

# Printable

## 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.js`，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。

### Tip

[createapp.dev](#) 是一个创建自定义 webpack 配置的在线工具。它允许你选择各种特性，其将会被添加到最后的配置文件中。它也会基于生成的 webpack 配置初始化一个示例项目，你可以在浏览器中查看该项目并且下载。

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，然后 webpack 会自动使用它。

下面指定了所有可用的配置选项。

### Tip

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！

## 使用不同的配置文件

如果出于某些原因，需要根据特定情况使用不同的配置文件，则可以通过在命令行中使用 `--config` 标志修改。

### package.json

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

## 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。

### Tip

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](#)。

## Warning

Notice that many array configurations allow to reference the default value via `"..."`.

### webpack.config.js

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

module.exports = {
  <mode "/configuration/mode">
    <default>
      mode: "production", // "production" | "development" | "none"
    </default>
    mode: "production", // enable many optimizations for production builds
    mode: "development", // enabled useful tools for development
    mode: "none", // no defaults
  </mode>
  // Chosen mode tells webpack to use its built-in optimizations accordingly.
  <entry "/configuration/entry-context/#entry">
    <default>
      entry: "./app/entry", // string | object | array
    </default>
    entry: ["./app/entry1", "./app/entry2"],
    entry: {
      a: "./app/entry-a",
      b: ["./app/entry-b1", "./app/entry-b2"]
    },
    entry: {
      a: {
        import: "./app/entry-a",
        library: { type: "commonjs-module" },
        dependOn: ["vendors"],
        filename: "entries/a.js",
        runtime: "dashboard",
        chunkLoading: "jsonp",
        wasmLoading: "fetch-streaming"
      },
      vendors: ["react", "react-dom"]
    }
  </entry>
  // 默认为 ./src
  // 这里应用程序开始执行
  // webpack 开始打包
  <link "/configuration/output">
    <default>
      output: {
    </default>
```

```
</link>
// webpack 如何输出结果的相关选项
<path "/configuration/output/#outputpath">
  <default>
    path: path.resolve(__dirname, "dist"), // string (default)
  </default>
</path>
// 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 "/configuration/output/#outputfilename">
  <default>
    filename: "[name].js", // string (default)
  </default>
  filename: "[contenthash].js", // 用于长效缓存
</filename>
// entry chunk 的文件名模板
<publicPath "/configuration/output/#outputpublicpath">
  <default>
    publicPath: "/assets/", // string
  </default>
  publicPath: "auto", // determined automatically from script tag (default)
  publicPath: "", // relative to HTML file
  publicPath: "/", // relative to server root
  publicPath: "https://cdn.example.com/", // absolute URL
</publicPath>
// 输出解析文件的目录, url 相对于 HTML 页面
<library "/configuration/output/#outputlibrary">
  <default>
    library: { // 这里有一种旧的语法形式可以使用（点击显示）
  </default>
  // 旧语法:
  library: "MyLibrary", // string,
  libraryTarget: "umd", // = library.type
  auxiliaryComment: "comment", // = library.auxiliaryComment
  libraryExport: "default", // = library.export
</library>
<libraryType "/configuration/output/#outputlibrarytype">
  <default>
    type: "umd", // 通用模块定义
  </default>
  type: "umd2", // 通用模块定义 (but optional externals will use globals)
  type: "commonjs-module", // exported with module.exports
  type: "commonjs2", // old name for "commonjs-module"
  type: "commonjs", // exported as properties to exports
  type: "amd", // defined with AMD define() method
  type: "amd-require", // defined with AMD require() method (can't have exports)
  type: "system", // exported to System.js
  type: "this", // property set on this
  type: "var", // variable defined in root scope (default)
  type: "assign", // blind assignment
  type: "global", // property set to global object as specified in output.globalObject
  type: "window", // property set to window object
  type: "self", // property set to self object
  type: "jsonp", // jsonp wrapper
  type: "module", // EcmaScript Module (not implemented yet)
```

```

</libraryType>
// the type of the exported library
<libraryName "/configuration/output/#outputlibraryname">
  <default>
    name: "MyLibrary", // string | string[]
  </default>
  name: ["MyCompany", "MyLibrary"], // Some types support creating nested objects
  name: undefined, // Some types support unnamed library (default)
</libraryName>
// the name of the exported library

<advancedLibrary "#">
  <default>
    /* Advanced output.library configuration (click to show) */
  </default>
  export: "default", // string | string[]
  export: undefined, // expose the whole namespace object resp. module.exports value
  export: "named", // expose named exports
  export: ["named", "property"], // expose nested properties
  // the export of the entry module that should be exposed
  auxiliaryComment: "comment",
  auxiliaryComment: { amd: "comment", commonjs: "comment", commonjs2: "comment", root
  // Add a comment in the UMD wrapper
  umdNamedDefine: false,
  // Use a named define() for the AMD part of the UMD wrapper
</advancedLibrary>
},
uniqueName: "my-application", // (defaults to package.json "name")
// unique name for this build to avoid conflicts with other builds in the same HTML
name: "my-config",
// name of the configuration, shown in output
<advancedOutput "#">
  <default>
    /* 高级输出配置（点击显示） */
  </default>
  chunkFilename: "[name].js",
  chunkFilename: "[id].js",
  chunkFilename: "[contenthash].js", // 长效缓存
  // the filename template for additional chunks
  assetModuleFilename: "[hash][ext][query]", // string
  // the filename template for asset modules
  webassemblyModuleFilename: "[hash].module.wasm", // string
  // the filename template for wasm modules
  sourceMapFilename: "[file].map", // string
  sourceMapFilename: "sourcemaps/[file].map", // string
  // source map location 的文件名模板
  devtoolModuleFilenameTemplate: "webpack:///[resource-path]", // string
  // devtool 模块的文件名模板
  devtoolFallbackModuleFilenameTemplate: "webpack:///[resource-path]?[hash]", // string
  // devtool 模块的文件名模板（用于冲突）
  crossOriginLoading: "use-credentials", // enum
  crossOriginLoading: "anonymous",
  crossOriginLoading: false,
  // specifies how cross origin request are issued by the runtime
  trustedTypes: false, // boolean (default) | string | object

```

```

// Trusted Types support
importFunctionName: "import", // string (default)
// expression that is called when using import()
// can be replaced to use polyfills
importMetaName: "import.meta", // string (default)
// expression that is used when using import.meta
// can be replaced to use polyfills
</advancedOutput>
<expertOutput "#">
  <default>
    /* Expert output configuration 1 (on own risk) */
  </default>
  pathinfo: true, // boolean
  // include useful path info about modules, exports, requests, etc. into the generated
  charset: true, // string
  // add charset attribute to injected script tags
  chunkLoadTimeout: 120000, // number (default)
  // timeout for loading chunks
  compareBeforeEmit: true, // boolean (default)
  // compare the generated asset with the asset on disk before writing to disk
  strictModuleExceptionHandling: true, // boolean
  // handle errors in module evaluation correctly, but for a performance cost
  devtoolNamespace: "MyLibrary", // string
  // prefix in the source names for devtools
  // defaults to output.uniqueName
  environment: {
    // Properties about the environment
    arrowFunction: true,
    bigIntLiteral: true,
    const: true,
    destructuring: true,
    dynamicImport: true,
    forOf: true,
    module: true
  },
  globalObject: "self", // string (default),
  // expression that points to the global object
  iife: true, // boolean (default)
  // wrap the bundle in a IIFE for isolation
  module: false, // boolean (default)
  // generate a module type javascript file instead of a classic script
  scriptType: "module"
  // adds a type attribute to injected script tags
</expertOutput>
<expertOutputB "#">
  <default>
    /* Expert output configuration 2 (on own risk) */
  </default>
  chunkLoading: "jsonp" // "jsonp" | "import-scripts" | "require" | "async-node" | false
  // method of loading chunks
  chunkLoadingGlobal: "myWebpackJsonp", // string
  // name of the global variable used to load chunks
  enabledChunkLoadingTypes: ["jsonp"], // string[]
  // the chunk loading methods that are available
  enabledLibraryTypes: ["var"], // string[]

```

```

// the library types that are available
enabledWasmLoadingTypes: ["var"], // string[]
// the wasm loading methods that are available
chunkFormat: "array-push",
chunkFormat: "commonjs",
chunkFormat: false,
// the format of chunks
hotUpdateMainFilename: "[fullhash].hot-update.json", // string
// filename template for HMR manifest
hotUpdateChunkFilename: "[id].[fullhash].hot-update.js", // string
// filename template for HMR chunks
hotUpdateGlobal: "hmrUpdateFunction", // string
// the name of the global variable used to load hot update chunks
sourcePrefix: "\t", // string
// prefix module sources in bundle for better readability
// but breaks multi-line template strings
hashFunction: "md4", // string (default)
// hash function used in general
hashDigest: "hex", // string (default)
// hash digest type used
hashDigestLength: 20, // number (default)
// length of hashes
hashSalt: "salt", // string | Buffer
// an additional hash salt to fix hash related issues or change the hash in general
</expertOutputB>
},
module: {
  // 模块配置相关
  rules: [
    // 模块规则（配置 loader、解析器等选项）
    {
      // Conditions:
      test: /\.jsx?$/,
      include: [
        path.resolve(__dirname, "app")
      ],
      exclude: [
        path.resolve(__dirname, "app/demo-files")
      ],
      // these are matching conditions, each accepting a regular expression or string
      // test and include have the same behavior, both must be matched
      // exclude must not be matched (takes preference over test and include)
      // Best practices:
      // - Use RegExp only in test and for filename matching
      // - Use arrays of absolute paths in include and exclude to match the full path
      // - Try to avoid exclude and prefer include
      // Each condition can also receive an object with "and", "or" or "not" properties
      // which are an array of conditions.
      issuer: /\.css$/,
      issuer: path.resolve(__dirname, "app"),
      issuer: { and: [ /\.css$/, path.resolve(__dirname, "app") ] },
      issuer: { or: [ /\.css$/, path.resolve(__dirname, "app") ] },
      issuer: { not: [ /\.css$/ ] },
      issuer: [ /\.css$/, path.resolve(__dirname, "app") ], // like "or"
      // conditions for the issuer (the origin of the import)

```



```

<advancedConditions "#">
  <default>
    /* Advanced conditions (click to show) */
  </default>
  resource: /\.css$/,
  // matches the resource of the module, behaves equal to "test" and "include"
  compiler: /html-webpack-plugin/,
  // matches the name of the child compilation
  dependency: "esm", // import-style dependencies
  dependency: "commonjs", // require-style dependencies
  dependency: "amd", // AMD-style dependency
  dependency: "wasm", // WebAssembly imports section
  dependency: "url", // new URL(), url() and similar
  dependency: "worker", // new Worker() and similar
  dependency: "loader", // this.loadModule in loaders
  // matches the type of dependency
  descriptionData: { type: "module" },
  // matches information from the package.json
  mimetype: "text/javascript",
  // matches the mimetype in DataUri
  realResource: /\.css$/,
  // matches the resource but ignores when resource was been renamed
  resourceFragment: "#blah",
  // matches the fragment part of the resource request
  resourceQuery: "?blah"
  // matches the query part of the resource request
</advancedConditions>

// Actions:
loader: "babel-loader",
// 应该应用的 loader，它相对上下文解析
options: {
  presets: ["es2015"]
},
// options for the loader
use: [
  // apply multiple loaders and options instead
  "htmlhint-loader",
  {
    loader: "html-loader",
    options: {
      // ...
    }
  }
],
<moduleType "#">
  <default>
    type: "javascript/auto",
  </default>
  type: "javascript/auto", // JS with all features
  type: "javascript/esm", // JS enforced to strict ESM
  type: "javascript/dynamic", // JS enforced to non-ESM
  type: "json", // JSON data
  type: "webassembly/async", // WebAssembly as async module
  type: "webassembly/sync", // WebAssembly as sync module

```

```

    </moduleType>
    // specifies the module type
    <advancedActions "#">
      <default>
        /* Advanced actions (click to show) */
      </default>
      enforce: "pre",
      enforce: "post",
      // flags to apply these rules, even if they are overridden
      generator: { /* ... */ },
      // Options for the generator (depends on module type)
      parser: { /* ... */ },
      // Options for the parser (depends on module type)
      resolve: { /* ... */ },
      // Resolve options (same as "resolve" in configuration)
      sideEffects: false, // boolean
      // Overrides "sideEffects" from package.json
    </advancedActions>
  },
  {
    oneOf: [
      // ... (rules)
    ]
    // only use one of these nested rules
  },
  {
    // ... (conditions)
    rules: [
      // ... (rules)
    ]
    // use all of these nested rules (combine with conditions to be useful)
  },
],
<advancedModule "#">
  <default>
    /* 高级模块配置（点击展示） */
  </default>
  noParse: [
    /special-library\.js$/,
  ],
  // 不解析这里的模块
  unknownContextRequest: ".",
  unknownContextRecursive: true,
  unknownContextRegExp: /^\.\/.*$/,
  unknownContextCritical: true,
  exprContextRequest: ".",
  exprContextRegExp: /^\.\/.*$/,
  exprContextRecursive: true,
  exprContextCritical: true,
  wrappedContextRegExp: /\.*/,
  wrappedContextRecursive: true,
  wrappedContextCritical: false,
  // specifies default behavior for dynamic requests
</advancedModule>
},

```



```

resolve: {
  // options for resolving module requests
  // (does not apply to resolving of loaders)
  <modules "/configuration/resolve/#resolvemodules">
    <default>
      modules: [
        "node_modules",
        path.resolve(__dirname, "app")
      ],
    </default>
    modules: [ "node_modules" ],
    // A filename (non-absolute) means this folder is looked up
    // in all parent directories
    modules: [ path.resolve(__dirname, "app") ],
    // An absolute path means exactly this folder
  </modules>
  // directories where to look for modules (in order)
  extensions: [".js", ".json", ".jsx", ".css"],
  // 使用的扩展名
  alias: {
    // a list of module name aliases
    // aliases are imported relative to the current context
    "module": "new-module",
    // 别名: "module" -> "new-module" 和 "module/path/file" -> "new-module/path/file"
    "only-module$": "new-module",
    // 别名 "only-module" -> "new-module", 但不匹配 "only-module/path/file" -> "new-module/
    "module": path.resolve(__dirname, "app/third/module.js"),
    // alias "module" -> "./app/third/module.js" and "module/file" results in error
    "module": path.resolve(__dirname, "app/third"),
    // alias "module" -> "./app/third" and "module/file" -> "./app/third/file"
    [path.resolve(__dirname, "app/module.js")]: path.resolve(__dirname, "app/alternative-
    // alias "./app/module.js" -> "./app/alternative-module.js"
  },
  <alias "/configuration/resolve/#resolvealias">
    <default>
      /* 可供选择的别名语法（点击展示） */
    </default>
    alias: [
      {
        name: "module",
        // 旧的 request
        alias: "new-module",
        // 新的 request
        onlyModule: true
        // 如果为 true, 只有 "module" 是别名
        // 如果为 false, "module/inner/path" 也是别名
      }
    ],
  </alias>
  <advancedResolve "/configuration/resolve/">
    <default>
      /* 高级解析选项（点击展示） */
    </default>
    conditionNames: ["myCompanyCondition", "..."],
    // conditions used for the "exports" and "imports" field in description file

```

```

    roots: [path.resolve(__dirname, "app/root")],
    // locations where to resolve server-relative requests (starting with "/")
    // This behavior is only applied when the request doesn't resolve as absolute path
    fallback: { "events": path.resolve(__dirname, "events.js") },
    // Similar to alias, but only applied when the normal resolving fails
    mainFields: ["main"],
    // properties that are read from description file
    // when a folder is requested
    restrictions: [ /\.js$/, path.resolve(__dirname, "app") ],
    // To successful resolve the result must match these criteria
    cache: true, // boolean
    // enable safe caching of resolving
    // this is safe as it tracks and validates all resolving dependencies
    unsafeCache: true,
    unsafeCache: {},
    // enables unsafe caching for resolved requests
    // this is unsafe as there is no validation
    // but performance improvement is really big
    plugins: [
      // ...
    ],
    // additional plugins applied to the resolver
  </advancedResolve>
  <expertResolve "/configuration/resolve/">
    <default>
      /* Expert resolve configuration (click to show) */
    </default>
    symlinks: true, // (default)
    // follow symlinks to new location
    descriptionFiles: ["package.json"], // (default)
    // files that are read for package description
    aliasFields: ["browser"],
    // properties that are read from description file
    // to alias requests in this package
    exportsFields: ["exports"], // (default)
    // fields in description file that are used for external module request
    importsFields: ["imports"], // (default)
    // fields in description file that are used for internal request
    mainFiles: ["index"],
    // files that are used when resolving in a directory and no mainField applies
    fullySpecified: true, // boolean
    // Input request is already full specified (it includes filename and extension)
    // Module requests are still resolved as usual
    preferRelative: true, // boolean
    // Try to resolve module requests also a relative request
    enforceExtension: false,
    // if true request must not include an extension
    // if false request may already include an extension
    cachePredicate: ({ path, request }) => true,
    // predicate function which selects requests for caching
    cacheWithContext: false, // (default)
    // include context information in cache key
    // This must be set to true when custom plugins resolve depending on
    // those information
    useSyncFilesystemCalls: false, // (default)
  </expertResolve>

```

```

    // use sync fs calls instead of async fs calls
    byDependency: { commonjs: { extensions: [".js", "..."] } },
    // change resolving depending on issuer dependency
  </expertResolve>
},
performance: {
  <hints "/configuration/performance/#performancehints">
    <default>
      hints: "warning", // 枚举
    </default>
    hints: "error", // 性能提示中抛出错误
    hints: false, // 关闭性能提示
  </hints>
  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('.js');
  }
},
<devtool "/configuration/devtool">
  <default>
    devtool: "source-map", // enum
  </default>
  devtool: "inline-source-map", // inlines SourceMap into original file
  devtool: "hidden-source-map", // SourceMap without reference in original file
  devtool: "eval-source-map", // inlines SourceMap per module
  devtool: "cheap-source-map", // cheap-variant of SourceMap without module mappings
  devtool: "cheap-module-source-map", // cheap-variant of SourceMap with module mappings
  devtool: "eval-cheap-module-source-map", // like above but per module
  devtool: "eval", // no SourceMap, but named modules. Fastest at the expense of detail.
  devtool: false, // no SourceMap
</devtool>
// 通过为浏览器调试工具提供极其详细的源映射的元信息来增强调试能力，
// 但会牺牲构建速度。
context: __dirname, // string（绝对路径！）
// webpack 的主目录
// entry 和 module.rules.loader 选项
// 都相对于此目录解析
<target "/configuration/target">
  <default>
    target: "web", // 枚举
  </default>
  target: "browserslist", // use browserslist
  target: "browserslist:modern", // use browserslist "modern" preset
  target: "browserslist:Chrome >= 43", // use browserslist query
  target: `browserslist:${path.resolve(__dirname, "browserslist.json")}`,
  target: `browserslist:${path.resolve(__dirname, "browserslist.json")}:modern`,
  target: "webworker", // WebWorker
  target: "node", // Node.js via require
  target: "node10.13", // Node.js via require
  target: "async-node10.13", // Node.js via fs and vm
  target: "nwjs0.43", // nw.js
  target: "electron11.0-main", // electron, main process
  target: "electron11-renderer", // electron, renderer process

```

```

    target: "electron-preload", // electron, preload script
    target: ["web", "es5"], // combining targets
    target: ["web", "es2020"],
    target: false, // custom target, via plugin
  </target>
  // the environment in which the bundle should run
  // changes chunk loading behavior, available external modules
  // and generated code style
  <externals "/configuration/externals">
    <default>
      externals: ["react", /^@angular/],
    </default>
    externals: "react", // string (exact match)
    externals: /^[a-z\-\-]+(\$|\/)/, // Regex
    externals: { // object
      angular: "this angular", // this["angular"]
      react: { // UMD
        commonjs: "react",
        commonjs2: "react",
        amd: "react",
        root: "React"
      }
    },
    externals: ({ context, request }, callback) => { /* ... */ callback(null, "commonjs " +
  </externals>
  // Don't follow/bundle these modules, but request them at runtime from the environment
  <externalsType "/configuration/externals">
    <default>
      externalsType: "var", // (defaults to output.library.type)
    </default>
    externalsType: "this", // this["EXTERNAL"]
    externalsType: "window", // window["EXTERNAL"]
    externalsType: "self", // self["EXTERNAL"]
    externalsType: "global", // property from output.globalObject
    externalsType: "commonjs", // require("EXTERNAL")
    externalsType: "amd", // define(["EXTERNAL"], ...), only with AMD library
    externalsType: "umd", // only with UMD library
    externalsType: "system", // only with System.js library
    externalsType: "jsonp", // only with jsonp library
    externalsType: "import", // import("EXTERNAL")
    externalsType: "module", // import X from "EXTERNAL"
    externalsType: "var", // EXTERNAL (name is an expression)
    externalsType: "promise", // await EXTERNAL (name is an expression giving a Promise)
  </externalsType>
  // Type of externals, when not specified inline in externals
  <externalsPresets "#">
    <default>
      externalsPresets: { /* ... */ },
    </default>
    externalsPresets: {
      electron: true,
      electronMain: true,
      electronPreload: true,
      electronRenderer: true,
      node: true,

```

```

    nwjs: true,
    web: true,
    webAsync: true,
  }
</externalsPresets>
// presets of externals
<ignoreWarnings "#">
  <default>
    ignoreWarnings: [/warning/],
  </default>
  ignoreWarnings: [
    /warning/,
    {
      file: /asset/,
      module: /module/,
      message: /warning/,
    },
    (warning, compilation) => true
  ],
</ignoreWarnings>
<stats "/configuration/stats">
  <default>
    stats: "errors-only",
  </default>
  stats: "verbose", // nearly all information
  stats: "detailed", // much information
  stats: "minimal", // summarized information
  stats: "errors-warnings", // only errors and warnings
  stats: "errors-only", // only errors
  stats: "summary", // only one line summary
  stats: "none", // none at all
</stats>
stats: {
  // lets you precisely control what bundle information gets displayed
  <preset "/configuration/stats/#stats-presets">
    <default>
      preset: "errors-only",
    </default>
    preset: "verbose", // nearly all information
    preset: "detailed", // much information
    preset: "minimal", // summarized information
    preset: "errors-warnings", // only errors and warnings
    preset: "errors-only", // only errors
    preset: "summary", // only one line summary
    preset: "none", // none at all
  </preset>
  // A stats preset

  <advancedGlobal "/configuration/stats/">
    <default>
      /* Advanced global settings (click to show) */
    </default>
    all: false,
    // switch all flags on or off
    colors: true,

```

```
// switch colors on and off
context: __dirname,
// all paths will be relative to this directory
ids: true,
// include module and chunk ids in the output
</advancedGlobal>

env: true,
// include value of --env in the output
outputPath: true,
// include absolute output path in the output
publicPath: true,
// include public path in the output

assets: true,
// show list of assets in output
<advancedAssets "/configuration/stats/">
  <default>
    /* Advanced assets settings (click to show) */
  </default>
  assetsSort: "size",
  // sorting of assets
  assetsSpace: 50,
  // number of asset lines to display
  cachedAssets: false,
  // show assets that are caching in output
  excludeAssets: /\.png$/,
  // hide some assets
  groupAssetsByPath: true,
  // group assets by their path in the output directory
  groupAssetsByExtension: true,
  // group assets by their extension
  groupAssetsByEmitStatus: true,
  // group assets depending if they are cached, emitted or compared
  groupAssetsByChunk: true,
  // group assets by how they relate to chunks
  groupAssetsByInfo: true,
  // group assets by meta information like immutable, development, etc.
  relatedAssets: true,
  // show assets that are related to other assets, like SourceMaps, compressed version,
  performance: true,
  // show performance hints next to assets and modules
</advancedAssets>

entrypoints: true,
// show entrypoints list
chunkGroups: true,
// show named chunk group list
<advancedChunkGroups "/configuration/stats/">
  <default>
    /* Advanced chunk group settings (click to show) */
  </default>
  chunkGroupAuxiliary: true,
  // show auxiliary assets for entrypoints/chunk groups
  chunkGroupChildren
```



```
// show child chunk groups for entrypoints/chunk groups
chunkGroupMaxAssets: 5,
// collapse chunk group assets lists when this limit has been reached
</advancedChunkGroups>

chunks: true,
// show list of chunks in output
<advancedChunks "/configuration/stats/">
  <default>
    /* Advanced chunk group settings (click to show) */
  </default>
  chunksSort: "size",
  // sort chunks list
  chunkModules: true,
  // show modules contained in each chunk
  chunkOrigins: true,
  // show the origin of a chunk (why was this chunk created)
  chunkRelations: true,
  // show relations to other chunks (parents, children, siblings)
  dependentModules: true,
  // show modules that are dependencies of other modules in that chunk
</advancedChunks>

modules: true,
// show list of modules in output
<advancedModules "/configuration/stats/">
  <default>
    /* Advanced module settings (click to show) */
  </default>
  modulesSpace: 50,
  // number of modules lines to display
  nestedModules: true,
  // show nested modules (when concatenated)
  cachedModules: true,
  // show modules that were cached
  orphanModules: true,
  // show modules that are not referenced in optimized graph anymore
  excludeModules: /\.css$/,
  // hide some modules
  reasons: true,
  // show the reasons why modules are included
  source: true,
  // include the Source Code of modules (only in JSON)
</advancedModules>
<expertModules "/configuration/stats/">
  <default>
    /* Expert module settings (click to show) */
  </default>
  modulesSort: "size",
  // sort modules list
  groupModulesByPath: true,
  // group modules by their resource path
  groupModulesByExtension: true
  // group modules by their extension
  groupModulesByAttributes: true
```



```
// group modules by attributes like if the have errors/warnings/assets
// or are optional
groupModulesByCacheStatus: true,
// group modules depending if they are built, code was generated or if
// they are cacheable in general
depth: true,
// show depth in the module graph of modules
moduleAssets: true,
// show assets emitted by modules in module list
runtimeModules: true,
// show runtime modules in the modules list
</expertModules>

<advancedStatsOptimization "/configuration/stats/">
  <default>
    /* Advanced optimization settings (click to show) */
  </default>
  providedExports: true,
  // show exports provided by modules
  usedExports: true,
  // show which exports are used by modules
  optimizationBailout: true,
  // show information why optimizations bailed out for modules
</advancedStatsOptimization>

children: true,
// show stats for child compilations

logging: true,
// show logging in output
loggingDebug: /webpack/,
// show debug type logging for some loggers
loggingTrace: true,
// show stack traces for warnings and errors in logging output

warnings: true,
// show warnings

errors: true,
// show errors
errorDetails: true,
// show details for errors
errorStack: true,
// show internal stack trace for errors
moduleTrace: true,
// show module trace for errors
// (why was causing module referenced)

builtAt: true,
// show timestamp in summary
errorsCount: true,
// show errors count in summary
warningsCount: true,
// show warnings count in summary
timings: true,
```

```

    // show build timing in summary
    version: true,
    // show webpack version in summary
    hash: true,
    // show build hash in summary
  },
  devServer: {
    proxy: { // proxy URLs to backend development server
      '/api': 'http://localhost:3000'
    },
    static: path.join(__dirname, 'public'), // boolean | string | array | object, static file
    compress: true, // enable gzip compression
    historyApiFallback: true, // true for index.html upon 404, object for multiple paths
    hot: true, // hot module replacement. Depends on HotModuleReplacementPlugin
    https: false, // true for self-signed, object for cert authority
    // ...
  },
  experiments: {
    asyncWebAssembly: true,
    // WebAssembly as async module (Proposal)
    syncWebAssembly: true,
    // WebAssembly as sync module (deprecated)
    outputModule: true,
    // Allow to output ESM
    topLevelAwait: true,
    // Allow to use await on module evaluation (Proposal)
  },
  plugins: [
    // ...
  ],
  // list of additional plugins
  optimization: {
    chunkIds: "size",
    // method of generating ids for chunks
    moduleIds: "size",
    // method of generating ids for modules
    mangleExports: "size",
    // rename export names to shorter names
    minimize: true,
    // minimize the output files
    minimizer: [new CssMinimizer(), "..."],
    // minimizers to use for the output files

    <advancedOptimization "<#>">
      <default>
        /* Advanced optimizations (click to show) */
      </default>
    concatenateModules: true,
    // concatenate multiple modules into a single one
    emitOnErrors: true,
    // emit output files even if there are build errors
    flagIncludedChunks: true,
    // avoid downloading a chunk if it's fully contained in
    // an already loaded chunk
    innerGraph: true,
  },

```

