be called with the filename where extracted comments have been stored. Will be wrapped into comment.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        extractComments: {
          condition: true,
          filename(file) {
            return `${file}.LICENSE`;
          },
          banner(commentsFile) {
            return `My custom banner about license information ${comment}`;
          },
        },
      }),
    ],
  },
};
```

warningsFilter

Type: Function<(warning, source) -> Boolean> Default: () => true

Allow to filter `uglify-js` warnings. Return `true` to keep the warning, `false` otherwise.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        warningsFilter: (warning, source) => {
          if (/Dropping unreachable code/i.test(warning)) {
            return true;
          }

          if (/filename\.js/i.test(source)) {
            return true;
          }

          return false;
        },
      }),
    ],
  },
};
```

Examples

Cache And Parallel

Enable cache and multi-process parallel running.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        cache: true,
        parallel: true,
      }),
    ],
  },
};
```

Preserve Comments

Extract all legal comments (i.e. `/^**!|@preserve|@license|@cc_on/i`) and preserve `@license/i` comments.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        uglifyOptions: {
          output: {
            comments: /@license/i,
          },
        },
        extractComments: true,
      }),
    ],
  },
};
```

Remove Comments

If you avoid building with comments, set **uglifyOptions.output.comments** to **false** as in this config:

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        uglifyOptions: {
          output: {
            comments: false,
          },
        },
      }),
    ],
  },
};
```

Custom Minify Function

Override default minify function - use terser for minification.

webpack.config.js

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        // Uncomment lines below for cache invalidation correctly
        // cache: true,
        // cacheKeys(defaultCacheKeys) {
        //   delete defaultCacheKeys['uglify-js'];
        //
        //   return Object.assign(
        //     {},
        //   );
      })
    ],
  }
};
```

```
//      defaultCacheKeys,
//      { 'uglify-js': require('uglify-js/package.json').version
//    );
//  },
minify(file, sourceMap) {
  // https://github.com/mishoo/UglifyJS2#minify-options
  const uglifyJsOptions = {
    /* your `uglify-js` package options */
  };

  if (sourceMap) {
    uglifyJsOptions.sourceMap = {
      content: sourceMap,
    };
  }

  return require('terser').minify(file, uglifyJsOptions);
},
),
],
},
};
```

Contributing

Please take a moment to read our contributing guidelines if you haven't yet done so.

CONTRIBUTING

License

MIT

WatchIgnorePlugin

无视指定的文件。换句话说，当处于监视模式([watch mode](#))下，符合给定地址的文件或者满足给定正则表达式的文件的改动不会触发重编译。

```
new webpack.WatchIgnorePlugin(paths);
```

选项

- 路径(paths) (array): 一个正则表达式或者绝对路径的数组。表示符合条件的文件将不会被监视

Internal webpack plugins

This is a list of plugins which are used by webpack internally.

You should only care about them if you are building your own compiler based on webpack, or introspect the internals.

Categories of internal plugins:

- [environment](#)
- [compiler](#)
- [entry](#)
- [output](#)
- [source](#)
- [optimize](#)

environment

Plugins affecting the environment of the compiler.

NodeEnvironmentPlugin

```
node/NodeEnvironmentPlugin()
```

Applies Node.js style filesystem to the compiler.

compiler

Plugins affecting the compiler

CachePlugin

```
CachePlugin([cache])
```

Adds a cache to the compiler, where modules are cached.

You can pass a `cache` object, where the modules are cached. Otherwise one is created per plugin instance.

ProgressPlugin

```
ProgressPlugin(handler)
```

Hook into the compiler to extract progress information. The `handler` must have the signature `function(percentage, message)`. Percentage is called with a value between 0 and 1, where 0 indicates the start and 1 the end.

RecordIdsPlugin

```
RecordIdsPlugin()
```

Saves and restores module and chunk ids from records.

entry

Plugins, which add entry chunks to the compilation.

SingleEntryPlugin

```
SingleEntryPlugin(context, request, chunkName)
```

Adds an entry chunk on compilation. The chunk is named `chunkName` and contains only one module (plus dependencies). The module is resolved from `request` in `context` (absolute path).

MultiEntryPlugin

```
MultiEntryPlugin(context, requests, chunkName)
```

Adds an entry chunk on compilation. The chunk is named `chunkName` and contains a module for each item in the `requests` array (plus dependencies). Each item in `requests` is resolved in `context` (absolute path).

