**webpack优化打包流程**

**目录**

[一、 背景 2](#_Toc75511828)

[二、 Webpack解决的问题 2](#_Toc75511829)

[三、 使用webpack模块化打包 2](#_Toc75511830)

[四、 Webpack使用loader实现特殊资源加载 2](#_Toc75511831)

[五、 Webpack使用plugin扩展功能 2](#_Toc75511832)

[六、 Webpack的整体架构 3](#_Toc75511833)

[七、 Webpack的工作流程 3](#_Toc75511834)

[八、 Webpack的核心概念 3](#_Toc75511835)

摘要

关键词

正文

# 背景

只有在了解Webpack的构建流程和原理后，我们才能去对其全链路进行性能优化。

# Webpack解决的问题

实现前端项目的模块化，就是如何在前端项目中更高效地管理和维护每一个资源。

# Webpack的整体架构

# Webpack的工作流程

Webpack的运行流程是一个串行的过程，从启动到结束会依次执行以下流程：

1. 初始化参数：从配置文件和shell语句中读取与合并参数，得出最终的参数。
2. 开始编译：用上一步得到的参数初始化compiler对象，加载所有配置的插件，通过执行对象的run方法开始执行编译。
3. 确定入口：根据配置中的entry找到所有入口文件。
4. 编译模块：从入口文件出发，调用所有配置的loader对模块进行编译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理。
5. 完成模块编译：在经过第4步使用loader编译完所有模块后，得到了每个模块被编译后的最终内容以及它们之间的依赖关系。
6. 输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的chunk，再将每一个chunk转换成一个单独的文件加入输出列表中，这是可以修改输出内容的最后机会。
7. 输出完成：再确定好输出内容后，根据配置确定输出的路径和文件名，将文件的内容写入文件系统中。

# Webpack的核心概念

本质上，*webpack* 是一个现代 JavaScript 应用程序的*静态模块打包器(module bundler)*。当 webpack 处理应用程序时，它会递归地构建一个*依赖关系图(dependency graph)*，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 *bundle*。

1. Entry：入口，webpack打包之前从这个配置中获得所有文件
2. Module：模块，一个模块对应一个文件
3. Plugins：扩展插件，提供webpack更多功能
4. Output：输出，提供webpack输出出口，webpack会将多个chunk组合成的文件列表输出到对应的系统文件
5. Chunk：代码块，是由一个个代码模块组成，用于代码的合并和分割
6. Loader：代码编译工具，将原内容通过函数转化为新内容

# 使用webpack模块化打包

# Webpack使用loader实现特殊资源加载

## Loader的概念

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

## Loader特性

* loader 支持链式传递。能够对资源使用流水线(pipeline)。一组链式的 loader 将按照相反的顺序执行，从右向左或从下向上执行。loader 链中的第一个 loader 返回值给下一个 loader。在最后一个 loader，返回 webpack 所预期的 JavaScript。
* loader 可以是同步的，也可以是异步的。
* loader 运行在 Node.js 中，并且能够执行任何可能的操作。
* loader 接收查询参数。用于对 loader 传递配置。
* loader 也能够使用 options 对象进行配置。
* 除了使用 package.json 常见的 main 属性，还可以将普通的 npm 模块导出为 loader，做法是在 package.json 里定义一个 loader 字段。
* 插件(plugin)可以为 loader 带来更多特性。
* loader 能够产生额外的任意文件。

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

## 使用loader

  module: {

    rules: [

      {

        test: /\.css$/,

        use: [

          { loader: 'style-loader' },

          {

            loader: 'css-loader',

            options: {

              modules: true

            }

          },

          {

            test: /\.jsx?/,

            include: [

              path.resolve(\_\_dirname, 'src')

            ]

          }

        ]

      }

    ]

  }

## 匹配规则

loader 的匹配规则中有两个最关键的因素：一个是匹配条件，一个是匹配规则后的应用。

webpack 的规则提供了多种配置形式：

* { test: ... } 匹配特定条件
* { include: ... } 匹配特定路径
* { exclude: ... } 排除特定路径
* { and: [...] }必须匹配数组中所有条件
* { or: [...] } 匹配数组中任意一个条件
* { not: [...] } 排除匹配数组中所有条件

## 应用顺序

一个匹配规则中可以配置使用多个 loader，即一个模块文件可以经过多个 loader 的转换处理，执行顺序是从最后配置的 loader 开始，一步步往前。例如，对于上面的 less 规则配置，一个 style.less 文件会途径 less-loader、css-loader、style-loader 处理，成为一个可以打包的模块。

