**安装和使用**

我们使用 npm 或者 yarn 来安装 webpack，可以作为一个全局的命令来使用：

npm install webpack webpack-cli -g

# 或者

yarn global add webpack webpack-cli

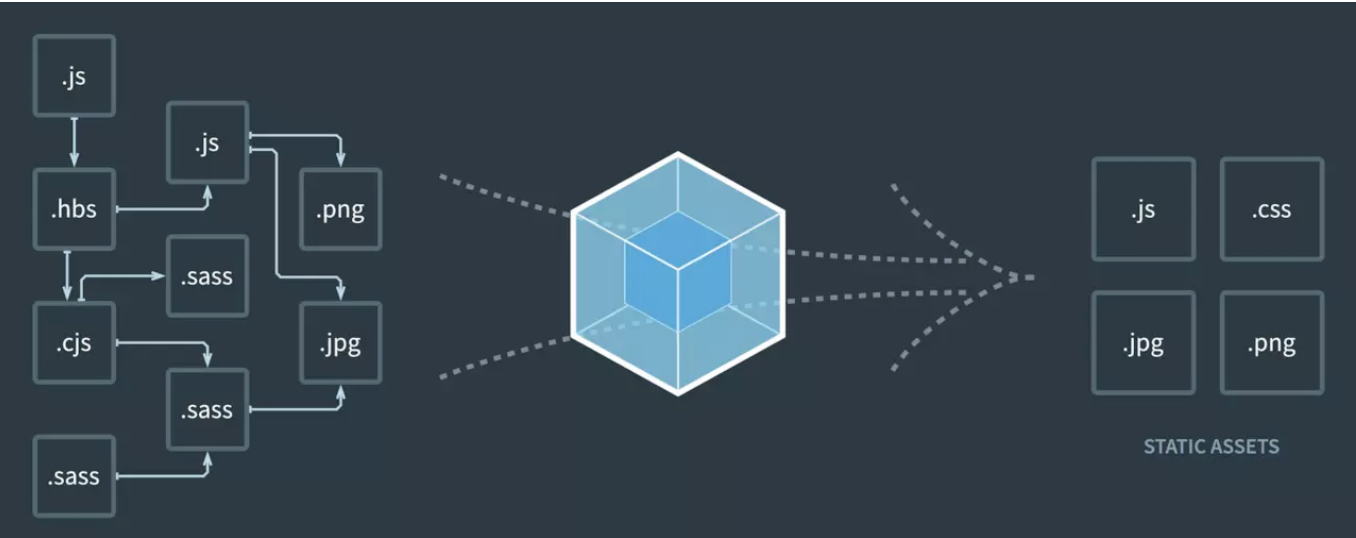
# 然后就可以全局执行命令了

webpack --help

[webpack-cli](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Fwebpack%2Fwebpack-cli) 是使用 webpack 的命令行工具，在 4.x 版本之后不再作为 webpack 的依赖了，我们使用时需要单独安装这个工具。

**webpack 的基本概念**

webpack 本质上是一个打包工具，它会根据代码的内容解析模块依赖，帮助我们把多个模块的代码打包：



如上图，webpack 会把我们项目中使用到的多个代码模块（可以是不同文件类型），打包构建成项目运行仅需要的几个静态文件。webpack 有着十分丰富的配置项，提供了十分强大的扩展能力，可以在打包构建的过程中做很多事情。我们先来看一下 webpack 中的几个基本概念。

**入口**

我们常见的项目中，如果是单页面应用，那么可能入口只有一个；如果是多个页面的项目，那么经常是一个页面会对应一个构建入口。

如上图所示，在多个代码模块中会有一个起始的 .js 文件，这个便是 webpack 构建的入口。webpack 会读取这个文件，并从它开始解析依赖，然后进行打包。如图，一开始我们使用 webpack 构建时，默认的入口文件就是 ./src/index.js

module.exports = {

entry: './src/index.js'

}

// 上述配置等同于

module.exports = {

entry: {

main: './src/index.js'

}

}

// 或者配置多个入口

module.exports = {

entry: {

foo: './src/page-foo.js',

bar: './src/page-bar.js',

// ...

