# webpack

## [Webpack] 为什么选择 webpack？【热度: 515】

### 为什么选择 webpack

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

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

### 立即调用函数表达式(IIFE) - Immediately invoked function expressions

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

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

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

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

### 感谢 Node.js，JavaScript 模块诞生了

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

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

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

npm + Node.js + modules - 大规模分发模块

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

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

### ESM - ECMAScript 模块

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

### 依赖自动收集

传统的任务构建工具基于 Google 的 Closure 编译器都要求你手动在顶部声明所有的依赖。然而像 webpack 一类的打包工具自动构建并基于你所引用或导出的内容推断出依赖的图谱。这个特性与其它的如插件 and 加载器一道让开发者的体验更好。

### 看起来都不是很好……

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

**这就是 webpack 存在的原因**。它是一个工具，可以打包你的 JavaScript 应用程序（支持 ESM 和 CommonJS），可以扩展为支持许多不同的静态资源，例如：images, fonts 和 stylesheets。

webpack 关心性能和加载时间；它始终在改进或添加新功能，例如：异步地加载 chunk 和预取，以便为你的项目和用户提供最佳体验。

## [Webpack] 有哪些基础概念【热度: 595】

Webpack是一个现代的JavaScript模块打包工具，它的核心概念包括以下几个方面：

* 入口（Entry）：指定Webpack开始构建依赖图谱的起点。可以通过配置文件中的entry属性来指定入口文件，也可以指定多个入口文件。
* 输出（Output）：指定Webpack打包后的文件输出的路径和文件名。可以通过配置文件中的output属性来定义输出路径和文件名的规则。
* 加载器（Loader）：Webpack本身只能处理JavaScript文件，通过加载器，Webpack可以处理其他类型的文件，如CSS、图片、字体等。加载器会在打包过程中对文件进行转换和处理。
* 插件（Plugin）：插件是Webpack的核心功能扩展机制，可以用于解决很多构建过程中的复杂问题或实现特定的需求。插件可以用于优化打包结果、自动生成HTML文件、提取CSS文件等。
* 模式（Mode）：Webpack提供了两种模式，分别是开发模式（development）和生产模式（production）。开发模式会启用一些有助于开发调试的功能，而生产模式则会启用代码压缩、优化等功能。
* 代码分割（Code Splitting）：Webpack支持将代码分割成多个块，实现按需加载和提高应用性能。可以使用动态导入、SplitChunks插件等方式进行代码分割。
* 解析（Resolve）：Webpack会解析模块之间的依赖关系，通过解析规则来确定模块的依赖关系。可以通过配置resolve属性来设置模块的解析规则。

**[Webpack]什么是loader？有哪些常见的Loader？你用过哪些Loader？**

Webpack默认只认识JS，对于非JS的文件，比方说样式，图片，文件，json等等，就需要一些工具来帮忙翻译。而loader，就是那个翻译官，可以解析非原生JS的代码或文件。

raw-loader：加载文件原始内容（utf-8）

file-loader：把文件输出到一个文件夹中，在代码中通过相对 URL 去引用输出的文件 (处理图片和字体)

url-loader：与 file-loader 类似，区别是用户可以设置一个阈值，大于阈值会交给 file-loader 处理，小于阈值时返回文件 base64 形式编码 (处理图片和字体)

source-map-loader：加载额外的 Source Map 文件，以方便断点调试

svg-inline-loader：将压缩后的 SVG 内容注入代码中

image-loader：加载并且压缩图片文件

json-loader 加载 JSON 文件（默认包含）

handlebars-loader: 将 Handlebars 模版编译成函数并返回

babel-loader：把 ES6 转换成 ES5

ts-loader: 将 TypeScript 转换成 JavaScript

awesome-typescript-loader：将 TypeScript 转换成 JavaScript，性能优于 ts-loader

sass-loader：将SCSS/SASS代码转换成CSS

css-loader：加载 CSS，支持模块化、压缩、文件导入等特性

style-loader：把 CSS 代码注入到 JavaScript 中，通过 DOM 操作去加载 CSS

postcss-loader：扩展 CSS 语法，使用下一代 CSS，可以配合 autoprefixer 插件自动补齐 CSS3 前缀

eslint-loader：通过 ESLint 检查 JavaScript 代码

tslint-loader：通过 TSLint检查 TypeScript 代码

mocha-loader：加载 Mocha 测试用例的代码

coverjs-loader：计算测试的覆盖率

vue-loader：加载 Vue.js 单文件组件

i18n-loader: 国际化

cache-loader: 可以在一些性能开销较大的 Loader 之前添加，目的是将结果缓存到磁盘里

loader配置步骤：

npm下载对应的loader。

在module选项里配置rules，每个rule是个对象，用来表示对一个文件的处理规则，test表示要处理的文件，use里可以通过配置多个loader来处理。要注意loader的执行顺序为：从下到上，从右到左。