那如果多个 rule 匹配了同一个模块文件，loader 的应用顺序又是怎样的呢？

  module: {

    rules: [

      {

        test: /\.js$/,

        exclude: /node\_modules/,

        loader: "eslint-loader",

      },

      {

        test: /\.js$/,

        exclude: /node\_modules/,

        loader: "babel-loader",

      },

    ],

  }

这样无法法保证 eslint-loader 在 babel-loader 应用前执行。webpack 在 rules 中提供了一个 enforce 的字段来配置当前 rule 的 loader 类型，没配置的话是普通类型，我们可以配置 pre 或 post，分别对应前置类型或后置类型的 loader。

eslint-loader 要检查的是人工编写的代码，如果在 babel-loader 之后使用，那么检查的是 Babel 转换后的代码，所以必须在 babel-loader 处理之前使用。

还有一种行内 loader，即我们在应用代码中引用依赖时直接声明使用的 loader，如 const json = require('json-loader!./file.json') 这种。不建议在应用开发中使用这种 loader，后续我们还会再提到。

顾名思义，所有的 loader 按照**前置 -> 行内 -> 普通 -> 后置**的顺序执行。所以当我们要确保 eslint-loader 在 babel-loader 之前执行时，可以如下添加 enforce 配置：

  module: {

    rules: [

      {

        enforce: 'pre',//前置类型

        test: /\.js$/,

        exclude: /node\_modules/,

        loader: "eslint-loader",

      },

      {

        test: /\.js$/,

        exclude: /node\_modules/,

        loader: "babel-loader",

      },

    ],

  }

## babel-loader实现原理

Loader其实就是一个函数。

使用过es6的同学肯定对babel-loader有了解，就是将es6通过babel转化为es5，然后让浏览器去执行。

let babel;

try {

  babel = require("@babel/core");

} catch (err) {

  if (err.code === "MODULE\_NOT\_FOUND") {

    err.message +=

      "\n babel-loader@8 requires Babel 7.x (the package '@babel/core'). " +

      "If you'd like to use Babel 6.x ('babel-core'), you should install 'babel-loader@7'.";

  }

  throw err;

}

// Since we've got the reverse bridge package at @babel/core@6.x, give

// people useful feedback if they try to use it alongside babel-loader.

if (/^6\./.test(babel.version)) {

  throw new Error(

    "\n babel-loader@8 will not work with the '@babel/core@6' bridge package. " +

      "If you want to use Babel 6.x, install 'babel-loader@7'.",

  );

}

const { version } = require("../package.json");

const cache = require("./cache");

const transform = require("./transform");

const injectCaller = require("./injectCaller");

const schema = require("./schema");

const { isAbsolute } = require("path");

//加载所有的loader配置

const loaderUtils = require("loader-utils");

const validateOptions = require("schema-utils");

function subscribe(subscriber, metadata, context) {

  if (context[subscriber]) {

    context[subscriber](metadata);

  }

}

module.exports = makeLoader();

module.exports.custom = makeLoader;

function makeLoader(callback) {

  const overrides = callback ? callback(babel) : undefined;

  return function (source, inputSourceMap) {

    // Make the loader async

    const callback = this.async();

    loader.call(this, source, inputSourceMap, overrides).then(

      args => callback(null, ...args),

      err => callback(err),

    );

  };

}

async function loader(source, inputSourceMap, overrides) {

  const filename = this.resourcePath;

// this 是构建运行时的一些上下文信息

  let loaderOptions = loaderUtils.getOptions(this) || {};

  validateOptions(schema, loaderOptions, {

    name: "Babel loader",

  });

  if (loaderOptions.customize != null) {

    if (typeof loaderOptions.customize !== "string") {

      throw new Error(

        "Customized loaders must be implemented as standalone modules.",

      );

    }

    if (!isAbsolute(loaderOptions.customize)) {

      throw new Error(

        "Customized loaders must be passed as absolute paths, since " +

          "babel-loader has no way to know what they would be relative to.",

      );

    }

    if (overrides) {

      throw new Error(

        "babel-loader's 'customize' option is not available when already " +

          "using a customized babel-loader wrapper.",

      );

    }

    let override = require(loaderOptions.customize);

    if (override.\_\_esModule) override = override.default;

    if (typeof override !== "function") {

      throw new Error("Custom overrides must be functions.");

    }

    overrides = override(babel);

  }

  let customOptions;

  if (overrides && overrides.customOptions) {

    const result = await overrides.customOptions.call(this, loaderOptions, {

      source,

      map: inputSourceMap,

    });

    customOptions = result.custom;

    loaderOptions = result.loader;

  }

  // Deprecation handling

  if ("forceEnv" in loaderOptions) {

    console.warn(

      "The option `forceEnv` has been removed in favor of `envName` in Babel 7.",

    );

  }

  if (typeof loaderOptions.babelrc === "string") {

    console.warn(

      "The option `babelrc` should not be set to a string anymore in the babel-loader config. " +