}

}

// 使用数组来对多个文件进行打包

module.exports = {

entry: {

main: [

'./src/foo.js',

'./src/bar.js'

]

}

}

最后的例子，可以理解为多个文件作为一个入口，webpack 会解析两个文件的依赖后进行打包。

**loader**

webpack 中提供一种处理多种文件格式的机制，便是使用 loader。我们可以把 loader 理解为是一个转换器，负责把某种文件格式的内容转换成 webpack 可以支持打包的模块。

举个例子，在没有添加额外插件的情况下，webpack 会默认把所有依赖打包成 js 文件，如果入口文件依赖一个 .hbs 的模板文件以及一个 .css 的样式文件，那么我们需要 handlebars-loader 来处理 .hbs 文件，需要 css-loader 来处理 .css 文件（这里其实还需要 style-loader，后续详解），最终把不同格式的文件都解析成 js 代码，以便打包后在浏览器中运行。... https://juejin.im 掘金 — 一个帮助开发者成长的社区

当我们需要使用不同的 loader 来解析处理不同类型的文件时，我们可以在 module.rules 字段下来配置相关的规则，例如使用 Babel 来处理 .js 文件：

module: {

// ...

rules: [

{

test: /\.jsx?/, // 匹配文件路径的正则表达式，通常我们都是匹配文件类型后缀

include: [

path.resolve(\_\_dirname, 'src') // 指定哪些路径下的文件需要经过 loader 处理

],

use: 'babel-loader', // 指定使用的 loader

},

],

}...

loader 是 webpack 中比较复杂的一块内容，它支撑着 webpack 来处理文件的多样性

**plugin**

在 webpack 的构建流程中，plugin 用于处理更多其他的一些构建任务。可以这么理解，模块代码转换的工作由 loader 来处理，除此之外的其他任何工作都可以交由 plugin 来完成。通过添加我们需要的 plugin，可以满足更多构建中特殊的需求。例如，要使用压缩 JS 代码的 uglifyjs-webpack-plugin 插件，只需在配置中通过 plugins 字段添加新的 plugin 即可

除了压缩 JS 代码的 [uglifyjs-webpack-plugin](https://link.juejin.im/?target=https%3A%2F%2Fwebpack.js.org%2Fplugins%2Fuglifyjs-webpack-plugin%2F)，常用的还有定义环境变量的 [DefinePlugin](https://link.juejin.im/?target=https%3A%2F%2Fwebpack.js.org%2Fplugins%2Fdefine-plugin%2F)，生成 CSS 文件的 [ExtractTextWebpackPlugin](https://link.juejin.im/?target=https%3A%2F%2Fwebpack.js.org%2Fplugins%2Fextract-text-webpack-plugin%2F) 等。

plugin 理论上可以干涉 webpack 整个构建流程，可以在流程的每一个步骤中定制自己的构建需求。第 15 小节我们会介绍如何开发 plugin，让开发者们在必要时，也可以在 webpack 的基础上开发 plugin 来应对一些项目的特殊构建需求

**输出**

webpack 的输出即指 webpack 最终构建出来的静态文件，可以看看上面 webpack 官方图片右侧的那些文件。当然，构建结果的文件名、路径等都是可以配置的，使用 output 字段：

// 路径中使用 hash，每次构建时会有一个不同 hash 值，避免发布新版本时线上使用浏览器缓存

module.exports = {

// ...

output: {

filename: '[name].js',

path: \_\_dirname + '/dist/[hash]',

},

}...

**一个简单的 webpack 配置**

我们把上述涉及的几部分配置内容合到一起，就可以创建一个简单的 webpack 配置了，webpack 运行时默认读取项目下的 webpack.config.js 文件作为配置。

创建了 webpack.config.js 后再执行 webpack 命令，webpack 就会使用这个配置文件的配置了。 有的时候我们开始一个新的前端项目，并不需要从零开始配置 webpack，而可以使用一些工具来帮助快速生成 webpack 配置。

webpack 的配置其实是一个 Node.js 的脚本，这个脚本对外暴露一个配置对象，webpack 通过这个对象来读取相关的一些配置。因为是 Node.js 脚本，所以可玩性非常高，你可以使用任何的 Node.js 模块，如上述用到的 path 模块，当然第三方的模块也可以。