module.exports = {

// loader

module: {

rules: [

{

test: /\.css$/,

use: [

{ loader: 'style-loader' },

{

loader: 'css-loader',

options: {

modules: true,

},

},

{ loader: 'sass-loader' },

],

},

],

},

};

**[Webpack]什么是plugin？有哪些常见的Plugin？你用过哪些Plugin？**

plugin，即插件。Webpack插件是对Webpack功能的扩展和增强，可以帮助我们在打包过程中自动执行一些额外的操作，例如生成HTML文件、压缩代码、提取CSS等。

define-plugin：定义环境变量 (Webpack4 之后指定 mode 会自动配置)

ignore-plugin：忽略部分文件

html-webpack-plugin：简化 HTML 文件创建 (依赖于 html-loader)

web-webpack-plugin：可方便地为单页应用输出 HTML，比 html-webpack-plugin 好用

uglifyjs-webpack-plugin：不支持 ES6 压缩 (Webpack4 以前)

terser-webpack-plugin: 支持压缩 ES6 (Webpack4)

webpack-parallel-uglify-plugin: 多进程执行代码压缩，提升构建速度

mini-css-extract-plugin: 分离样式文件，CSS 提取为独立文件，支持按需加载 (替代extract-text-webpack-plugin)

serviceworker-webpack-plugin：为网页应用增加离线缓存功能

clean-webpack-plugin: 目录清理

ModuleConcatenationPlugin: 开启 Scope Hoisting

speed-measure-webpack-plugin: 可以看到每个 Loader 和 Plugin 执行耗时 (整个打包耗时、每个 Plugin 和 Loader 耗时)

webpack-bundle-analyzer: 可视化 Webpack 输出文件的体积 (业务组件、依赖第三方模块)

plugin配置步骤：

npm下载要用的plugin。

在plugins选项里配置plugin，每个plugin是一个类，new这个类，然后可以根据文档和需求配置option即可。

const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {

// ...其他配置

plugins: [

new HtmlWebpackPlugin({

template: './src/index.html'

})

]

}

**[Webpack] loader和plugin区别**

【Loader】：用于对模块源码的转换，loader描述了webpack如何处理非javascript模块，并且在buld中引入这些依赖。loader可以将文件从不同的语言（如TypeScript）转换为JavaScript，或者将内联图像转换为data URL。比如说：CSS-Loader，Style-Loader等。

loader的使用很简单：

在webpack.config.js中指定loader。module.rules可以指定多个loader，对项目中的各个loader有个全局概览。

loader是运行在NodeJS中，可以用options对象进行配置。plugin可以为loader带来更多特性。loader可以进行压缩，打包，语言翻译等等。

loader从模板路径解析，npm install node\_modules。也可以自定义loader，命名XXX-loader。

语言类的处理器loader：CoffeeScript，TypeScript，ESNext（Bable）,Sass,Less,Stylus。任何开发技术栈都可以使用webpack。

【Plugin】：目的在于解决loader无法实现的其他事，从打包优化和压缩，到重新定义环境变量，功能强大到可以用来处理各种各样的任务。webpack提供了很多开箱即用的插件：CommonChunkPlugin主要用于提取第三方库和公共模块，避免首屏加载的bundle文件，或者按需加载的bundle文件体积过大，导致加载时间过长，是一把优化的利器。而在多页面应用中，更是能够为每个页面间的应用程序共享代码创建bundle。