PrefetchPlugin

```
PrefetchPlugin(context, request)
```

Prefetches `request` and dependencies to enable a more parallel compilation. It doesn't create any chunk. The module is resolved from `request` in `context` (absolute path).

output

FunctionModulePlugin

```
FunctionModulePlugin(context, options)
```

Each emitted module is wrapped in a function.

`options` are the output options.

If `options.pathinfo` is set, each module function is annotated with a comment containing the module identifier shortened to `context` (absolute path).

JsonpTemplatePlugin

`JsonpTemplatePlugin(options)`

Chunks are wrapped into JSONP-calls. A loading algorithm is included in entry chunks. It loads chunks by adding a `<script>` tag.

`options` are the output options.

`options.jsonpFunction` is the JSONP function.

`options.publicPath` is used as path for loading the chunks.

`options.chunkFilename` is the filename under that chunks are expected.

NodeTemplatePlugin

`node/NodeTemplatePlugin(options)`

Chunks are wrapped into Node.js modules exporting the bundled modules. The entry chunks loads chunks by requiring them.

`options` are the output options.

`options.chunkFilename` is the filename under that chunks are expected.

LibraryTemplatePlugin

`LibraryTemplatePlugin(name, target)`

The entries chunks are decorated to form a library `name` of type `type`.

WebWorkerTemplatePlugin

`webworker/WebWorkerTemplatePlugin(options)`

Chunks are loaded by `importScripts`. Else it's similar to [JsonpTemplatePlugin](#).

`options` are the output options.

EvalDevToolModulePlugin

Decorates the module template by wrapping each module in a `eval` annotated with `// @sourceURL`.

SourceMapDevToolPlugin

```
SourceMapDevToolPlugin(sourceMapFilename, sourceMappingURLComment,  
moduleFilenameTemplate, fallbackModuleFilenameTemplate)
```

Decorates the templates by generating a SourceMap for each chunk.

`sourceMapFilename` the filename template of the SourceMap. `[hash]`, `[name]`, `[id]`, `[file]` and `[filebase]` are replaced. If this argument is missing, the SourceMap will be inlined as DataUrl.

NoHotModuleReplacementPlugin

```
NoHotModuleReplacementPlugin()
```

Defines `module.hot` as `false` to remove hot module replacement code.

HotModuleReplacementPlugin

```
HotModuleReplacementPlugin(options)
```

Add support for hot module replacement. Decorates the templates to add runtime code.
Adds `module.hot` API.

`options.hotUpdateChunkFilename` the filename for hot update chunks.

`options.hotUpdateMainFilename` the filename for the hot update manifest.

`options.hotUpdateFunction` JSON function name for the hot update.

Source

Plugins affecting the source code of modules.

APIPlugin

Make `webpack_public_path`, `webpack_require`, `webpack_modules` and `webpack_chunk_load` accessible. Ensures that `require.valueOf` and `require.onError` are not processed by other plugins.

CompatibilityPlugin

Currently useless. Ensures compatibility with other module loaders.

ConsolePlugin

Offers a pseudo `console` if it is not available.

ConstPlugin

Tries to evaluate expressions in `if (. . .)` statements and ternaries to replace them with `true/false` for further possible dead branch elimination using hooks fired by the parser.

There are multiple optimizations in production mode regarding dead branches:

- The ones performed by Terser
- The ones performed by webpack

webpack will try to evaluate conditional statements. If it succeeds then the dead branch is removed. webpack can't do constant folding unless the compiler knows it. For example:

```
import { calculateTax } from './tax';

const FOO = 1;
if (FOO === 0) {
  // dead branch
  calculateTax();
}
```

In the above example, webpack is unable to prune the branch, but Terser does. However, if `FOO` is defined using [DefinePlugin](#), webpack will succeed.

It is important to mention that `import { calculateTax } from './tax';` will also get pruned because `calculateTax()` call was in the dead branch and got eliminated.

ProvidePlugin

```
ProvidePlugin(name, request)
```

If `name` is used in a module it is filled by a module loaded by `require(<request>)`.

NodeStuffPlugin

```
NodeStuffPlugin(options, context)
```

Provide stuff that is normally available in Node.js modules.

It also ensures that `module` is filled with some Node.js stuff if you use it.

RequireJsStuffPlugin

Provide stuff that is normally available in require.js.