**搭建基本的前端开发环境**

我们日常使用的前端开发环境的需求应该是：

* 构建我们发布需要的 HTML、CSS、JS 文件
* 使用 CSS 预处理器来编写样式
* 处理和压缩图片
* 使用 Babel 来支持 ES 新特性
* 本地提供静态服务以方便开发调试

**关联 HTML**

使用html-webpack-plugin 插件，动态生成html模版

module.exports = {

// ...

plugins: [

new HtmlWebpackPlugin({

filename: 'index.html', // 配置输出文件名和路径

template: 'assets/index.html', // 配置文件模板

}),

],

}

这样，通过 html-webpack-plugin 就可以将我们的页面和构建 JS 关联起来，回归日常，从页面开始开发。如果需要添加多个页面关联，那么实例化多个 html-webpack-plugin， 并将它们都放到 plugins 字段数组中就可以了。

**构建 CSS**

我们编写 CSS，并且希望使用 webpack 来进行构建，为此，需要在配置中引入 loader 来解析和处理 CSS 文件：

* css-loader 负责解析 CSS 代码，主要是为了处理 CSS 中的依赖，例如 @import 和 url() 等引用外部文件的声明；
* style-loader 会将 css-loader 解析的结果转变成 JS 代码，运行时动态插入 style 标签来让 CSS 代码生效。

经由上述两个 loader 的处理后，CSS 代码会转变为 JS，和 index.js 一起打包了。如果需要单独把 CSS 文件分离出来，我们需要使用 [extract-text-webpack-plugin](https://link.juejin.im/?target=https%3A%2F%2Fdoc.webpack-china.org%2Fplugins%2Fextract-text-webpack-plugin) 插件。

const ExtractTextPlugin = require('extract-text-webpack-plugin')

module.exports = {

// ...

module: {

rules: [

{

test: /\.css$/,

// 因为这个插件需要干涉模块转换的内容，所以需要使用它对应的 loader

use: ExtractTextPlugin.extract({

fallback: 'style-loader',

use: 'css-loader',

}),

},

],

},

plugins: [

// 引入插件，配置文件名，这里同样可以使用 [hash]

new ExtractTextPlugin('index.css'),

],

}...

**CSS 预处理器**

在上述使用 CSS 的基础上，通常我们会使用 Less/Sass 等 CSS 预处理器，webpack 可以通过添加对应的 loader 来支持，以使用 Less 为例，我们可以在官方文档中找到对应的 [loader](https://link.juejin.im/?target=https%3A%2F%2Fdoc.webpack-china.org%2Floaders%2Fless-loader)。

**使用 Babel**

[Babel](https://link.juejin.im/?target=http%3A%2F%2Fbabeljs.io%2F) 是一个让我们能够使用 ES 新特性的 JS 编译工具，我们可以在 webpack 中配置 Babel，以便使用 ES6、ES7 标准来编写 JS 代码。

module.exports = {

// ...

module: {

rules: [

{

test: /\.jsx?/, // 支持 js 和 jsx

include: [

path.resolve(\_\_dirname, 'src'), // src 目录下的才需要经过 babel-loader 处理

],

loader: 'babel-loader',

},

],

},

}

Babel 的相关配置可以在目录下使用 .babelrc 文件来处理，详细参考 Babel 官方文档 [.babelrc](https://link.juejin.im/?target=http%3A%2F%2Fbabeljs.io%2Fdocs%2Fusage%2Fbabelrc%2F)。

**启动静态服务**

至此，我们完成了处理多种文件类型的 webpack 配置。我们可以使用 [webpack-dev-server](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Fwebpack%2Fwebpack-dev-server) 在本地开启一个简单的静态服务来进行开发。

在项目下安装 webpack-dev-server，然后添加启动命令到 package.json 中：

"scripts": {

"build": "webpack --mode production",

"start": "webpack-dev-server --mode development"

}

**webpack 如何解析代码模块路径**

在 webpack 支持的前端代码模块化中，我们可以使用类似 import \* as m from './index.js' 来引用代码模块 index.js。