**[Webpack] Webpack事件机制了解吗？**

Webpack常见的事件有：

before-run: 在Webpack开始执行构建之前触发，可以用于清理上一次构建的临时文件或状态。

run: 在Webpack开始执行构建时触发。

before-compile: 在Webpack开始编译代码之前触发，可以用于添加一些额外的编译配置或预处理代码。

compile: 在Webpack开始编译代码时触发，可以用于监听编译过程或处理编译错误。

this-compilation: 在创建新的Compilation对象时触发，Compilation对象代表当前编译过程中的所有状态和信息。

compilation: 在Webpack编译代码期间触发，可以用于监听编译过程或处理编译错误。

emit: 在Webpack生成输出文件之前触发，可以用于修改输出文件或生成一些附加文件。

after-emit: 在Webpack生成输出文件后触发，可以用于清理中间文件或执行一些其他操作。

done: 在Webpack完成构建时触发，可以用于生成构建报告或通知开发者构建结果。

2. Webpack的事件机制是基于Tapable实现的，Tapable是Webpack事件机制的核心类，它封装了事件的订阅和发布机制。在Webpack中，Compiler对象和Compilation对象都是Tapable类的实例对象。

**[Webpack]什么是HMR，原理是什么？**

HMR：即**热更新**，简单说就是在我们写代码保存后不需要手动刷新浏览器，就能直接看到更新后的结果，而且只改变我们更改的那部分内容。

HMR的原理：将需要更新的模块通过websocket与Webpack Dev Server建立连接，当模块发生变化时，Webpack Dev Server会将新的模块代码推送给浏览器端，浏览器端通过将新代码插入到运行时环境中，来实现实时更新。

怎么配置HMR：

在配置文件中添加webpack.HotModuleReplacementPlugin插件。

在webpack-dev-server的配置中添加hot: true，启用热替换。

在entry中添加hot module replacement runtime。

在模块代码中使用module.hot.accept方法，以接受新模块的更新。

HMR只适用于开发环境，不能用于生产环境，因为HMR需要额外的代码和性能消耗。在生产环境中，应该禁用HMR，使用正常的文件更新机制。

**[Webpack]说一下 Webpack 的热更新原理吧(必考)**

Webpack 的热更新又称热替换（Hot Module Replacement），缩写为 HMR。 这个机制可以做到不用刷新浏览器而将新变更的模块替换掉旧的模块。

HMR的核心就是**客户端从服务端拉取更新后的文件**，准确的说是 chunk diff (chunk 需要更新的部分)，实际上 WDS 与浏览器之间维护了一个 Websocket，当本地资源发生变化时，WDS 会向浏览器推送更新，并带上构建时的 hash，让客户端与上一次资源进行对比。客户端对比出差异后会向 WDS 发起 Ajax 请求来获取更改内容(文件列表、hash)，这样客户端就可以再借助这些信息继续向 WDS 发起 jsonp 请求获取该chunk的增量更新。

后续的部分(拿到增量更新之后如何处理？哪些状态该保留？哪些又需要更新？)由 HotModulePlugin 来完成，提供了相关 API 以供开发者针对自身场景进行处理，像react-hot-loader 和 vue-loader 都是借助这些 API 实现 HMR。

**[Webpack]模块打包原理知道吗？**

Webpack 实际上**为每个模块创造了一个可以导出和导入的环境**，本质上并没有修改代码的执行逻辑，代码执行顺序与模块加载顺序也完全一致。

**[Webpack]文件监听原理呢？**

在发现源码发生变化时，自动重新构建出新的输出文件。

Webpack开启监听模式，有两种方式：

* 启动 webpack 命令时，带上 --watch 参数
* 在配置 webpack.config.js 中设置 watch:true

缺点：每次需要手动刷新浏览器

原理：轮询判断文件的最后编辑时间是否变化，如果某个文件发生了变化，并不会立刻告诉监听者，而是先缓存起来，等 aggregateTimeout 后再执行。

module.export = { // 默认false,也就是不开启 watch: true, // 只有开启监听模式时，watchOptions才有意义 watchOptions: { // 默认为空，不监听的文件或者文件夹，支持正则匹配 ignored: /node\_modules/, // 监听到变化发生后会等300ms再去执行，默认300ms aggregateTimeout:300, // 判断文件是否发生变化是通过不停询问系统指定文件有没有变化实现的，默认每秒问1000次 poll:1000 }}

**[Webpack]如何对bundle体积进行监控和分析？**

VSCode 中有一个插件 Import Cost 可以帮助我们对引入模块的大小进行实时监测，还可以使用 webpack-bundle-analyzer 生成 bundle 的模块组成图，显示所占体积。

bundlesize 工具包可以进行自动化资源体积监控。

**[Webpack]文件指纹是什么？怎么用？**

文件指纹是打包后输出的文件名的后缀。

Hash：和整个项目的构建相关，只要项目文件有修改，整个项目构建的 hash 值就会更改

Chunkhash：和 Webpack 打包的 chunk 有关，不同的 entry 会生出不同的 chunkhash

Contenthash：根据文件内容来定义 hash，文件内容不变，则 contenthash 不变

**什么是文件指纹？**

Webpack的文件指纹是指在打包过程中为每个文件生成唯一的标识符，以便于版本管理和缓存控制。比方说Vue项目打包后生成的css文件和js文件，一般都会有奇奇怪怪的文件名，那就是文件指纹。