        "Please update your configuration and set `babelrc` to true or false.\n" +

        "If you want to specify a specific babel config file to inherit config from " +

        "please use the `extends` option.\nFor more information about this options see " +

        "https://babeljs.io/docs/core-packages/#options",

    );

  }

  // Standardize on 'sourceMaps' as the key passed through to Webpack, so that

  // users may safely use either one alongside our default use of

  // 'this.sourceMap' below without getting error about conflicting aliases.

  if (

    Object.prototype.hasOwnProperty.call(loaderOptions, "sourceMap") &&

    !Object.prototype.hasOwnProperty.call(loaderOptions, "sourceMaps")

  ) {

    loaderOptions = Object.assign({}, loaderOptions, {

      sourceMaps: loaderOptions.sourceMap,

    });

    delete loaderOptions.sourceMap;

  }

  const programmaticOptions = Object.assign({}, loaderOptions, {

    filename,

    inputSourceMap: inputSourceMap || undefined,

    // Set the default sourcemap behavior based on Webpack's mapping flag,

    // but allow users to override if they want.

    sourceMaps:

      loaderOptions.sourceMaps === undefined

        ? this.sourceMap

        : loaderOptions.sourceMaps,

    // Ensure that Webpack will get a full absolute path in the sourcemap

    // so that it can properly map the module back to its internal cached

    // modules.

    sourceFileName: filename,

  });

  // Remove loader related options

  delete programmaticOptions.customize;

  delete programmaticOptions.cacheDirectory;

  delete programmaticOptions.cacheIdentifier;

  delete programmaticOptions.cacheCompression;

  delete programmaticOptions.metadataSubscribers;

  if (!babel.loadPartialConfig) {

    throw new Error(

      `babel-loader ^8.0.0-beta.3 requires @babel/core@7.0.0-beta.41, but ` +

        `you appear to be using "${babel.version}". Either update your ` +

        `@babel/core version, or pin you babel-loader version to 8.0.0-beta.2`,

    );

  }

  // babel.loadPartialConfigAsync is available in v7.8.0+

  const { loadPartialConfigAsync = babel.loadPartialConfig } = babel;

  const config = await loadPartialConfigAsync(

    injectCaller(programmaticOptions, this.target),

  );

  if (config) {

    let options = config.options;

    if (overrides && overrides.config) {

      options = await overrides.config.call(this, config, {

        source,

        map: inputSourceMap,

        customOptions,

      });

    }

    if (options.sourceMaps === "inline") {

      // Babel has this weird behavior where if you set "inline", we

      // inline the sourcemap, and set 'result.map = null'. This results

      // in bad behavior from Babel since the maps get put into the code,

      // which Webpack does not expect, and because the map we return to

      // Webpack is null, which is also bad. To avoid that, we override the

      // behavior here so "inline" just behaves like 'true'.

      options.sourceMaps = true;

    }

    const {

      cacheDirectory = null,

      cacheIdentifier = JSON.stringify({

        options,

        "@babel/core": transform.version,

        "@babel/loader": version,

      }),

      cacheCompression = true,

      metadataSubscribers = [],

    } = loaderOptions;

    let result;

    if (cacheDirectory) {

      result = await cache({

        source,

        options,

        transform,

        cacheDirectory,

        cacheIdentifier,

        cacheCompression,

      });

    } else {

      result = await transform(source, options);

    }

    // Availabe since Babel 7.12

    // https://github.com/babel/babel/pull/11907

    if (config.files) {

      config.files.forEach(configFile => this.addDependency(configFile));

    } else {

      // .babelrc.json

      if (typeof config.babelrc === "string") {

        this.addDependency(config.babelrc);

      }

      // babel.config.js

      if (config.config) {

        this.addDependency(config.config);

      }

    }

    if (result) {

      if (overrides && overrides.result) {

        result = await overrides.result.call(this, result, {

          source,

          map: inputSourceMap,

          customOptions,

          config,

          options,

        });

      }

      const { code, map, metadata } = result;

      metadataSubscribers.forEach(subscriber => {

        subscribe(subscriber, metadata, this);

      });

      return [code, map];

    }

  }

  // If the file was ignored, pass through the original content.

  return [source, inputSourceMap];

}

我们可以在 webpack 配置中直接使用路径来指定使用本地的 loader，或者在 loader 路径解析中加入本地开发 loader 的目录。看看配置例子：

// ...

module: {

    rules: [

      {

        test: /\.js$/,

        exclude: /node\_modules/,

        loader: path.resolve('./loader/index.js'), // 使用本地的 ./loader/index.js 作为 loader

      },

    ],

  },

  // 在 resolveLoader 中添加本地开发的 loaders 存放路径

  // 如果你同时需要开发多个 loader，那么这个方式会更加适合你

  resolveLoader: {

    modules: [

      'node\_modules',

      path.resolver(\_\_dirname, 'loaders')