`require[js].config` is removed. `require.version` is 0.0.0. `requirejs.onError` is mapped to `require.onError`.

NodeSourcePlugin

```
node/NodeSourcePlugin(options)
```

This module adds stuff from Node.js that is not available in non Node.js environments.

It adds polyfills for `process`, `console`, `Buffer` and `global` if used. It also binds the built in Node.js replacement modules.

NodeTargetPlugin

```
node/NodeTargetPlugin()
```

The plugins should be used if you run the bundle in a Node.js environment.

If ensures that native modules are loaded correctly even if bundled.

AMDPlugin

```
dependencies/AMDPlugin(options)
```

Provides AMD-style `define` and `require` to modules. Also bind `require.amd`, `define.amd` and `webpack_amd_options##` to the `options` passed as parameter.

CommonJsPlugin

```
dependencies/CommonJsPlugin
```

Provides CommonJs-style `require` to modules.

LabeledModulesPlugin

```
dependencies/LabeledModulesPlugin()
```

Provide labels `require:` and `exports:` to modules.

RequireContextPlugin

```
dependencies/RequireContextPlugin(modulesDirectories, extensions)
```

Provides `require.context`. The parameter `modulesDirectories` and `extensions` are used to find alternative requests for files. It's useful to provide the same arrays as you provide to the resolver.

RequireEnsurePlugin

```
dependencies/RequireEnsurePlugin()
```

Provides `require.ensure`.

RequireIncludePlugin

```
dependencies/RequireIncludePlugin()
```

Provides `require.include`.

DefinePlugin

```
DefinePlugin(definitions)
```

Define constants for identifier.

`definitions` is an object.

optimize

LimitChunkCountPlugin

```
optimize/LimitChunkCountPlugin(options)
```

Merge chunks limit chunk count is lower than `options.maxChunks`.

The overhead for each chunks is provided by `options.chunkOverhead` or defaults to 10000. Entry chunks sizes are multiplied by `options.entryChunkMultiplicator` (or 10).

Chunks that reduce the total size the most are merged first. If multiple combinations are equal the minimal merged size wins.

MergeDuplicateChunksPlugin

```
optimize/MergeDuplicateChunksPlugin()
```

Chunks with the same modules are merged.

RemoveEmptyChunksPlugin

```
optimize/RemoveEmptyChunksPlugin()
```

Modules that are included in every parent chunk are removed from the chunk.

MinChunkSizePlugin

```
optimize/MinChunkSizePlugin(minChunkSize)
```

Merges chunks until each chunk has the minimum size of `minChunkSize`.

FlagIncludedChunksPlugin

```
optimize/FlagIncludedChunksPlugin()
```

Adds chunk ids of chunks which are included in the chunk. This eliminates unnecessary chunk loads.

OccurrenceOrderPlugin

```
optimize/OccurrenceOrderPlugin(preferEntry)
```

Order the modules and chunks by occurrence. This saves space, because often referenced modules and chunks get smaller ids.

`preferEntry` If true, references in entry chunks have higher priority

DedupePlugin

```
optimize/DedupePlugin()
```

Deduplicates modules and adds runtime code.

迁移

此章节包含关于从较早版本的 webpack 迁移到较新版本的信息。

To v4 from v3

This guide only shows major changes that affect end users. For more details please see [the changelog](#).

Node.js v4

If you are still using Node.js v4 or lower, you need to upgrade your Node.js installation to Node.js v6 or higher.

CLI

The CLI has moved to a separate package: `webpack-cli`. You need to install it before using `webpack`, see [basic setup](#).

Update plugins

Many 3rd-party plugins need to be upgraded to their latest version to be compatible.

mode

Add the new `mode` option to your config. Set it to production or development in your configuration depending on config type.

webpack.config.js

```
module.exports = {
  // ...
  mode: 'production',
}
```

Alternatively you can pass it via CLI: `--mode production`/`--mode development`

Deprecated/Removed plugins

These plugins can be removed from configuration as they are default in production mode:

webpack.config.js

```
module.exports = {
  // ...
  plugins: [
```

```
- new NoEmitOnErrorsPlugin(),
- new ModuleConcatenationPlugin(),
- new DefinePlugin({ "process.env.NODE_ENV": JSON.stringify("product"),
- new UglifyJsPlugin()
],
}
```

These plugins are default in development mode

webpack.config.js

```
module.exports = {
  // ...
  plugins: [
-   new NamedModulesPlugin()
  ],
}
```

These plugins were deprecated and are now removed:

webpack.config.js

```
module.exports = {
  // ...
  plugins: [
-   new NoErrorsPlugin(),
-   new NewWatchingPlugin()
  ],
}
```

CommonsChunkPlugin

The CommonsChunkPlugin was removed. Instead the `optimization.splitChunks` options can be used.