文件指纹的实现原理是根据文件内容生成哈希值，一般是利用Webpack内置的HashedModuleIdsPlugin和MiniCssExtractPlugin来实现。

**JS的文件指纹设置**

设置 output 的 filename，用 chunkhash。

module.exports = {

entry: {

app: './scr/app.js',

search: './src/search.js'

},

output: {

filename: '[name][chunkhash:8].js',

path: \_\_dirname + '/dist'

}

}

**CSS的文件指纹设置**

设置 MiniCssExtractPlugin 的 filename，使用 contenthash。

module.exports = {

entry: {

app: './scr/app.js',

search: './src/search.js'

},

output: {

filename: '[name][chunkhash:8].js',

path: \_\_dirname + '/dist'

},

plugins: [

new MiniCssExtractPlugin({

filename: `[name][contenthash:8].css`

})

]

}

**图片的文件指纹设置**

设置file-loader的name，使用hash。

占位符名称及含义

* ext 资源后缀名
* name 文件名称
* path 文件的相对路径
* folder 文件所在的文件夹
* contenthash 文件的内容hash，默认是md5生成
* hash 文件内容的hash，默认是md5生成
* emoji 一个随机的指代文件内容的emoj

const path = require('path');

module.exports = {

entry: './src/index.js',

output: {

filename: 'bundle.js',

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

},

module: {

rules: [{

test: /\.(png|svg|jpg|gif)$/,

use: [{

loader: 'file-loader',

options: {

name: 'img/[name][hash:8].[ext]'

}

}]

}]

}

}

**[Webpack]在实际工程中，配置文件上百行乃是常事，如何保证各个loader按照预想方式工作？**

可以使用 enforce 强制执行 loader 的作用顺序，pre 代表在所有正常 loader 之前执行，post 是所有 loader 之后执行。(inline 官方不推荐使用)

**[Webpack] 为什么要进行代码分割？**

前端代码体积变大，调试和上线都需要很长的编译时间，开发时修改一行代码也要重新打包整个脚本。

用户需要花额外的时间和带宽下载更大体积的脚本文件。

一、按需加载 首次加载只加载必要的内容，提升用户的首次加载的速度。其他的模块可以根据用户的交互进行按需加载，即用户跳转新路由或者点击的页面的时候再进行加载。

二、有效利用缓存 通过webpack在打包是对代码进行分割，可以有效的利用缓存：打包编译的时候，只需要编译需要更新的部分；用户访问的时候只需要下载被修改的文件即可。

场景： 你有一个体积巨大的文件，并且只改了一行代码，用户仍然需要重新下载整个文件。但是如果你把它分为了两个文件，那么用户只需要下载那个被修改的文件，而浏览器则可以从缓存中加载另一个文件。

三、预获取/预加载模块 prefetch(预获取)：将来某些导航下可能需要的资源：这会生成 <link rel="prefetch" href="login-modal-chunk.js">并追加到页面头部，指示着浏览器在闲置时间预取login-modal-chunk.js 文件。 preload(预加载)：当前导航下可能需要资源 --- preload chunk 会在父 chunk加载时，以并行方式开始加载。prefetch chunk 会在父 chunk 加载结束后开始加载。 --- preload chunk 具有中等优先级，并立即下载。prefetch chunk 在浏览器闲置时下载。 --- preload chunk 会在父 chunk 中立即请求，用于当下时刻。prefetch chunk 会用于未来的某个时刻。 --- 浏览器支持程度不同。

**[Webpack]是否写过Loader？简单描述一下编写loader的思路？**

Loader 支持链式调用，所以开发上需要严格遵循“单一职责”，每个 Loader 只负责自己需要负责的事情。

[Loader的API](https://link.juejin.cn?target=https://www.webpackjs.com/api/loaders/" \o "https://www.webpackjs.com/api/loaders/" \t "/Users/gdj/Documents\\x/_blank) 可以去官网查阅