    ],

  },

首先 loader 函数接受的参数是有三个的：content, map, meta。content 是模块内容，但不仅限于字符串，也可以是 buffer，例如一些图片或者字体等文件。map 则是 sourcemap 对象，meta 是其他的一些元数据。loader 函数单纯返回一个值，这个值是当成 content 去处理，但如果你需要返回 sourcemap 对象或者 meta 数据，甚至是抛出一个 loader 异常给 webpack 时，你需要使用 this.callback(err, content, map, meta) 来传递这些数据。

## 手写一个loader

使用Webpack往往离不开loader的安装配置，手写一个loader其实非常简单，类似手写一个功能函数，下面我们来实现一个替换字符串的loader

### 初始化项目

创建一个根目录mack-loader，此目录下 npm init -y生成默认的package.json文件 ,在文件中配置打包命令

"scripts"**:** {

"build"**:** "webpack"}

之后npm i -D webpack webpack-cli,安装完webpack，在根目录 创建配置文件webpack.config.js

**const** path **=** require('path')

module.exports **=** {

mode**:** 'development', *// 先设置为development，不压缩代码，方便调试* entry**:** {

main**:** './src/index.js'

},

output**:** {

filename**:** '[name].js',

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

}}

在根目录创建一个src目录，里面创建index.js,输入console.log('hello, world')

之后运行npm run build即可打包项目，初始化项目完成

### 编写 replace-loader

根目录创建loaders文件夹，里面创建replaceLoader.js

**const** loaderUtils **=** require('loader-utils');

module.exports **=** **function** (source) {

**const** options **=** loaderUtils.getOptions(**this**);

**const** result **=** source.replace('world', options.name);

**return** result;}

这里我们采用官方推荐的loader-utils读取options配置，也可用this.query获取配置对象，name是我们在loader配置项输入的字段名，source是源文件内容，最后需要返回，注意这里不能使用箭头函数，否则this指向会有错误，之后便可在webpack.config.js配置文件使用这个loader

module**:** {

rules**:** [{

test**:** /\.js$/,

use**:** {

loader**:** path.resolve(\_\_dirname, './loaders/replaceLoader.js'),

options**:** {

name**:** 'echo'

}

}

}]

}

效果是会把world替换为name中得字符串，npm run build 后在main.js里面则可以看到此效果

### loader 返回更多内容

官方文档[配置的 API 中](https://link.zhihu.com/?target=https://webpack.js.org/api/loaders%23thiscallback" \t "https://zhuanlan.zhihu.com/p/_blank)有loader的许多API ,除了this.query，常用的还有this.callback

**this**.callback(

err**:** Error **|** **null**,

content**:** string **|** Buffer,

sourceMap**?:** SourceMap, *// 可选参数，返回source-map* meta**?:** any *// 可选参数，返回meta*);

可使用此API替代return

**const** loaderUtils **=** require('loader-utils');

module.exports **=** **function** (source) {

**const** options **=** loaderUtils.getOptions(**this**);

**const** result **=** source.replace('world', options.name);

*// return result;* **this**.callback(**null**, result);}

### loader 中编写异步代码

在loader中编写异步代码需要用this.async，我们可以再实现一个异步loader，创建replaceLoderAsync.js

**const** loaderUtils **=** require('loader-utils');

module.exports **=** **function** (source) {

**const** options **=** loaderUtils.getOptions(**this**);

**const** callback **=** **this**.async()

setTimeout(() => {

**const** result **=** source.replace('world', options.name);

callback(**null**, result)

}, 1000)}

其中this.async返回的是this.callback，因此可以当做return来使用，将replaceLoder.js中的代码改为

module.exports **=** **function** (source) {

**const** result **=** source.replace('echo', 'world');

**this**.callback(**null**, result)}

我们实现先调用异步loader，将world改为echo，之后再调用同步loader将echo改为world，在配置文件的相应配置为

**const** path **=** require('path')

module.exports **=** {

mode**:** 'development', *// 先设置为development，不压缩代码，方便调试* entry**:** {

main**:** './src/index.js'

},

output**:** {

filename**:** '[name].js',

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

},

resolveLoader**:** {

*// 会依次在node\_modules、loaders文件夹中查找是否存在对应loader* modules**:** ['node\_modules', './loaders']

},

module**:** {

rules**:** [{

test**:** /\.js$/,

use**:** [{

loader**:** 'replaceLoader.js'

},{

loader**:** 'replaceLoaderAsync.js',

options**:** {

name**:** 'echo'

}

}]

}]