See documentation of the `optimization.splitChunks` for more details. The default configuration may already suit your needs.

When generating the HTML from the stats you can use

`optimization.splitChunks.chunks: "all"` which is the optimal configuration in most cases.

import() and CommonJS

When using `import()` to load non-ESM the result has changed in webpack 4. Now you need to access the `default` property to get the value of `module.exports`.

non-esm.js

```
module.exports = {
  sayHello: () => {
    console.log('hello world');
  }
};
```

example.js

```
function sayHello() {
  import('./non-esm.js').then(module => {
    module.default.sayHello();
  });
}
```

json and loaders

When using a custom loader to transform `.json` files you now need to change the `module.type`:

webpack.config.js

```
module.exports = {
  // ...
  rules: [
    {
      test: /config\.json$/,
      loader: 'special-loader',
+     type: 'javascript/auto',
      options: {...}
    }
  ]
};
```

When still using the `json-loader`, it can be removed:

webpack.config.js

```
module.exports = {
  // ...
  rules: [
    {
      test: /\.json$/,
      loader: 'json-loader'
    }
  ]
};
```

module.loaders

`module.loaders` were deprecated since webpack 2 and are now removed in favor of `module.rules`.

从 v1 升级到 v2 或 v3

以下各节描述从 webpack 1 到 webpack 2 的重大变化。

webpack 从 1 到 2 的变化，比从 2 到 3 要少很多，所以版本迁移起来难度应该不大。如果你遇到了问题，请查看 [更新日志](#) 以了解更多细节。

resolve.root, resolve.fallback, resolve.modulesDirectories

这些选项被一个单独的选项 `resolve.modules` 取代。更多用法请查看 [解析](#)。

```
resolve: {  
-   root: path.join(__dirname, "src")  
+   modules: [  
+     path.join(__dirname, "src"),  
+     "node_modules"  
+   ]  
}
```

resolve.extensions

此选项不再需要传一个空字符串。此行为被迁移到 `resolve.enforceExtension`。更多用法请查看 [解析](#)。

resolve.*

这里更改了几个 API。由于不常用，不在这里详细列出。更多用法请查看 [解析](#)。

module.loaders 改为 module.rules

旧的 loader 配置被更强大的 rules 系统取代，后者允许配置 loader 以及其他更多选项。为了兼容旧版，`module.loaders` 语法仍然有效，旧的属性名依然可以被解析。新的命名约定更易于理解，并且是升级配置使用 `module.rules` 的好理由。

```
module: {  
-   loaders: [  
+   rules: [  
     {  
       test: /\.css$/,  
-       loaders: [  
-         "style-loader",  
-         "css-loader?modules=true"  
+       use: [  
-
```

```

+
+         {
+             loader: "style-loader"
+         },
+         {
+             loader: "css-loader",
+             options: {
+                 modules: true
+             }
+         }
+     ]
}
{
    test: /\.jsx$/,
    loader: "babel-loader", // 这里不再使用 "use"
    options: {
        // ...
    }
}
]
}

```

链式 loader

就像在 webpack 1 中，loader 可以链式调用，上一个 loader 的输出被作为输入传给下一个 loader。使用 rule.use 配置选项，use 可以设置为一个 loader 数组。在 webpack 1 中，loader 通常被用 ! 连写。这一写法在 webpack 2 中只在使用旧的选项 module.loaders 时才有效。

```

module: {
-   loaders: [{ 
+   rules: [ {
        test: /\.less$/,
-   loader: "style-loader!css-loader!less-loader"
+   use: [
        "style-loader",
        "css-loader",
        "less-loader"
+   ]
} ]
}

```

取消「在模块名中自动添加 -loader 后缀」

在引用 loader 时，不能再省略 -loader 后缀了：

```

module: {
  rules: [
  {
    use: [
-     "style",
+     "style-loader",
-     "css",