```

    // determine references without modules between symbols
    mergeDuplicateChunks: true,
    // merge chunks if they are equal
    nodeEnv: "production",
    // value of process.env.NODE_ENV inside of modules
    portableRecords: true,
    // use relative paths in records
    providedExports: true,
    // determine which exports are exposed by modules
    usedExports: true,
    // determine which exports are used by modules and
    // remove the unused ones
    realContentHash: true,
    // calculate a contenthash for assets based on the content
    removeAvailableModules: true,
    // run extra pass to determine modules that are already in
    // parent chunks and remove them
    removeEmptyChunks: true,
    // remove chunks that are empty
    runtimeChunk: "single",
    // change placement of runtime code
    sideEffects: true,
    // skip modules that are side effect free when using reexports
  </advancedOptimization>

```

```

splitChunks: {
  cacheGroups: {
    "my-name": {
      // define groups of modules with specific
      // caching behavior
      test: /\.sass$/,
      type: "css/mini-extract",

      <cacheGroupAdvancedSelectors "#">
        <default>
          /* Advanced selectors (click to show) */
        </default>
        chunks: "async",
        minChunks: 1,
        enforceSizeThreshold: 100000,
        minSize: 0,
        minRemainingSize: 0,
        usedExports: true,
        maxAsyncRequests: 30,
        maxInitialRequests: 30,
      </cacheGroupAdvancedSelectors>

      <cacheGroupAdvancedEffects "#">
        <default>
          /* Advanced effects (click to show) */
        </default>
        maxAsyncSize: 200000,
        maxInitialSize: 100000,
        maxSize: 200000,
        filename: "my-name-[contenthash].js",

```

```
      idHint: "my-name",
      name: false,
      hidePathInfo: true,
      automaticNameDelimiter: "-",
    </cacheGroupAdvancedEffects>
  }
},

<fallbackCacheGroup "#">
  <default>
    fallbackCacheGroup: { /* Advanced (click to show) */ }
  </default>
  fallbackCacheGroup: {
    automaticNameDelimiter: "-"
    minSize: 20000,
    maxAsyncSize: 200000,
    maxInitialSize: 100000,
    maxSize: 200000,
  },
</fallbackCacheGroup>

<advancedSelectors "#">
  <default>
    /* Advanced selectors (click to show) */
  </default>
  chunks: "all",
  // select which chunks should be optimized
  usedExports: true,
  // treat modules as equal only when used exports are equal
  minChunks: 1,
  // minimum number of chunks a module must be in
  enforceSizeThreshold: 100000,
  // ignore when following criteria when size of modules
  // is above this threshold
  minSize: 20000,
  // size of modules must be above this threshold
  minRemainingSize: 20000,
  // when modules are removed from a single chunk
  // the size of the modules that are remaining
  // must be above this threshold
  maxAsyncRequests: 30,
  maxInitialRequests: 30,
  // number of parallel requests for a single on demand loading
  // resp. entrypoint but be above this threshold
</advancedSelectors>

<advancedEffects "#">
  <default>
    /* Advanced effects (click to show) */
  </default>
  maxAsyncSize: 200000,
  maxInitialSize: 100000,
  maxSize: 200000,
  // when size of modules in the new chunk is above this
  // threshold, split it further
```

```

    filename: "[contenthash].js",
    // give the new chunk a different filename
    name: false, // false | string | (module, chunks, key) => string
    // give the new chunk a different name
    // when an existing name is used, chunks are merged
    // non-splitChunks chunks can only be selected, when they are
    // a parent or sibling chunk of all selected modules
    hidePathInfo: true,
    // hide path info when splitting via "maxSize"
    automaticNameDelimiter: "-",
    // use this separator to separate original name from path info
    // when splitting via "maxSize"
  </advancedEffects>

  <expert "#">
    <default>
      /* Expert settings (click to show) */
    </default>
    defaultSizeTypes: ["javascript", "..."]
    // when using numbers for sizes measure these size types
    // minSize: { javascript: 10000 } allows to be more specific
  </expert>
}
},
<advanced "#">
  <default>
    /* 高级配置（点击展示） */
  </default>
  loader: { /* ... */ },
  // add custom API or properties to loader context
  resolveLoader: { /* same as resolve */ }
  // separate resolve options for loaders
  node: {
    // Polyfills and mocks to run Node.js-
    // environment code in non-Node environments.
    global: true, // boolean
    // replace "global" with the output.globalObject
    __filename: "mock", // boolean | "mock" | "eval-only"
    __dirname: "mock", // boolean | "mock" | "eval-only"
    // true: includes the real path
    // "mock": includes a fake path
    // "eval-only": only defines it at compile-time
    // false: disables all handling
  },
  recordsPath: path.resolve(__dirname, "build/records.json"),
  recordsInputPath: path.resolve(__dirname, "build/records.json"),
  recordsOutputPath: path.resolve(__dirname, "build/records.json"),
  // store ids into a file to make the build even more deterministic
</advanced>
<advancedCaching "#">
  <default>
    /* Advanced caching configuration (click to show) */
  </default>
  cache: false, // boolean
  // disable/enable caching

```

```

    snapshot: {
      managedPaths: [ path.resolve(__dirname, "node_modules") ],
      // paths that are snapshot using only package.json name and version
      immutablePaths: [ path.resolve(__dirname, ".yarn/cache") ],
      // paths that doesn't need to be snapshot as they are immutable
      module: { timestamp: true, hash: true },
      resolve: { timestamp: true, hash: false },
      resolveBuildDependencies: { timestamp: true, hash: false },
      buildDependencies: { timestamp: true, hash: true },
      // snapshot method for different operations
    },
    watch: true, // boolean
    // 启用 watch 模式
    watchOptions: {
      aggregateTimeout: 1000, // 以毫秒为单位
      // 将多个修改聚合到单个 rebuild
      poll: true,
      poll: 500, // 以毫秒为间隔单位
      // 在 watch 模式中启用轮询
      // 必须用在不通知更改的文件系统中
      // 即 nfs shares
    },
  },
</advancedCaching>
<advancedBuild "#">
  <default>
    /* Advanced build configuration (click to show) */
  </default>
  infrastructureLogging: {
    level: "none",
    level: "error",
    level: "warn",
    level: "info", // (default)
    level: "log",
    level: "verbose",
    debug: true,
    debug: /webpack/,
    debug: [ "MyPlugin", /webpack/ ]
  },
  parallelism: 1, // number
  // limit the number of parallel processed modules
  profile: true, // boolean
  // capture timing information
  bail: true, //boolean
  // fail out on the first error instead of tolerating it.
  dependencies: ["name"],
  // When using an array of configs this can be used to reference other
  // configs and let this config run after the other config at initial build
</advancedBuild>
}

```

## Warning

在应用 [插件默认值](#) 之后，webpack 将应用配置默认值。

为了快速生成符合项目要求的 webpack 配置文件，在使用 [webpack-cli](#) 的 `init` 命令时，会在创建配置文件之前会询问你几个问题。

```
npx webpack-cli init
```

如果尚未在项目或全局安装 `@webpack-cli/init`，npx 可能会提示你安装。根据你在配置生成过程中的选择，你也可能会安装额外的 package 到你的项目中。

```
npx webpack-cli init
```

```
! INFO For more information and a detailed description of each question, have a look at htt
! INFO Alternatively, run `webpack(-cli) --help` for usage info.
```

```
? Will your application have multiple bundles? No
? Which module will be the first to enter the application? [default: ./src/index]
? Which folder will your generated bundles be in? [default: dist]:
? Will you be using ES2015? Yes
? Will you use one of the below CSS solutions? No
```

```
+ babel-plugin-syntax-dynamic-import@6.18.0
+ uglifyjs-webpack-plugin@2.0.1
+ webpack-cli@3.2.3
+ @babel/core@7.2.2
+ babel-loader@8.0.4
+ @babel/preset-env@7.1.0
+ webpack@4.29.3
added 124 packages from 39 contributors, updated 4 packages and audited 25221 packages in 7
found 0 vulnerabilities
```

```
Congratulations! Your new webpack configuration file has been created!
```

## Configuration Languages

Webpack 支持使用多种编程语言和数据描述格式来编写配置文件。在 [node-interpret](#) 中你可以找到当前所支持的文件类型列表，通过 [node-interpret](#)，webpack 能够处理这些类型的配置文件。

## TypeScript

要使用 [Typescript](#) 来编写 webpack 配置，你需要先安装必要的依赖，比如 Typescript 以及其相应的类型声明，类型声明可以从 [DefinitelyTyped](#) 项目中获取，依赖安装如下所示：

```
npm install --save-dev typescript ts-node @types/node @types/webpack
# 如果使用 webpack-dev-server，还需要安装以下依赖
```



```
npm install --save-dev @types/webpack-dev-server
```

完成依赖安装后便可以开始编写配置文件，示例如下：

### webpack.config.ts

```
import * as path from 'path';
import * as webpack from 'webpack';
// in case you run into any typescript error when configuring `devServer`
import 'webpack-dev-server';

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

export default config;
```

该示例需要 typescript 版本在 2.7 及以上，并在 tsconfig.json 文件的 compilerOptions 中添加 esModuleInterop 和 allowSyntheticDefaultImports 两个配置项。

值得注意的是你需要确保 tsconfig.json 的 compilerOptions 中 module 选项的值为 commonjs ,否则 webpack 的运行会失败报错，因为 ts-node 不支持 commonjs 以外的其他模块规范。

你可以通过三个途径来完成 module 的设置：

- 直接修改 tsconfig.json 文件
- 修改 tsconfig.json 并且添加 ts-node 的设置。
- 使用 tsconfig-paths

**第一种方法**就是打开你的 tsconfig.json 文件，找到 compilerOptions 的配置，然后设置 target 和 module 的选项分别为 "ES5" 和 "CommonJs" (在 target 设置为 es5 时你也可以不显示编写 module 配置)。

**第二种方法** 就是添加 ts-node 设置：

你可以为 tsc 保持 "module": "ESNext" 配置，如果你是用 webpack 或者其他构建工具的话，为 ts-node 设置一个重载 (override) 。[ts-node 配置项](#)

```
{
  "compilerOptions": {
    "module": "ESNext",
  },
}
```

```
"ts-node": {
  "compilerOptions": {
    "module": "CommonJS"
  }
}
```

**第二种方法**需要先安装 `tsconfig-paths` 这个 npm 包，如下所示：

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

安装后你可以为 webpack 配置创建一个单独的 TypeScript 配置文件，示例如下：

#### tsconfig-for-webpack-config.json

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

### Tip

ts-node 可以根据 `tsconfig-paths` 提供的环境变量 `process.env.TS_NODE_PROJECT` 来找到 `tsconfig.json` 文件路径。

`process.env.TS_NODE_PROJECT` 变量的设置如下所示：

#### package.json

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

之所以要添加 `cross-env`，是因为我们在直接使用 `TS_NODE_PROJECT` 时遇到过 `"TS_NODE_PROJECT" unrecognized command` 报错的反馈，添加 `cross-env` 之后该问题也似乎得到了解决，你可以查看[这个 issue](#)获取到关于该问题的更多信息。

## CoffeeScript

与 Typescript 类似，在使用 CoffeeScript 前需要先安装其依赖，如下所示：

```
npm install --save-dev coffeescript
```

完成安装之后便可以开始编写 webpack 配置，示例如下：

### 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

下述的示例中使用了 JSX（用于 React 的 JavaScript 标记语言）和 babel 来创建格式为 json 的 webpack 配置文件。

*感谢 [Jason Miller](#) 提供示例代码*

首先，需要安装一些必要依赖，如下所示：

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

### .babelrc

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

### webpack.config.babel.js

```
import jsxobj from 'jsxobj';

// 插件引入示例
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 `modules` 的值设置为 `false`，则必须维护两份 `.babelrc` 的文件，或者你也可以将上述示例中的 `import jsxobj from 'jsxobj'`；替换为 `const jsxobj = require('jsxobj')`；并将新的 `export` 语法替换为 `module.exports`，因为尽管 Node 目前已经支持了 ES6 的许多新特性，但是仍然没有支持 ES6 的模块语法。

## Configuration Types

除了导出单个配置外，还有一些能满足更多需求的使用方式。

### 导出函数

你可能会遇到需要区分开发环境和生产环境的情况。有很多种方式可以做到这一点。其中一种选择是由 webpack 配置导出一个函数而非对象，这个函数包含两个参数：

- 参数一是环境（environment）。请查阅 [environment 选项文档](#) 了解更多。
- 参数二是传递给 webpack 的配置项，其中包含 `output-path` 和 `mode` 等。

```
-module.exports = {
+module.exports = function(env, argv) {
+  return {
+    mode: env.production ? 'production' : 'development',
+    devtool: env.production ? 'source-map' : 'eval',
```

```
    plugins: [  
      new TerserPlugin({  
        terserOptions: {  
+         compress: argv.mode === 'production' // only if '--mode production' was passed  
        },  
      })  
    ],  
+  };  
};
```

## 导出 Promise

当需要异步加载配置变量时，webpack 将执行函数并导出一个配置文件，同时返回一个 Promise。

### Tip

支持使用 `Promise.all([/* Your promises */])` 导出多个 Promise。

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

### Warning

只有通过 webpack 命令行工具返回的 `Promise` 才生效。 `webpack()` 只接受对象。

## 导出多种配置

除了导出单个配置对象/函数，你可能也会需要导出多种配置（webpack 3.1.0 起支持）。当运行 webpack 时，所有配置项都会构建。比如，对于多 [targets](#)（如 AMD 和 CommonJS）[构建 library](#) 时会非常有用。

```
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',
},
];
```

## Tip

如果你只传了一个 `--config-name` 名字标识, webpack 将只会构建指定的配置项。

## dependencies

以防你的某个配置依赖于另一个配置的输出, 你可以使用一个 `dependencies` 列表指定一个依赖列表。

### webpack.config.js

```
module.exports = [
  {
    name: 'client',
    target: 'web',
    // ...
  },
  {
    name: 'server',
    target: 'node',
    dependencies: ['client'],
  },
];
```

## parallelism

如果你导出了多个配置, 你可以在配置中使用 `parallelism` 选项来指定编译的最大并发数。

- 类型: `number`

- 支持版本: 5.22.0+

## webpack.config.js

```
module.exports = [  
  {  
    //config-1  
  },  
  {  
    //config-2  
  },  
];  
module.exports.parallelism = 1;
```

# 入口和上下文

入口对象是用于 webpack 查找开始构建 bundle 的地方。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点(entry point)和 加载器(loader)。

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

默认使用 Node.js 进程的当前工作目录，但是推荐在配置中传入一个值。这使得你的配置独立于 CWD(current working directory, 当前工作目录)。

## entry

```
string [string] object = { <key> string | [string] | object = { import  
string | [string], dependOn string | [string], filename string, layer string  
}} (function() => string | [string] | object = { <key> string | [string] } |  
object = { import string | [string], dependOn string | [string], filename  
string })
```



开始应用程序打包过程的一个或多个起点。如果传入数组，则会处理所有条目。

动态加载的模块 **不是** 入口起点。

一个需要考虑的规则：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',
  },
};
```

## Naming

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，则每个属性的键(key)会是 chunk 的名称，该属性的值描述了 chunk 的入口点。

## Entry descriptor

如果传入一个对象，对象的属性的值可以是一个字符串、字符串数组或者一个描述符(descriptor):

```
module.exports = {
  //...
  entry: {
    home: './home.js',
    shared: ['react', 'react-dom', 'redux', 'react-redux'],
    catalog: {
      import: './catalog.js',
      filename: 'pages/catalog.js',
      dependOn: 'shared',
      chunkLoading: false, // Disable chunks that are loaded on demand and put everything i
    },
    personal: {
      import: './personal.js',
      filename: 'pages/personal.js',
      dependOn: 'shared',
      chunkLoading: 'jsonp',
      asyncChunks: true, // Create async chunks that are loaded on demand.
      layer: 'name of layer', // set the layer for an entry point
    },
  },
};
```

描述符语法可以用来传入额外的选项给入口。

## Output filename

默认情况下，入口 chunk 的输出文件名是从 `output.filename` 中提取出来的，但你可以为特定的入口指定一个自定义的输出文件名。

```
module.exports = {
  //...
  entry: {
    app: './app.js',
    home: { import: './contact.js', filename: 'pages/[name][ext]' },
    about: { import: './about.js', filename: 'pages/[name][ext]' },
  },
};
```

描述符语法在这里被用来将 `filename` 选项传递给指定的入口点。

## Dependencies

默认情况下，每个入口 chunk 保存了全部其用的模块(modules)。使用 `dependOn` 选项你可以与另一个入口 chunk 共享模块：

```
module.exports = {
  //...
  entry: {
    app: { import: './app.js', dependOn: 'react-vendors' },
    'react-vendors': ['react', 'react-dom', 'prop-types'],
  },
};
```

app 这个 chunk 就不会包含 `react-vendors` 拥有的模块了。

`dependOn` 选项的也可以为字符串数组：

```
module.exports = {
  //...
  entry: {
    moment: { import: 'moment-mini', runtime: 'runtime' },
    reactvendors: { import: ['react', 'react-dom'], runtime: 'runtime' },
    testapp: {
      import: './wwwroot/component/TestApp.tsx',
      dependOn: ['reactvendors', 'moment'],
    },
  },
};
```

此外，你还可以使用数组为每个入口指定多个文件：

```
module.exports = {
  //...
  entry: {
    app: { import: ['./app.js', './app2.js'], dependOn: 'react-vendors' },
    'react-vendors': ['react', 'react-dom', 'prop-types'],
  },
};
```

## Dynamic entry

如果传入一个函数，那么它将会在每次 `make` 事件中被调用。

要注意的是，`make` 事件在 webpack 启动和每当 [监听文件变化](#) 时都会触发。

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

或者

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

例如，你可以使用动态入口来从外部来源（远程服务器，文件系统内容或者数据库）获取真正的入口：

**webpack.config.js**

```
module.exports = {
  entry() {
    return fetchPathsFromSomeExternalSource(); // 返回一个会被用像 ['src/main-layout.js', 'src/...']
  },
};
```

当和 `output.library` 选项结合：如果传入的是一个数组，只有数组的最后一个条目会被导出。

## 模式(Mode)

提供 `mode` 配置选项，告知 webpack 使用相应模式的内置优化。

```
string = 'production': 'none' | 'development' | 'production'
```

# 用法

只需在配置对象中提供 `mode` 选项：

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

或者从 [CLI](#) 参数中传递：

```
webpack --mode=development
```

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> 。为模块和 chunk 启用有效的名。

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> 。为模块和 chunk 启用确定性的混淆名称, <code>FlagDependencyUsagePlugin</code> , <code>FlagIncludedChunksPlugin</code> , <code>ModuleConcatenationPlugin</code> , <code>NoEmitOnErrorsPlugin</code> 和 <code>TerserPlugin</code> 。

选项	none
描述	不使用任何默认优化选项

如果没有设置，webpack 会给 `mode` 的默认值设置为 `production` 。

Tip

如果 `mode` 未通过配置或 CLI 赋值，CLI 将使用可能有效的 `NODE_ENV` 值作为 `mode` 。

## Mode: development

```
// webpack.development.config.js
module.exports = {
  mode: 'development',
};
```

## Mode: production

```
// webpack.production.config.js
module.exports = {
  mode: 'production',
};
```

## Mode: none

```
// webpack.custom.config.js
module.exports = {
  mode: 'none',
};
```

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函数，而不是导出对象：

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

module.exports = (env, argv) => {
  if (argv.mode === 'development') {
    config.devtool = 'source-map';
  }

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

  return config;
};
```

## 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.assetModuleFilename

### \$#outputassetModulefilename\$

string = '[hash][ext][query]'

与 `output.filename` 相同，不过应用于 `Asset Modules`。

## output.asyncChunks

boolean = true

创建按需加载的异步 chunk。

**webpack.config.js**

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

## output.auxiliaryComment

### Warning

最好使用 `output.library.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:

### someLibName.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.charset

boolean = true

告诉 webpack 为 HTML 的 `<script>` 标签添加 `charset="utf-8"` 标识。



## Tip

尽管 `<script>` 已弃用了 `charset` 属性，当 webpack 还是默认添加了它，目的是与非现代浏览器兼容。

# output.chunkFilename

```
string = '[id].js' function (pathData, assetInfo) => string
```

此选项决定了非初始 (non-initial) 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，这些文件名需要在运行时根据 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].`)。

## webpack.config.js

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

Usage as a function:

## webpack.config.js

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

# output.chunkFormat

```
false string: 'array-push' | 'commonjs' | 'module' | <any string>
```

chunk 的格式 (formats 默认包含 'array-push' (web/WebWorker)、 'commonjs' (node.js)、 'module' (ESM), 还有其他情况可由插件添加) 。

#### webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    //...  
    chunkFormat: 'commonjs',  
  },  
};
```

## output.chunkLoadTimeout

### \$#outputchunkLoadtimeout\$

```
number = 120000
```

chunk 请求到期之前的毫秒数, 默认为 120000。从 webpack 2.6.0 开始支持此选项。

#### webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    //...  
    chunkLoadTimeout: 30000,  
  },  
};
```

## output.chunkLoadingGlobal

```
string = 'webpackChunkwebpack'
```

webpack 用于加载 chunk 的全局变量。

#### webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    //...  
    chunkLoadingGlobal: 'myCustomFunc',  
  },  
};
```

```
},  
};
```

## output.chunkLoading

false string: 'jsonp' | 'import-scripts' | 'require' | 'async-node' | 'import' | <any string>

加载 chunk 的方法（默认值有 'jsonp' (web)、'import' (ESM)、'importScripts' (WebWorker)、'require' (sync node.js)、'async-node' (async node.js)，还有其他值可由插件添加）。

### webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    //...  
    chunkLoading: 'async-node',  
  },  
};
```

## output.clean

5.20.0+

boolean { dry?: boolean, keep?: RegExp | string | ((filename: string) => boolean) }

```
module.exports = {  
  //...  
  output: {  
    clean: true, // 在生成文件之前清空 output 目录  
  },  
};
```

```
module.exports = {  
  //...  
  output: {  
    clean: {  
      dry: true, // 打印而不是删除应该移除的静态资源  
    },  
  },  
};
```

```
module.exports = {
  //...
  output: {
    clean: {
      keep: /ignored\/dir\/, // 保留 'ignored/dir' 下的静态资源
    },
  },
};

// 或者

module.exports = {
  //...
  output: {
    clean: {
      keep(asset) {
        return asset.includes('ignored/dir');
      },
    },
  },
};
```

你也可以使用钩子函数：

```
webpack.CleanPlugin.getCompilationHooks(compilation).keep.tap(
  'Test',
  (asset) => {
    if (/ignored\/dir\/.test(asset)) return true;
  }
);
```

## output.compareBeforeEmit

boolean = true

告知 webpack 在写入到输出文件系统时检查输出的文件是否已经存在并且拥有相同内容。

### Warning

当在磁盘中已经存在有相同内容的文件时，webpack 将不会写入输出文件。

```
module.exports = {
  //...
  output: {
    compareBeforeEmit: false,
  },
};
```

## output.crossOriginLoading

boolean = false string: 'anonymous' | 'use-credentials'

告诉 webpack 启用 [cross-origin](#) 属性加载 chunk。仅在 [target](#) 设置为 'web' 时生效，通过使用 JSONP 来添加脚本标签，实现按需加载模块。

- 'anonymous' - **不带凭据(credential)** 启用跨域加载
- 'use-credentials' - **携带凭据(credential)** 启用跨域加载

## 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.devtoolModuleFilenameTemplate

string = 'webpack://[namespace]/[resource-path]?[loaders]' function (info)  
=> string

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]?[loaders]',  
  },  
};
```

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

Template	[absolute-resource-path]
----------	--------------------------

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

[string: 'jsonp' | 'import-scripts' | 'require' | 'async-node' | <any string>]

允许入口点使用的 chunk 加载类型列表。将被 webpack 自动填充。只有当使用一个函数作为入口配置项并从那里返回 chunkLoading 配置项时才需要。

**webpack.config.js**

```
module.exports = {
  //...
  output: {
    //...
    enabledChunkLoadingTypes: ['jsonp', 'require'],
  },
};
```

## output.enabledLibraryTypes

[string]

入口点可用的 library 类型列表.

```
module.exports = {
  //...
  output: {
    enabledLibraryTypes: ['module'],
  },
};
```



## output.enabledWasmLoadingTypes

[string]


用于设置入口支持的 wasm 加载类型的列表。

```
module.exports = {
  //...
  output: {
    enabledWasmLoadingTypes: ['fetch'],
  },
};

## `output.environment` $#outputenvironment$
```

告诉 webpack 在生成的运行时代码中可以使用哪个版本的 ES 特性。

```
```javascript
module.exports = {
  output: {
    environment: {
      // The environment supports arrow functions ('() => { ... }').
      arrowFunction: true,
      // The environment supports BigInt as literal (123n).
      bigintLiteral: false,
      // The environment supports const and let for variable declarations.
      const: true,
      // The environment supports destructuring ('{ a, b } = obj').
      destructuring: true,
      // The environment supports an async import() function to import EcmaScript modules.
      dynamicImport: false,
      // The environment supports 'for of' iteration ('for (const x of array) { ... }').
      forOf: true,
      // The environment supports ECMAScript Module syntax to import ECMAScript modules (in
      module: false,
      // The environment supports optional chaining ('obj?.a' or 'obj?.()').
      optionalChaining: true,
      // The environment supports template literals.
      templateLiteral: true,
    },
  },
};
```



## output.filename

string function (pathData, assetInfo) => string

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 一个唯一的名称.....