}}

之后运行npm run build即可在dist的main.js验证效果

### 编写loader的应用场景

1. 监控前端方法错误：可以自己编写loader检测业务代码中含有function关键字时自动用try...catch...包含代码块捕获错误，可以避免自己手写try...catch...导致的业务代码的臃肿
2. 实现网站的中英文替换：可以将文字用占位符包裹，检测到占位符则根据环境变量替换为中英文，伪代码如下

module.exports **=** **function** (source) {

**if**(Node全局变量 **===** '中文') {

source.replace('{{title}}', '中文标题')

} **else** {

source.replace('{{title}}', 'english title')

}

**const** result **=** source.replace('echo', 'world');

**this**.callback(**null**, result)}

## 常用第三方loader

### **文件**

* [raw-loader](https://www.webpackjs.com/loaders/raw-loader) 加载文件原始内容（utf-8）
* [val-loader](https://www.webpackjs.com/loaders/val-loader) 将代码作为模块执行，并将 exports 转为 JS 代码
* [url-loader](https://www.webpackjs.com/loaders/url-loader) 像 file loader 一样工作，但如果文件小于限制，可以返回 [data URL](https://tools.ietf.org/html/rfc2397)
* [file-loader](https://www.webpackjs.com/loaders/file-loader) 将文件发送到输出文件夹，并返回（相对）URL

### JSON

* [json-loader](https://www.webpackjs.com/loaders/json-loader) 加载 [JSON](http://json.org/) 文件（默认包含）
* [json5-loader](https://www.webpackjs.com/loaders/json5-loader) 加载和转译 [JSON 5](https://json5.org/) 文件
* cson-loader 加载和转译 [CSON](https://github.com/bevry/cson" \l "what-is-cson) 文件

### **转换编译(Transpiling)**

* [script-loader](https://www.webpackjs.com/loaders/script-loader) 在全局上下文中执行一次 JavaScript 文件（如在 script 标签），不需要解析
* [babel-loader](https://www.webpackjs.com/loaders/babel-loader) 加载 ES2015+ 代码，然后使用 [Babel](https://babeljs.io/) 转译为 ES5
* buble-loader 使用 [Bublé](https://buble.surge.sh/guide/) 加载 ES2015+ 代码，并且将代码转译为 ES5
* traceur-loader 加载 ES2015+ 代码，然后使用 [Traceur](https://github.com/google/traceur-compiler" \l "readme) 转译为 ES5
* [ts-loader](https://github.com/TypeStrong/ts-loader) 或 [awesome-typescript-loader](https://github.com/s-panferov/awesome-typescript-loader) 像 JavaScript 一样加载 [TypeScript](https://www.typescriptlang.org/) 2.0+
* [coffee-loader](https://www.webpackjs.com/loaders/coffee-loader) 像 JavaScript 一样加载 [CoffeeScript](http://coffeescript.org/)

### **模板(Templating)**

* [html-loader](https://www.webpackjs.com/loaders/html-loader) 导出 HTML 为字符串，需要引用静态资源
* pug-loader 加载 Pug 模板并返回一个函数
* jade-loader 加载 Jade 模板并返回一个函数
* markdown-loader 将 Markdown 转译为 HTML
* [react-markdown-loader](https://github.com/javiercf/react-markdown-loader) 使用 markdown-parse parser(解析器) 将 Markdown 编译为 React 组件
* posthtml-loader 使用 [PostHTML](https://github.com/posthtml/posthtml) 加载并转换 HTML 文件
* handlebars-loader 将 Handlebars 转移为 HTML
* [markup-inline-loader](https://github.com/asnowwolf/markup-inline-loader) 将内联的 SVG/MathML 文件转换为 HTML。在应用于图标字体，或将 CSS 动画应用于 SVG 时非常有用。

### **样式**

* [style-loader](https://www.webpackjs.com/loaders/style-loader) 将模块的导出作为样式添加到 DOM 中
* [css-loader](https://www.webpackjs.com/loaders/css-loader) 解析 CSS 文件后，使用 import 加载，并且返回 CSS 代码
* [less-loader](https://www.webpackjs.com/loaders/less-loader) 加载和转译 LESS 文件
* [sass-loader](https://www.webpackjs.com/loaders/sass-loader) 加载和转译 SASS/SCSS 文件
* [postcss-loader](https://www.webpackjs.com/loaders/postcss-loader) 使用 [PostCSS](http://postcss.org/) 加载和转译 CSS/SSS 文件
* stylus-loader 加载和转译 Stylus 文件