```

```
+         "css-loader",
-         "less",
+         "less-loader",
     ]
 }
]
}
```

你仍然可以通过配置 `resolveLoader.moduleExtensions` 配置选项，启用这一旧有行为，但是我们不推荐这么做。

```
+ resolveLoader: {
+   moduleExtensions: ["-loader"]
+ }
```

了解这一改变背后的原因，请查看 [#2986](#)。

json-loader 不再需要手动添加

如果没有为 JSON 文件配置 loader，webpack 将自动尝试通过 `json-loader` 加载 JSON 文件。

```
- module: {
-   rules: [
-     {
-       test: /\.json$/,
-       loader: "json-loader"
-     }
-   ]
- }
```

我们决定这么做是为了消除 webpack、node.js 和 browserify 之间的环境差异。

配置中的 loader 默认相对于 context 进行解析

在 **webpack 1** 中，默认配置下 loader 解析相对于被匹配的文件。然而，在 **webpack 2** 中，默认配置下 loader 解析相对于 `context` 选项。

这解决了「在使用 `npm link` 或引用 `context` 上下文目录之外的模块时，loader 所导致的模块重复载入」的问题。

你可以移除掉那些为解决此问题的 hack 方案了：

```
- module: {
-   rules: [
-     {
-       // ...
-       loader: require.resolve("my-loader")
-     }
-   ]
+   loader: "my-loader"
+ }
```

```
        ],
      },
      resolveLoader: {
-        root: path.resolve(__dirname, "node_modules")
      }
    }
  }
}
```

移除 module.preLoaders 和 module.postLoaders

```
  module: {
-    preLoaders: [
+    rules: [
      {
        test: /\.js$/,
+        enforce: "pre",
        loader: "eslint-loader"
      }
    ]
  }
}
```

UglifyJsPlugin sourceMap

UglifyJsPlugin 的 `sourceMap` 选项现在默认为 `false` 而不是 `true`。这意味着如果你在压缩代码时启用了 source map，或者想要让 uglifyjs 的警告能够对应到正确的代码行，你需要将 UglifyJsPlugin 的 `sourceMap` 设为 `true`。

```
devtool: "source-map",
plugins: [
  new UglifyJsPlugin({
+    sourceMap: true
  })
]
```

UglifyJsPlugin warnings

UglifyJsPlugin 的 `compress.warnings` 选项现在默认为 `false` 而不是 `true`。这意味着如果你想要看到 uglifyjs 的警告信息，你需要将 `compress.warnings` 设为 `true`。

```
devtool: "source-map",
plugins: [
  new UglifyJsPlugin({
+    compress: {
+      warnings: true
+    }
  })
]
```

UglifyJsPlugin 压缩 loaders

`UglifyJsPlugin` 不再压缩 loaders。在未来很长一段时间里，需要通过设置 `minimize: true` 来压缩 loaders。参考 loader 文档里的相关选项。

loaders 的压缩模式将在 webpack 3 或后续版本中取消。

为了兼容旧的 loaders，loaders 可以通过插件来切换到压缩模式：

```
plugins: [
+   new webpack.LoaderOptionsPlugin({
+     minimize: true
+   })
]
```

移除 DedupePlugin

不再需要 `webpack.optimize.DedupePlugin`。请从配置中移除。

BannerPlugin - 破坏性改动

`BannerPlugin` 不再接受两个参数，而是只接受单独的 options 对象。

```
plugins: [
-   new webpack.BannerPlugin('Banner', {raw: true, entryOnly: true});
+   new webpack.BannerPlugin({banner: 'Banner', raw: true, entryOnly: true})
]
```

默认加载 OccurrenceOrderPlugin

`OccurrenceOrderPlugin` 现在默认启用，并已重命名（在 webpack 1 中为 `OccurrenceOrderPlugin`）。因此，请确保从你的配置中删除该插件：

```
plugins: [
  // webpack 1
-  new webpack.optimize.OccurrenceOrderPlugin()
  // webpack 2
-  new webpack.optimize.OccurrenceOrderPlugin()
]
```

ExtractTextWebpackPlugin - 破坏性改动

`ExtractTextPlugin` 需要使用版本 2，才能在 webpack 2 下正常运行。

```
npm install --save-dev extract-text-webpack-plugin
```