引用第三方类库则是像这样：import React from 'react'。webpack 构建的时候，会解析依赖后，然后再去加载依赖的模块文件，那么 webpack 如何将上述编写的 ./index.js 或 react 解析成对应的模块文件路径呢？

webpack 中有一个很关键的模块 [enhanced-resolve](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Fwebpack%2Fenhanced-resolve%2F) 就是处理依赖模块路径的解析的，这个模块可以说是 Node.js 那一套模块路径解析的增强版本，有很多可以自定义的解析配置。

**模块解析规则**

简单整理一下基本的模块解析规则，以便更好地理解后续 webpack 的一些配置会产生的影响。

* 解析相对路径

查找相对当前模块的路径下是否有对应文件或文件夹

是文件则直接加载

是文件夹则继续查找文件夹下的 package.json 文件

有 package.json 文件则按照文件中 main 字段的文件名来查找文件

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

在 webpack 配置中，和模块路径解析相关的配置都在 resolve 字段下：

module.exports = {

resolve: {

// ...

}

}

**常用的一些配置**

我们先从一些简单的需求来阐述 webpack 可以支持哪些解析路径规则的自定义配置。

**resolve.alias**

假设我们有个 utils 模块极其常用，经常编写相对路径很麻烦，希望可以直接 import 'utils' 来引用，那么我们可以配置某个模块的别名，

**resolve.extensions**

这个配置可以定义在进行模块路径解析时，webpack 会尝试帮你补全那些后缀名来进行查找，例如有了上述的配置，当你在 src/utils/ 目录下有一个 common.js 文件时。

webpack 会尝试给你依赖的路径添加上 extensions 字段所配置的后缀，然后进行依赖路径查找，所以可以命中 src/utils/common.js 文件。

**resolve.modules**

对于直接声明依赖名的模块（如 react ），webpack 会类似 Node.js 一样进行路径搜索，搜索 node\_modules 目录，这个目录就是使用 resolve.modules 字段进行配置的，默认就是：

resolve: {

modules: ['node\_modules'],

},

**配置 loader**

webpack 的 loader 用于处理不同的文件类型，在日常的项目中使用 loader 时，可能会遇到比较复杂的情况

**loader 匹配规则**

当我们需要配置 loader 时，都是在 module.rules 中添加新的配置项，在该字段中，每一项被视为一条匹配使用 loader 的规则。

module.exports = {

// ...

module: {

rules: [

{

test: /\.jsx?/, // 条件

include: [

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

], // 条件

use: 'babel-loader', // 规则应用结果

}, // 一个 object 即一条规则

// ...

],

},

}...

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。

匹配条件通常都使用请求资源文件的绝对路径来进行匹配，在官方文档中称为 resource，除此之外还有比较少用到的 issuer，则是声明依赖请求的源文件的绝对路径。举个例子：在 /path/to/app.js 中声明引入 import './src/style.scss'，resource 是 /path/to/src/style.scss，issuer 是 /path/to/app.js，规则条件会对这两个值来尝试匹配

当规则的条件匹配时，便会使用对应的 loader 配置

**规则条件配置**

大多数情况下，配置 loader 的匹配条件时，只要使用 test 字段就好了，很多时候都只需要匹配文件后缀名来决定使用什么 loader，但也不排除在某些特殊场景下，我们需要配置比较复杂的匹配条件。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

上述的所谓条件的值可以是：

字符串：必须以提供的字符串开始，所以是字符串的话，这里我们需要提供绝对路径

正则表达式：调用正则的 test 方法来判断匹配

函数：(path) => boolean，返回 true 表示匹配

数组：至少包含一个条件的数组

对象：匹配所有属性值的条件

rules: [

{

test: /\.jsx?/, // 正则

include: [

path.resolve(\_\_dirname, 'src'), // 字符串，注意是绝对路径

], // 数组

// ...