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',  
  },  
};
```

使用由生成的内容产生的 hash:

#### webpack.config.js

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

结合多个替换组合使用：

webpack.config.js

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

使用函数返回 filename：

webpack.config.js

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

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。它只影响最初加载的输出文件。对于按需加载的 chunk 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。

## Template strings

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

可在编译层面进行替换的内容：

模板	[fullhash]
描述	compilation 完整的 hash 值

模板	[hash]
描述	同上，但已弃用

可在 chunk 层面进行替换的内容：

模板	[id]
描述	此 chunk 的 ID

模板	[name]
描述	如果设置，则为此 chunk 的名称，否则使用 chunk 的 ID

模板	[chunkhash]
描述	此 chunk 的 hash 值，包含该 chunk 的所有元素

模板	[contenthash]
描述	此 chunk 的 hash 值，只包括该内容类型的元素（受 optimization.realContentHash 影响）

可在模块层面替换的内容：

模板	[id]
描述	模块的 ID

模板	[moduleid]
描述	同上，但已弃用

模板	[hash]
描述	模块的 Hash 值

模板	[modulehash]
描述	同上，但已弃用

模板	[contenthash]
描述	模块内容的 Hash 值

可在文件层面替换的内容：

模板	[file]
描述	filename 和路径，不含 query 或 fragment

模板	[query]
描述	带前缀 ? 的 query

模板	[fragment]
描述	带前缀 # 的 fragment

模板	[base]
描述	只有 filename（包含扩展名），不含 path

模板	[filebase]
描述	同上，但已弃用

模板	[path]
描述	只有 path，不含 filename

模板	[name]
描述	只有 filename，不含扩展名或 path

模板	[ext]
描述	带前缀 . 的扩展名（对 <code>output.filename</code> 不可用）

可在 URL 层面替换的内容：

模块[url]
描述URL

Tip

[file] 等价于 [path][base] 。 [base] 等价于 [name][ext] 。完整的路径为 [path][name][ext][query][fragment] 或 [path][base][query][fragment] 或 [file][query][fragment] 。

[hash] ， [contenthash] 或者 [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。

当你要在实际文件名中使用占位符时，webpack 会过滤出需要替换的占位符。例如，输出一个文件 [name].js ，你必须通过在括号之间添加反斜杠来转义 [name] 占位符。因此，`[\name\]` 生成 [name] 而不是 name 。

例如：`[\id\]` 生成 [id] 而不是 id 。

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含有替换信息数据的对象。替换也会被应用到返回的字符串中。传递的对象将具有如下类型（取决于上下文的属性）：

```
type PathData = {
  hash: string;
  hashWithLength: (number) => string;
  chunk: Chunk | ChunkPathData;
  module: Module | ModulePathData;
  contentHashType: string;
  contentHash: string;
  contentHashWithLength: (number) => string;
  filename: string;
  url: string;
  runtime: string | SortableSet<string>;
  chunkGraph: ChunkGraph;
};

type ChunkPathData = {
  id: string | number;
  name: string;
  hash: string;
  hashWithLength: (number) => string;
  contentHash: Record<string, string>;
  contentHashWithLength: Record<string, (number) => string>;
};

type ModulePathData = {
  id: string | number;
  hash: string;
  hashWithLength: (number) => string;
};
```

## Tip

在某些上下文中，属性将使用 JavaScript 代码表达式代替原始值。在此情况下，`WithLength` 变量是可用的，应该使用它来代替对原始值的分片操作。

## output.globalObject

```
string = 'window'
```

当输出为 `library` 时，尤其是当 `libraryTarget` 为 `'umd'` 时，此选项将决定使用哪个全局对象来挂载 `library`。为了使 UMD 构建在浏览器和 Node.js 上均可用，应将

`output.globalObject` 选项设置为 `'this'`。对于类似 web 的目标，默认为 `self`。

示例：

### webpack.config.js

```
module.exports = {
  // ...
  output: {
    library: 'myLib',
```

```
libraryTarget: 'umd',
filename: 'myLib.js',
globalObject: 'this',
},
};
```

## output.hashDigest

```
string = 'hex'
```

在生成 hash 时使用的编码方式。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

```
number = 20
```

散列摘要的前缀长度。

### Tip

针对 webpack v5.65.0+，在 `experiments.futureDefaults` 启用时，`hashDigestLength` 配置项会以 16 作为默认值。

## output.hashFunction

```
string = 'md4' function
```

散列算法。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` 和 `digest` 方法。

## Tip

从 webpack v5.54.0+ 起, `hashFunction` 支持将 `xxhash64` 作为更快的算法, 当启用 `experiments.futureDefaults` 时, 此算法将被默认使用。

## output.hashSalt

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

## output.hotUpdateChunkFilename

```
string = '[id].[fullhash].hot-update.js'
```

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。

其中值唯一的占位符是 `[id]` 和 `[fullhash]`, 其默认为:

`webpack.config.js`

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

## Tip

通常, 你不需要修改 `output.hotUpdateChunkFilename`。

## output.hotUpdateGlobal

```
string
```

只在 `target` 设置为 `'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。

欲了解详情, 请查阅 `output.chunkLoadingGlobal`。



## output.hotUpdateMainFilename

```
string = '[runtime].[fullhash].hot-update.json' function
```

自定义热更新的主文件名(main filename)。 `[fullhash]` 和 `[runtime]` 均可作为占位符。

### Tip

通常, 你不需要修改 `output.hotUpdateMainFilename`。

## output.iife

```
boolean = true
```

告诉 webpack 添加 **IIFE** 外层包裹生成的代码。

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

## output.importFunctionName

```
string = 'import'
```

内部 `import()` 函数的名称. 可用于 polyfilling, 例如 通过 [dynamic-import-polyfill](#)。

**webpack.config.js**

```
module.exports = {  
  //...  
  output: {  
    importFunctionName: '__import__',  
  },  
};
```

## output.library

输出一个库，为你的入口做导出。

- 类型: `string` | `string[]` | `object`

一起来看一个简单的示例。

### webpack.config.js

```
module.exports = {  
  // ...  
  entry: './src/index.js',  
  output: {  
    library: 'MyLibrary',  
  },  
};
```

假设你在 `src/index.js` 的入口中导出了如下函数：

```
export function hello(name) {  
  console.log(`hello ${name}`);  
}
```

此时，变量 `MyLibrary` 将与你的入口文件所导出的文件进行绑定，下面是如何使用 webpack 构建的库的实现：

```
<script src="https://example.org/path/to/my-library.js"></script>  
<script>  
  MyLibrary.hello('webpack');  
</script>
```

在上面的例子中，我们为 `entry` 设置了一个入口文件，然而 webpack 可以接受 [多个入口](#)，例如一个 `array` 或者一个 `object`。

1. 如果你将 `entry` 设置为一个 `array`，那么只有数组中的最后一个会被暴露。

```
module.exports = {  
  // ...  
  entry: ['./src/a.js', './src/b.js'], // 只有在 b.js 中导出的内容才会被暴露  
  output: {  
    library: 'MyLibrary',  
  },  
};
```

2. 如果你将 `entry` 设置为一个 `object`，所以入口都可以通过 `library` 的 `array` 语法暴露：

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

```
    b: './src/b.js',
  },
  output: {
    filename: '[name].js',
    library: ['MyLibrary', '[name]'], // name is a placeholder here
  },
};
```

假设 a.js 与 b.js 导出名为 hello 的函数，这就是如何使用这些库的方法：

```
<script src="https://example.org/path/to/a.js"></script>
<script src="https://example.org/path/to/b.js"></script>
<script>
  MyLibrary.a.hello('webpack');
  MyLibrary.b.hello('webpack');
</script>
```

查看 [示例](#) 获取更多内容。

请注意，如果你打算在每个入口点配置 library 配置项的话，以上配置将不能按照预期执行。这里是如何 [在每个入口点下](#) 做的方法：

```
module.exports = {
  // ...
  entry: {
    main: {
      import: './src/index.js',
      library: {
        // `output.library` 下的所有配置项可以在这里使用
        name: 'MyLibrary',
        type: 'umd',
        umdNamedDefine: true,
      },
    },
  },
  another: {
    import: './src/another.js',
    library: {
      name: 'AnotherLibrary',
      type: 'commonjs2',
    },
  },
},
};
```

## output.library.name

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

```
    },
  },
};
```

指定库的名称。

- 类型:

```
string | string[] | {amd?: string, commonjs?: string, root?: string | string[]}
```

## output.library.type

配置将库暴露的方式。

- 类型: string

类型默认包括 'var'、'module'、'assign'、'assign-properties'、'this'、'window'、'self'、'global'、'commonjs'、'common-module'、'amd'、'amd-require'、'umd'、'umd2'、'jsonp' 以及 'system'，除此之外也可以通过插件添加。

对于接下来的实例，我们将会使用 `__entry_return__` 表示被入口点返回的值。

### Expose a Variable

These options assign the return value of the entry point (e.g. whatever the entry point exported) to the name provided by `output.library.name` at whatever scope the bundle was included at.

**type: 'var'**

```
module.exports = {
  // ...
  output: {
    library: {
      name: 'MyLibrary',
      type: 'var',
    },
  },
};
```

让你的库加载之后，**入口起点的返回值** 将会被赋值给一个变量：

```
var MyLibrary = __entry_return__;

// 在加载了 `MyLibrary` 的单独脚本中
MyLibrary.doSomething();
```

**type: 'assign'**

```
module.exports = {
  // ...
  output: {
    library: {
      name: 'MyLibrary',
      type: 'assign',
    },
  },
};
```

这将生成一个隐含的全局变量，它有可能重新分配一个现有的值（请谨慎使用）：

```
MyLibrary = _entry_return_;
```

请注意，如果 `MyLibrary` 没有在你的库之前定义，那么它将会被设置在全局作用域。

**type: 'assign-properties'** 5.16.0+

```
module.exports = {
  // ...
  output: {
    library: {
      name: 'MyLibrary',
      type: 'assign-properties',
    },
  },
};
```

与 `type: 'assign'` 相似但是更安全，因为如果 `MyLibrary` 已经存在的话，它将被重用：

```
// 仅在当其不存在是创建 MyLibrary
MyLibrary = typeof MyLibrary === 'undefined' ? {} : MyLibrary;
// 然后复制返回值到 MyLibrary
// 与 Object.assign 行为类似

// 例如，你像下面这样在你的入口导出一个 `hello` 函数
export function hello(name) {
  console.log(`Hello ${name}`);
}

// 在另外一个已经加载 MyLibrary 的脚本中
// 你可以像这样运行 `hello` 函数
MyLibrary.hello('World');
```

## Expose Via Object Assignment

这些配置项分配入口点的返回值（例如：无论入口点导出的什么内容）到一个名为 `output.library.name` 的对象中。

**type: 'this'**

```
module.exports = {
  // ...
  output: {
    library: {
      name: 'MyLibrary',
      type: 'this',
    },
  },
};
```

**入口起点的返回值** 将会被赋值给 this 对象下的 `output.library.name` 属性。this 的含义取决于你：

```
this['MyLibrary'] = _entry_return_;

// 在一个单独的脚本中
this.MyLibrary.doSomething();
MyLibrary.doSomething(); // 如果 `this` 为 window 对象
```

### type: 'window'

```
module.exports = {
  // ...
  output: {
    library: {
      name: 'MyLibrary',
      type: 'window',
    },
  },
};
```

**入口起点的返回值** 将会被赋值给 window 对象下的 `output.library.name`。

```
window['MyLibrary'] = _entry_return_;

window.MyLibrary.doSomething();
```

### type: 'global'

```
module.exports = {
  // ...
  output: {
    library: {
      name: 'MyLibrary',
      type: 'global',
    },
  },
};
```

**入口起点的返回值** 将会被复制给全局对象下的 `output.library.name`。取决于 `target` 值，全局对象可以分别改变,例如， `self`、`global` 或者 `globalThis`。

```
global['MyLibrary'] = _entry_return_;

global.MyLibrary.doSomething();
```

### type: 'commonjs'

```
module.exports = {
  // ...
  output: {
    library: {
      name: 'MyLibrary',
      type: 'commonjs',
    },
  },
};
```

**入口起点的返回值** 将使用 `output.library.name` 赋值给 `exports` 对象。顾名思义，这是在 CommonJS 环境中使用。

```
exports['MyLibrary'] = _entry_return_;

require('MyLibrary').doSomething();
```

## Warning

注意，不设置 `output.library.name` 将导致入口起点返回的所有属性都被赋值给给定的对象；不检查现有的属性名。

## Module Definition Systems

这些配置项将生成一个带有完整 header 的 bundle，以确保与各种模块系统兼容。

`output.library.name` 配置项在不同的 `output.library.type` 中有不同的含义。

### type: 'module'

```
module.exports = {
  // ...
  experiments: {
    outputModule: true,
  },
  output: {
    library: {
      // do not specify a `name` here
      type: 'module',
    },
  },
};
```

```
  },  
};
```

输出 ES 模块。

然而该特性仍然是实验性的，并且没有完全支持，所以请确保事先启用 `experiments.outputModule`。除此之外，你可以在 [这里](#) 追踪开发进度。

### `type: 'commonjs2'`

```
module.exports = {  
  // ...  
  output: {  
    library: {  
      // note there's no `name` here  
      type: 'commonjs2',  
    },  
  },  
};
```

**入口起点的返回值** 将会被赋值给 `module.exports`。顾名思义，这是在 Node.js (CommonJS) 环境中使用的：

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

如果我们指定 `output.library.name` 为 `type: commonjs2`，你的入口起点的返回值将会被赋值给 `module.exports[output.library.name]`。

## Tip

在考虑 CommonJS 与 CommonJS2 之间的区别？虽然它们很相似，但是它们之间有一些细微的差别，这些差别在 webpack 的上下文中通常是不相关的。（要获取更多详细内容，请[阅读这个 issue](#)。）

### `type: 'amd'`

可以将你的库暴露为 AMD 模块。

AMD module 要求入口 chunk（例如，第一个通过 `<script>` 标签加载的脚本）使用特定的属性来定义，例如 `define` 与 `require`，这通常由 RequireJS 或任何兼容的 loader（如 almond）提供。否则，直接加载产生的 AMD bundle 将导致一个错误，如 `define is not defined`。

按照下面的配置

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



```
output: {
  library: {
    name: 'MyLibrary',
    type: 'amd',
  },
},
};
```

生成的输出将被定义为 "MyLibrary"，例如：

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

该 bundle 可以使用 script 标签引入，并且可以被这样引入：

```
require(['MyLibrary'], function (MyLibrary) {
  // Do something with the library...
});
```

如果没有定义 `output.library.name` 的话，会生成以下内容。

```
define(function () {
  return _entry_return_;
});
```

如果使用一个 `<script>` 标签直接加载。它只能通过 RequireJS 兼容的异步模块 loader 通过文件的实际路径工作，所以在这种情况下，如果 `output.path` 与 `output.filename` 直接在服务端暴露，那么对于这种特殊设置可能会变得很重要。

### **type: 'amd-require'**

```
module.exports = {
  //...
  output: {
    library: {
      name: 'MyLibrary',
      type: 'amd-require',
    },
  },
};
```

它会用一个立即执行的 AMD `require(dependencies, factory)` 包装器来打包输出。

'amd-require' 类型允许使用 AMD 的依赖，而不需要单独的后续调用。与 'amd' 类型一样，这取决于在加载 webpack 输出的环境中适当的 `require` 函数是否可用。

使用该类型的话，不能使用库的名称。

## type: 'umd'

这将在所有模块定义下暴露你的库, 允许它与 CommonJS、AMD 和作为全局变量工作。可以查看 [UMD Repository](#) 获取更多内容。

在这种情况下, 你需要使用 `library.name` 属性命名你的模块:

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

最终的输出为:

```
(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();
})(global, function () {
  return _entry_return_;
});
```

请注意, 根据 [对象赋值部分](#), 省略 `library.name` 将导致入口起点返回的所有属性直接赋值给根对象。示例:

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

输出将会是:

```
(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] = a[i];
  }
})(global, function () {
```

```
    return _entry_return_;
  });
```

你可以为 `library.name` 指定一个对象，每个目标的名称不同：

```
module.exports = {
  //...
  output: {
    library: {
      name: {
        root: 'MyLibrary',
        amd: 'my-library',
        commonjs: 'my-common-library',
      },
      type: 'umd',
    },
  },
};
```

### **type: 'system'**

这将会把你的库暴露为一个 `System.register` 模块。这个特性最初是在 [webpack 4.30.0](#) 中发布。

`System` 模块要求当 webpack bundle 执行时，全局变量 `System` 出现在浏览器中。编译的 `System.register` 格式允许你在没有额外配置的情况下使用 `System.import('/bundle.js')`，并将你的 webpack bundle 加载到系统模块注册表中。

```
module.exports = {
  //...
  output: {
    library: {
      type: 'system',
    },
  },
};
```

输出：

```
System.register([], function (__WEBPACK_DYNAMIC_EXPORT__, __system_context__) {
  return {
    execute: function () {
      // ...
    },
  };
});
```

除了设置 `output.library.type` 为 `system`，还要将 `output.library.name` 添加到配置中，输出的 bundle 将以库名作为 `System.register` 的参数：

```
System.register(  
  'MyLibrary',  
  [],  
  function (__WEBPACK_DYNAMIC_EXPORT__, __system_context__) {  
    return {  
      execute: function () {  
        // ...  
      },  
    };  
  }  
);
```

## Other Types

### type: 'jsonp'

```
module.exports = {  
  // ...  
  output: {  
    library: {  
      name: 'MyLibrary',  
      type: 'jsonp',  
    },  
  },  
};
```

这会将入口起点的返回值包装到 jsonp 包装器中。

```
MyLibrary(_entry_return_);
```

你的库的依赖将由 `externals` 配置定义。

### Tip

阅读 [authoring libraries guide](#) 指南获取更多关于 `output.library.name` 与 `output.library.type` 的相关信息。

## output.library.export

指定哪一个导出应该被暴露为一个库。

- 类型: `string | string[]`

默认为 `undefined`，将会导出整个（命名空间）对象。下面的例子演示了使用 `output.library.type: 'var'` 配置项产生的作用。

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

```
    library: {
      name: 'MyLibrary',
      type: 'var',
      export: 'default',
    },
  },
};
```

入口起点的默认导出将会被赋值为库名称：

```
// 如果入口有一个默认导出
var MyLibrary = _entry_return_.default;
```

你也可以向 `output.library.export` 传递一个数组，它将被解析为一个要分配给库名的模块的路径：

```
module.exports = {
  output: {
    library: {
      name: 'MyLibrary',
      type: 'var',
      export: ['default', 'subModule'],
    },
  },
};
```

这里就是库代码：

```
var MyLibrary = _entry_return_.default.subModule;
```

## output.library.auxiliaryComment

在 UMD 包装器中添加注释。

- 类型: `string | { amd?: string, commonjs?: string, commonjs2?: string, root?: string }`

为每个 `umd` 类型插入相同的注释，将 `auxiliaryComment` 设置为 `string`。

```
module.exports = {
  // ...
  mode: 'development',
  output: {
    library: {
      name: 'MyLibrary',
      type: 'umd',
      auxiliaryComment: 'Test Comment',
    },
  },
};
```

```
},  
};
```

这将产生以下结果：

```
(function webpackUniversalModuleDefinition(root, factory) {  
  //Test Comment  
  if (typeof exports === 'object' && typeof module === 'object')  
    module.exports = factory();  
  //Test Comment  
  else if (typeof define === 'function' && define.amd) define([], factory);  
  //Test Comment  
  else if (typeof exports === 'object') exports['MyLibrary'] = factory();  
  //Test Comment  
  else root['MyLibrary'] = factory();  
})(self, function () {  
  return __entry_return__;  
});
```

对于细粒度控制，可以传递一个对象：

```
module.exports = {  
  // ...  
  mode: 'development',  
  output: {  
    library: {  
      name: 'MyLibrary',  
      type: 'umd',  
      auxiliaryComment: {  
        root: 'Root Comment',  
        commonjs: 'CommonJS Comment',  
        commonjs2: 'CommonJS2 Comment',  
        amd: 'AMD Comment',  
      },  
    },  
  },  
};
```

## output.library.umdNamedDefine

boolean

当使用 `output.library.type: "umd"` 时，将 `output.library.umdNamedDefine` 设置为 `true` 将会把 AMD 模块命名为 UMD 构建。否则使用匿名 `define`。

```
module.exports = {  
  //...  
  output: {  
    library: {  
      name: 'MyLibrary',
```

```
    type: 'umd',
    umdNamedDefine: true,
  },
},
};
```

AMD module 将会是这样：

```
define('MyLibrary', [], factory);
```

## output.libraryExport

### Warning

我们可能会停止对此的支持，所以最好使用 `output.library.export`，其作用与 `libraryExport` 一致。

```
string [string]
```

通过配置 `libraryTarget` 决定暴露哪些模块。默认情况下为 `undefined`，如果你将 `libraryTarget` 设置为空字符串，则与默认情况具有相同的行为。例如，如果设置为 `''`，将导出整个（命名空间）对象。下述 demo 演示了当设置 `libraryTarget: 'var'` 时的效果。

支持以下配置：

`libraryExport: 'default'` - **入口的默认导出** 将分配给 `library target`：

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

`libraryExport: 'MyModule'` - 这个 **确定的模块** 将被分配给 `library target`：

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

`libraryExport: ['MyModule', 'MySubModule']` - 数组将被解析为要分配给 `library target` 的 **模块路径**：

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

使用上述指定的 `libraryExport` 配置时，`library` 的结果可以这样使用：

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

```
MySubModule.doSomething();
```

## output.libraryTarget

```
string = 'var'
```

### Warning

请使用 `output.library.type` 代理，因为我们可能在未来放弃对 `output.libraryTarget` 的支持。

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`。

### Tip

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' \$#libraryTarget-var\$**

### Warning

最好使用 `output.library.type: 'var'`。

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();
```

**libraryTarget: 'assign' \$#libraryTarget-assign\$**

### Warning



最好使用 `output.library.type: '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。

**libraryTarget: 'assign-properties'** 5.16.0+ `$.libraryTarget=assign-properties$`

## Warning

最好使用 `output.library.type: 'assign-properties'` 。

如果目标对象存在，则将返回值 `copy` 到目标对象，否则先创建目标对象：

```
// 如果不存在的话就创建目标对象
MyLibrary = typeof MyLibrary === 'undefined' ? {} : MyLibrary;
// 然后复制返回值到 MyLibrary
// 与 Object.assign 行为类似

// 例如，你在入口导出了一个 `hello` 函数
export function hello(name) {
  console.log(`Hello ${name}`);
}

// 在另一个脚本中运行 MyLibrary
// 你可以像这样运行 `hello` 函数
MyLibrary.hello('World');
```

## Warning

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.libraryTarget, _entry_return_);
```

## Warning

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'

## Warning

最好使用 `output.library.type: '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'

## Warning

最好使用 `output.library.type: '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'

## Warning

最好使用 `output.library.type: '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'

### Warning

最好使用 `output.library.type: '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

这些选项将使得 `bundle` 带有更完整的模块头，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: 'module'

### Warning

最好使用 `output.library.type: 'module'`。

输出 ES 模块。请确保事先启用 `experiments.outputModule`。

需要注意的是，该功能还未完全支持，请在[此处](#)跟进进度。

## libraryTarget: 'commonjs2'

### Warning

最好使用 `output.library.type: '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` 不能与 `output.libraryTarget` 一起使用，具体原因请参照[此 issue](#)。

## Tip

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'

### Warning

最好使用 `output.library.type: '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`。

配置如下：

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

生成的 output 名称将被定义为 "MyLibrary"：

```
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) {
  // Do something with the library...
});
```

如果 `output.library` 未定义, 将会生成以下内容。

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

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'

### Warning

最好使用 `output.library.type: 'amd-require'`。

这将使用立即执行的 AMD `require(dependencies, factory)` 包装器包装您的输出。

'amd-require' 目标 (target) 允许使用 AMD 依赖项, 而无需单独的后续调用。与 'amd' 目标 (target) 一样, 这取决于在加载 webpack 输出的环境中适当可用的 `require function`。

对于此 target, 库名称将被忽略。

## libraryTarget: 'umd'

### Warning

最好使用 `output.library.type: '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',  
  },  
};
```

最终的输出结果为:

```
(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_;
  });

```

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] = a[i];
  }
})(typeof self !== 'undefined' ? self : this, function () {
  return _entry_return_;
});

```

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',
  },
};

```

**libraryTarget: 'system'**

## Warning

最好使用 `output.library.type: 'system'`。

这将暴露你的 library 作为一个由 `System.register` 的模块。此特性首次发布于 [webpack 4.30.0](#)。

当 webpack bundle 被执行时，系统模块依赖全局的变量 `System`。编译为 `System.register` 形式后，你可以使用 `System.import('/bundle.js')` 而无需额外配置，并将你的 webpack bundle 包加载到系统模块注册表中。

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

输出：

```
System.register([], function (_export) {
  return {
    setters: [],
    execute: function () {
      // ...
    },
  };
});
```

除了将 `output.libraryTarget` 设置为 `system` 之外，还可将 `output.library` 添加到配置中，输出 bundle 的 library 名将作为 `System.register` 的参数：

```
System.register('my-library', [], function (_export) {
  return {
    setters: [],
    execute: function () {
      // ...
    },
  };
});
```

你可以通过 `__system_context__` 访问 [SystemJS context](#)：

```
// 记录当前系统模块的 URL
console.log(__system_context__.meta.url);

// 导入一个系统模块，通过将当前的系统模块的 url 作为 parentUrl
__system_context__.import('./other-file.js').then((m) => {
  console.log(m);
});
```

## 其他 Targets

### libraryTarget: 'jsonp'

#### Warning

最好使用 `output.library.type: 'jsonp'`。

这将把入口起点的返回值，包裹到一个 jsonp 包装容器中

```
MyLibrary(_entry_return_);
```

你的 library 的依赖将由 `externals` 配置定义。

## output.module

```
boolean = false
```

以模块类型输出 JavaScript 文件。由于此功能还处于实验阶段，默认禁用。

当启用时，webpack 会在内部将 `output.iife` 设置为 `false`，将 `output.scriptType` 为 `'module'`，并将 `terserOptions.module` 设置为 `true`

如果你需要使用 webpack 构建一个库以供别人使用，当 `output.module` 为 `true` 时，一定要将 `output.libraryTarget` 设置为 `'module'`。

```
module.exports = {  
  //...  
  experiments: {  
    outputModule: true,  
  },  
  output: {  
    module: true,  
  },  
};
```

#### Warning

`output.module` 是一个实验性的功能，想要使用的话，通过设置 `experiments.outputModule` 为 `true`。

## output.path



```
string = path.join(process.cwd(), 'dist')
```

output 目录对应一个**绝对路径**。

#### webpack.config.js

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

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

注意，`[fullhash]` 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=true string: 'verbose'
```

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](#) 时的默认值为 `true`，而在 production [模式](#) 时的默认值为 `false`。当值为 `'verbose'` 时，会显示更多信息，如 export，运行时依赖以及 bailouts。

### Warning

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,
  },
};
```

### Tip

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

# output.publicPath

- Type:
  - function
  - string

targets 设置为 web 与 web-worker 时 output.publicPath 默认为 'auto'，查看[该指南](#)获取其用例

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**。

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

## webpack.config.js

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

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 `[fullhash]` 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: 'auto', // It automatically determines the public path from either `import.`
    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.scriptType

```
string: 'module' | 'text/javascript' boolean = false
```

这个配置项允许使用自定义 script 类型加载异步 chunk，例如 `<script type="module">...</script>`。

## Tip

如果将 `output.module` 设置为 `true`，`output.scriptType` 将会默认设置为 `'module'` 而不是 `false`。

```
module.exports = {
  //...
  output: {
    scriptType: 'module',
  },
};
```

## output.sourceMapFilename

```
string = '[file].map[query]'
```

仅在 `devtool` 设置为 `'source-map'` 时有效，此选项会向硬盘写入一个输出文件。

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 [Template strings](#) 在 `Filename-level` 下替换。

## output.sourcePrefix

```
string = ''
```

修改输出 bundle 中每行的前缀。

**webpack.config.js**

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

## Tip

使用一些缩进会使 bundle 看起来更美观，但会导致多行字符串的问题。

## Tip

通常，你不需要修改 `output.sourcePrefix`。

## output.strictModuleErrorHandling

按照 ES Module 规范处理 module 加载时的错误，会有性能损失。

- 类型： `boolean`
- 可用版本： 5.25.0+

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

## output.strictModuleExceptionHandling

### Warning

已废弃，请使用 `output.strictModuleErrorHandling`。

```
boolean = false
```

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.trustedTypes

boolean = false string object

5.37.0+

控制 [Trusted Types](#) 兼容性。启用时, webpack 将检测 Trusted Types 支持, 如果支持, 则使用 Trusted Types 策略创建它动态加载的脚本 url。当应用程序在 [require-trusted-types-for](#) 内容安全策略指令下运行时使用。

默认为 `false` (不兼容, 脚本 URL 为字符串)。

- 设置为 `true` 时, webpack 将会使用 `output.uniqueName` 作为 Trusted Types 策略名称。
- 设置为非空字符串时, 它的值将被用作策略名称。
- 设置为一个对象时, 策略名称取自对象 `policyName` 属性。

### webpack.config.js

```
module.exports = {
  //...
  output: {
    //...
    trustedTypes: {
      policyName: 'my-application#webpack',
    },
  },
};
```

## output.umdNamedDefine

### Warning

最好使用 `output.library.umdNamedDefine`。

boolean

当使用 `libraryTarget: "umd"` 时, 设置 `output.umdNamedDefine` 为 `true` 将命名由 UMD 构建的 AMD 模块。否则将使用一个匿名的 `define`。

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

## output.uniqueName

string

在全局环境下为防止多个 webpack 运行时 冲突所使用的唯一名称。默认使用

`output.library` 名称或者上下文中的 `package.json` 的包名称(package name), 如果两者都不存在, 值为 `''`。

`output.uniqueName` 将用于生成唯一全局变量:

- `output.chunkLoadingGlobal`

### webpack.config.js

```
module.exports = {
  // ...
  output: {
    uniqueName: 'my-package-xyz',
  },
};
```

## output.wasmLoading

boolean = false string

此选项用于设置加载 WebAssembly 模块的方式。默认可使用的方式有

`'fetch'` (web/WebWorker), `'async-node'` (Node.js), 其他额外方式可由插件提供。

其默认值会根据 `target` 的变化而变化:

- 如果 `target` 设置为 `'web'` , `'webworker'` , `'electron-renderer'` 或 `'node-webkit'` 其中之一, 其默认值为 `'fetch'` 。
- 如果 `target` 设置为 `'node'` , `'async-node'` , `'electron-main'` 或 `'electron-preload'` , 其默认值为 `'async-node'` 。

```
module.exports = {  
  //...  
  output: {  
    wasmLoading: 'fetch',  
  },  
};
```

## output.workerChunkLoading

string: `'require'` | `'import-scripts'` | `'async-node'` | `'import'` | `'universal'` boolean: `false`

新选项 `workerChunkLoading` 用于控制 workder 的 chunk 加载。

### Tip

此选项默认值取决于 `target` 的设置。欲了解更多详情, 请在 [webpack 默认值文件中搜索 "workerChunkLoading"](#) 。

### webpack.config.js

```
module.exports = {  
  //...  
  output: {  
    workerChunkLoading: false,  
  },  
};
```

## Module

这些选项决定了如何处理项目中的[不同类型的模块](#)。

## module.generator

5.12.0+

可以使用 `module.generator` 在一个地方配置所有生成器的选项。



## webpack.config.js

```
module.exports = {
  module: {
    generator: {
      asset: {
        // asset 模块的 generator 选项

        // 自定义 asset 模块的 publicPath, 自 webpack 5.28.0 起可用
        publicPath: 'assets/',
      },
      'asset/inline': {
        // asset/内联模块的 generator 选项
      },
      'asset/resource': {
        // asset/资源模块的 generator 选项

        // 自定义 asset/resource 模块的 publicPath, 自 webpack 5.28.0 起可用
        publicPath: 'assets/',
      },
      javascript: {
        // 该模块类型尚不支持 generator 选项
      },
      'javascript/auto': {
        // 同上
      },
      'javascript/dynamic': {
        // 同上
      },
      'javascript/esm': {
        // 同上
      },
      // 其他...
    },
  },
};
```