这一插件的配置变化主要体现在语法上。

`ExtractTextPlugin.extract`

```
module: {
  rules: [
    {
      test: /\.css$/,
      - loader: ExtractTextPlugin.extract("style-loader", "css-loader",
      + use: ExtractTextPlugin.extract({
      +   fallback: "style-loader",
      +   use: "css-loader",
      +   publicPath: "/dist"
      + })
    }
  ]
}
```

`new ExtractTextPlugin({options})`

```
plugins: [
- new ExtractTextPlugin("bundle.css", { allChunks: true, disable: false,
+ new ExtractTextPlugin({
+   filename: "bundle.css",
+   disable: false,
+   allChunks: true
+ })
]
```

全动态 `require` 现在默认会失败

只有一个表达式的依赖（例如 `require(expr)`）将创建一个空的 `context` 而不是一个完整目录的 `context`。

这样的代码应该进行重构，因为它不能与 ES2015 模块一起使用。如果你确定不会有 ES2015 模块，你可以使用 `ContextReplacementPlugin` 来指示 compiler 进行正确的解析。

[Link to an article about dynamic dependencies.](#)

在 CLI 和配置中使用自定义参数

如果你之前滥用 CLI 来传自定义参数到配置中，比如：

```
webpack --custom-stuff
```

```
// webpack.config.js
var customStuff = process.argv.indexOf('--custom-stuff') >= 0;
/* ... */
module.exports = config;
```

你将会发现新版中不再允许这么做。CLI 现在更加严格了。

替代地，现在提供了一个接口来传递参数给配置。我们应该采用这种新方式，在未来许多工具将可能依赖于此。

```
webpack --env.customStuff

module.exports = function(env) {
  var customStuff = env.customStuff;
  /* ... */
  return config;
};
```

详见 [CLI](#)。

require.ensure 以及 AMD **require** 将采用异步式调用

现在这些函数总是异步的，而不是当 chunk 已经加载完成的时候同步调用它们的回调函数(callback)。

require.ensure 现在依赖于原生的 **Promise**。如果在不支持 **Promise** 的环境里使用 **require.ensure**，你需要添加 **polyfill**。

通过 **options** 配置 loader

你不能再通过 `webpack.config.js` 的自定义属性来配置 loader。只能通过 `options` 来配置。下面配置的 `ts` 属性在 webpack 2 下不再有效：

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        loader: 'ts-loader'
      }
    ],
    // 在 webpack 2 中无效
    ts: { transpileOnly: false }
};
```

什么是 **options**？

好问题。严格来说，有两种办法，都可以用来配置 webpack 的 loader。典型的 `options` 被称为 `query`，是一个可以被添加到 loader 名之后的字符串。它比较像一个 `query string`，但是实际上有更强大的能力：

```
module.exports = {
//...
module: {
  rules: [
    test: /\.tsx?$/,
    loader: 'ts-loader?' + JSON.stringify({ transpileOnly: false })
  ]
}
};
```

不过它也可以分开来，写成一个单独的对象，紧跟在 loader 属性后面：

```
module.exports = {
//...
module: {
  rules: [
    test: /\.tsx?$/,
    loader: 'ts-loader',
    options: { transpileOnly: false }
  ]
}
};
```

LoaderOptionsPlugin context

有的 loader 需要从配置中读取一些 context 信息。在未来很长一段时间里，这将需要通过 loader options 传入。详见 loader 文档的相关选项。

为了保持对旧 loaders 的兼容，这些信息可以通过插件传进来：

```
plugins: [
+  new webpack.LoaderOptionsPlugin({
+    options: {
+      context: __dirname
+    }
+  })
]
```

debug

在 webpack 1 中 `debug` 选项可以将 loader 切换到调试模式(debug mode)。在未来很长一段时间里，这将需要通过 loader 选项传递。详见 loader 文档的相关选项。

loaders 的调试模式将在 webpack 3 或后续版本中取消。

为了保持对旧 loaders 的兼容，loader 可以通过插件来切换到调试模式：

```
- debug: true,
plugins: [
+  new webpack.LoaderOptionsPlugin({
+    debug: true
}
```

```
+   })
]
```

ES2015 的代码分割

在 webpack 1 中，可以使用 `require.ensure` 作为实现应用程序的懒加载 chunks 的一种方法：