### **清理和测试(Linting && Testing)**

* [mocha-loader](https://www.webpackjs.com/loaders/mocha-loader) 使用 [mocha](https://mochajs.org/) 测试（浏览器/NodeJS）
* [eslint-loader](https://github.com/webpack-contrib/eslint-loader) PreLoader，使用 [ESLint](https://eslint.org/) 清理代码
* [jshint-loader](https://www.webpackjs.com/loaders/jshint-loader) PreLoader，使用 [JSHint](http://jshint.com/about/) 清理代码
* jscs-loader PreLoader，使用 [JSCS](http://jscs.info/) 检查代码样式
* [coverjs-loader](https://www.webpackjs.com/loaders/coverjs-loader) PreLoader，使用 [CoverJS](https://github.com/arian/CoverJS) 确定测试覆盖率

### **框架(Frameworks)**

* vue-loader 加载和转译 [Vue 组件](https://vuejs.org/v2/guide/components.html)

# Webpack使用plugin扩展功能

## Plugin的概念

插件目的在于解决 [loader](https://www.webpackjs.com/concepts/loaders) 无法实现的**其他事**。

webpack **插件**是一个具有 [apply](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply) 属性的 JavaScript 对象。apply 属性会被 webpack compiler 调用，并且 compiler 对象可在**整个**编译生命周期访问。

## 使用plugin

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

const HtmlWebpackPlugin = require('html-webpack-plugin'); //通过 npm 安装

const webpack = require('webpack'); //访问内置的插件

const path = require('path');

const config = {

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

output: {

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

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

},

module: {

rules: [

{

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

use: 'babel-loader'

}

]

},

plugins: [

new webpack.optimize.UglifyJsPlugin(),

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

]

};

module.exports = config;

## 常用的第三方plugin

除了压缩 JS 代码的 [uglifyjs-webpack-plugin](https://webpack.js.org/plugins/uglifyjs-webpack-plugin/" \t "_blank)，

常用的还有定义环境变量的 [DefinePlugin](https://webpack.js.org/plugins/define-plugin/" \t "_blank)，

生成 CSS 文件的 [ExtractTextWebpackPlugin](https://webpack.js.org/plugins/extract-text-webpack-plugin/" \t "_blank)

默认html模板的 html-webpack-plugin

## 手写一个plugin

webpack 插件由以下组成：

* 一个 JavaScript 命名函数。
* 在插件函数的 prototype 上定义一个 apply 方法。
* 指定一个绑定到 webpack 自身的[事件钩子](https://www.webpackjs.com/api/compiler-hooks/)。
* 处理 webpack 内部实例的特定数据。
* 功能完成后调用 webpack 提供的回调。

// 一个 JavaScript 命名函数。function MyExampleWebpackPlugin() {

};

// 在插件函数的 prototype 上定义一个 `apply` 方法。

MyExampleWebpackPlugin.prototype.apply = function(compiler) {

// 指定一个挂载到 webpack 自身的事件钩子。

compiler.plugin('webpacksEventHook', function(compilation /\* 处理 webpack 内部实例的特定数据。\*/, callback) {

console.log("This is an example plugin!!!");

// 功能完成后调用 webpack 提供的回调。

callback();

});};

compiler 对象代表了完整的 webpack 环境配置。这个对象在启动 webpack 时被一次性建立，并配置好所有可操作的设置，包括 options，loader 和 plugin。当在 webpack 环境中应用一个插件时，插件将收到此 compiler 对象的引用。可以使用它来访问 webpack 的主环境。

compilation 对象代表了一次资源版本构建。当运行 webpack 开发环境中间件时，每当检测到一个文件变化，就会创建一个新的 compilation，从而生成一组新的编译资源。一个 compilation 对象表现了当前的模块资源、编译生成资源、变化的文件、以及被跟踪依赖的状态信息。compilation 对象也提供了很多关键时机的回调，以供插件做自定义处理时选择使用。

插件是由「具有 apply 方法的 prototype 对象」所实例化出来的。这个 apply 方法在安装插件时，会被 webpack compiler 调用一次。apply 方法可以接收一个 webpack compiler 对象的引用，从而可以在回调函数中访问到 compiler 对象。一个简单的插件结构如下：

function HelloWorldPlugin(options) {

// 使用 options 设置插件实例……}

HelloWorldPlugin.prototype.apply = function(compiler) {

compiler.plugin('done', function() {

console.log('Hello World!');

});};

module.exports = HelloWorldPlugin;

然后，要安装这个插件，只需要在你的 webpack 配置的 plugin 数组中添加一个实例：

var HelloWorldPlugin = require('hello-world');

var webpackConfig = {

// ... 这里是其他配置 ...

plugins: [

new HelloWorldPlugin({options: true})

]};

### **访问 compilation 对象**

使用 compiler 对象时，你可以绑定提供了编译 compilation 引用的回调函数，然后拿到每次新的 compilation 对象。这些 compilation 对象提供了一些钩子函数，来钩入到构建流程的很多步骤中。

function HelloCompilationPlugin(options) {}

HelloCompilationPlugin.prototype.apply = function(compiler) {

// 设置回调来访问 compilation 对象：

compiler.plugin("compilation", function(compilation) {

// 现在，设置回调来访问 compilation 中的步骤：

compilation.plugin("optimize", function() {

console.log("Assets are being optimized.");