## module.parser

5.12.0+

类似于 `module.generator` , 你可以用 `module.parser` 在一个地方配置所有解析器的选项。

## webpack.config.js

```
module.exports = {
  module: {
    parser: {
      asset: {
```

```
// 资产模块的 parser 选项
},
'asset/inline': {
  // 该模块类型尚不支持 parser 选项
},
'asset/resource': {
  // 同上
},
'asset/source': {
  // 同上
},
javascript: {
  // javascript 模块的解析器选项
  // 例如, 启用解析 require.ensure 语法的功能
  requireEnsure: true,
},
'javascript/auto': {
  // 同上
},
'javascript/dynamic': {
  // 同上
},
'javascript/esm': {
  // 同上
},
// 其他...
},
};
```

## module.parser.javascript

JavaScript parser 的配置项。

```
module.exports = {
  module: {
    parser: {
      javascript: {
        // ...
        commonjsMagicComments: true,
      },
    },
  },
};
```

允许在 `Rule.parser` 中配置这些选项。也可以是特定模块。

### module.parser.javascript.commonjsMagicComments

为 CommonJS 启用 [魔法注释](#)。

- 类型: boolean
- 可用版本: 5.17.0+
- 示例:

```
module.exports = {
  module: {
    parser: {
      javascript: {
        commonjsMagicComments: true,
      },
    },
  },
};
```

请注意, 目前只支持 webpackIgnore 注释:

```
const x = require(/* webpackIgnore: true */ 'x');
```

## module.parser.javascript.exportsPresence

指出在 `"import ... from ..."` 与 `"export ... from ..."` 中无效导出名称的行为。

- Type: 'error' | 'warn' | 'auto' | false
- Available: 5.62.0+
- Example:

```
module.exports = {
  module: {
    parser: {
      javascript: {
        exportsPresence: 'error',
      },
    },
  },
};
```

## module.parser.javascript.importExportsPresence

指出在 `"import ... from ..."` 中无效导出名称的行为。

- Type: 'error' | 'warn' | 'auto' | false
- Available: 5.62.0+
- Example:

```
module.exports = {
  module: {
```

```
    parser: {
      javascript: {
        importExportsPresence: 'error',
      },
    },
  },
};
```

## module.parser.javascript.reexportExportsPresence

指出在 `"export ... from ..."` 中无效导出名称的行为。当在 TypeScript 重新导出类型, 从 `"export ... from ..."` 迁移到 `"export type ... from ..."` 时禁用该配置项是有用的。

- Type: 'error' | 'warn' | 'auto' | false
- Available: 5.62.0+
- Example:

```
module.exports = {
  module: {
    parser: {
      javascript: {
        reexportExportsPresence: 'error',
      },
    },
  },
};
```

## module.parser.javascript.url

启用 `new URL()` 语法解析。

- 类型: boolean = true | 'relative'
- 示例:

```
module.exports = {
  module: {
    parser: {
      javascript: {
        url: false, // disable parsing of `new URL()` syntax
      },
    },
  },
};
```

自 webpack 5.23.0 起, `module.parser.javascript.url` 的 'relative' 值可用。当使用 'relative' 时, webpack 将为 `new URL()` 语法生成相对的 URL, 即结果 URL 中不包含根 URL:

```
<!-- with 'relative' -->
<img src='c43188443804f1b1f534.svg' />

<!-- without 'relative' -->
<img src='file:///path/to/project/dist/c43188443804f1b1f534.svg' />
```

1. 当服务器不知道根 URL 时，这对 SSR（服务端渲染）很有用（而且可用节省一些字节）。为了使其相同，它也必须用于客户端构建。
2. 这也适用于 SSG（静态网站生成器），mini-css-plugin 和 html-plugin 等通常需要进行服务端渲染。

## 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.unsafeCache

boolean function (module)

缓存模块请求的解析。 module.unsafeCache 包含如下几个默认值：

- 如果 cache 未被启用，则默认值为 false 。

- 如果 `cache` 被启用，并且此模块的来自 `node_modules`，则值为 `true`，否则为 `false`。

### webpack.config.js

```
module.exports = {  
  //...  
  module: {  
    unsafeCache: false,  
  },  
};
```

## 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：资源文件的绝对路径。它已经根据 `resolve` 规则解析。
2. issuer: 请求者的文件绝对路径。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。

### Warning

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。每个嵌套规则包含它自己的条件。

被解析的顺序基于以下规则:

1. 父规则
2. `rules`
3. `oneOf`

## 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(inline)、普通(normal)、前置(pre) 的顺序调用。更多详细信息，请查看 [Pitching Loader](#)。
2. **Normal** 阶段: loader 上的 常规方法，按照 前置(pre)、普通(normal)、行内(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）。

```
// 禁用普通 loaders
import { a } from '!../file1.js';

// 禁用前置和普通 loaders
import { b } from '-!./file2.js';

// 禁用所有的 loaders
import { c } from '!!!./file3.js';
```

不应使用内联 loader 和 `!` 前缀，因为它是非标准的。它们可能会被 loader 生成代码使用。

## Rule.exclude

排除所有符合条件的模块。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

引入符合以下任何条件的模块。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](#)，用来与被发出的 request 对应的模块项匹配。在以下示例中，`a.js` request 的发出者(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.issuerLayer

允许按 issuer 的 layer 进行过滤/匹配。

**webpack.config.js**

```
module.exports = {  
  // ...  
  module: {  
    rules: [  
      {  
        issuerLayer: 'other-layer',  
      },  
    ],  
  },  
};
```

## Rule.layer

string

指定模块应放置在哪个 layer。

**webpack.config.js**

```
module.exports = {  
  // ...  
  module: {  
    rules: [  
      {  
        test: /module-layer-change/,  
        layer: 'layer',  
      },  
    ],  
  },  
};
```

## Rule.loader

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

## Rule.loaders

### Warning

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.mimetype

你可以使用 `mimetype` 使 `rules` 配置与 `data` 的 `uri` 进行匹配。

### webpack.config.js

```
module.exports = {
  // ...
  module: {
    rules: [
      {
        mimetype: 'application/json',
        type: 'json',
      },
    ],
  },
};
```

`mimetype` 已默认包含了 `application/json`，`text/javascript`，`application/javascript`，`application/node` 以及 `application/wasm`。

## 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',
    },
  ],
},
],
},
};
```

## Tip

详情请参阅 [嵌套规则](#)。

## 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](#)。

## Warning

由于需要支持 `Rule.options`，`UseEntry.options` 以及 `Rule.use`，因此 `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` 插件，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.extensions, require.main
          node: {...}, // 在模块级别(module level)上重新配置 [node] (/configuration/node) 层(layer)
          worker: ["default from web-worker", "..."] // 自定义 WebWorker 对 JavaScript 的处理
        }
      }
    ]
  }
}

```

如果 `Rule.type` 的值为 `asset`，那么 `Rules.parser` 选项可能是一个对象或一个函数，其作用可能是将文件内容编码为 Base64，还可能是将其当做单独文件 emit 到输出目录。

如果 `Rule.type` 的值为 `asset` 或 `asset/inline`，那么 `Rule.generator` 选项可能是一个描述模块源编码方式的对象，还可能是一个通过自定义算法对模块源代码进行编码的函数。

更多信息及用例请查阅 [Asset 模块指南](#)。

## Rule.parser.dataUrlCondition

```

object = { maxSize number = 8096 } function (source, { filename, module })
=> boolean

```

如果一个模块源码大小小于 `maxSize`，那么模块会被作为一个 Base64 编码的字符串注入到包中，否则模块文件会被生成到输出的目标目录中。

### webpack.config.js

```

module.exports = {
  //...
  module: {

```

```
rules: [
  {
    //...
    parser: {
      dataUrlCondition: {
        maxSize: 4 * 1024,
      },
    },
  },
],
};
```

当提供函数时，返回 `true` 值时告知 webpack 将模块作为一个 Base64 编码的字符串注入到包中，否则模块文件会被生成到输出的目标目录中。

### webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        //...
        parser: {
          dataUrlCondition: (source, { filename, module }) => {
            const content = source.toString();
            return content.includes('some marker');
          },
        },
      },
    ],
  },
};
```

## Rule.generator

### Rule.generator.dataUrl

```
object = { encoding string = 'base64' | false, mimetype string = undefined | false } function (content, { filename, module }) => string
```

当 `Rule.generator.dataUrl` 被用作一个对象，你可以配置两个属性：

- `encoding`: 当被设置为 `'base64'`，模块源码会用 Base64 算法编码。设置 `encoding` 为 `false`，会禁用编码。
- `mimetype`: 为数据链接(data URI)设置的一个 `mimetype` 值。默认根据模块资源后缀设置。

## webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        //...
        generator: {
          dataUrl: {
            encoding: 'base64',
            mimetype: 'mimetype/png',
          },
        },
      },
    ],
  },
};
```

当被作为一个函数使用，它必须为每个模块执行且并须返回一个数据链接(data URI)字符串。

```
module.exports = {
  //...
  module: {
    rules: [
      {
        //...
        generator: {
          dataUrl: (content) => {
            const svgToMiniDataURI = require('mini-svg-data-uri');
            if (typeof content !== 'string') {
              content = content.toString();
            }
            return svgToMiniDataURI(content);
          },
        },
      },
    ],
  },
};
```

## Rule.generator.emit

配置不从 [Asset Modules](#) 写入资源文件, 你可能会在 SSR 中使用它。

- 类型: `boolean = true`
- 可用版本: `5.25.0+`
- 示例:

```
module.exports = {
  // ...
  module: {
    rules: [
      {
        test: /\.png$/i,
        type: 'asset/resource',
        generator: {
          emit: false,
        },
      },
    ],
  },
};
```

## Rule.generator.filename

对某些特定的规则而言与 `output.assetModuleFilename` 相同. 覆盖了

`output.assetModuleFilename` 选项并且仅与 `asset` 和 `asset/resource` 模块类型一同起作用。

### webpack.config.js

```
module.exports = {
  //...
  output: {
    assetModuleFilename: 'images/[hash][ext][query]',
  },
  module: {
    rules: [
      {
        test: /\.png/,
        type: 'asset/resource',
      },
      {
        test: /\.html/,
        type: 'asset/resource',
        generator: {
          filename: 'static/[hash][ext]',
        },
      },
    ],
  },
};
```

## Rule.generator.publicPath

对指定的资源模式指定 `publicPath`。

- 类型: `string | ((pathData: PathData, assetInfo?: AssetInfo) => string)`
- 可用版本: 5.28.0+

```
module.exports = {
  //...
  output: {
    publicPath: 'static/',
  },
  module: {
    rules: [
      {
        test: /\.png$/i,
        type: 'asset/resource',
        generator: {
          publicPath: 'assets/',
        },
      },
    ],
  },
};
```

## Rule.resource

[条件](#) 会匹配 `resource`。在 [Rule 条件](#) 中查阅详情。

## Rule.resourceQuery

与资源查询相匹配的 [Condition](#)。此选项用于测试请求字符串的查询部分（即从问号开始）。如果你需要通过 `import Foo from './foo.css?inline'` 导入 `Foo`，则需符合以下条件：

**webpack.config.js**

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



# Rule.parser.parse

function(input) => string | object

如果 `Rule.type` 被设置成 `'json'`，那么 `Rules.parser.parse` 选择可能会是一个方法，该方法实现自定义的逻辑，以解析模块的源和并将它转换成 JavaScript 对象。它可能在没有特定加载器的时候，对将 `toml`，`yaml` 和其它非 JSON 文件导入成导入非常有用：

## webpack.config.js

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

module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.toml/,
        type: 'json',
        parser: {
          parse: toml.parse,
        },
      },
    ],
  },
};
```

# Rule.rules

规则 数组，当规则匹配时使用。

## Tip

详情参阅 [嵌套规则](#)。

# Rule.scheme

匹配使用的 schema，例如 `data`，`http`。

- 类型: `string` | `RegExp` | `((value: string) => boolean)` | `RuleSetLogicalConditions` | `RuleSetCondition[]`

- 可用: 5.38.0+

### webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        scheme: 'data',
        type: 'asset/resource',
      },
    ],
  },
};
```

## Rule.sideEffects

bool

表明模块的哪一部份包含副作用。详情参阅 [Tree Shaking](#)。

## Rule.test

引入所有通过断言测试的模块。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

可设置值: 'javascript/auto' | 'javascript/dynamic' | 'javascript/esm' | 'json' | 'webassembly/sync' | 'webassembly/async' | 'asset' | 'asset/source' | 'asset/resource' | 'asset/inline'

`Rule.type` 设置类型用于匹配模块。它防止了 `defaultRules` 和它们的默认导入行为发生。例如，如果你想通过自定义 loader 加载一个 `.json` 文件，你会需要将 `type` 设置为 `javascript/auto` 以绕过 webpack 内置的 json 导入。（详参[v4.0 更新日志](#)）

### webpack.config.js

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

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

关于 `asset*` 类型详参 [资源模块指南](#)。

## 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 可以通过传入多个 loaders 以达到链式调用的效果，它们会从右到左被应用（从最后到最先配置）。

### 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` 也可以是一个函数，接受对象参数，描述被加载的模块，而且必须返回 `UseEntry` 元素的数组。

`info` 对象参数有以下的字段：

- `compiler` : 当前 webpack 的编译器（可以是 `undefined` 值）
- `issuer` : 模块的路径，该元素正在导入一个被加载的模块(resource)
- `realResource` : 总会是被加载模块的路径
- `resource` : 被加载的模块路径，它常常与 `realResource` 相等，只有当资源名称被 `request` 字符串中的 `!!` 覆盖时才不相等

相同的作为数组的快捷方式可以用作返回值（如 `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),
                  },
                },
              ],
            },
          },
        ],
      },
    ],
  },
},
];

```

```
  },  
};
```

参参 [UseEntry](#)。

## Rule.resolve

### Warning

`Rule.resolve` 从 webpack 4.36.1 起可用

模块解析可以在模块层被配置。[resolve 配置页面](#)可查看所有可用的配置。所有被应用的 `resolve` 选项被更高层级的`resolve`配置合并。

例如，假设我们有一个入口文件在 `./src/index.js`，`./src/footer/default.js` 和一个 `./src/footer/overridden.js`，用以论证模块层级的解析。

`./src/index.js`

```
import footer from 'footer';  
console.log(footer);
```

`./src/footer/default.js`

```
export default 'default footer';
```

`./src/footer/overridden.js`

```
export default 'overridden footer';
```

webpack.js.org

```
module.exports = {  
  resolve: {  
    alias: {  
      footer: './footer/default.js',  
    },  
  },  
};
```

当用该配置创建一个包，`console.log(footer)` 会输出 'default footer'。让我们为 `.js` 文件设置 `Rule.resolve`，以及 `footer` 作为 `overridden.js` 的别名。

webpack.js.org

```
module.exports = {
  resolve: {
    alias: {
      footer: './footer/default.js',
    },
  },
  module: {
    rules: [
      {
        resolve: {
          alias: {
            footer: './footer/overridden.js',
          },
        },
      },
    ],
  },
};
```

当用该配置创建一个包， `console.log(footer)` 会输出 'overridden footer'。

## resolve.fullySpecified

boolean = true

启用后，你若在 `.mjs` 文件或其他 `.js` 文件中导入模块，并且它们最近的 `package.json` 中包含 `"type"` 字段，其值为 `"module"` 时，你应为此文件提供扩展名，否则 webpack 会提示 `Module not found` 的错误且编译失败。并且 webpack 不会解析 `resolve.mainFiles` 中定义的文件目录，你必须自己指定文件名。

### webpack.config.js

```
module.exports = {
  // ...
  module: {
    rules: [
      {
        test: /\.m?js$/,
        resolve: {
          fullySpecified: false, // disable the behaviour
        },
      },
    ],
  },
};
```

## Warning

`resolve.fullySpecified` 不影响来自 `mainFields`，`aliasFields` 或 `aliases` 的请求。

# Condition

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。

{ 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:**

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

module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.css$/,
        include: [
          // will include any paths relative to the current directory starting with `app/st
          // e.g. `app/styles.css`, `app/styles/styles.css`, `app/stylesheet.css`
          path.resolve(__dirname, 'app/styles'),
          // add an extra slash to only include the content of the directory `vendor/styles
          path.join(__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)

`UseEntry` 也可以是一个函数，接受对象参数，描述被加载的模块，而且必须返回一个参数对象。这可用于按模块更改 `loader` 选项。

`info` 对象参数有以下的字段：

- `compiler` : 当前 webpack 的编译器（可以是 `undefined` 值）
- `issuer` : 模块的路径，该元素正在导入一个被加载的模块(resource)
- `realResource` : 总会是被加载模块的路径
- `resource` : 被加载的模块路径，它常常与 `realResource` 相等，只有当资源名称被 `request` 字符串中的 `!!` 覆盖时才不相等

### 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 Contexts)

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,  
  },  
};
```

## Tip

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` 将缺失的导出提示成错误而不是警告
- 为部分动态依赖项设置内部正则表达式： `wrappedContextRegExp: /\\\.\\*/`

## Warning

为支持 `exportsPresence` 配置项， `strictExportPresence` 已被废弃。

# 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，但是还是可能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简单。例如，一些位于 `src/` 文件夹下的常用模块：

### webpack.config.js

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

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

alias:	{}
import 'xyz'	/abc/node_modules/xyz/index.js
import 'xyz/file.js'	/abc/node_modules/xyz/file.js

alias:	{ xyz: '/abc/path/to/file.js' }
import 'xyz'	/abc/path/to/file.js
import 'xyz/file.js'	error

alias:	{ xyz\$: '/abc/path/to/file.js' }
import 'xyz'	/abc/path/to/file.js
import 'xyz/file.js'	/abc/node_modules/xyz/file.js

alias:	{ xyz: './dir/file.js' }
import 'xyz'	/abc/dir/file.js
import 'xyz/file.js'	error

alias:	{ xyz\$: './dir/file.js' }
import 'xyz'	/abc/dir/file.js
import 'xyz/file.js'	/abc/node_modules/xyz/file.js

alias:	{ xyz: '/some/dir' }
import 'xyz'	/some/dir/index.js
import 'xyz/file.js'	/some/dir/file.js

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

alias:	{ xyz: './dir' }
import 'xyz'	/abc/dir/index.js
import 'xyz/file.js'	/abc/dir/file.js

alias:	{ xyz: 'modu' }
--------	-----------------

import 'xyz'	/abc/node_modules/modu/index.js
import 'xyz/file.js'	/abc/node_modules/modu/file.js

alias:	{ xyz\$: 'modu' }
import 'xyz'	/abc/node_modules/modu/index.js
import 'xyz/file.js'	/abc/node_modules/xyz/file.js

alias:	{ xyz: 'modu/some/file.js' }
import 'xyz'	/abc/node_modules/modu/some/file.js
import 'xyz/file.js'	error

alias:	{ xyz: 'modu/dir' }
import 'xyz'	/abc/node_modules/modu/dir/index.js
import 'xyz/file.js'	/abc/node_modules/modu/dir/file.js

alias:	{ xyz\$: 'modu/dir' }
import 'xyz'	/abc/node_modules/modu/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。

Warning

resolve.alias 优先级高于其它模块解析方式。

Warning

`null-loader` 在webpack5中被废弃。使用 `alias: { xyz$: false }` 或绝对路径 `alias: {[path.resolve(__dirname, './path/to/module')]: false }`

Warning

`[string]` 字符串值在 webapck 5 中被支持。

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

```
    alias: {
      _: [
        path.resolve(__dirname, 'src/utilities/'),
        path.resolve(__dirname, 'src/templates/'),
      ],
    },
  },
};
```

将 `resolve.alias` 设置为 `false` 将告知 webpack 忽略模块。

```
module.exports = {
  //...
  resolve: {
    alias: {
      'ignored-module': false,
      './ignored-module': false,
    },
  },
};
```

## resolve.aliasFields

[string]: ['browser']

指定一个字段，例如 `browser`，根据 [此规范](#) 进行解析。

### webpack.config.js

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

## resolve.cacheWithContext

boolean

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.conditionNames

string[]

`exports` 配置项（定义一个库的入口）的 `conditionName`。

### webpack.config.js

```
module.exports = {  
  //...  
  resolve: {  
    conditionNames: ['require', 'node'],  
  },  
};
```

## resolve.descriptionFiles

[string] = ['package.json']

用于描述的 JSON 文件。

### 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。

### webpack.config.js

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

## resolve.extensions

[string] = ['.js', '.json', '.wasm']

尝试按顺序解析这些后缀名。如果有多个文件有相同的名字，但后缀名不同，webpack 会解析列在数组首位的后缀的文件并跳过其余的后缀。

### webpack.config.js

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

能够使用户在引入模块时不带扩展：

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

请注意，以上这样使用 `resolve.extensions` 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。然而你可以使用 `'...'` 访问默认拓展名：

```
module.exports = {  
  //...  
  resolve: {  
    extensions: ['.ts', '...'],  
  },  
};
```

## resolve.fallback

object

当正常解析失败时，重定向模块请求。

### webpack.config.js

```
module.exports = {  
  //...  
  resolve: {  
    fallback: {  
      abc: false, // do not include a polyfill for abc  
      xyz: path.resolve(__dirname, 'path/to/file.js'), // include a polyfill for xyz  
    },  
  },  
};
```

webpack 5 不再自动 polyfill Node.js 的核心模块，这意味着如果你在浏览器或类似的环境中运行的代码中使用它们，你必须从 NPM 中安装兼容的模块，并自己包含它们。以下是 webpack 在 webpack 5 之前使用过的 polyfill 包列表：



```
module.exports = {
  //...
  resolve: {
    fallback: {
      assert: require.resolve('assert'),
      buffer: require.resolve('buffer'),
      console: require.resolve('console-browserify'),
      constants: require.resolve('constants-browserify'),
      crypto: require.resolve('crypto-browserify'),
      domain: require.resolve('domain-browser'),
      events: require.resolve('events'),
      http: require.resolve('stream-http'),
      https: require.resolve('https-browserify'),
      os: require.resolve('os-browserify/browser'),
      path: require.resolve('path-browserify'),
      punycode: require.resolve('punycode'),
      process: require.resolve('process/browser'),
      querystring: require.resolve('querystring-es3'),
      stream: require.resolve('stream-browserify'),
      string_decoder: require.resolve('string_decoder'),
      sys: require.resolve('util'),
      timers: require.resolve('timers-browserify'),
      tty: require.resolve('tty-browserify'),
      url: require.resolve('url'),
      util: require.resolve('util'),
      vm: require.resolve('vm-browserify'),
      zlib: require.resolve('browserify-zlib'),
    },
  },
};
```

## 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：

### 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 的类库 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.mainFields

[string] = ['index']

解析目录时要使用的文件名。

#### webpack.config.js

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

## resolve.exportsFields

[string] = ['exports']

在 package.json 中用于解析模块请求的字段。欲了解更多信息，请查阅 [package-exports guideline](#) 文档。

#### webpack.config.js

```
module.exports = {
  //...
  resolve: {
    exportsFields: ['exports', 'myCompanyExports'],
  },
};
```

## 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

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

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

## resolve.unsafeCache

RegExp [RegExp] boolean: true

启用，会主动缓存模块，但并不**安全**。传递 `true` 将缓存一切。

## 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/,  
  },  
};
```

### Warning

修改缓存路径可能在极少数情况下导致失败。

## resolve.useSyncFilesystemCalls

boolean

对 resolver 使用同步文件系统调用。

## webpack.config.js

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

## 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.preferRelative

boolean

当启用此选项时，webpack 更倾向于将模块请求解析为相对请求，而不使用来自 `node_modules` 目录下的模块。

## webpack.config.js

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

## src/index.js

```
// let's say `src/logo.svg` exists
import logo1 from 'logo.svg'; // this is viable when `preferRelative` enabled
import logo2 from './logo.svg'; // otherwise you can only use relative path to resolve logc

// `preferRelative` is enabled by default for `new URL()` case
const b = new URL('module/path', import.meta.url);
const a = new URL('./module/path', import.meta.url);
```

## resolve.preferAbsolute

boolean

5.13.0+

解析时，首选的绝对路径为 `resolve.roots`。

## webpack.config.js

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

```
    resolve: {
      preferAbsolute: true,
    },
  };
};
```

## 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)。

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)的位置。注意，当使用创建符号链接包的工具（如 `npm link`）时，这种方式可能会导致模块解析失败。

### webpack.config.js

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

## resolve.cachePredicate

function(module) => boolean

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;
    },
  },
};
```

## resolve.restrictions

[string, RegExp]

解析限制列表，用于限制可以解析请求的路径。

### webpack.config.js

```
module.exports = {  
  //...  
  resolve: {  
    restrictions: [/\. (sass|scss|css)$/],  
  },  
};
```

## resolve.roots

[string]

A list of directories where requests of server-relative URLs (starting with '/') are resolved, defaults to [context configuration option](#). On non-Windows systems these requests are resolved as an absolute path first.

### webpack.config.js

```
const fixtures = path.resolve(__dirname, 'fixtures');  
module.exports = {  
  //...  
  resolve: {  
    roots: [__dirname, fixtures],  
  },  
};
```

## resolve.importsFields

[string]

Fields from `package.json` which are used to provide the internal requests of a package (requests starting with `#` are considered internal).