```
require.ensure([], function(require) {
  var foo = require('./module');
});
```

ES2015 模块加载规范定义了 `import()` 方法，可以在运行时(runtime)动态地加载 ES2015 模块。webpack 将 `import()` 作为分割点(split-point)并将所要请求的模块(requested module)放置到一个单独的 chunk 中。`import()` 接收模块名作为参数，并返回一个 Promise。

```
function onClick() {
  import('./module').then(module => {
    return module.default;
  }).catch(err => {
    console.log('Chunk loading failed');
  });
}
```

好消息是：如果加载 chunk 失败，我们现在可以进行处理，因为现在它基于 `Promise`。

动态表达式

可以传递部分表达式给 `import()`。这与 CommonJS 对表达式的处理方式一致（webpack 为所有可能匹配的文件创建 `context`）。

`import()` 为每一个可能的模块创建独立的 chunk。

```
function route(path, query) {
  return import(`./routes/${path}/route`)
    .then(route => new route.Route(query));
}
// 上面代码为每个可能的路由创建独立的 chunk
```

混合使用 ES2015、AMD 和 CommonJS

你可以自由混合使用三种模块类型（甚至在同一个文件中）。在这个情况下 webpack 的行为和 babel 以及 node-eps 一致：

```
// CommonJS 调用 ES2015 模块
```

```
var book = require('./book');

book.currentPage;
book.readPage();
book.default === 'This is a book';

// ES2015 模块调用 CommonJS
import fs from 'fs'; // module.exports 映射到 default
import { readFileSync } from 'fs'; // 从返回对象(returned object+)中读取值
typeof fs.readFileSync === 'function';
typeof readFileSync === 'function';
```

值得注意的是，你需要让 Babel 不解析这些模块符号，从而让 webpack 可以使用它们。你可以通过设置如下配置到 .babelrc 或 babel-loader 来实现这一点。

.babelrc

```
{
  "presets": [
    ["es2015", { "modules": false }]
  ]
}
```

Hints

不需要改变什么，但有机会改变。

模板字符串

webpack 现在支持表达式中的模板字符串了。这意味着你可以在 webpack 构建中使用它们：

```
- require("./templates/" + name);
+ require(`./templates/${name}`);
```

配置中使用 Promise

webpack 现在支持在配置文件中返回 `Promise` 了。这让你能在配置文件中做异步处理。

webpack.config.js

```
module.exports = function() {
  return fetchLangs().then(lang => ({
    entry: '...',
    // ...
    plugins: [
      new DefinePlugin({ LANGUAGE: lang })
    ]
  })
}
```

```
});  
};
```

高级 loader 匹配

webpack 现在支持对 loader 进行更多方式的匹配。

```
module.exports = {  
  //...  
  module: {  
    rules: [  
      {  
        resource: /filename/, // 匹配 "/path/filename.js"  
        resourceQuery: /^\\?querystring$/, // 匹配 "?querystring"  
        issuer: /filename/, // 如果请求 "/path/filename.js" 则匹配 "/path/  
      }  
    ]  
  }  
};
```

更多的 CLI 参数项

你可以使用一些新的 CLI 参数项：

--define process.env.NODE_ENV="production" 见 [DefinePlugin](#)。

--display-depth 显示每个模块到入口的距离。

--display-used-exports 显示一个模块中被使用的 exports 信息。

--display-max-modules 设置输出时显示的模块数量（默认是 15）。

-p 能够定义 process.env.NODE_ENV 为 "production"。

Loader 变更

以下变更仅影响 loader 的开发者。

Cacheable

Loaders 现在默认可被缓存。Loaders 如果不想被缓存，需要选择不被缓存。

```
// 缓存 loader  
module.exports = function(source) {  
-  this.cacheable();  
  return source;  
}  
  
// 不缓存 loader
```

```
module.exports = function(source) {  
+  this.cacheable(false);  
  return source;  
}
```

复合 options

webpack 1 只支持能够「可 `JSON.stringify` 的对象」作为 loader 的 options。

webpack 2 现在支持任意 JS 对象作为 loader 的 options.

webpack 2.2.1之前（即从 2.0.0 到 2.2.0），使用复合 options，需要在 `options` 对象上添加 `ident`，允许它能够被其他 loader 引用。这在 2.2.1 中被删除，因此目前的迁移不再需要使用 `ident` 键。

```
{  
  test: /\.ext/  
  use: {  
    loader: '...',  
    options: {  
-      ident: 'id',  
-      fn: () => require('./foo.js')  
    }  
  }  
}
```