});

});};

module.exports = HelloCompilationPlugin;

### **异步编译插件**

有一些编译插件中的步骤是异步的，这样就需要额外传入一个 callback 回调函数，并且在插件运行结束时，\_必须\_调用这个回调函数。

function HelloAsyncPlugin(options) {}

HelloAsyncPlugin.prototype.apply = function(compiler) {

compiler.plugin("emit", function(compilation, callback) {

// 做一些异步处理……

setTimeout(function() {

console.log("Done with async work...");

callback();

}, 1000);

});};

module.exports = HelloAsyncPlugin;

让我们来写一个简单的示例插件，生成一个叫做 filelist.md 的新文件；文件内容是所有构建生成的文件的列表。这个插件大概像下面这样：

function FileListPlugin(options) {}

FileListPlugin.prototype.apply = function(compiler) {

compiler.plugin('emit', function(compilation, callback) {

// 在生成文件中，创建一个头部字符串：

var filelist = 'In this build:\n\n';

// 遍历所有编译过的资源文件，

// 对于每个文件名称，都添加一行内容。

for (var filename in compilation.assets) {

filelist += ('- '+ filename +'\n');

}

// 将这个列表作为一个新的文件资源，插入到 webpack 构建中：

compilation.assets['filelist.md'] = {

source: function() {

return filelist;

},

size: function() {

return filelist.length;

}

};

callback();

});};

module.exports = FileListPlugin;

## 第三方插件

[AggressiveSplittingPlugin](https://www.webpackjs.com/plugins/aggressive-splitting-plugin)

将原来的 chunk 分成更小的 chunk

[BabelMinifyWebpackPlugin](https://www.webpackjs.com/plugins/babel-minify-webpack-plugin)

使用 [babel-minify](https://github.com/babel/minify)进行压缩

[BannerPlugin](https://www.webpackjs.com/plugins/banner-plugin)

在每个生成的 chunk 顶部添加 banner

[CommonsChunkPlugin](https://www.webpackjs.com/plugins/commons-chunk-plugin)

提取 chunks 之间共享的通用模块

[CompressionWebpackPlugin](https://www.webpackjs.com/plugins/compression-webpack-plugin)

预先准备的资源压缩版本，使用 Content-Encoding 提供访问服务

[ContextReplacementPlugin](https://www.webpackjs.com/plugins/context-replacement-plugin)

重写 require 表达式的推断上下文

[CopyWebpackPlugin](https://www.webpackjs.com/plugins/copy-webpack-plugin)

将单个文件或整个目录复制到构建目录

[DefinePlugin](https://www.webpackjs.com/plugins/define-plugin)

允许在编译时(compile time)配置的全局常量

[DllPlugin](https://www.webpackjs.com/plugins/dll-plugin)

为了极大减少构建时间，进行分离打包

[EnvironmentPlugin](https://www.webpackjs.com/plugins/environment-plugin)

[DefinePlugin](https://www.webpackjs.com/plugins/define-plugin) 中 process.env 键的简写方式。

[ExtractTextWebpackPlugin](https://www.webpackjs.com/plugins/extract-text-webpack-plugin)

从 bundle 中提取文本（CSS）到单独的文件

[HotModuleReplacementPlugin](https://www.webpackjs.com/plugins/hot-module-replacement-plugin)

启用模块热替换(Enable Hot Module Replacement - HMR)

[HtmlWebpackPlugin](https://www.webpackjs.com/plugins/html-webpack-plugin)

简单创建 HTML 文件，用于服务器访问

[I18nWebpackPlugin](https://www.webpackjs.com/plugins/i18n-webpack-plugin)

为 bundle 增加国际化支持

[IgnorePlugin](https://www.webpackjs.com/plugins/ignore-plugin)

从 bundle 中排除某些模块

[LimitChunkCountPlugin](https://www.webpackjs.com/plugins/limit-chunk-count-plugin)

设置 chunk 的最小/最大限制，以微调和控制 chunk

[LoaderOptionsPlugin](https://www.webpackjs.com/plugins/loader-options-plugin)

用于从 webpack 1 迁移到 webpack 2

[MinChunkSizePlugin](https://www.webpackjs.com/plugins/min-chunk-size-plugin)

确保 chunk 大小超过指定限制

[NoEmitOnErrorsPlugin](https://www.webpackjs.com/plugins/no-emit-on-errors-plugin)

在输出阶段时，遇到编译错误跳过

[NormalModuleReplacementPlugin](https://www.webpackjs.com/plugins/normal-module-replacement-plugin)