### webpack.config.js

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

## resolve.byDependency

通过 module 请求类型来配置 resolve 选项。

- Type: [type: string]: ResolveOptions
- Example:

```
module.exports = {
  // ...
  resolve: {
    byDependency: {
      // ...
      esm: {
        mainFields: ['browser', 'module'],
      },
      commonjs: {
        aliasFields: ['browser'],
      },
      url: {
        preferRelative: true,
      },
    },
  },
};
```

## resolveLoader

object { modules [string] = ['node\_modules'], extensions [string] = ['.js', '.json'], mainFields [string] = ['loader', 'main']}

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。

### webpack.config.js

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

### Tip

请注意你可以在这里使用别名和 resolve 中熟悉的其他特性。例如 { txt: 'raw-loader' } 将会 shim txt!templates/demo.txt 使用 raw-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.chunkIds

```
boolean = false  string: 'natural' | 'named' | 'size' | 'total-size' | 'deterministic'
```

告知 webpack 当选择模块 id 时需要使用哪种算法。将 `optimization.chunkIds` 设置为 `false` 会告知 webpack 没有任何内置的算法会被使用，但自定义的算法会由插件提供。  
`optimization.chunkIds` 的默认值是 `false`：

- 如果环境是开发环境，那么 `optimization.chunkIds` 会被设置成 `'named'`，但当在生产环境中时，它会被设置成 `'deterministic'`
- 如果上述的条件都不符合，`optimization.chunkIds` 会被默认设置为 `'natural'`

下述选项字符串值均为被支持：

选项值	'natural'
描述	按使用顺序的数字 id。
选项值	'named'
描述	对调试更友好的可读的 id。
选项值	'deterministic'
描述	在不同的编译中不变的短数字 id。有益于长期缓存。在生产模式中会默认开启。
选项值	'size'
描述	专注于让初始下载包大小更小的数字 id。
选项值	'total-size'
描述	专注于让总下载包大小更小的数字 id。

### webpack.config.js

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

```
    optimization: {
      chunkIds: 'named',
    },
  };
```

默认地, 当 `optimization.chunkIds` 被设置为 `'deterministic'` 时, 最少3位数字会被使用。要覆盖默认的行为, 要将 `optimization.chunkIds` 设置为 `false`, 同时要使用 `webpack.ids.DeterministicChunkIdsPlugin`。

#### webpack.config.js

```
module.exports = {
  //...
  optimization: {
    chunkIds: false,
  },
  plugins: [
    new webpack.ids.DeterministicChunkIdsPlugin({
      maxLength: 5,
    }),
  ],
};
```

## optimization.concatenateModules

boolean

告知 webpack 去寻找模块图形中的片段, 哪些是可以安全地被合并到单一模块中。这取决于 `optimization.providedExports` 和 `optimization.usedExports`。默认 `optimization.concatenateModules` 在 **生产模式** 下被启用, 而在其它情况下被禁用。

#### webpack.config.js

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

## optimization.emitOnErrors

boolean = false

使用 `optimization.emitOnErrors` 在编译时每当有错误时，就会发送静态资源。这样可以确保出错的静态资源被发送出来。关键错误会被发送到生成的代码中，并会在运行时报错。

#### webpack.config.js

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

### Warning

如果你使用的是 webpack 的 [CLI](#)，当这个插件被启用时，webpack 进程不会以错误代码退出。如果你想让 webpack 在使用 CLI 时 "fail"，请查阅 [bail 选项](#)。

## optimization.flagIncludedChunks

boolean

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 模式](#) 中启用，其他情况禁用。

#### webpack.config.js

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

## optimization.innerGraph

boolean = true

`optimization.innerGraph` 告知 webpack 是否对未使用的导出内容，实施内部图形分析 (graph analysis)。

#### webpack.config.js

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

# optimization.mangleExports

boolean string: 'deterministic' | 'size'

optimization.mangleExports 允许控制导出处理(export mangling)。

默认 optimization.mangleExports: 'deterministic' 会在 production 模式下启用而其它情况会被禁用。

此选项支持以下选项：

选项	'size'
描述	简写形式 — 通常只有一个字符 — 专注于最小的下载 size。

选项	'deterministic'
描述	简写形式 - 通常两个字符 — 在添加或移除 export 时不会改变。适用于长效缓存。

选项	true
描述	等价于 'deterministic'

选项	false
描述	保留原名，有利于阅读和调试。

## webpack.config.js

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

## optimization.mangleWasmImports

boolean = 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.mergeDuplicateChunks

boolean = 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.minimize

boolean = true

告知 webpack 使用 [TerserPlugin](#) 或其它在 `optimization.minimizer` 定义的插件压缩 bundle。

**webpack.config.js**

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

```
    minimize: false,  
  },  
};
```

## Tip

了解 [mode](#) 工作机制。

# optimization.minimizer

[TerserPlugin] 或 [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({  
        parallel: true,  
        terserOptions: {  
          // https://github.com/webpack-contrib/terser-webpack-plugin#terseroptions  
        },  
      })),  
    ],  
  },  
};
```

或，使用函数的形式：

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

在 `optimization.minimizer` 中可以使用 `'...'` 来访问默认值。

```
module.exports = {
  optimization: {
    minimizer: [new CssMinimizer(), '...'],
  },
};
```

# optimization.moduleIds

boolean: false   string: 'natural' | 'named' | 'deterministic' | 'size'

告知 webpack 当选择模块 id 时需要使用哪种算法。将 `optimization.moduleIds` 设置为 `false` 会告知 webpack 没有任何内置的算法会被使用，但自定义的算法会由插件提供。

下述选项字符串值均为被支持:

选荐值	natural
描述	按使用顺序的数字 id。
选荐值	named
描述	对调试更友好的可读的 id。
选荐值	deterministic
描述	被哈希转化成的小位数值模块名。
选荐值	size
描述	专注于让初始下载包大小更小的数字 id。

## webpack.config.js

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

`deterministic` 选项有益于长期缓存，但对比于 `hashed` 来说，它会导致更小的文件 bundles。数字值的长度会被选作用于填满最多 80% 的 id 空间。当 `optimization.moduleIds` 被设置成 `deterministic`，默认最小 3 位数字会被使用。要覆盖默认行为，要将 `optimization.moduleIds` 设置成 `false`，并且要使用 `webpack.ids.DeterministicModuleIdsPlugin`。

## webpack.config.js

```
module.exports = {
  //...
  optimization: {
    moduleIds: false,
  },
  plugins: [
    new webpack.ids.DeterministicModuleIdsPlugin({
      maxLength: 5,
    }),
  ],
};
```

## Warning

`moduleIds: 'deterministic'` 在 webpack 5 中被添加, 而且 `moduleIds: 'hashed'` 相应地会在 webpack 5 中废弃。

## Warning

`moduleIds: total-size` 在 webpack 5 中被废弃。

# optimization.nodeEnv

boolean = false string

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',
  },
};
```

## Tip



当 `mode` 设置为 `'none'` 时, `optimization.nodeEnv` 的默认值为 `false`。

## optimization.portableRecords

boolean

`optimization.portableRecords` 告知 webpack 生成带有相对路径的记录(records)使得可以移动上下文目录。

默认 `optimization.portableRecords` 被禁用。如果下列至少一个选项在 webpack 中被设置, 该选项也会自动启用: `recordsPath`, `recordsInputPath`, `recordsOutputPath`。

**webpack.config.js**

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

## optimization.providedExports

boolean

告知 webpack 去确定那些由模块提供的导出内容, 为 `export * from ...` 生成更多高效的代码。默认 `optimization.providedExports` 会被启用。

**webpack.config.js**

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

## optimization.realContentHash

boolean = true

在处理静态资源后添加额外的哈希编译，以获得正确的静态资源内容哈希。如果 `realContentHash` 设置为 `false`，内部数据用于计算哈希值，当静态资源相同时，它可以改变。

#### webpack.config.js

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

## optimization.removeAvailableModules

boolean = false

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 `true` 以启用这项优化。在 [production 模式](#) 中默认会被开启。

#### webpack.config.js

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

### Warning

`optimization.removeAvailableModules` 会削减了 webapck 的性能表现，而且将会在下一个主要发布版本中，在 `production` 模式下会被禁用。如果你想获得额外的构建性能，请在 `production` 模式中禁用它。

## optimization.removeEmptyChunks

boolean = 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.runtimeChunk

object string boolean

将 `optimization.runtimeChunk` 设置为 `true` 或 `'multiple'`，会为每个入口添加一个只含有 `runtime` 的额外 `chunk`。此配置的别名如下：

### 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`。

## Warning

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.sideEffects

boolean = true string: 'flag'

告知 webpack 去辨识 package.json 中的 [副作用](#) 标记或规则，以跳过那些当导出不被使用且被标记不包含副作用的模块。

### package.json

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

### Tip

需要注意的是（副作用）sideEffects 需要在 npm 模块的 package.json 文件中，但并不意味着你需要在你自己的引用那个大模块的项目中的 package.json 中将 sideEffects 设置成 false。

optimization.sideEffects 取决于 [optimization.providedExports](#) 被设置成启用。这个依赖会有构建时间的损耗，但去掉模块会对性能有正面的影响，因为更少的代码被生成。该优化的效果取决于你的代码库，可以尝试这个特性以获取一些可能的性能优化。

### webpack.config.js

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

```
    sideEffects: true,
  },
};
```

只使用手动 flag，并且不对源码进行分析：

```
module.exports = {
  //...
  optimization: {
    sideEffects: 'flag',
  },
};
```

此处的 'flag' 值在非生产环境默认使用。

## Tip

设置为 `optimization.sideEffects` 时，当模块只包含无副作用的语句时，此模块也会被标记为无副作用。

## 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.usedExports

boolean = true string: 'global'

告知 webpack 去决定每个模块使用的导出内容。这取决于 `optimization.providedExports` 选项。由 `optimization.usedExports` 收集的信息会被其它优化手段或者代码生成使用，比如未使用的导出内容不会被生成，当所有的使用都适配，导出名称会被处理做单个标记字符。在压缩工具中的无用代码清除会受益于该选项，而且能够去除未使用的导出内容。

### webpack.config.js

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

选择退出每次运行时使用 export 分享：

```
module.exports = {
  //...
  optimization: {
    usedExports: 'global',
  },
};
```

## 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。

### Tip

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。An array of webpack plugins. For example, `DefinePlugin` allows you to create global constants which can be configured at compile time. This can be useful for allowing different behavior between development builds and release builds.

### 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 DashboardPlugin = require('webpack-dashboard/plugin');

// 在配置中添加插件
module.exports = {
  //...
  plugins: [
    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(),
  ],
};
```

## DevServer

[webpack-dev-server](#) 可用于快速开发应用程序。请查阅 [开发指南](#) 开始使用。

当前页面记录了影响 `webpack-dev-server` (简写: `dev-server`) `version >= 4.0.0` 配置的选项。可以在 [这里](#) 找到 `v3` 到 `v4` 的迁移指南

### Warning

`webpack-dev-server v4.0.0+` 要求 `node >= v12.13.0`、`webpack >= v4.37.0` (但是我们推荐使用 `webpack >= v5.0.0`) 和 `webpack-cli >= v4.7.0`。

## devServer

object

通过 [webpack-dev-server](#) 的这些配置，能够以多种方式改变其行为。这是一个基本的示例，利用 `gzip`s 压缩 `public/` 目录当中的所有内容并提供一个本地服务(serve)：

### webpack.config.js

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

module.exports = {
  //...
  devServer: {
    static: {
```

```
    directory: path.join(__dirname, 'public'),
  },
  compress: true,
  port: 9000,
},
};
```

当服务( `server` )启动后, 在解析模块列表之前输出一条消息:

```
<i> [webpack-dev-server] Project is running at:
<i> [webpack-dev-server] Loopback: http://localhost:9000/
<i> [webpack-dev-server] On Your Network (IPv4): http://197.158.164.104:9000/
<i> [webpack-dev-server] On Your Network (IPv6): http://[fe80::1]:9000/
<i> [webpack-dev-server] Content not from webpack is served from '/path/to/public' director
```

这里将会给出服务启动位置以及内容的一些基本信息。

如果你通过 Node.js API 使用 dev-server, 则 `devServer` 中的配置选项将被忽略。但可以将配置选项作为第一个参数传入: `new WebpackDevServer({...}, compiler)`。此示例展示了如何通过 Node.js API 使用 webpack-dev-server。

## Warning

使用 `WebpackDevServer` 时, 不能使用第二个 `compiler` 参数 (一个回调)。

## Warning

请注意, 当[导出多个配置对象](#)时, 只会使用 `devServer` 的第一个配置选项, 并将其应用于所有的配置当中。

## Tip

如果你碰到了问题, 请将路由导航至 `/webpack-dev-server` 将会为你展示服务文件的位置。例如: `http://localhost:9000/webpack-dev-server`。

## Tip

如果你需要手动重新编译 `bundle`, 将路由导航至 `/invalidate` 使当前编译的 `bundle` 无效, 并通过 `webpack-dev-middleware` 为你重新编译。根据你的配置, URL 可能看起来像 `http://localhost:9000/invalidate`。

## Tip



当启动本地服务的时候 HTML 模板是必须提供的，通常是 `index.html`。确保将脚本引用添加到 HTML 中，`webpack-dev-server` 不会自动注入它们。

## Usage via CLI

你可以通过 CLI 调用 `webpack-dev-server`，方式是：

```
npx webpack serve
```

CLI 配置项列表可以在 [这里](#) 查询。

## Usage via API

虽然推荐通过 CLI 运行 `webpack-dev-server`，但是你也可以通过 API 来启用服务器。

查看相关的 [webpack-dev-server API 文档](#)。

## devServer.allowedHosts

'auto' | 'all' [string]

该选项允许将允许访问开发服务器的服务列入白名单。

### webpack.config.js

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

模仿 django 的 `ALLOWED_HOSTS`，用 `.` 作为子域通配符。`.host.com` 会与 `host.com`，`www.host.com` 以及 `host.com` 等其他任何其他子域匹配。

### webpack.config.js

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

```
// this achieves the same effect as the first example
// with the bonus of not having to update your config
// if new subdomains need to access the dev server
allowedHosts: ['.host.com', 'host2.com'],
},
};
```

通过 CLI 使用:

```
npx webpack serve --allowed-hosts .host.com --allowed-hosts host2.com
```

当设置为 'all' 时会跳过 host 检查。**并不推荐这样做**，因为不检查 host 的应用程序容易受到 DNS 重绑定攻击。

### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    allowedHosts: 'all',
  },
};
```

通过 CLI 使用:

```
npx webpack serve --allowed-hosts all
```

当设置为 'auto' 时，此配置项总是允许 localhost、host 和 `client.webSocketURL.hostname` :

### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    allowedHosts: 'auto',
  },
};
```

通过 CLI 使用:

```
npx webpack serve --allowed-hosts auto
```

## devServer.bonjour

boolean = false    object

这个配置用于在启动时通过 [ZeroConf](#) 网络广播你的开发服务器。

#### webpack.config.js

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

通过命令行使用:

```
npx webpack serve --bonjour
```

如需禁用:

```
npx webpack serve --no-bonjour
```

你也可以为 bonjour 设置 [自定义配置项](#), 例如:

#### webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    bonjour: {  
      type: 'http',  
      protocol: 'udp',  
    },  
  },  
};
```

## devServer.client

### logging

'log' | 'info' | 'warn' | 'error' | 'none' | 'verbose'

允许在浏览器中设置日志级别, 例如在重载之前, 在一个错误之前或者 [热模块替换](#) 启用时。

#### webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    client: {
```

```
    logging: 'info',
  },
},
};
```

通过命令行使用:

```
npx webpack serve --client-logging info
```

## overlay

```
boolean = true  object: { errors boolean = true, warnings boolean = true }
```

当出现编译错误或警告时，在浏览器中显示全屏覆盖。

### webpack.config.js

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

通过命令行使用:

```
npx webpack serve --client-overlay
```

如需禁用:

```
npx webpack serve --no-client-overlay
```

如果你只想显示错误信息:

### webpack.config.js

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

通过命令行使用：

```
npx webpack serve --client-overlay-errors --no-client-overlay-warnings
```

## progress

boolean

在浏览器中以百分比显示编译进度。

**webpack.config.js**

```
module.exports = {  
  //...  
  devServer: {  
    client: {  
      progress: true,  
    },  
  },  
};
```

通过命令行使用：

```
npx webpack serve --client-progress
```

如需禁用：

```
npx webpack serve --no-client-progress
```

## reconnect

boolean = true    number

v4.4.0+

告诉 dev-server 它应该尝试重新连接客户端的次数。当为 `true` 时，它将无限次尝试重新连接。

**webpack.config.js**

```
module.exports = {  
  //...  
  devServer: {  
    client: {  
      reconnect: true,  
    },  
  },  
};
```

```
    },  
  };  
};
```

通过 CLI 使用:

```
npx webpack serve --client-reconnect
```

当设置为 `false` 时, 它将不会尝试连接。

```
module.exports = {  
  //...  
  devServer: {  
    client: {  
      reconnect: false,  
    },  
  },  
};
```

通过 CLI 使用:

```
npx webpack serve --no-client-reconnect
```

您还可以指定客户端应该尝试重新连接的确切次数。

```
module.exports = {  
  //...  
  devServer: {  
    client: {  
      reconnect: 5,  
    },  
  },  
};
```

通过 CLI 使用:

```
npx webpack serve --client-reconnect 5
```

## WebSocketTransport

`'ws' | 'sockjs' string`

该配置项允许我们为客户端单独选择当前的 `devServer` 传输模式, 或者提供自定义的客户端实现。这允许指定浏览器或其他客户端与 `devServer` 的通信方式。

### Tip

为 `WebSocketServer` 设置 `'ws'` 或者 `'sockjs'` 是一个设置 `devServer.client.webSocketTransport` 和 `devServer.webSocketServer` 的快捷方式。

### webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    client: {  
      webSocketTransport: 'ws',  
    },  
    webSocketServer: 'ws',  
  },  
};
```

通过命令行使用：

```
npm run webpack serve --client-web-socket-transport ws --web-socket-server ws
```

## Tip

在提供自定义客户端和服务端实现时，要确保它们彼此兼容，以成功通信。

创建一个自定义客户端实现，创建一个类拓展 `BaseClient`。

使用 `CustomClient.js` 的路径，一个自定义 `WebSocket` 客户端实现，连同兼容的 `'ws'` 服务器：

### webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    client: {  
      webSocketTransport: require.resolve('./CustomClient'),  
    },  
    webSocketServer: 'ws',  
  },  
};
```

使用自定义且兼容的 `WebSocket` 客户端和服务端实现：

### webpack.config.js

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

```
    client: {
      websocketTransport: require.resolve('./CustomClient'),
    },
    websocketServer: require.resolve('./CustomServer'),
  },
};
```

## websocketURL

string object

这个选项允许指定 URL 到 web socket 服务器（当你代理开发服务器和客户端脚本不总是知道连接到哪里时很有用）。

### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    client: {
      websocketURL: 'ws://0.0.0.0:8080/ws',
    },
  },
};
```

通过命令行使用：

```
npx webpack serve --client-web-socket-url ws://0.0.0.0:8080/ws
```

您还可以指定具有以下属性的对象：

- `hostname` : 告诉连接到 devServer 的客户端使用提供的 hostname 进行连接。
- `pathname` : 告诉连接到 devServer 的客户端使用提供的路径进行连接。
- `password` : 告诉连接到 devServer 的客户端使用提供的密码进行认证。
- `port` : 告诉连接到 devServer 的客户端使用提供的端口进行连接。
- `protocol` : 告诉连接到 devServer 的客户端使用提供的协议进行连接。
- `username` : 告诉连接到 devServer 的客户端使用提供的用户名进行认证。

### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    client: {
      websocketURL: {
        hostname: '0.0.0.0',
```



```
    pathname: '/ws',
    password: 'dev-server',
    port: 8080,
    protocol: 'ws',
    username: 'webpack',
  },
},
},
};
```

## Tip

要从浏览器中获取 `protocol / hostname / port` , 请使用 `WebSocketURL`:  
`'auto://0.0.0.0:0/ws'` 。

## devServer.compress

`boolean = true`

启用 `gzip compression`:

`webpack.config.js`

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

通过命令行使用:

```
npx webpack serve --compress
```

如需禁用

```
npx webpack serve --no-compress
```

## devServer.devMiddleware

`object`

为 `webpack-dev-middleware` 提供处理 webpack 资源的配置项。

## webpack.config.js

```
module.exports = {
  devServer: {
    devMiddleware: {
      index: true,
      mimeTypes: { phtml: 'text/html' },
      publicPath: '/publicPathForDevServe',
      serverSideRender: true,
      writeToDisk: true,
    },
  },
};
```

## devServer.http2

boolean

使用 [spdy](#) 提供 HTTP/2 服务。对于 Node 15.0.0 及更高版本，此选项将被忽略，因为 spdy 在这些版本中已被破坏。一旦 [Express](#) 支持，开发服务器将迁移到 Node 内置的 HTTP/2。

HTTP/2 带有自签名证书：

## webpack.config.js

```
module.exports = {
  //...
  devServer: {
    http2: true,
  },
};
```

通过 CLI 使用：

```
npx webpack serve --http2
```

禁用：

```
npx webpack serve --no-http2
```

通过 [https](#) 配置你自己的证书文件：

## webpack.config.js

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

module.exports = {
```

```
//...
devServer: {
  http2: true,
  https: {
    key: fs.readFileSync('/path/to/server.key'),
    cert: fs.readFileSync('/path/to/server.crt'),
    ca: fs.readFileSync('/path/to/ca.pem'),
  },
},
};
```

要通过 CLI 使用你的证书，请使用以下配置项：

```
npx webpack serve --http2 --https-key ./path/to/server.key --https-cert ./path/to/server.cr
```

## Warning

此配置项已被弃用，以支持 `devServer.server`。

## devServer.https

boolean object

默认情况下，开发服务器将通过 HTTP 提供服务。可以选择使用 HTTPS 提供服务：

### webpack.config.js

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

通过 CLI 使用：

```
npx webpack serve --https
```

禁用：

```
npx webpack serve --no-https
```

根据上述配置，将使用自签名证书，但是你也可以提供自己的证书：

### webpack.config.js

```
module.exports = {
  devServer: {
    https: {
      ca: './path/to/server.pem',
      pfx: './path/to/server.pfx',
      key: './path/to/server.key',
      cert: './path/to/server.crt',
      passphrase: 'webpack-dev-server',
      requestCert: true,
    },
  },
};
```

该对象直接传递到 Node.js HTTPS 模块，因此请参阅 [HTTPS documentation](#) 以获取更多信息。

要通过 CLI 使用自己的证书，请使用以下选项：

```
npx webpack serve --https-request-cert --https-key ./path/to/server.key --https-cert ./path/to/server.crt
```

webpack-dev-server >= v4.2.0 允许你设置额外的 [TLS 配置项](#)，比如 `minVersion`。同样你可以直接传递各自文件的内容：

### webpack.config.js

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

module.exports = {
  devServer: {
    https: {
      minVersion: 'TLSv1.1',
      key: fs.readFileSync(path.join(__dirname, './server.key')),
      pfx: fs.readFileSync(path.join(__dirname, './server.pfx')),
      cert: fs.readFileSync(path.join(__dirname, './server.crt')),
      ca: fs.readFileSync(path.join(__dirname, './ca.pem')),
      passphrase: 'webpack-dev-server',
      requestCert: true,
    },
  },
};
```

## Warning

不要同时指定 `https.ca` 与 `https.cacert` 配置项，如果指定的话将会使用 `https.ca`。`https.cacert` 已弃用并且将会在下个主要版本中被移除。

## Warning

该配置项已弃用，以支持 [devServer.server](#)。

## devServer.headers

array function object

为所有响应添加 headers:

**webpack.config.js**

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

你也可以传递一个数组:

**webpack.config.js**

```
module.exports = {  
  //...  
  devServer: {  
    headers: [  
      {  
        key: 'X-Custom',  
        value: 'foo',  
      },  
      {  
        key: 'Y-Custom',  
        value: 'bar',  
      },  
    ],  
  },  
};
```

你也可以传递一个函数:

```
module.exports = {  
  //...  
  devServer: {  
    headers: () => {  
      return { 'X-Bar': ['key1=value1', 'key2=value2'] };  
    },  
  },  
};
```

```
  },  
};
```

## devServer.historyApiFallback

boolean = false object

When using the [HTML5 History API](#), the `index.html` page will likely have to be served in place of any 404 responses. Enable `devServer.historyApiFallback` by setting it to `true` :

### webpack.config.js

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

通过 CLI 使用:

```
npx webpack serve --history-api-fallback
```

禁用:

```
npx webpack serve --no-history-api-fallback
```

通过提供一个对象, 这种行为可以通过像 `rewrites` 这样的配置项进一步控制:

### webpack.config.js

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

在你的路径中使用点 (与 Angular 相同) , 你可能需要使用 `disableDotRule` :

### webpack.config.js

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

For more options and information, see the [connect-history-api-fallback](#) documentation.

## devServer.host

'local-ip' | 'local-ipv4' | 'local-ipv6' string

指定要使用的 host。如果你想让你的服务器可以被外部访问，像这样指定：

### webpack.config.js

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

通过命令行使用：

```
npx webpack serve --host 0.0.0.0
```

这也适用于 IPv6：

```
npx webpack serve --host ::
```

## local-ip

Specifying `local-ip` as host will try to resolve the host option as your local IPv4 address if available, if IPv4 is not available it will try to resolve your local IPv6 address.

```
npx webpack serve --host local-ip
```

## local-ipv4

Specifying `local-ipv4` as host will try to resolve the host option as your local IPv4 address.

```
npx webpack serve --host local-ipv4
```

## local-ipv6

指定 local-ipv6 作为主机将尝试将主机选项解析为您的本地 IPv6 地址。

```
npx webpack serve --host local-ipv6
```

## devServer.hot

```
'only' boolean = true
```

启用 webpack 的 [热模块替换](#) 特性：

### webpack.config.js

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

通过命令行使用：

```
npx webpack serve --hot
```

如需禁用：

```
npx webpack serve --no-hot
```

启用热模块替换功能，在构建失败时不刷新页面作为回退，使用 `hot: 'only'`：

### webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    hot: 'only',  
  },  
};
```

通过命令行使用：

```
npx webpack serve --hot only
```



## Tip

从 webpack-dev-server v4 开始, HMR 是默认启用的。它会自动应用 `webpack.HotModuleReplacementPlugin`, 这是启用 HMR 所必需的。因此当 `hot` 设置为 `true` 或者通过 CLI 设置 `--hot`, 你不需要在你的 `webpack.config.js` 添加该插件。查看 [HMR concepts page](#) 以获取更多信息。

## devServer.ipc

`true` `string`

The Unix socket to listen to (instead of a `host` ).

将其设置为 `true` 将会监听 `/your-os-temp-dir/webpack-dev-server.sock` 的 socket:

### webpack.config.js

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

通过命令行使用:

```
npx webpack serve --ipc
```

你也可以这样监听一个不同的 socket:

### webpack.config.js

```
const path = require('path');  
  
module.exports = {  
  //...  
  devServer: {  
    ipc: path.join(__dirname, 'my-socket.sock'),  
  },  
};
```

## devServer.liveReload

```
boolean = true
```

默认情况下，当监听到文件变化时 dev-server 将会重新加载或刷新页面。为了 `liveReload` 能够生效，`devServer.hot` 配置项必须禁用或者 `devServer.watchFiles` 配置项必须启用。将其设置为 `false` 以禁用 `devServer.liveReload`：

### webpack.config.js

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

通过命令行使用：

```
npx webpack serve --live-reload
```

禁用该功能：

```
npx webpack serve --no-live-reload
```

## Warning

热加载仅对像 `web`、`webworker`、`electron-renderer` 和 `node-webkit` 的 `targets` 生效。

## devServer.magicHtml

```
boolean = true
```

v4.1.0+

Tell dev-server to enable/disable magic HTML routes (routes corresponding to your webpack output, for example `/main` for `main.js`).

### webpack.config.js

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

通过 CLI 使用:

```
npx webpack serve --magic-html
```

禁用:

```
npx webpack serve --no-magic-html
```

## devServer.onAfterSetupMiddleware

```
function (devServer)
```

提供服务器内部在所有其他中间件之后执行 自定义中间件的能力

**webpack.config.js**

```
module.exports = {  
  //...  
  devServer: {  
    onAfterSetupMiddleware: function (devServer) {  
      if (!devServer) {  
        throw new Error('webpack-dev-server is not defined');  
      }  
  
      devServer.app.get('/some/path', function (req, res) {  
        res.json({ custom: 'response' });  
      });  
    },  
  },  
};
```

### Warning

该配置项已弃用，以支持 [devServer.setupMiddlewares](#)。

## devServer.onBeforeSetupMiddleware

```
function (devServer)
```

提供在服务器内部执行所有其他中间件之前执行自定义中间件的能力。这可以用来定义自定义处理程序，例如:

**webpack.config.js**

```
module.exports = {
  //...
  devServer: {
    onBeforeSetupMiddleware: function (devServer) {
      if (!devServer) {
        throw new Error('webpack-dev-server is not defined');
      }

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

## Warning

该配置项已弃用，以支持 `devServer.setupMiddlewares`。

# devserver.onListening

```
function (devServer)
```

提供在 webpack-dev-server 开始监听端口连接时执行自定义函数的能力。

## webpack.config.js

```
module.exports = {
  //...
  devServer: {
    onListening: function (devServer) {
      if (!devServer) {
        throw new Error('webpack-dev-server is not defined');
      }

      const port = devServer.server.address().port;
      console.log('Listening on port:', port);
    },
  },
};
```

# devServer.open

```
boolean string [string] object [object]
```

告诉 dev-server 在服务器已经启动后打开浏览器。设置其为 `true` 以打开你的默认浏览器。

### webpack.config.js

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

通过命令行使用:

```
npx webpack serve --open
```

如需禁用:

```
npx webpack serve --no-open
```

在浏览器中打开指定页面:

### webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    open: ['/my-page'],  
  },  
};
```

通过命令行使用:

```
npx webpack serve --open /my-page
```

在浏览器中打开多个指定页面:

### webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    open: ['/my-page', '/another-page'],  
  },  
};
```

通过命令行使用:

```
npx webpack serve --open /my-page --open /another-page
```

提供要使用的浏览器名称，而不是默认名称：

### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    open: {
      app: {
        name: 'google-chrome',
      },
    },
  },
};
```

通过命令行使用：

```
npx webpack serve --open-app-name 'google-chrome'
```

该对象接收所有 `open` 配置项：

### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    open: {
      target: ['first.html', 'http://localhost:8080/second.html'],
      app: {
        name: 'google-chrome',
        arguments: ['--incognito', '--new-window'],
      },
    },
  },
};
```

## Tip

浏览器应用程序名称与平台相关。不要在可重用模块中硬编码它。例如，`'Chrome'` 在 macOS 是 `'Google Chrome'`，在 Linux 是 `'google-chrome'`，在 Windows 是 `'chrome'`。

## devServer.port

`'auto'` string number

指定监听请求的端口号：

## webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    port: 8080,  
  },  
};
```

通过命令行使用:

```
npx webpack serve --port 8080
```

port 配置项不能设置为 null 或者空字符串, 要想自动使用一个可用端口请使用 port: 'auto' :

## webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    port: 'auto',  
  },  
};
```

通过 CLI 使用:

```
npx webpack serve --port auto
```

# devServer.proxy

object [object, function]