},

{

test: {

js: /\.js/,

jsx: /\.jsx/,

}, // 对象，不建议使用

not: [

(value) => { /\* ... \*/ return true; }, // 函数，通常需要高度自定义时才会使用

],

},

],

上述多个配置形式结合起来就能够基本满足各种各样的构建场景了，通常我们会结合使用 test/and和 include&exclude 来配置条件，如上述那个简单的例子。

**使用 loader 配置**

当然，在当前版本的 webpack 中，module.rules 的匹配规则最重要的还是用于配置 loader，我们可以使用 use 字段：

rules: [

{

test: /\.less/,

use: [

'style-loader', // 直接使用字符串表示 loader

{

loader: 'css-loader',

options: {

importLoaders: 1

},

}, // 用对象表示 loader，可以传递 loader 配置等

{

loader: 'less-loader',

options: {

noIeCompat: true

}, // 传递 loader 配置

},

],

},

]

我们看下上述的例子，先忽略 loader 的使用情况，单纯看看如何配置。use 字段可以是一个数组，也可以是一个字符串或者表示 loader 的对象。如果只需要一个 loader，也可以这样：use: { loader: 'babel-loader', options: { ... } }

我们还可以使用 options 给对应的 loader 传递一些配置项，这里不再展开。当你使用一些 loader 时，loader 的说明一般都有相关配置的描述。

**loader 应用顺序**

前面提到，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。

loader 的应用顺序在配置多个 loader 一起工作时很重要，通常会使用在 CSS 配置上，除了 style-loader 和 css-loader，你可能还要配置 less-loader 然后再加个 postcss 的 autoprefixer 等。

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。

所有的 loader 按照**前置 -> 行内 -> 普通 -> 后置**的顺序执行

rules: [

{

enforce: 'pre', // 指定为前置类型

test: /\.js$/,

exclude: /node\_modules/,

loader: "eslint-loader",

},

]

**使用 noParse**

在 webpack 中，我们需要使用的 loader 是在 module.rules 下配置的，webpack 配置中的 module 用于控制如何处理项目中不同类型的模块。

除了 module.rules 字段用于配置 loader 之外，还有一个 module.noParse 字段，可以用于配置哪些模块文件的内容不需要进行解析。对于一些不需要解析依赖（即无依赖） 的第三方大型类库等，可以通过这个字段来配置，以提高整体的构建速度。

module.exports = {

// ...

module: {

noParse: /jquery|lodash/, // 正则表达式

// 或者使用 function

noParse(content) {

return /jquery|lodash/.test(content)

},

}

}

webpack 的 loader 相关配置都在 module.rules 字段下，我们需要通过 test、include、exclude 等配置好应用 loader 的条件规则，然后使用 use 来指定需要用到的 loader，配置应用的 loader 时还需要注意一下 loader 的执行顺序。

**使用 plugin**

webpack 中的 plugin 大多都提供额外的能力，它们在 webpack 中的配置都只是把插件实例添加到 plugins 字段的数组中。不过由于需要提供不同的功能，不同的插件本身的配置比较多样化。

**常用插件**

webpack.DefinePlugin

copy-webpack-plugin

html-webpack-plugin

extract-text-webpack-plugin

**开发和生产环境的构建配置差异**

我们在日常的前端开发工作中，一般都会有两套构建环境：一套开发时使用，构建结果用于本地开发调试，不进行代码压缩，打印 debug 信息，包含 sourcemap 文件；另外一套构建后的结果是直接应用于线上的，即代码都是压缩后，运行时不打印 debug 信息，静态文件不包括 sourcemap 的。有的时候可能还需要多一套测试环境，在运行时直接进行请求 mock 等工作。

webpack 的运行时环境是 Node.js，我们可以通过 Node.js 提供的机制给要运行的 webpack 程序传递环境变量，来控制不同环境下的构建行为。例如，我们在 npm 中的 scripts 字段添加一个用于生产环境的构建命令

{

"scripts": {

"build": "NODE\_ENV=production webpack",

"dev": "NODE\_ENV=development webpack-dev-server"

}

}

然后在 webpack.config.js 文件中可以通过 process.env.NODE\_ENV 来获取命令传入的环境变量

**常见的环境差异配置**

前面提及的使用环境变量的方式可以让我们在不同的构建环境中完成不同的构建需求，这里列举一下常见的 webpack 构建差异配置：

生产环境可能需要分离 CSS 成单独的文件，以便多个页面共享同一个 CSS 文件

生产环境需要压缩 HTML/CSS/JS 代码

生产环境需要压缩图片

开发环境需要生成 sourcemap 文件

开发环境需要打印 debug 信息

开发环境需要 live reload 或者 hot reload 的功能

以上是常见的构建环境需求差异，可能更加复杂的项目中会有更多的构建需求（如划分静态域名等），但是我们都可以通过判断环境变量来实现这些有环境差异的构建需求。

**优化前端资源加载**

**图片加载优化**

css sprites 用 webpack-spritesmith 或者 sprite-webpack-plugin

图片压缩 image-webpack-loader 来压缩图片文件

使用 DataURL url－loader在处理文件的时候，可以通过配置指定一个大小，当文件小于这个配置值时，url-loader 会将其转换为一个 base64 编码的 DataURL