替换与正则表达式匹配的资源

[NpmInstallWebpackPlugin](https://www.webpackjs.com/plugins/npm-install-webpack-plugin)

在开发时自动安装缺少的依赖

[ProvidePlugin](https://www.webpackjs.com/plugins/provide-plugin)

不必通过 import/require 使用模块

[SourceMapDevToolPlugin](https://www.webpackjs.com/plugins/source-map-dev-tool-plugin)

对 source map 进行更细粒度的控制

[EvalSourceMapDevToolPlugin](https://www.webpackjs.com/plugins/eval-source-map-dev-tool-plugin)

对 eval source map 进行更细粒度的控制

[UglifyjsWebpackPlugin](https://www.webpackjs.com/plugins/uglifyjs-webpack-plugin)

可以控制项目中 UglifyJS 的版本

[ZopfliWebpackPlugin](https://www.webpackjs.com/plugins/zopfli-webpack-plugin)

通过 node-zopfli 将资源预先压缩的版本

# Webpack解析代码模块路径

## 模块解析规则

* 解析相对路径

1. 查找相对当前模块的路径下是否有对应文件或文件夹
2. 是文件则直接加载
3. 是文件夹则继续查找文件夹下的 package.json 文件
4. 有 package.json 文件则按照文件中 main 字段的文件名来查找文件
5. 无 package.json 或者无 main 字段则查找 index.js 文件

* 解析模块名  
  查找当前文件目录下，父级目录及以上目录下的 node\_modules 文件夹，看是否有对应名称的模块
* 解析绝对路径（不建议使用）  
  直接查找对应路径的文件

### Resolve.alias

创建别名，来确保引入模块变得更简单。

      resolve: {

        alias: {

          utils: path.resolve(\_\_dirname, 'src/utils/'),

          views: path.resolve(\_\_dirname, 'src/views/'),

          cpt: path.resolve(\_\_dirname, 'src/components/')

        }

      }

引入就变成

import utils from 'utils'

import views from 'views'

import cpt from 'cpt'

### resolve.extensions

自动解析确定的扩展。

      resolve: {

        extensions:['.js']

      }

我们就可以这样导入

import file from '../file'

### resolve.modules

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。

      resolve: {

        modules: ['node\_modules']

      }

### Resolve.mainFields

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](https://www.webpackjs.com/concepts/targets) 不同，默认值也会有所不同。

      resolve: {

        // target为webworker, web

        mainFields: ['browser', 'module', 'main'],

        // target为node

        mainFields: ['module', 'main']

      }

因为通常情况下，模块的 package 都不会声明 browser 或 module 字段，所以便是使用 main 了。

在 NPM packages 中，会有些 package 提供了两个实现，分别给浏览器和 Node.js 两个不同的运行时使用，这个时候就需要区分不同的实现入口在哪里。如果你有留意一些社区开源模块的 package.json 的话，你也许会发现 browser 或者 module 等字段的声明。

### Resolve.mainFiles

解析目录时要使用的文件名。当目录下没有 package.json 文件时，我们说会默认使用目录下的 index.js 这个文件

      resolve: {

        mainFiles: ['index']

      }

### Resolve.resolveLoader

这组选项与上面的 resolve 对象的属性集合相同

# **模块热替换**

模块热替换(HMR - Hot Module Replacement)功能会在应用程序运行过程中替换、添加或删除[模块](https://www.webpackjs.com/concepts/modules/)，而无需重新加载整个页面。主要是通过以下几种方式，来显著加快开发速度：

* 保留在完全重新加载页面时丢失的应用程序状态。
* 只更新变更内容，以节省宝贵的开发时间。
* 调整样式更加快速 - 几乎相当于在浏览器调试器中更改样式。

# **tree shaking**

*tree shaking* 是一个术语，通常用于描述移除 JavaScript 上下文中的未引用代码(dead-code)。它依赖于 ES2015 模块系统中的[静态结构特性](http://exploringjs.com/es6/ch_modules.html" \l "static-module-structure)，例如 [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) 和 [export](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export)。这个术语和概念实际上是兴起于 ES2015 模块打包工具 [rollup](https://github.com/rollup/rollup)。

新的 webpack 4 正式版本，扩展了这个检测能力，通过 package.json 的 "sideEffects" 属性作为标记，向 compiler 提供提示，表明项目中的哪些文件是 "pure(纯的 ES2015 模块)"，由此可以安全地删除文件中未使用的部分。

# **TypeScript**

[TypeScript](https://www.typescriptlang.org/) 是 JavaScript 的超集，为其增加了类型系统，可以编译为普通的 JavaScript 代码。这篇指南里我们将会学习 webpack 是如何跟 TypeScript 进行集成。

总结