当拥有单独的 API 后端开发服务器并且希望在同一域上发送 API 请求时, 代理某些 URL 可能会很有用。

开发服务器使用功能强大的 [http-proxy-middleware](#) 软件包。查看其 [documentation](#) 了解更多高级用法。请注意, http-proxy-middleware 的某些功能不需要 target 键, 例如 它的 router 功能, 但是仍然需要在此处的配置中包含 target , 否则 webpack-dev-server 不会将其传递给 http-proxy-middleware ) 。

使用后端在 localhost:3000 上, 可以使用它来启用代理:

## webpack.config.js

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

现在, 对 `/api/users` 的请求会将请求代理到 `http://localhost:3000/api/users`。

如果不希望传递 `/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, 将不接受在 HTTPS 上运行且证书无效的后端服务器。如果需要, 可以这样修改配置:

#### webpack.config.js

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

有时不想代理所有内容。可以基于函数的返回值绕过代理。

在该功能中, 可以访问请求, 响应和代理选项。

- 返回 `null` 或 `undefined` 以继续使用代理处理请求。
- 返回 `false` 会为请求产生 404 错误。



- 返回提供服务的路径，而不是继续代理请求。

例如。对于浏览器请求，想要提供 HTML 页面，但是对于 API 请求，想要代理它。可以执行以下操作：

#### 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';
          }
        },
      },
    },
  },
};
```

如果想将多个特定路径代理到同一目标，则可以使用一个或多个带有 `context` 属性的对象的数组：

#### webpack.config.js

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

请注意，默认情况下不会代理对 `root` 的请求。要启用根代理，应将 `devMiddleware.index` 选项指定为虚假值：

#### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    devMiddleware: {
      index: false, // specify to enable root proxying
    },
  },
};
```

```
    },
    proxy: {
      context: () => true,
      target: 'http://localhost:1234',
    },
  },
};
```

默认情况下，代理时会保留主机头的来源，可以将 `changeOrigin` 设置为 `true` 以覆盖此行为。在某些情况下，例如使用 [name-based virtual hosted sites](#)，它很有用。

### webpack.config.js

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

## devServer.server

'http' | 'https' | 'spdy' string object

v4.4.0+

允许设置服务器和配置项（默认为 'http'）。

### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    server: 'http',
  },
};
```

通过 CLI 使用：

```
npx webpack serve --server-type http
```

使用自签名证书通过 HTTPS 提供服务：

## webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    server: 'https',  
  },  
};
```

通过 CLI 使用:

```
npx webpack serve --server-type https
```

使用 [spdy](#) 使用自签名证书通过 HTTP/2 提供服务:

## webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    server: 'spdy',  
  },  
};
```

通过 CLI 使用:

```
npx webpack serve --server-type spdy
```

## Warning

该配置项在 Node 15.0.0 及以上的版本会被忽略，因为 [spdy](#) 在这些版本中不会正常工作。一旦 [Express](#) 支持 Node 内建 HTTP/2，dev server 会进行迁移。

使用对象语法提供自己的证书:

## webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    server: {  
      type: 'https',  
      options: {  
        ca: './path/to/server.pem',  
        pfx: './path/to/server.pfx',  
        key: './path/to/server.key',  
        cert: './path/to/server.crt',  
        passphrase: 'webpack-dev-server',  
        requestCert: true,  
      },  
    },  
  },  
};
```

```
    },  
    },  
  },  
};
```

通过 CLI 使用:

```
npx webpack serve --server-type https --server-options-key ./path/to/server.key --server-op
```

It also allows you to set additional [TLS options](#) like `minVersion` and you can directly pass the contents of respective files:

### webpack.config.js

```
const fs = require('fs');  
const path = require('path');  
  
module.exports = {  
  //...  
  devServer: {  
    server: {  
      type: 'https',  
      options: {  
        minVersion: 'TLSv1.1',  
        key: fs.readFileSync(path.join(__dirname, './server.key')),  
        pfx: fs.readFileSync(path.join(__dirname, './server.pfx')),  
        cert: fs.readFileSync(path.join(__dirname, './server.crt')),  
        ca: fs.readFileSync(path.join(__dirname, './ca.pem')),  
        passphrase: 'webpack-dev-server',  
        requestCert: true,  
      },  
    },  
  },  
};
```

### Warning

如果指定的 `server.options.ca` 将会被使用的话, 不要同时指定 `server.options.ca` 与 `server.options.cacert` 配置项。 `server.options.cacert` 已弃用并且会在下一个大版本中移除。

## devServer.setupExitSignals

boolean = true

允许在 `SIGINT` 和 `SIGTERM` 信号时关闭开发服务器和退出进程。

## webpack.config.js

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

## devServer.setupMiddlewares

```
function (middlewares, devServer)
```

v4.7.0+

提供执行自定义函数和应用自定义中间件的能力。

## webpack.config.js

```
module.exports = {  
  // ...  
  devServer: {  
    setupMiddlewares: (middlewares, devServer) => {  
      if (!devServer) {  
        throw new Error('webpack-dev-server is not defined');  
      }  
  
      devServer.app.get('/setup-middleware/some/path', (_, response) => {  
        response.send('setup-middlewares option GET');  
      });  
  
      middlewares.push({  
        name: 'hello-world-test-one',  
        // `path` is optional  
        path: '/foo/bar',  
        middleware: (req, res) => {  
          res.send('Foo Bar!');  
        },  
      });  
  
      middlewares.push((req, res) => {  
        res.send('Hello World!');  
      });  
  
      return middlewares;  
    },  
  },  
};
```

# devServer.static

boolean string [string] object [object]

该配置项允许配置从目录提供静态文件的选项（默认是 'public' 文件夹）。将其设置为 `false` 以禁用：

## webpack.config.js

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

通过 CLI 使用：

```
npx webpack serve --static
```

禁用该功能：

```
npx webpack serve --no-static
```

监听单个目录：

## webpack.config.js

```
module.exports = {  
  // ...  
  devServer: {  
    static: ['assets'],  
  },  
};
```

通过 CLI 使用：

```
npx webpack serve --static assets
```

监听多个进后台资源目录：

## webpack.config.js

```
module.exports = {  
  // ...  
  devServer: {  
    static: ['assets', 'css'],  
  },  
};
```

```
  },  
};
```

通过 CLI 使用:

```
npx webpack serve --static assets --static css
```

## directory

```
string = path.join(process.cwd(), 'public')
```

告诉服务器从哪里提供内容。只有在你希望提供静态文件时才需要这样做。

`static.publicPath` 将会被用来决定应该从哪里提供 bundle，并具有优先级。

### webpack.config.js

```
const path = require('path');  
  
module.exports = {  
  //...  
  devServer: {  
    static: {  
      directory: path.join(__dirname, 'public'),  
    },  
  },  
};
```

提供一个对象数组，以防你有多个静态资源文件夹：

### webpack.config.js

```
const path = require('path');  
  
module.exports = {  
  //...  
  devServer: {  
    static: [  
      {  
        directory: path.join(__dirname, 'assets'),  
      },  
      {  
        directory: path.join(__dirname, 'css'),  
      },  
    ],  
  },  
};
```

## Tip

推荐使用绝对路径。

## staticOptions

object

可以配置从 `static.directory` 提供静态文件的高级选项。关于可用配置项可以插件 [Express 文档](#)。

### webpack.config.js

```
module.exports = {  
  //...  
  devServer: {  
    static: {  
      staticOptions: {  
        redirect: true,  
      },  
    },  
  },  
};
```

## publicPath

string = '/' [string]

告诉服务器在哪个 URL 上提供 `static.directory` 的内容。例如为在 `/serve-public-path-url/manifest.json` 中的 `assets/manifest.json` 提供服务，你的配置项应该像下面这样：

### webpack.config.js

```
const path = require('path');  
  
module.exports = {  
  //...  
  devServer: {  
    static: {  
      directory: path.join(__dirname, 'assets'),  
      publicPath: '/serve-public-path-url',  
    },  
  },  
};
```

提供一个对象数组，以防你有多个静态文件夹：

### webpack.config.js



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

module.exports = {
  //...
  devServer: {
    static: [
      {
        directory: path.join(__dirname, 'assets'),
        publicPath: '/serve-public-path-url',
      },
      {
        directory: path.join(__dirname, 'css'),
        publicPath: '/other-serve-public-path-url',
      },
    ],
  },
};
```

## serveIndex

```
boolean object = { icons: true }
```

告诉开发服务器启用后使用 `serveIndex` 中间件。

`serveIndex` 中间件会在查看没有 `index.html` 文件的目录时生成目录列表。

### webpack.config.js

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

module.exports = {
  //...
  devServer: {
    static: {
      directory: path.join(__dirname, 'public'),
      serveIndex: true,
    },
  },
};
```

通过 CLI 使用:

```
npx webpack serve --static-serve-index
```

禁用该功能:

```
npx webpack serve --no-static-serve-index
```

# watch

boolean object

通过 `static.directory` 配置项告诉 dev-server 监听文件。默认启用，文件更改将触发整个页面重新加载。可以通过将 `watch` 设置为 `false` 禁用。

## webpack.config.js

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

module.exports = {
  //...
  devServer: {
    static: {
      directory: path.join(__dirname, 'public'),
      watch: false,
    },
  },
};
```

通过 CLI 使用:

```
npx webpack serve --static-watch
```

禁用该功能:

```
npx webpack serve --no-static-watch
```

可以配置高级选项以监听 `static.directory` 中的静态文件。关于可用选项可以查看 [chokidar](#) 文档。

## webpack.config.js

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

module.exports = {
  //...
  devServer: {
    static: {
      directory: path.join(__dirname, 'public'),
      watch: {
        ignored: '*.txt',
        usePolling: false,
      },
    },
  },
};
```

## devServer.watchFiles

string [string] object [object]

该配置项允许你配置 globs/directories/files 来监听文件变化。例如：

### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    watchFiles: ['src/**/*.php', 'public/**/*.php'],
  },
};
```

可以配置高级选项来监听文件。关于可用选项可以查看 [chokidar](#) 文档。

### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    watchFiles: {
      paths: ['src/**/*.php', 'public/**/*.php'],
      options: {
        usePolling: false,
      },
    },
  },
};
```

## devServer.webSocketServer

false | 'sockjs' | 'ws' string function object

该配置项允许我们选择当前的 web-socket 服务器或者提供自定义的 web-socket 服务器实现。

当前默认模式为 'ws'。该模式使用 [ws](#) 作为服务器，客户端中的 WebSockets。

### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    webSocketServer: 'ws',
  },
};
```

为了创建一个自定义服务端实现，可以创建一个拓展 `BaseServer` 的类。

使用 `CustomServer.js` 导出的类实现自定义 WebSocket 客户端并兼容 ws 服务端：

#### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    client: {
      websocketTransport: 'ws',
    },
    websocketServer: require.resolve('./CustomServer'),
  },
};
```

使用自定义且兼容的 WebSocket 客户端以及服务端实现：

#### webpack.config.js

```
module.exports = {
  //...
  devServer: {
    client: {
      websocketTransport: require.resolve('./CustomClient'),
    },
    websocketServer: require.resolve('./CustomServer'),
  },
};
```

## Cache

### cache

boolean object

缓存生成的 webpack 模块和 chunk，来改善构建速度。cache 会在 [开发 模式](#) 被设置成 `type: 'memory'` 而且在 [生产 模式](#) 中被禁用。cache: true 与 cache: { type: 'memory' } 配置作用一致。传入 false 会禁用缓存：

#### webpack.config.js

```
module.exports = {
  //...
  cache: false,
};
```

当将 `cache.type` 设置为 `'filesystem'` 是会增加更多的可配置项。

## cache.allowCollectingMemory

收集在反序列化期间分配的未使用的内存，仅当 `cache.type` 设置为 `'filesystem'` 时生效。这需要将数据复制到更小的缓冲区中，并有性能成本。

- Type: boolean
  - It defaults to `false` in production mode and `true` in development mode.
- 5.35.0+

### webpack.config.js

```
module.exports = {
  cache: {
    type: 'filesystem',
    allowCollectingMemory: true,
  },
};
```

## cache.buildDependencies

object

`cache.buildDependencies` 是一个针对构建的额外代码依赖的数组对象。webpack 将使用这些项和所有依赖项的哈希值来使文件系统缓存失效。

默认是 `webpack/lib` 来获取 webpack 的所有依赖项。

### Tip

推荐在 webpack 配置中设置 `cache.buildDependencies.config: [__filename]` 来获取最新配置以及所有依赖项。

### webpack.config.js

```
module.exports = {
  cache: {
    buildDependencies: {
      // This makes all dependencies of this file - build dependencies
      config: [__filename],
      // 默认情况下 webpack 与 loader 是构建依赖。
    },
  },
};
```

## cache.cacheDirectory

string

缓存的。默认为 `node_modules/.cache/webpack`。

`cache.cacheDirectory` 选项仅当 `cache.type` 被设置成 `'filesystem'` 才可用。

### webpack.config.js

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

module.exports = {
  //...
  cache: {
    type: 'filesystem',
    cacheDirectory: path.resolve(__dirname, '.temp_cache'),
  },
};
```

### Warning

最终的缓存目标是 `cache.cacheDirectory + cache.name` 的混合。

## cache.cacheLocation

string

缓存的路径。默认值为 `path.resolve(cache.cacheDirectory, cache.name)`。

### webpack.config.js

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

module.exports = {
  //...
  cache: {
    type: 'filesystem',
    cacheLocation: path.resolve(__dirname, '.test_cache'),
  },
};
```

## cache.cacheUnaffected

对未改变的模块进行缓存计算，只引用未改变的模块。它只能在 `cache.type` 值为 `'memory'` 时使用，除此之外，必须启用 `experiments.cacheUnaffected` 配置项。

- 类型: `boolean`
- `v5.54.0+`

#### webpack.config.js

```
module.exports = {  
  //...  
  cache: {  
    type: 'memory',  
    cacheUnaffected: true,  
  },  
};
```

## cache.compression

`false` | `'gzip'` | `'brotli'`

`5.42.0+`

用于缓存文件的压缩类型。 `development` 模式下默认为 `false` , `production` 模式下默认为 `'gzip'` 。

`cache.compression` 配置项仅在 `cache.type` 设为 `'filesystem'` 时可用。

#### webpack.config.js

```
module.exports = {  
  //...  
  cache: {  
    type: 'filesystem',  
    compression: 'gzip',  
  },  
};
```

## cache.hashAlgorithm

`string`

用于哈希生成的算法。详情请参阅 [Node.js crypto](#)。默认值为 `md4` 。

`cache.hashAlgorithm` 选项仅当 `cache.type` 设置成 `'filesystem'` 才可配置。

#### webpack.config.js

```
module.exports = {  
  //...  
  cache: {  
    type: 'filesystem',  
    hashAlgorithm: 'md4',  
  },  
};
```

```
    hashAlgorithm: 'md4',  
  },  
};
```

## cache.idleTimeout

number = 60000

时间以毫秒为单位。 `cache.idleTimeout` 表示缓存存储发生的时间间隔。

`cache.idleTimeout` 配置项仅在 `[ cache.type ] 'filesystem'` 时生效。

### webpack.config.js

```
module.exports = {  
  //..  
  cache: {  
    type: 'filesystem',  
    idleTimeout: 60000,  
  },  
};
```

## cache.idleTimeoutAfterLargeChanges

number = 1000

5.41.0+

以毫秒为单位。 `cache.idleTimeoutAfterLargeChanges` 是当检测到较大的更改时，缓存存储应在此之后发生的时间段。

`cache.idleTimeoutAfterLargeChanges` 仅在 `cache.type` 设为 `'filesystem'` 时可用。

### webpack.config.js

```
module.exports = {  
  //..  
  cache: {  
    type: 'filesystem',  
    idleTimeoutAfterLargeChanges: 1000,  
  },  
};
```

## cache.idleTimeoutForInitialStore

number = 5000



单位毫秒。 `cache.idleTimeoutForInitialStore` 是在初始缓存存储发生后的时间段。

`cache.idleTimeoutForInitialStore` 配置项仅在 `[ cache.type ] 'filesystem'` 时生效。

#### webpack.config.js

```
module.exports = {  
  //..  
  cache: {  
    type: 'filesystem',  
    idleTimeoutForInitialStore: 0,  
  },  
};
```

## cache.managedPaths

```
[string] = ['./node_modules']
```

### Warning

已迁移到 [snapshot.managedPaths](#)

`cache.managedPaths` 是仅托管路径的包管理器数组。webpack 将避免将他们进行哈希和时间戳处理，假设版本是唯一的，并将其用作快照（用于内存和文件系统缓存）。

## cache.maxAge

```
number = 5184000000
```

5.30.0+

允许未使用的缓存留在文件系统缓存中的时间（以毫秒为单位）；默认为一个月。

`cache.maxAge` 仅在 `cache.type` 设置为 `'filesystem'` 时生效。

#### webpack.config.js

```
module.exports = {  
  // ...  
  cache: {  
    type: 'filesystem',  
    maxAge: 5184000000,  
  },  
};
```

## cache.maxGenerations

number

5.30.0+

定义内存缓存中未使用的缓存项的生命周期。

- `cache.maxGenerations: 1`: 在一次编译中未使用的缓存被删除。
- `cache.maxGenerations: Infinity`: 缓存将永远保存。

`cache.maxGenerations` 配置项仅在 `cache.type` 设置为 `'memory'` 时有效。

### webpack.config.js

```
module.exports = {  
  // ...  
  cache: {  
    type: 'memory',  
    maxGenerations: Infinity,  
  },  
};
```

## cache.maxMemoryGenerations

### \$#cachemaxMemorygenerations\$

number

5.30.0+

定义内存缓存中未使用的缓存项的生命周期。

- `cache.maxMemoryGenerations: 0`: 持久化缓存不会使用额外的内存缓存。它只将项目缓存到内存中，直到它们被序列化到磁盘。一旦序列化，下一次读取将再次从磁盘反序列化它们。这种模式将最小化内存使用，但会带来性能成本。
- `cache.maxMemoryGenerations: 1`: 这将从内存缓存中清除已序列化且在至少一次编译中未使用的项。当再次使用它们时，它们将从磁盘反序列化。这种模式将最小化内存使用量，同时仍将活动项保留在内存缓存中。
- `cache.maxMemoryGenerations`: 大于 0 的小数字将为 GC 操作带来性能成本。它会随着数字的增加而降低。
- `cache.maxMemoryGenerations`: `development` 模式下默认为 10，`production` 模式下默认为 `Infinity`。

`cache.maxMemoryGenerations` 配置项仅在 `cache.type` 设置为 `'filesystem'` 时有效。

## webpack.config.js

```
module.exports = {
  // ...
  cache: {
    type: 'filesystem',
    maxMemoryGenerations: Infinity,
  },
};
```

## cache.memoryCacheUnaffected

对未改变的模块进行缓存计算，并且只引用内存中未改变的模块。它只能在 `cache.type` 值为 'filesystem' 时使用，除此之外，必须启用 `experiments.cacheUnaffected` 配置项。

- 类型： `boolean`
- v5.54.0+

## webpack.config.js

```
module.exports = {
  //...
  cache: {
    type: 'filesystem',
    memoryCacheUnaffected: true,
  },
};
```

## cache.name

string

缓存的名称。不同的名字会导致不同的共存的缓存。默认值为

`${config.name}-${config.mode}`。使用 `cache.name` 当你有多份配置的时候，是比较合理的因为会有配置会有独立的缓存。

`cache.name` 选项仅当 `cache.type` 被设置成 'filesystem' 的时候可进行配置。

## webpack.config.js

```
module.exports = {
  //...
  cache: {
    type: 'filesystem',
    name: 'AppBuildCache',
  },
};
```

## cache.profile

boolean = false

跟踪并记录各个 'filesystem' 缓存项的详细时间信息。

### webpack.config.js

```
module.exports = {  
  //...  
  cache: {  
    type: 'filesystem',  
    profile: true,  
  },  
};
```

## cache.store

string = 'pack': 'pack'

cache.store 告诉 webpack 什么时候将数据存放在文件系统中。

- 'pack' : 当编译器闲置时候, 将缓存数据都存放在一个文件中

cache.store 选项仅当 `cache.type` 设置成 'filesystem' 才可配置。

### Warning

pack 是 webpack 5.0.x 起唯一支持的类型

### webpack.config.js

```
module.exports = {  
  //...  
  cache: {  
    type: 'filesystem',  
    store: 'pack',  
  },  
};
```

## cache.type

string: 'memory' | 'filesystem'

将 `cache` 类型设置为内存或者文件系统。 `memory` 选项很简单，它告诉 webpack 在内存中存储缓存，不允许额外的配置：

#### webpack.config.js

```
module.exports = {  
  //...  
  cache: {  
    type: 'memory',  
  },  
};
```

## cache.version

```
string = ''
```

缓存数据的版本。不同版本不会允许重用缓存和重载当前的内容。当配置以一种无法重用缓存的方式改变时，要更新缓存的版本。这会让缓存失效。

`cache.version` 选项仅当 `cache.type` 设置成 `'filesystem'` 才可配置。

#### webpack.config.js

```
module.exports = {  
  //...  
  cache: {  
    type: 'filesystem',  
    version: 'your_version',  
  },  
};
```

### Warning

在具有不同选项的调用之间不要共享缓存。

## 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 = 'eval' false
```

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

**Tip**

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。

**Tip**

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)
performance	<b>build:</b> fastest
	<b>rebuild:</b> fastest
production	yes
quality	bundle
comment	Recommended choice for production builds with maximum performance.

devtool	eval
performance	<b>build:</b> fast
	<b>rebuild:</b> fastest
production	no
quality	generated
comment	Recommended choice for development builds with maximum performance.

devtool	eval-cheap-source-map
performance	<b>build:</b> ok
	<b>rebuild:</b> fast

production	no
quality	transformed
comment	Tradeoff choice for development builds.

devtool	eval-cheap-module-source-map
performance	<b>build:</b> slow  <b>rebuild:</b> fast
production	no
quality	original lines
comment	Tradeoff choice for development builds.

devtool	eval-source-map
performance	<b>build:</b> slowest  <b>rebuild:</b> ok
production	no
quality	original
comment	Recommended choice for development builds with high quality SourceMaps.

devtool	cheap-source-map
performance	<b>build:</b> ok  <b>rebuild:</b> slow
production	no
quality	transformed

devtool	cheap-module-source-map
performance	<b>build:</b> slow  <b>rebuild:</b> slow
production	no
quality	original lines

devtool	source-map
performance	<b>build:</b> slowest

	<b>rebuild:</b> slowest
<b>production</b>	yes
<b>quality</b>	original
<b>comment</b>	Recommended choice for production builds with high quality SourceMaps.

<b>devtool</b>	inline-cheap-source-map
<b>performance</b>	<b>build:</b> ok
	<b>rebuild:</b> slow
<b>production</b>	no
<b>quality</b>	transformed

<b>devtool</b>	inline-cheap-module-source-map
<b>performance</b>	<b>build:</b> slow
	<b>rebuild:</b> slow
<b>production</b>	no
<b>quality</b>	original lines

<b>devtool</b>	inline-source-map
<b>performance</b>	<b>build:</b> slowest
	<b>rebuild:</b> slowest
<b>production</b>	no
<b>quality</b>	original
<b>comment</b>	Possible choice when publishing a single file

<b>devtool</b>	eval-nosources-cheap-source-map
<b>performance</b>	<b>build:</b> ok
	<b>rebuild:</b> fast
<b>production</b>	no
<b>quality</b>	transformed
<b>comment</b>	source code not included



devtool	eval-nosources-cheap-module-source-map
performance	<b>build:</b> slow
	<b>rebuild:</b> fast
production	no
quality	original lines
comment	source code not included

devtool	eval-nosources-source-map
performance	<b>build:</b> slowest
	<b>rebuild:</b> ok
production	no
quality	original
comment	source code not included

devtool	inline-nosources-cheap-source-map
performance	<b>build:</b> ok
	<b>rebuild:</b> slow
production	no
quality	transformed
comment	source code not included

devtool	inline-nosources-cheap-module-source-map
performance	<b>build:</b> slow
	<b>rebuild:</b> slow
production	no
quality	original lines
comment	source code not included

devtool	inline-nosources-source-map
performance	<b>build:</b> slowest

	<b>rebuild:</b> slowest
<b>production</b>	no
<b>quality</b>	original
<b>comment</b>	source code not included

<b>devtool</b>	nosources-cheap-source-map
<b>performance</b>	<b>build:</b> ok
	<b>rebuild:</b> slow
<b>production</b>	no
<b>quality</b>	transformed
<b>comment</b>	source code not included

<b>devtool</b>	nosources-cheap-module-source-map
<b>performance</b>	<b>build:</b> slow
	<b>rebuild:</b> slow
<b>production</b>	no
<b>quality</b>	original lines
<b>comment</b>	source code not included

<b>devtool</b>	nosources-source-map
<b>performance</b>	<b>build:</b> slowest
	<b>rebuild:</b> slowest
<b>production</b>	yes
<b>quality</b>	original
<b>comment</b>	source code not included

<b>devtool</b>	hidden-nosources-cheap-source-map
<b>performance</b>	<b>build:</b> ok
	<b>rebuild:</b> slow
<b>production</b>	no
<b>quality</b>	transformed
<b>comment</b>	no reference, source code not included

devtool	hidden-nosources-cheap-module-source-map
performance	<b>build:</b> slow  <b>rebuild:</b> slow
production	no
quality	original lines
comment	no reference, source code not included

devtool	hidden-nosources-source-map
performance	<b>build:</b> slowest  <b>rebuild:</b> slowest
production	yes
quality	original
comment	no reference, source code not included

devtool	hidden-cheap-source-map
performance	<b>build:</b> ok  <b>rebuild:</b> slow
production	no
quality	transformed
comment	no reference

devtool	hidden-cheap-module-source-map
performance	<b>build:</b> slow  <b>rebuild:</b> slow
production	no
quality	original lines
comment	no reference

devtool	hidden-source-map
performance	<b>build:</b> slowest  <b>rebuild:</b> slowest

<b>production</b>	yes
<b>quality</b>	original
<b>comment</b>	no reference. Possible choice when using SourceMap only for error reporting purposes.

<b>shortcut</b>	performance: build
<b>explanation</b>	How is the performance of the initial build affected by the devtool setting?

<b>shortcut</b>	performance: rebuild
<b>explanation</b>	How is the performance of the incremental build affected by the devtool setting? Slow devtools might reduce development feedback loop in watch mode. The scale is different compared to the build performance, as one would expect rebuilds to be faster than builds.

<b>shortcut</b>	production
<b>explanation</b>	Does it make sense to use this devtool for production builds? It's usually no when the devtool has a negative effect on user experience.

<b>shortcut</b>	quality: bundled
<b>explanation</b>	You will see all generated code of a chunk in a single blob of code. This is the raw output file without any devtooling support

<b>shortcut</b>	quality: generated
<b>explanation</b>	You will see the generated code, but each module is shown as separate code file in browser devtools.

<b>shortcut</b>	quality: transformed
<b>explanation</b>	You will see generated code after the preprocessing by loaders but before additional webpack transformations. Only source lines will be mapped and column information will be discarded resp. not generated. This prevents

	setting breakpoints in the middle of lines which doesn't work together with minimizer.
<b>shortcut</b>	quality: original lines
<b>explanation</b>	You will see the original code that you wrote, assuming all loaders support SourceMapping. Only source lines will be mapped and column information will be discarded resp. not generated. This prevents setting breakpoints in the middle of lines which doesn't work together with minimizer.
<b>shortcut</b>	quality: original
<b>explanation</b>	You will see the original code that you wrote, assuming all loaders support SourceMapping.
<b>shortcut</b>	eval-* addition
<b>explanation</b>	generate SourceMap per module and attach it via eval. Recommended for development, because of improved rebuild performance. Note that there is a windows defender issue, which causes huge slowdown due to virus scanning.
<b>shortcut</b>	inline-* addition
<b>explanation</b>	inline the SourceMap to the original file instead of creating a separate file.
<b>shortcut</b>	hidden-* addition
<b>explanation</b>	no reference to the SourceMap added. When SourceMap is not deployed, but should still be generated, e. g. for error reporting purposes.
<b>shortcut</b>	nosources-* addition
<b>explanation</b>	source code is not included in SourceMap. This can be useful when the original files should be referenced (further config options needed).

## Tip

验证 devtool 名称时，我们期望使用某种模式，注意不要混淆 devtool 字符串的顺序，模式是： `[inline-|hidden-|eval-][nosources-][cheap-[module-]]source-map` .

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，通常希望更快速的 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。

## Tip

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 `//@ sourceMappingURL`。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。

`eval-cheap-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`。

`eval-cheap-module-source-map` - 类似 `eval-cheap-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。

### Warning

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。

Warning

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。

Warning

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。

Tip

如果默认的 webpack minimizer 被覆盖 (例如自定义 `terser-webpack-plugin` 选项), 请确保将其替换配置为 `sourceMap: true` 以启用 SourceMap 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 [string] false`

告知 webpack 为目标(target)指定一个环境。默认值为 `"browserslist"` , 如果没有找到 `browserslist` 的配置, 则默认为 `"web"`

## string

通过 `WebpackOptionsApply` , 可以支持以下字符串值:

选项	<code>async-node</code>
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	<code>electron-main</code>
描述	编译为 <a href="#">Electron</a> 主进程。



选项	electron-renderer
描述	编译为 <a href="#">Electron</a> 渲染进程，使用 <code>JsonTemplatePlugin</code> ，
选项	<code>FunctionModulePlugin</code> 来为浏览器环境提供目标，使用 <code>NodeTargetPlugin</code> 和 <code>ExternalsPlugin</code>
选项	为 CommonJS 和 Electron 内置模块提供目标。
选项	electron-preload
描述	编译为 <a href="#">Electron</a> 渲染进程，
选项	使用 <code>NodeTemplatePlugin</code> 且 <code>asyncChunkLoading</code> 设置为 <code>true</code> ， <code>FunctionModulePlugin</code> 来为浏览器提供目标，使用 <code>NodeTargetPlugin</code> 和 <code>ExternalsPlugin</code> 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 <code>require</code> 加载 chunks）
选项	node-webkit
描述	编译为 Webkit 可用，并且使用 <code>jsonp</code> 去加载分块。支持 Node.js 内置模块和 <a href="#">nw.gui</a>
选项	导入（实验性质）
选项	<code>nwjs[[X].Y]</code>
描述	等价于 <code>node-webkit</code>
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

选项	esX
描述	编译为指定版本的 ECMAScript。例如, es5, es2020

选项	browserslist
描述	从 browserslist-config 中推断出平台和 ES 特性 (如果 browserslist 可用, 其值则为默认)

例如, 当 `target` 设置为 `"electron-main"`, webpack 引入多个 electron 特定的变量。

可指定 node 或者 electron 的版本。上表中使用 `[[X].Y]` 表示。

#### webpack.config.js