**代码压缩**

webpack 4.x 版本运行时，mode 为 production 即会启动压缩 JS 代码的插件，而对于 webpack 3.x，使用压缩 JS 代码插件的方式也已经介绍过了。在生产环境中，压缩 JS 代码基本是一个必不可少的步骤，这样可以大大减小 JavaScript 的体积，相关内容这里不再赘述

**分离代码文件**

css分离在 webpack 中使用 extract-text-webpack-plugin 插件即可

如何使用 webpack 来把代码中公共使用的部分分离成为独立的文件呢？

3.x 以前的版本是使用 CommonsChunkPlugin 来做代码分离的，而 webpack 4.x 则是把相关的功能包到了 optimize.splitChunks 中，直接使用该配置就可以实现代码分离。

webpack 4.x 的 optimization

module.exports = {

// ... webpack 配置

optimization: {

splitChunks: {

chunks: "all", // 所有的 chunks 代码公共的部分分离出来成为一个单独的文件

},

},

}

之前我们提到拆分文件是为了更好地利用缓存，分离公共类库很大程度上是为了让多页面利用缓存，从而减少下载的代码量，同时，也有代码变更时可以利用缓存减少下载代码量的好处。从这个角度出发，笔者建议将公共使用的第三方类库显式地配置为公共的部分，而不是 webpack 自己去判断处理。因为公共的第三方类库通常升级频率相对低一些，这样可以避免因公共 chunk 的频繁变更而导致缓存失效。

显式配置共享类库可以这么操作：

**module.exports = {**

**entry: {**

**vendor: ["react", "lodash", "angular", ...], // 指定公共使用的第三方类库**

**},**

**optimization: {**

**splitChunks: {**

**cacheGroups: {**

**vendor: {**

**chunks: "initial",**

**test: "vendor",**

**name: "vendor", // 使用 vendor 入口作为公共部分**

**enforce: true,**

**},**

**},**

**},**

**},**

**// ... 其他配置**

**}**

**// 或者**

**module.exports = {**

**optimization: {**

**splitChunks: {**

**cacheGroups: {**

**vendor: {**

**test: /react|angluar|lodash/, // 直接使用 test 来做路径匹配**

**chunks: "initial",**

**name: "vendor",**

**enforce: true,**

**},**

**},**

**},**

**},**

**}**

**// 或者**

**module.exports = {**

**optimization: {**

**splitChunks: {**

**cacheGroups: {**

**vendor: {**

**chunks: "initial",**

**test: path.resolve(\_\_dirname, "node\_modules") // 路径在 node\_modules 目录下的都作为公共部分**

**name: "vendor", // 使用 vendor 入口作为公共部分**

**enforce: true,**

**},**

**},**

**},**

**},**

**}**

上述第一种做法是显示指定哪些类库作为公共部分，第二种做法实现的功能差不多，只是利用了 test 来做模块路径的匹配，第三种做法是把所有在 node\_modules 下的模块，即作为依赖安装的，都作为公共部分。你可以针对项目情况，选择最合适的做法。

webpack 3.x 的 CommonsChunkPlugin

module.exports = {

// ...

plugins: [

new webpack.optimize.CommonsChunkPlugin({

name: "commons", // 公共使用的 chunk 的名称

filename: "commons.js", // 公共 chunk 的生成文件名

minChunks: 3, // 公共的部分必须被 3 个 chunk 共享

}),

],

}

CommonsChunkPlugin 也是支持显式配置共享类库的：

module.exports = {

entry: {

vendor: ['react', 'react-redux'], // 指定公共使用的第三方类库

app: './src/entry',

// ...

},

// ...

plugins: [

new webpack.optimize.CommonsChunkPlugin({

name: 'vendor' // 使用 vendor 入口作为公共部分

filename: "vendor.js",

minChunks: Infinity, // 这个配置会让 webpack 不再自动抽离公共模块

}),

],

}

按需加载模块

前面讲述了如何把大的代码文件进行拆分，抽离出多个页面共享的部分，但是当你的 Web 应用是单个页面，并且极其复杂的时候，你会发现有一些代码并不是每一个用户都需要用到的。你可能希望将这一部分代码抽离出去，仅当用户真正需要用到时才加载，这个时候就需要用到 webpack 提供的一个优化功能 —— 按需加载代码模块。

在 webpack 的构建环境中，要按需加载代码模块很简单，遵循 ES 标准的动态加载语法 [dynamic-import](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Ftc39%2Fproposal-dynamic-import) 来编写代码即可，webpack 会自动处理使用该语法编写的模块：

**提升 webpack 的构建速度**

1. 减少 resolve 的解析
   * 1. 使用绝对路径指定 node\_modules，不做过多查询
     2. 删除不必要的后缀自动补全，少了文件后缀的自动匹配，即减少了文件路径查询的工作
     3. 避免新增默认文件，编码时使用详细的文件路径，代码会更容易解读，也有益于提高构建速度
2. 把 loader 应用的文件范围缩小
   * 1. 我们在使用 loader 的时候，尽可能把 loader 应用的文件范围缩小，只在最少数必须的代码模块中去使用必要的 loader，例如 node\_modules 目录下的其他依赖类库文件，基本就是直接编译好可用的代码，无须再经过 loader 处理了
3. 减少 plugin 的消耗
   * 1. webpack 的 plugin 会在构建的过程中加入其它的工作步骤，如果可以的话，适当地移除掉一些没有必要的 plugin。

**webpack 内部工作流程**

webpack 本质上就是一个 JS Module Bundler，用于将多个代码模块进行打包

**JS Module Bunlder 的基础工作流程：**

1. 首先，bundler 从一个构建入口出发，解析代码，分析出代码模块依赖关系，然后将依赖的代码模块组合在一起，在 JavaScript bundler 中，还需要提供一些胶水代码让多个代码模块可以协同工作，相互引用。
2. 首先是解析代码，分析依赖关系：根据入口解析出依赖代码文件，继续解析其依赖，递归下去，直至没有更多的依赖模块，最终形成一颗模块依赖树。
3. 分析出依赖关系后，bunlder 需要将依赖关系中涉及的所有文件组合到一起，但由于依赖代码的执行是有先后顺序以及会引用模块内部不同的内容，不能简单地将代码拼接到一起。webpack 会利用 JavaScript Function 的特性提供一些代码来将各个模块整合到一起，即是将每一个模块包装成一个 JS Function，提供一个引用依赖模块的方法

**webpack 工作的主要流程和其中几个重要的概念**

1. Compiler，webpack 的运行入口，实例化时定义 webpack 构建主要流程，同时创建构建时使用的核心对象 compilation
2. Compilation，由 Compiler 实例化，存储构建过程中各流程使用到的数据，用于控制这些数据的变化
3. Chunk，即用于表示 chunk 的类，对于构建时需要的 chunk 对象由 Compilation 创建后保存管理
4. Module，用于表示代码模块的类，衍生出很多子类用于处理不同的情况，关于代码模块的所有信息都会存在 Module 实例中，例如 dependencies 记录代码模块的依赖等
5. Parser，其中相对复杂的一个部分，基于 acorn 来分析 AST 语法树，解析出代码模块的依赖
6. Dependency，解析时用于保存代码模块对应的依赖使用的对象
7. Template，生成最终代码要使用到的代码模板，像上述提到的胶水代码就是用对应的 Template 来生成

**webpack 运行的大概工作流程是这样的：**

* 创建 Compiler ->
* 调用 compiler.run 开始构建 ->
* 创建 Compilation ->
* 基于配置开始创建 Chunk ->
* 使用 Parser 从 Chunk 开始解析依赖 ->
* 使用 Module 和 Dependency 管理代码模块相互关系 ->
* 使用 Template 基于 Compilation 的数据生成结果代码

参考：

https://github.com/DDFE/DDFE-blog/issues/12