```
module.exports = {  
  // ...  
  target: 'node12.18',  
};
```

它有助于确定可能用于生成运行时代码的 ES 特性 (所有的 chunk 和模块都被运行时代码所包裹)

### browserslist

如果一个项目有 browserslist 配置, 那么 webpack 将会使用它:

- 确定可用于运行时代码的 ES 特性。
- 推断环境 (例如: `last 2 node versions` 等价于 `target: node`, 并会进行一些 `output.environment` 设置)。

支持的 browserslist 值:

- `browserslist` - 使用自动解析的 browserslist 配置和环境 (从最近的 `package.json` 或 `BROWSERSLIST` 环境变量中获取, 具体请查阅 [browserslist 文档](#))
- `browserslist:modern` - 使用自动解析的 browserslist 配置中的 `modern` 环境
- `browserslist:last 2 versions` - 使用显式 browserslist 查询 (配置将被忽略)
- `browserslist:/path/to/config` - 明确指定 browserslist 配置路径
- `browserslist:/path/to/config:modern` - 明确指定 browserslist 的配置路径和环境

### [string]

当传递多个目标时, 将使用共同的特性子集:

## webpack.config.js

```
module.exports = {  
  // ...  
  target: ['web', 'es5'],  
};
```

Webpack 将生成 web 平台的运行时代码，并且只使用 ES5 相关的特性。

目前并不是所有的 target 都可以进行混合。

## webpack.config.js

```
module.exports = {  
  // ...  
  target: ['web', 'node'],  
};
```

此时会导致错误。webpack 暂时不支持 universal 的 target。

## false

如果上述列表中的预设 target 都不符合你的需求，你可以将 target 设置为 false，这将告诉 webpack 不使用任何插件。

## webpack.config.js

```
module.exports = {  
  // ...  
  target: false,  
};
```

或者可以使用你想要指定的插件

## webpack.config.js

```
const webpack = require('webpack');  
  
module.exports = {  
  // ...  
  target: false,  
  plugins: [  
    new webpack.web.JsonpTemplatePlugin(options.output),  
    new webpack.LoaderTargetPlugin('web'),  
  ],  
};
```

当没有提供 target 或 [environment](#) 特性的信息时，将默认使用 ES2015。

# 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 无法正常运行时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,  
};
```

### Tip

[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: 200,  
    poll: 1000,  
  },  
};
```

## watchOptions.aggregateTimeout

number = 200

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 string [string]

对于某些系统，监听大量文件会导致大量的 CPU 或内存占用。可以使用正则排除像 `node_modules` 如此庞大的文件夹：

### webpack.config.js

```
module.exports = {
  //...
  watchOptions: {
    ignored: /node_modules/,
  },
};
```

此外，还可以使用 glob 模式：

### webpack.config.js

```
module.exports = {
  //...
  watchOptions: {
    ignored: '**/node_modules',
  },
};
```

也可以使用多 glob 匹配模式：

### webpack.config.js

```
module.exports = {
  //...
  watchOptions: {
    ignored: ['**/files/**/*.js', '**/node_modules'],
  },
};
```

此外，你还可以指定一个绝对路径：

```
const path = require('path');
module.exports = {
  //...
  watchOptions: {
    ignored: [path.posix.resolve(__dirname, './ignored-dir')],
  },
};
```

当使用 glob 模式时，我们使用 [glob-to-regexp](#) 将其转为正则表达式，因此，在使用 `watchOptions.ignored` 的 glob 模式之前，请确保自己熟悉它。

## Tip

如果你使用 `require.context`，webpack 会监听你的整个目录。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后不会触发重新构建。

## 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
  }
};
```

## Tip

如果监听没生效，试试这个选项吧。这会对 VirtualBox、WSL、Containers 或者 Docker 有所帮助。在这些情况下，使用轮询间隔并忽略 `/node_modules/` 这样的大文件夹，以保持最小的 CPU 使用率。

## watchOptions.followSymlinks

根据软链接查找文件。这通常是不需要的，因为 webpack 已经使用 [resolve.symlinks](#) 解析了软链接。

- Type: boolean
- Example:

```
module.exports = {  
  //...  
  watchOptions: {  
    followSymlinks: true,  
  },  
};
```

## watchOptions.stdin

当 stdin 流结束时停止监听。

- 类型: `boolean`
- 示例:

```
module.exports = {  
  //...  
  watchOptions: {  
    stdin: true,  
  },  
};
```

## 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 `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。

## externals

```
string [string] object function RegExp
```



**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-slogkvB1K3V0kzAI8QITxV3VzpOnkeNVsKvtkYLMjfk="
  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，换句话说，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。

可以接受以下语法.....

## 字符串

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）。

另一方面，如果你想将一个符合 CommonJS 模块化规则类库外部化，你可以提供外联类库的类型以及类库的名称。

如果你想将 `fs-extra` 从输出的 bundle 中剔除并在运行时中引入它，你可以如下定义：

```
module.exports = {
  // ...
  externals: {
    'fs-extra': 'commonjs2 fs-extra',
  },
};
```

这样的做法会让任何依赖的模块都不变，正如以下所示的代码：

```
import fs from 'fs-extra';
```

会将代码编译成：

```
const fs = require('fs-extra');
```

## [string]

```
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。该例子会编译成 `require('./math').subtract`；

## 对象

### Warning

一个形如 `{ root, amd, commonjs, ... }` 的对象仅允许用于 `libraryTarget: 'umd'` 这样的配置.它不被允许 用于其它的 library targets 配置值.

```
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']`）。

## 函数

- `function ({ context, request, contextInfo, getResolve }, callback)`
- `function ({ context, request, contextInfo, getResolve }) => promise`  
5.15.0+

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)`。

函数接收两个入参：

- `ctx ( object )`：包含文件详情的对象。
  - `ctx.context ( string )`：包含引用的文件目录。
  - `ctx.request ( string )`：被请求引入的路径。
  - `ctx.contextInfo ( string )`：包含 issuer 的信息（如，layer）
  - `ctx.getResolve` 5.15.0+：获取当前解析器选项的解析函数。
- `callback ( function (err, result, type) )`：用于指明模块如何被外部化的回调函数

回调函数接收三个入参：

- `err ( Error )`：被用于表明在外部引用时是否会产生错误。如果有错误，这将会是唯一被用到的参数。
- `result ( string [string] object )`：描述外部化的模块。可以接受形如 `${type}${path}` 格式的字符串，或者其它标准化外部化模块格式，（ `string` ， `[string]` ， 或

`object` )。

- `type ( string )`: 可选的参数, 用于指明模块的类型 (如果它没在 `result` 参数中被指明) 。

作为例子, 要外部化所有匹配一个正则表达式的引入, 你可以像下面那样处理:

#### webpack.config.js

```
module.exports = {
  //...
  externals: [
    function ({ context, request }, callback) {
      if (/^yourregex$/.test(request)) {
        // 使用 request 路径, 将一个 commonjs 模块外部化
        return callback(null, 'commonjs ' + request);
      }

      // 继续下一步且不外部化引用
      callback();
    },
  ],
};
```

其它例子使用不同的模块格式:

#### webpack.config.js

```
module.exports = {
  externals: [
    function (ctx, callback) {
      // 该外部化的模块, 是一个 `commonjs2` 的模块, 且放在 `@scope/library` 目录中
      callback(null, '@scope/library', 'commonjs2');
    },
  ],
};
```

#### webpack.config.js

```
module.exports = {
  externals: [
    function (ctx, callback) {
      // 该外部化模块是一个全局变量叫作 `nameOfGlobal`.
      callback(null, 'nameOfGlobal');
    },
  ],
};
```

#### webpack.config.js

```
module.exports = {
  externals: [
    function (ctx, callback) {
      // 该外部化模块是一个在`@scope/library`模块里的命名导出 (named export)。
      callback(null, ['@scope/library', 'namedexport'], 'commonjs');
    },
  ],
};
```

### webpack.config.js

```
module.exports = {
  externals: [
    function (ctx, callback) {
      // 外部化模块是一个 UMD 模块
      callback(null, {
        root: 'componentsGlobal',
        commonjs: '@scope/components',
        commonjs2: '@scope/components',
        amd: 'components',
      });
    },
  ],
};
```

## RegExp

匹配给定正则表达式的每个依赖，都将从输出 bundle 中排除。

### webpack.config.js

```
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。

## 混用语法

有时你需要混用上面介绍的语法。这可以像以下这样操作：

### webpack.config.js

```
module.exports = {
  //...
  externals: [
    {
```

```

    // 字符串
    react: 'react',
    // 对象
    lodash: {
      commonjs: 'lodash',
      amd: 'lodash',
      root: '_', // indicates global variable
    },
    // 字符串数组
    subtract: ['./math', 'subtract'],
  },
  // 函数
  function ({ context, request }, callback) {
    if (/^yourregex$/.test(request)) {
      return callback(null, 'commonjs ' + request);
    }
    callback();
  },
  // 正则表达式
  /^(jquery|\$)$/i,
],
};

```

## Warning

如果你指定的 `externals` 未使用类型，则会使用[默认类型](#)。例如 `externals: { react: 'react' }` 会被替换成 `externals: { react: 'commonjs-module react' }`。

关于如何使用此 `externals` 配置的更多信息，请参考[如何编写 library](#)。

## byLayer

function object

按层指定 externals:

**webpack.config.js**

```

module.exports = {
  externals: {
    byLayer: {
      layer: {
        external1: 'var 43',
      },
    },
  },
};

```

# externalsType

```
string = 'var'
```

指定 externals 的默认类型。当 external 被设置为 `amd` , `umd` , `system` 以及 `jsonp` 时, `output.libraryTarget` 的值也应相同。例如, 你只能在 `amd` 库中使用 `amd` 的 externals。

支持的类型如下:

- `'amd'`
- `'amd-require'`
- `'assign'`
- `'commonjs'`
- `'commonjs-module'`
- `'global'`
- `'import'` - uses `import()` to load a native EcmaScript module (async module)
- `'jsonp'`
- `'module'`
- `'node-commonjs'`
- `'promise'` - 与 `'var'` 相同, 但是会 `await` 结果 (适用于 `async` 模块)
- `'self'`
- `'system'`
- `'script'`
- `'this'`
- `'umd'`
- `'umd2'`
- `'var'`
- `'window'`

## webpack.config.js

```
module.exports = {  
  //...  
  externalsType: 'promise',  
};
```

## externalsType.module

Specify the default type of externals as `'module'`. Webpack will generate code like `import * as X from '...'` for externals used in a module.

Make sure to enable `experiments.outputModule` first, otherwise webpack will throw errors.

### webpack.config.js

```
module.exports = {
  experiments: {
    outputModule: true,
  },
  externalsType: 'module',
};
```

## externalsType.node-commonjs

Specify the default type of externals as `'node-commonjs'`. Webpack will import `createRequire` from `'module'` to construct a require function for loading externals used in a module.

```
module.export = {
  experiments: {
    outputModule: true,
  },
  externalsType: 'node-commonjs',
};
```

## externalsType.script

Specify the default type of externals as `'script'`. Webpack will Load the external as a script exposing predefined global variables with HTML `<script>` element. The `<script>` tag would be removed after the script has been loaded.

### Syntax

```
module.exports = {
  externalsType: 'script',
  externals: {
    packageName: [
      'http://example.com/script.js',
      'global',
      'property',
      'property',
    ], // properties are optional
  },
};
```



```
  },  
};
```

如果你不打算定义任何熟悉，你可以使用简写形式：

```
module.exports = {  
  externalsType: 'script',  
  externals: {  
    packageName: 'global@http://example.com/script.js', // no properties here  
  },  
};
```

请注意，`output.publicPath` 不会被添加到提供的 URL 中。

示例

从CDN加载 `lodash`：

**webpack.config.js**

```
module.exports = {  
  // ...  
  externalsType: 'script',  
  externals: {  
    lodash: ['https://cdn.jsdelivr.net/npm/lodash@4.17.19/lodash.min.js', '_'],  
  },  
};
```

然后，代码中使用方式如下：

```
import _ from 'lodash';  
console.log(_.head([1, 2, 3]));
```

下面示例是针对上面示例新增了属性配置：

```
module.exports = {  
  // ...  
  externalsType: 'script',  
  externals: {  
    lodash: [  
      'https://cdn.jsdelivr.net/npm/lodash@4.17.19/lodash.min.js',  
      '_',  
      'head',  
    ],  
  },  
};
```

当你 `import 'loadsh'` 时，局部变量 `head` 和全局变量 `window._` 都会被暴露：

```
import head from 'lodash';
console.log(head([1, 2, 3])); // logs 1 here
console.log(window._.head(['a', 'b'])); // logs a here
```

Tip

当加载带有 HTML `<script>` 标签的代码时，webpack 的 runtime 将试图寻找一个已经存在的 `<script>` 标签，此标签需与 `src` 的属性相匹配，或者具有特定的 `data-webpack` 属性。对于 chunk 加载来说，`data-webpack` 属性的值为 `'[output.uniqueName]:chunk-[chunkId]'`，而 external 脚本的值为 `'[output.uniqueName]:[global]'`。

Tip

像 `output.chunkLoadTimeout`，`output.crossOriginLoading` 以及 `output.scriptType` 等选项也会对这种方式加载的 external 脚本产生影响。

externalsPresets

object

为特定的 target 启用 externals 的 preset。

选项	electron
描述	将 main 和预加载上下文中常见的 electron 内置模块视为 external 模块（如 electron，ipc 或 shell），使用时通过 require() 加载。
输入类型	boolean

选项	electronMain
描述	将 main 上下文中的 electron 内置模块视为 external 模块（如 app，ipc-main 或 shell），使用时通过 require() 加载。
输入类型	boolean

选项	electronPreload
描述	将预加载上下文的 electron 内置模块视为 external 模块（如 web-frame，ipc-

	renderer 或 shell ) , 使用时通过 require() 加载。
输入类型	boolean

选项	electronRenderer
描述	将 renderer 上下文的 electron 内置模块视为 external 模块 (如 web-frame 、 ipc-renderer 或 shell ) , 使用时通过 require() 加载。
输入类型	boolean

选项	node
描述	将 node.js 的内置模块视为 external 模块 (如 fs , path 或 vm ) , 使用时通过 require() 加载。
输入类型	boolean

选项	nwjs
描述	将 NW.js 遗留的 nw.gui 模块视为 external 模块, 使用时通过 require() 加载。
输入类型	boolean

选项	web
描述	将 http(s)://... 以及 std:... 视为 external 模块, 使用时通过 import 加载。 <b>(注意, 这将改变执行顺序, 因为 external 代码会在该块中的其他代码执行前被执行) 。</b>
输入类型	boolean

选项	webAsync
描述	将 'http(s)://...' 以及 'std:...' 的引用视为 external 模块, 使用时通过 async import() 加载。 <b>(注意, 此 external 类型为 async 模块, 它对执行会产生各种副作用) 。</b>
输入类型	boolean

Note that if you're going to output ES Modules with those node.js-related presets, webpack will set the default `externalsType` to `node-commonjs` which would use `createRequire` to

construct a require function instead of using `require()` .

### Example

使用 `node` 的 `preset` 不会构建内置模块，而会将其视为 `external` 模块，使用时通过 `require()` 加载。

#### webpack.config.js

```
module.exports = {  
  // ...  
  externalsPresets: {  
    node: true,  
  },  
};
```

## 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.assetFilter

function(assetFilename) => boolean

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。

## performance.hints

```
string = 'warning': 'error' | 'warning'  boolean: false
```

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.maxAssetSize

```
number = 250000
```

资源(asset)是从 webpack 生成的任何文件。此选项根据单个资源体积(单位: bytes), 控制 webpack 何时生成性能提示。

```
module.exports = {
  //...
  performance: {
    maxAssetSize: 100000,
  },
};
```

## performance.maxEntrypointSize

number = 250000

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。

```
module.exports = {
  //...
  performance: {
    maxEntrypointSize: 400000,
  },
};
```

# Node

这些选项可以配置是否 polyfill 或 mock 某些 [Node.js 全局变量](#)。

此功能由 webpack 内部的 [NodeStuffPlugin](#) 插件提供。

## Warning

从 webpack 5 开始, 你只能在 `node` 选项下配置 `global`、`__filename` 或 `__dirname`。如果需要在 webpack 5 下的 Node.js 中填充 `fs`, 请查阅 [resolve.fallback](#) 获取相关帮助。

## node

boolean: false    object

**webpack.config.js**

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

```
node: {  
  global: false,  
  __filename: false,  
  __dirname: false,  
},  
};
```

从 webpack 3.0.0 开始, `node` 选项可能被设置为 `false`, 以完全关闭 `NodeStuffPlugin` 插件。

## node.global

boolean 'warn'

### Tip

如果你正在使用一个需要全局变量的模块, 请使用 `ProvidePlugin` 替代 `global`。

关于此对象的准确行为, 请查看[Node.js 文档](#)。

选项:

- `true`: 提供 polyfill.
- `false`: 不提供任何 polyfill。代码可能会出现 `ReferenceError` 的崩溃。
- `'warn'`: 当使用 `global` 时展示一个警告。

## node.\_\_filename

boolean 'mock' | 'warn-mock' | 'eval-only'

选项:

- `true`: **输入**文件的文件名, 是相对于 `context` [选项](#)。
- `false`: webpack 不会更改 `__filename` 的代码。在 Node.js 环境中运行时, **出**文件的文件名。
- `'mock'`: value 填充为 `'index.js'`。
- `'warn-mock'`: 使用 `'/index.js'` 但是会展示一个警告。
- `'eval-only'`

# node.\_\_dirname

boolean 'mock' | 'warn-mock' | 'eval-only'

选项:

- true :**输入** 文件的目录名，是相对于 `context` 选项。
- false :webpack 不会更改 `__dirname` 的代码，这意味着你有常规 Node.js 中的 `__dirname` 的行为。在 Node.js 环境中运行时，**输出** 文件的目录名。
- 'mock' :value 填充为 `'/'` 。
- 'warn-mock' :使用 `'/'` 但是会显示一个警告。
- 'eval-only'

## Stats 对象

object string

`stats` 选项让你更精确地控制 bundle 信息该怎么显示。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。

Warning

在使用 Node.js API 时，此选项无效。你需要将统计配置项传递给 `stats.toString()` 和 `stats.toJson()` 调用。

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

## Stats Presets

webpack 有一些特定的预设选项给统计信息输出:

预设	'errors-only'
可选值	<i>none</i>
描述	只在发生错误时输出

预设	'errors-warnings'
----	-------------------



可选值	<i>none</i>
描述	只在发生错误或有新的编译时输出

预设	'minimal'
可选值	<i>none</i>
描述	只在发生错误或新的编译开始时输出

预设	'none'
可选值	false
描述	没有输出

预设	'normal'
可选值	true
描述	标准输出

预设	'verbose'
可选值	<i>none</i>
描述	全部输出

预设	'detailed'
可选值	<i>none</i>
描述	全部输出除了 chunkModules 和 chunkRootModules

预设	'summary'
可选值	<i>none</i>
描述	输出 webpack 版本，以及警告数和错误数

## Stats

你可以在统计输出里指定你想看到的信息。

Tip

所有在统计信息配置里的选项都是可选的。

## stats.all

当统计信息配置没被定义，则该值是一个回退值。它的优先级比本地的 webpack 默认值高。

```
module.exports = {  
  //...  
  stats: {  
    all: undefined,  
  },  
};
```

## stats.assets

boolean = true

告知 stats 是否展示资源信息。将 stats.assets 设置成 false 会禁用。

```
module.exports = {  
  //...  
  stats: {  
    assets: false,  
  },  
};
```

## stats.assetsSort

string = 'id'

告知 stats 基于给定的字段对资源进行排序。所有的 [排序字段](#) 都被允许作为 stats.assetsSort 的值。使用 ! 作为值的前缀以反转基于给定字段的排序结果。

```
module.exports = {  
  //...  
  stats: {  
    assetsSort: '!size',  
  },  
};
```

## stats.builtAt

boolean = true

告知 stats 是否添加构建日期与时间信息。将 stats.builtAt 设置成 false 来隐藏。

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

```
stats: {
  builtAt: false,
},
};
```

## stats.moduleAssets

boolean = true

告知 stats 是否添加模块内的资源信息。将 stats.moduleAssets 设置成 false 以隐藏。

```
module.exports = {
  //...
  stats: {
    moduleAssets: false,
  },
};
```

## stats.assetsSpace

number = 15

告诉 stats 应该显示多少个 asset 项目（将以组的方式折叠，以适应这个空间）。

```
module.exports = {
  //...
  stats: {
    assetsSpace: 15,
  },
};
```

## stats.modulesSpace

number = 15

告诉 stats 应该显示多少个模块项目（将以组的方式折叠，以适应这个空间）。

```
module.exports = {
  //...
  stats: {
    modulesSpace: 15,
  },
};
```

## stats.chunkModulesSpace

```
number = 10
```

告诉 `stats` 显示多少个 chunk 模块项目（将以组的方式折叠，以适应这个空间）。

```
module.exports = {  
  //...  
  stats: {  
    chunkModulesSpace: 15,  
  },  
};
```

## stats.nestedModules

```
boolean
```

告知 `stats` 是否添加嵌套在其他模块中的模块信息（比如模块联邦）。

```
module.exports = {  
  //...  
  stats: {  
    nestedModules: true,  
  },  
};
```

## stats.nestedModulesSpace

```
number = 10
```

告诉 `stats` 应该显示多少个嵌套模块的项目（将以组的方式折叠，以适应这个空间）。

```
module.exports = {  
  //...  
  stats: {  
    nestedModulesSpace: 15,  
  },  
};
```

## stats.cached

旧版的 `stats.cachedModules`。

## stats.cachedModules

```
boolean = true
```

告诉 `stats` 是否要缓存（非内置）模块的信息。

```
module.exports = {  
  //...  
  stats: {  
    cachedModules: false,  
  },  
};
```

## stats.runtimeModules

boolean = true

告诉 `stats` 是否要添加运行时模块的信息。

```
module.exports = {  
  //...  
  stats: {  
    runtimeModules: false,  
  },  
};
```

## stats.dependentModules

boolean

告诉 `stats` 是否要展示该 chunk 依赖的其他模块的 chunk 模块。

```
module.exports = {  
  //...  
  stats: {  
    dependentModules: false,  
  },  
};
```

## stats.groupAssetsByChunk

boolean

告诉 `stats` 是否按照 asset 与 chunk 的关系进行分组。

```
module.exports = {  
  //...  
  stats: {  
    groupAssetsByChunk: false,  
  },  
};
```

## stats.groupAssetsByEmitStatus

boolean

告诉 stats 是否按照 asset 的状态进行分组 (emitted, 对比 emit 或缓存) 。

```
module.exports = {
  //...
  stats: {
    groupAssetsByEmitStatus: false,
  },
};
```

## stats.groupAssetsByExtension

boolean

告诉 stats 是否根据它们的拓展名聚合静态资源。

```
module.exports = {
  //...
  stats: {
    groupAssetsByExtension: false,
  },
};
```

## stats.groupAssetsByInfo

boolean

告诉 stats 是否按照 asset 信息对 asset 进行分组 (immutable, development, hotModuleReplacement 等) 。

```
module.exports = {
  //...
  stats: {
    groupAssetsByInfo: false,
  },
};
```

## stats.groupAssetsByPath

boolean

告诉 `stats` 是否根据它们的路径聚合静态资源。

```
module.exports = {  
  //...  
  stats: {  
    groupAssetsByPath: false,  
  },  
};
```

## stats.groupModulesByAttributes

boolean

告诉 `stats` 是否按模块的属性进行分组 (`errors`, `warnings`, `assets`, `optional`, `orphan` 或者 `dependent`) 。

```
module.exports = {  
  //...  
  stats: {  
    groupModulesByAttributes: false,  
  },  
};
```

## stats.groupModulesByCacheStatus

boolean

告诉 `stats` 是否按模块的缓存状态进行分组（已缓存或者已构建并且可缓存）。

```
module.exports = {  
  //...  
  stats: {  
    groupModulesByCacheStatus: true,  
  },  
};
```

## stats.groupModulesByExtension

boolean

告诉 `stats` 是否按模块的拓展名进行分组。

```
module.exports = {  
  //...  
  stats: {  
    groupModulesByExtension: true,  
  },  
};
```

```
  },  
};
```

## stats.groupModulesByLayer

boolean

告诉 stats 是否按模块的 layer 进行分组。

```
module.exports = {  
  //...  
  stats: {  
    groupModulesByLayer: true,  
  },  
};
```

## stats.groupModulesByPath

boolean

告诉 stats 是否按模块的路径进行分组。

```
module.exports = {  
  //...  
  stats: {  
    groupModulesByPath: true,  
  },  
};
```

## stats.groupModulesByType

boolean

告诉 stats 是否按模块的类型进行分组。

```
module.exports = {  
  //...  
  stats: {  
    groupModulesByType: true,  
  },  
};
```

## stats.groupReasonsByOrigin

boolean



5.46.0+

Group `reasons` by their origin module to avoid large set of reasons.

```
module.exports = {  
  //...  
  stats: {  
    groupReasonsByOrigin: true,  
  },  
};
```

## stats.cacheAssets

boolean = true

告知 `stats` 是否添加关于缓存资源的信息。将 `stats.cacheAssets` 设置成 `false` 会告知 `stats` 仅展示被生成的文件 (并非被构建的模块)。

```
module.exports = {  
  //...  
  stats: {  
    cacheAssets: false,  
  },  
};
```

## stats.children

boolean = true

告知 `stats` 是否添加关于子模块的信息。

```
module.exports = {  
  //...  
  stats: {  
    children: false,  
  },  
};
```

## stats.chunks

boolean = true

告知 `stats` 是否添加关于 chunk 的信息。将 `stats.chunks` 设置为 `false` 会引发更少的输出。

```
module.exports = {  
  //...  
  stats: {  
    chunks: false,  
  },  
};
```

## stats.chunkGroups

boolean = true

告知 stats 是否添加关于 namedChunkGroups 的信息。

```
module.exports = {  
  //...  
  stats: {  
    chunkGroups: false,  
  },  
};
```

## stats.chunkModules

boolean = true

告知 stats 是否添加关于已构建模块和关于 chunk 的信息。

```
module.exports = {  
  //...  
  stats: {  
    chunkModules: false,  
  },  
};
```

## stats.chunkOrigins

boolean = true

告知 stats 是不添加关于 chunks 的来源和 chunk 合并的信息。

```
module.exports = {  
  //...  
  stats: {  
    chunkOrigins: false,  
  },  
};
```

## stats.chunksSort

```
string = 'id'
```

告知 `stats` 基于给定的字段给 `chunks` 排序。所有 [排序字段](#) 都被允许用于作为 `stats.chunksSort` 的值。使用 `!` 作为值里的前缀用以将基于给定字段排序的结果反转。

```
module.exports = {  
  //...  
  stats: {  
    chunksSort: 'name',  
  },  
};
```

## stats.context

```
string = '../src/'
```

设置上下文目录用以将文件请求信息变短。

```
module.exports = {  
  //...  
  stats: {  
    context: '../src/components/',  
  },  
};
```

## stats.colors

```
boolean = false  object
```

告知 `stats` 是否输出不同的颜色。

```
module.exports = {  
  //...  
  stats: {  
    colors: true,  
  },  
};
```

它也可用通过命令行的参数实现：

```
npx webpack --stats-colors
```

To disable:

```
npx webpack --no-stats-colors
```

你可以通过使用 [ANSI escape sequences](#) 指定你自己的命令行终端颜色。

```
module.exports = {  
  //...  
  colors: {  
    green: '\u001b[32m',  
  },  
};
```

## stats.depth

boolean = false

告知 stats 是否展示每个模块与入口文件的距离。

```
module.exports = {  
  //...  
  stats: {  
    depth: true,  
  },  
};
```

## stats.entrypoints

boolean = true string = 'auto'

告知 stats 是否展示入口文件与对应的文件 bundles。

```
module.exports = {  
  //...  
  stats: {  
    entrypoints: false,  
  },  
};
```

当 stats.entrypoints 被设置为 'auto' 时，webpack 将自动决定是否在 stats 输出中展示入口信息。

## stats.env

boolean = false

告知 stats 是否展示 --env 信息。

```
module.exports = {  
  //...  
  stats: {  
    env: true,  
  },  
};
```

## stats.orphanModules

boolean = false

告知 stats 是否隐藏 孤儿(orphan) 模块. 一个模块属于 孤儿(orphan) 如果它不被包含在任何一个 chunk 里。孤儿模块默认在 stats 中会被隐藏。

```
module.exports = {  
  //...  
  stats: {  
    orphanModules: true,  
  },  
};
```

## stats.errors

boolean = true

告知 stats 是否展示错误。

```
module.exports = {  
  //...  
  stats: {  
    errors: false,  
  },  
};
```

## stats.errorDetails

boolean string = "auto"

告知 stats 是否添加错误的详情。如果默认值为 'auto' , 当只有 2 个或更少的错误时, 它将显示错误详情。

```
module.exports = {  
  //...  
  stats: {  
    errorDetails: false,  
  },  
};
```

```
    },  
  };  
};
```

## stats.errorStack

boolean = true

告知 stats 是否展示错位的栈追踪信息。

```
module.exports = {  
  //...  
  stats: {  
    errorStack: false,  
  },  
};
```

## stats.excludeAssets

array = []: string | RegExp | function (assetName) => boolean string  
RegExp function (assetName) => boolean

告知 stats 排除掉匹配的资源信息。这个可以通过设置一个 字符串 ,一个 正则表达式 ,一个 函数 取得资源的名字作为入参且返回一个 布尔值 。 stats.excludeAssets 可以是一个包括上面任意一类型值的 数组 。

```
module.exports = {  
  //...  
  stats: {  
    excludeAssets: [  
      'filter',  
      /filter/,  
      (assetName) => assetName.contains('moduleA'),  
    ],  
  },  
};
```

## stats.excludeModules

array = []: string | RegExp | function (assetName) => boolean string  
RegExp function (assetName) => boolean boolean: false

告知 stats 排除掉匹配的资源信息。这个可以通过设置一个 字符串 ,一个 正则表达式 ,一个 函数 取得资源的名字作为入参且返回一个 布尔值 。 stats.excludeModules 可以是一个包括上面任意一类型值的 数组 。 stats.excludeModules 会与 stats.exclude 的配置值[进行合并](#)。

```
module.exports = {
  //...
  stats: {
    excludeModules: ['filter', /filter/, (moduleSource) => true],
  },
};
```

将 `stats.excludeModules` 设置为 `false` 会禁用以上的排除行为。

```
module.exports = {
  //...
  stats: {
    excludeModules: false,
  },
};
```

## stats.exclude

详参 [stats.excludeModules](#) .

## stats.hash

boolean = true

告知 `stats` 是否添加关于编译哈希值的信息。

```
module.exports = {
  //...
  stats: {
    hash: false,
  },
};
```

## stats.logging

string = 'info': 'none' | 'error' | 'warn' | 'info' | 'log' | 'verbose'  
boolean

告知 `stats` 是否添加日志输出。

- 'none' , false - 禁用日志
- 'error' - 仅显示错误
- 'warn' - 仅显示错误与告警
- 'info' - 显示错误, 告警与信息

- 'log' , true - 显示错误, 告警与信息, 日志, 组别, 清理。折叠组别会在折叠状态中被显示。
- 'verbose' - 输出所有日志除了调试与追踪。折叠组别会在扩展状态中被显示。

```
module.exports = {  
  //...  
  stats: {  
    logging: 'verbose',  
  },  
};
```

## stats.loggingDebug

array = []: string | RegExp | function (name) => boolean string RegExp  
function (name) => boolean

告知 stats 去包括特定的日志工具调试信息比如插件或加载器的日志工具。当 `stats.logging` 被设置为 `false` , `stats.loggingDebug` 配置会被忽略。

```
module.exports = {  
  //...  
  stats: {  
    loggingDebug: [  
      'MyPlugin',  
      /MyPlugin/,  
      /webpack/, // To get core logging  
      (name) => name.contains('MyPlugin'),  
    ],  
  },  
};
```

## stats.loggingTrace

boolean = true

启用错误, 告警与追踪的日志输出中的堆栈追踪。将 `stats.loggingTrace` 设置为 `false` 隐藏追踪。

```
module.exports = {  
  //...  
  stats: {  
    loggingTrace: false,  
  },  
};
```



## stats.modules

boolean = true

告知 `stats` 是否添加关于构建模块的信息。

```
module.exports = {
  //...
  stats: {
    modules: false,
  },
};
```

## stats.modulesSort

string = 'id'

告知 `stats` 基于给定的字段对资源进行排序。所有的 [排序字段](#) 都被允许作为 `stats.modulesSort` 的值。使用 `!` 作为值的前缀以反转基于给定字段的排序结果。

```
module.exports = {
  //...
  stats: {
    modulesSort: 'size',
  },
};
```

## stats.moduleTrace

boolean = true

告知 `stats` 展示依赖和告警/错误的来源。 `stats.moduleTrace` 从 webpack 2.5.0 起可用。

```
module.exports = {
  //...
  stats: {
    moduleTrace: false,
  },
};
```

## stats.optimizationBailout

boolean

告诉 `stats` 展示模块优化失效的原因。

```
module.exports = {  
  //...  
  stats: {  
    optimizationBailout: false,  
  },  
};
```

## stats.outputPath

boolean = true

告知 stats 展示 outputPath。

```
module.exports = {  
  //...  
  stats: {  
    outputPath: false,  
  },  
};
```

## stats.performance

boolean = true

告知 stats 当文件大小超过 `performance.maxAssetSize` 配置值时，展示性能提示。

```
module.exports = {  
  //...  
  stats: {  
    performance: false,  
  },  
};
```

## stats.preset

string boolean: false

为展示的信息类型设置 [预设值](#)。这对 [扩展统计信息行为](#) 非常有用。

```
module.exports = {  
  //...  
  stats: {  
    preset: 'minimal',  
  },  
};
```

将 `stats.preset` 的值设置为 `false` 告知 webpack 使用 `'none'` [统计信息预设值](#)。

## stats.providedExports

boolean = false

告知 `stats` 去展示模块的导出。

```
module.exports = {  
  //...  
  stats: {  
    providedExports: true,  
  },  
};
```

## stats.errorsCount

boolean = true

添加展示 `errors` 个数。

```
module.exports = {  
  //...  
  stats: {  
    errorsCount: false,  
  },  
};
```

## stats.warningsCount

boolean = true

添加展示 `warnings` 个数。

```
module.exports = {  
  //...  
  stats: {  
    warningsCount: false,  
  },  
};
```

## stats.publicPath

boolean = true

告知 `stats` 展示 `publicPath`。

```
module.exports = {
  //...
  stats: {
    publicPath: false,
  },
};
```

## stats.reasons

boolean = true

告知 `stats` 添加关于模块被引用的原因信息。

```
module.exports = {
  //...
  stats: {
    reasons: false,
  },
};
```

## stats.reasonsSpace

number

5.46.0+

Space to display `reasons` (groups will be collapsed to fit this space).

```
module.exports = {
  //...
  stats: {
    reasonsSpace: 1000,
  },
};
```

### Tip

The detailed preset limits `modulesSpace`, `assetsSpace`, and `reasonsSpace` to 1000 by default.

## stats.relatedAssets

boolean = false

告诉 `stats` 是否需添加与其他 `assets` 相关的信息（例如 `assets` 的 `SourceMaps`）。

```
module.exports = {  
  //...  
  stats: {  
    relatedAssets: true,  
  },  
};
```

## stats.source

boolean = false

告知 `stats` 去添加模块的源码。

```
module.exports = {  
  //...  
  stats: {  
    source: true,  
  },  
};
```

## stats.timings

boolean = true

告知 `stats` 添加时间信息。

```
module.exports = {  
  //...  
  stats: {  
    timings: false,  
  },  
};
```

## stats.ids

boolean = false

通知 `stats` 给 `module` 和 `chunk` 添加 `id`。

```
module.exports = {  
  //...  
  stats: {  
    ids: true,  
  },  
};
```

## stats.usedExports

boolean = false

告知 stats 是否展示模块用了哪些导出。

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

## stats.version

boolean = true

告知 stats 添加关于 webpack 版本的信息。

```
module.exports = {  
  //...  
  stats: {  
    version: false,  
  },  
};
```

## stats.chunkGroupAuxiliary

boolean = true

在 chunk 组中展示辅助 asset。

```
module.exports = {  
  //...  
  stats: {  
    chunkGroupAuxiliary: false,  
  },  
};
```

## stats.chunkGroupChildren

boolean = true

显示 chunk 组的子 chunk。（例如，预置（prefetched），预加载（preloaded）的 chunk 和 asset）。

```
module.exports = {
  //...
  stats: {
    chunkGroupChildren: false,
  },
};
```

## stats.chunkGroupMaxAssets

number

chunk 组中的 asset 数上限。

```
module.exports = {
  //...
  stats: {
    chunkGroupMaxAssets: 5,
  },
};
```

## stats.warnings

boolean = true

告知 stats 添加告警。

```
module.exports = {
  //...
  stats: {
    warnings: false,
  },
};
```

## stats.warningsFilter

array = []: string | RegExp | function (warning) => boolean string RegExp  
function (warning) => boolean

告知 stats 排除掉匹配的告警信息。这个可以通过设置一个 字符串，一个 正则表达式，一个 函数 取得资源的名字作为入参且返回一个 布尔值。stats.warningsFilter 可以是一个包括上面任意一类型值的 数组。

```
module.exports = {  
  //...  
  stats: {  
    warningsFilter: ['filter', /filter/, (warning) => true],  
  },  
};
```

## Warning

`stats.warningsFilter` 已被弃用, 请改用 `ignoreWarnings`。

## stats.chunkRelations

`boolean = false`

告知 `stats` 展示 chunk 的父 chunk, 孩子 chunk 和兄弟 chunk。

## 字段排序

对于 `assetsSort`, `chunksSort` 和 `modulesSort` 它们有几个可用的字段用于排序:

- `'id'` 是元素 (指资源, chunk 或模块, 下同) 的 id;
- `'name'` - 一个元素的名字, 在导引的时候被分配;
- `'size'` - 一个元素的大小, 单位字节 (bytes) ;
- `'chunks'` - 元素来源于哪些 chunks (例如, 一个 chunk 有多个子 chunks, - 子 chunks 会被基于主 chunk 组合到一起);
- `'errors'` - 元素组错误的数量;
- `'warnings'` - 元素中告警的数量;
- `'failed'` - 元素是被编译失败;
- `'cacheable'` - 元素是否被缓存;
- `'built'` - 资源是否被构建;
- `'prefetched'` - 资源是否被预拉取;
- `'optional'` - 资源是否可选;
- `'identifier'` - 元素的标识符;
- `'index'` - 元素加工指针;
- `'index2'`
- `'profile'`



- 'issuer' - 发起者(issuer)的标识符;
- 'issuerId' - 发起者(issuer)的 id;
- 'issuerName' - 发起者(issuer)的名字;
- 'issuerPath' - 一个完整的发起者(issuer)对象。基于这个字段排序没有现实的需要;

## 扩展统计信息行为

如果你想使用其中一个预定义的行为, 例如 'minimal', 但仍想重载一个或更多的规则: 请指定想要设置的 `stats.preset` 同时在后面添加自定义或额外的规则。

### webpack.config.js

```
module.exports = {  
  //..  
  stats: {  
    preset: 'minimal',  
    moduleTrace: true,  
    errorDetails: true,  
  },  
};
```

# 实验特性(Experiments)

## experiments

boolean: false

`experiments` 配置是在 webpack 5 中推出, 目的是为了给用户赋能去开启并试用一些实验的特性。

### Warning

由于实验特性具有相对宽松的语义版本, 可能会有重大的变更, 所以主确定你将 webpack 的版本固定为小版本号, 例如与其使用 `webpack: ~5.4.3` 不如使用 `webpack: ^5.4.3` 或者当使用 `experiments` 配置时使用版本锁定能力。

Available options:

- `asyncWebAssembly` : Support the new WebAssembly according to the [updated specification](#), it makes a WebAssembly module an async module. And it is enabled by default when `experiments.futureDefaults` is set to `true` .

- [backCompat](#)
- [buildHttp](#)
- [cacheUnaffected](#)
- [css](#)
- [futureDefaults](#)
- `layers` : Enable module and chunk layers.
- [lazyCompilation](#)
- [outputModule](#)
- `syncWebAssembly` : Support the old WebAssembly like in webpack 4.
- `topLevelAwait` : Support the [Top Level Await Stage 3 proposal](#), it makes the module an async module when `await` is used on the top-level. And it is enabled by default when [experiments.futureDefaults](#) is set to `true` .

## webpack.config.js

```
module.exports = {  
  //...  
  experiments: {  
    asyncWebAssembly: true,  
    buildHttp: true,  
    layers: true,  
    lazyCompilation: true,  
    outputModule: true,  
    syncWebAssembly: true,  
    topLevelAwait: true,  
  },  
};
```

## experiments.backCompat

为许多 webpack 4 api 启用后向兼容层，并发出弃用警告。

- 类型: `boolean`

```
module.exports = {  
  //...  
  experiments: {  
    backCompat: true,  
  },  
};
```

## experiments.buildHttp

When enabled, webpack can build remote resources that begin with the `http(s):` protocol.

- Type:
  - `(string | RegExp | ((uri: string) => boolean))[]`

A shortcut for `experiments.buildHttp.allowedUris`.

- `HttpUriOptions`

```
{
  allowedUris: (string|RegExp|(uri: string) => boolean)[],
  cacheLocation?: false | string,
  frozen?: boolean,
  lockfileLocation?: string,
  upgrade?: boolean
}
```

- Available: 5.49.0+

- Example

```
// webpack.config.js
module.exports = {
  //...
  experiments: {
    buildHttp: true,
  },
};
```

```
// src/index.js
import pMap1 from 'https://cdn.skypack.dev/p-map';
// with `buildHttp` enabled, webpack will build pMap1 just like a regular local module
console.log(pMap1);
```

## `experiments.buildHttp.allowedUris`

允许的 URI 列表。

- 类型: `(string|RegExp|(uri: string) => boolean)[]`
- 示例

```
// webpack.config.js
module.exports = {
  //...
  experiments: {
    buildHttp: {
      allowedUris: [
        'http://localhost:9990/',
        'https://raw.githubusercontent.com/',
      ],
    },
  },
};
```

```
    ],  
  },  
},  
};
```

## experiments.buildHttp.cacheLocation

Define the location for caching remote resources.

- Type
  - string
  - false
- Example

```
// webpack.config.js  
module.exports = {  
  //...  
  experiments: {  
    buildHttp: {  
      cacheLocation: false,  
    },  
  },  
};
```

By default webpack would use `<compiler-name>webpack.lock.data/` for caching, but you can disable it by setting its value to `false`.

Note that you should commit files under `experiments.buildHttp.cacheLocation` into a version control system as no network requests will be made during the `production` build.

## experiments.buildHttp.frozen

Freeze the remote resources and lockfile. Any modification to the lockfile or resource contents will result in an error.

- Type: boolean

## experiments.buildHttp.lockfileLocation

Define the location to store the lockfile.

- Type: string

By default webpack would generate a `<compiler-name>webpack.lock file>`. Make sure to commit it into a version control system. During the `production` build, webpack will build those modules beginning with `http(s):` protocol from the lockfile and caches under [experiments.buildHttp.cacheLocation](#).

## experiments.buildHttp.upgrade

Detect changes to remote resources and upgrade them automatically.

- Type: `boolean`

## experiments.css

启用原生 CSS 支持。请注意该实验特性仍处于开发状态并且将会在 webpack v6 中默认启用，你可以在 [GitHub](#) 中跟踪进度。

- Type: `boolean`

## experiments.cacheUnaffected

Enable additional in-memory caching of modules which are unchanged and reference only unchanged modules.

- Type: `boolean`

Defaults to the value of `futureDefaults` .

## experiments.futureDefaults

使用下一个 webpack 主版本的默认值，并在任何有问题的地方显示警告。

### webpack.config.js

```
module.exports = {  
  //...  
  experiments: {  
    futureDefaults: true,  
  },  
};
```

## experiments.lazyCompilation

Compile entrypoints and dynamic `import` s only when they are in use. It can be used for either Web or Node.js.

- Type
  - `boolean`
  - `object`

```

{
  // define a custom backend
  backend?: ((
    compiler: Compiler,
    callback: (err?: Error, api?: BackendApi) => void
  ) => void)
  | ((compiler: Compiler) => Promise<BackendApi>)
  | {
    /**
     * A custom client.
     */
    client?: string;

    /**
     * Specify where to listen to from the server.
     */
    listen?: number | ListenOptions | ((server: typeof Server) => void);

    /**
     * Specify the protocol the client should use to connect to the server.
     */
    protocol?: "http" | "https";

    /**
     * Specify how to create the server handling the EventSource requests.
     */
    server?: ServerOptionsImport | ServerOptionsHttps | (() => typeof Server);
  },
  entries?: boolean,
  imports?: boolean,
  test?: string | RegExp | ((module: Module) => boolean)
}

```

- `backend` : Customize the backend.
- `entries` : Enable lazy compilation for entries.
- `imports` 5.20.0+ : Enable lazy compilation for dynamic imports.
- `test` 5.20.0+ : Specify which imported modules should be lazily compiled.

- Available: 5.17.0+

- Example 1:

```

module.exports = {
  // ...
  experiments: {
    lazyCompilation: true,
  },
};

```

- Example 2:

```
module.exports = {
  // ...
  experiments: {
    lazyCompilation: {
      // disable lazy compilation for dynamic imports
      imports: false,

      // disable lazy compilation for entries
      entries: false,

      // do not lazily compile moduleB
      test: (module) => !/moduleB/.test(module.nameForCondition()),
    },
  },
};
```

## experiments.outputModule

boolean

Once enabled, webpack will output ECMAScript module syntax whenever possible. For instance, `import()` to load chunks, ESM exports to expose chunk data, among others.

```
module.exports = {
  experiments: {
    outputModule: true,
  },
};
```

## 其它选项

这里是 webpack 支持的其它选项。

### Warning

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 boolean: false

设置 `require.amd` 或 `define.amd` 的值。设置 `amd` 为 `false` 会禁用 webpack 的 AMD 支持。

## webpack.config.js

```
module.exports = {  
  //...  
  amd: {  
    jQuery: true,  
  },  
};
```

某些流行的模块是按照 AMD 规范编写的，最引人注目的 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 = false
```

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。

### Warning

避免在 `watch` 模式下使用 `bail` 选项，因为这将会致使 webpack 在发现错误时尽快退出。

## dependencies

```
[string]
```

一个 `name` 列表，定义它所依赖的所有兄弟 (sibling) 配置。需要首先编译依赖的配置。



在 watch 模式下，当出现以下情况时，依赖项将使编译器失效：

1. 依赖项发生变化
2. 依赖项当前正在编译或者处于无效状态

请记住，在完成依赖项编译之前，不会编译此配置。

### webpack.config.js

```
module.exports = [  
  {  
    name: 'client',  
    target: 'web',  
    // ...  
  },  
  {  
    name: 'server',  
    target: 'node',  
    dependencies: ['client'],  
  },  
];
```

## ignoreWarnings

[RegExp, function (WebpackError, Compilation) => boolean, {module?: RegExp, file?: RegExp, message?: RegExp}]

告诉 webpack 忽略掉特定的警告。类型可以是 RegExp，可以是自定义 function。如果类型为函数，可基于原始 warning 来选择性展示警告，其参数分别为 WebpackError 和 Compilation，且返回值为 boolean。还可以包含以下属性的 object：

- file：类型为 RegExp，用于选择出现警告的源文件。
- message：类型为 RegExp，用于选择警告的内容。
- module：类型为 RegExp，用于选择警告来源的模块。

ignoreWarnings 必须是上述任意或所有类型组成的 array。

```
module.exports = {  
  //...  
  ignoreWarnings: [  
    {  
      module: /module2\.js\?[34]/, // A RegExp  
    },  
    {  
      module: /[13]/,  
      message: /homepage/,  
    },  
  ],  
};
```

```
    /warning from compiler/,  
    (warning) => true,  
  ],  
};
```

## infrastructureLogging

用于基础设施水平的日志选项。

```
object = {}
```

### appendOnly

5.31.0+

boolean

将内容追加到现有输出中，而非更新现有输出，这对于展示状态信息来说非常有用。此选项仅在未提供自定义 `console` 的情况下使用。

#### webpack.config.js

```
module.exports = {  
  //...  
  infrastructureLogging: {  
    appendOnly: true,  
    level: 'verbose',  
  },  
  plugins: [  
    (compiler) => {  
      const logger = compiler.getInfrastructureLogger('MyPlugin');  
      logger.status('first output'); // this line won't be overridden with `appendOnly` en  
      logger.status('second output');  
    },  
  ],  
};
```

## colors

5.31.0+

boolean

为基础设施日志启用带有颜色的输出。此选项仅在未提供自定义 `console` 的情况下使用。

#### webpack.config.js

```
module.exports = {
  //...
  infrastructureLogging: {
    colors: true,
    level: 'verbose',
  },
  plugins: [
    (compiler) => {
      const logger = compiler.getInfrastructureLogger('MyPlugin');
      logger.log('this output will be colorful');
    },
  ],
};
```

## console

5.31.0+

Console

为基础设施日志提供自定义方案。

**webpack.config.js**

```
module.exports = {
  //...
  infrastructureLogging: {
    console: yourCustomConsole(),
  },
};
```

## debug

string boolean = false    RegExp    function(name) => boolean    [string, RegExp, function(name) => boolean]

开启特定日志比如插件(plugins)和加载器(loaders)的调试信息。与 `stats.loggingDebug` 选项类似但仅仅对于基础设施而言。默认为 `false`。

**webpack.config.js**

```
module.exports = {
  //...
  infrastructureLogging: {
    level: 'info',
    debug: ['MyPlugin', /MyPlugin/, (name) => name.contains('MyPlugin')],
  },
};
```

## level

```
string = 'info' : 'none' | 'error' | 'warn' | 'info' | 'log' | 'verbose'
```

开启基础设施日志输出。与 `stats.logging` 选项类似但仅仅是对于基础设施而言。默认值为 `'info'`。

可能的取值：

- `'none'` - 禁用日志
- `'error'` - 仅仅显示错误
- `'warn'` - 仅仅显示错误与告警
- `'info'` - 显示错误、告警与信息
- `'log'` - 显示错误、告警，信息，日志信息，组别，清楚。收缩的组别会在收缩的状态中被显示。
- `'verbose'` - 输出所有日志除了调试与追踪。收缩的组别会在扩展的状态中被显示。

### webpack.config.js

```
module.exports = {  
  //...  
  infrastructureLogging: {  
    level: 'info',  
  },  
};
```

## stream

5.31.0+

```
NodeJS.WritableStream = process.stderr
```

用于日志输出的流。默认为 `process.stderr`。此选项仅在未提供自定义 `console` 的情况下使用。

### webpack.config.js

```
module.exports = {  
  //...  
  infrastructureLogging: {  
    stream: process.stderr,  
  },  
};
```

## Tip

在 TTY 流的情况下，会启用 `colors` 并禁用 `appendOnly`，反之亦然。

# loader

object

在 [loader 上下文](#) 中暴露自定义值。

例如，你可以在 loader 上下文中定义一个新变量：

**webpack.config.js**

```
module.exports = {  
  // ...  
  loader: {  
    answer: 42,  
  },  
};
```

然后使用 `this.answer` 在 loader 中获取该值：

**custom-loader.js**

```
module.exports = function (source) {  
  // ...  
  console.log(this.answer); // will log `42` here  
  return source;  
};
```

## Tip

你可以覆盖 loader 上下文中的属性，因为 webpack 会将所有定义在 `loader` 中的属性负责到 loader 上下文中。

# name

string

配置的名称。当加载不同的配置时会被使用。

**webpack.config.js**

```
module.exports = {  
  //...  
  name: 'admin-app',  
};
```

## parallelism

number = 100

限制并行处理的模块数量。可以用于调优性能或获取更可靠的性能分析结果。

## 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。

### Tip

使用 [StatsPlugin](#) 可以更好地控制生成的配置文件。

### Tip

与 `parallelism: 1` 混用以达到更好的结果。需要注意的是, 这样做也会减慢建造速度。

## 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

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

module.exports = {
  //...
  recordsInputPath: path.join(__dirname, 'records.json'),
  recordsOutputPath: path.join(__dirname, 'newRecords.json'),
};
```

## 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

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

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。

### Tip

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。

### Warning

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：

# snapshot

object

snapshot 配置项决定文件系统是如何创建和无效快照。

## webpack.config.js

```
const path = require('path');
module.exports = {
  // ...
  snapshot: {
    managedPaths: [path.resolve(__dirname, '../node_modules')],
    immutablePaths: [],
    buildDependencies: {
      hash: true,
      timestamp: true,
    },
    module: {
      timestamp: true,
    },
    resolve: {
      timestamp: true,
    },
    resolveBuildDependencies: {
      hash: true,
      timestamp: true,
    },
  },
};
```

## buildDependencies

object = { hash boolean = true, timestamp boolean = true }

使用持久化缓存时的依赖构建关系快照。

- hash : 比较内容哈希以确定无效 (比 timestamp 更昂贵, 但更改的频率较低)。
- timestamp : 比较 timestamps 以确定无效。

hash 与 timestamp 都是可选配置。

- { hash: true } : 对 CI 缓存很有帮助, 使用新的 checkout, 不需要保存时间戳, 并且使用哈希。
- { timestamp: true } : 对应本地开发缓存很有帮助。
- { timestamp: true, hash: true } : 对于以上提到的两者都很有帮助。首先比较时间戳, 这代价很低, 因为 webpack 不需要读取文件来计算它们的哈希值。仅当时间戳相同时



才会比较内容哈希，这对初始构建的性能影响很小。

## immutablePaths

[string]

由包管理器管理的路径数组，在其路径中包含一个版本或哈希，以便所有文件都是不可变的（immutable）。

## managedPaths

[string]

由包管理器管理的路径数组，可以信任它不会被修改。

## module

```
object = {hash boolean = true, timestamp boolean = true}
```

构建模块的快照。

- hash：比较内容哈希以判断无效。（比 timestamp 更昂贵，但更改的频率较低）。
- timestamp：比较时间戳以确定无效。

## resolve

```
object = {hash boolean = true, timestamp boolean = true}
```

解析请求的快照。

- hash：比较内容哈希以判断无效。（比 timestamp 更昂贵，但更改的频率较低）。
- timestamp：比较时间戳以确定无效。

## resolveBuildDependencies

```
object = {hash boolean = true, timestamp boolean = true}
```

使用持久缓存时用于解析构建依赖项的快照。

- hash：比较内容哈希以判断无效。（比 timestamp 更昂贵，但更改的频率较低）。
- timestamp：比较时间戳以确定无效。