Loader 运行在 Node.js 中，我们可以调用任意 Node.js 自带的 API 或者安装第三方模块进行调用

Webpack 传给 Loader 的原内容都是 UTF-8 格式编码的字符串，当某些场景下 Loader 处理二进制文件时，需要通过 exports.raw = true 告诉 Webpack 该 Loader 是否需要二进制数据

尽可能的异步化 Loader，如果计算量很小，同步也可以

Loader 是无状态的，我们不应该在 Loader 中保留状态

使用 loader-utils 和 schema-utils 为我们提供的实用工具

加载本地 Loader 方法

Npm link

ResolveLoader

**[Webpack]是否写过Plugin？简单描述一下编写Plugin的思路？**

webpack在运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在特定的阶段钩入想要添加的自定义功能。Webpack 的 Tapable 事件流机制保证了插件的有序性，使得整个系统扩展性良好。

[Plugin的API](https://link.juejin.cn?target=https://www.webpackjs.com/api/plugins/" \o "https://www.webpackjs.com/api/plugins/" \t "/Users/gdj/Documents\\x/_blank) 可以去官网查阅

compiler 暴露了和 Webpack 整个生命周期相关的钩子

compilation 暴露了与模块和依赖有关的粒度更小的事件钩子

插件需要在其原型上绑定apply方法，才能访问 compiler 实例

传给每个插件的 compiler 和 compilation对象都是同一个引用，若在一个插件中修改了它们身上的属性，会影响后面的插件

找出合适的事件点去完成想要的功能

emit 事件发生时，可以读取到最终输出的资源、代码块、模块及其依赖，并进行修改(emit 事件是修改 Webpack 输出资源的最后时机)

watch-run 当依赖的文件发生变化时会触发

异步的事件需要在插件处理完任务时调用回调函数通知 Webpack 进入下一个流程，不然会卡住

**[Webpack] 怎么优化开发环境？**

**开发环境常见的问题有**

启动慢。

编译慢，尤其是当项目变大时，修改一行代码要等好几秒甚至十几秒才能看到效果。

占用内存高，当模块数变多时，项目运行占用内存飙升，导致电脑卡顿，影响开发效率。

**优化措施**

**使用缓存**：可以使用Webpack的缓存功能，将打包结果缓存起来以避免重复构建。可以使用cache-loader或hard-source-webpack-plugin等插件来实现缓存。

**使用 DllPlugin分包**：DllPlugin 是 Webpack 的一个插件，它可以将一些不经常变动的第三方库预先打包好，然后在开发过程中直接使用。这样可以减少每次构建时对这些库的重复打包，提高构建速度。

**配置合适的SourceMap策略**：在开发环境下，开启 SourceMap 可以帮助我们快速定位错误和调试代码。但是开启 SourceMap 会降低构建速度，所以需要权衡是否开启。

**多线程并行打包**：可以使用thread-loader或happypack开启多线程并行构建，但是并不是一定会提升性能，需要结合场景来自行取舍，比较适合单个耗时长的任务。

**配置模块解析**：Webpack 在模块解析时会搜索 node\_modules 目录，这个过程比较耗时。为了减少搜索时间，我们可以使用 resolve.alias 配置选项来告诉 Webpack 直接使用特定的路径来查找模块。

**使用新技术，比如Webpack5或者Vite这些性能更好的构建工具**。

**[Webpack]有了解过Webpack5吗，相比于Webpack4有哪些提升？**

Webpack5相对于Webpack4有以下提升：

**更快的构建速度**：Webpack5在构建速度方面进行了大量优化，尤其是在开发模式下，构建速度有了明显提升。

**Tree Shaking优化**：Webpack5进一步改进了Tree Shaking算法，可以更准确地判断哪些代码是无用的，从而更好地优化构建输出的文件大小。

**内置的持久化缓存**：Webpack5在持久化缓存方面进行了优化，可以缓存每个模块的编译结果，从而加速后续的构建。

**支持WebAssembly**：Webpack5增加了对WebAssembly的原生支持。

**模块联邦（Module Federation）**：Webpack5引入了模块联邦的概念，可以实现多个独立的Webpack构建之间的模块共享和远程加载，为微前端架构提供了更好的支持。

**[Webpack]配置代码太多，达到数千行，这个时候该如何优化配置代码【热度: 186】**

当Webpack配置代码变得冗⻓和难以管理时，可以采取以下⽅法来优化配置：

* 配置⽂件拆分

将配置⽂件分成多个部分，每个⽂件只负责⼀部分逻辑。⽐如基础配置、开发环境配置、⽣产环境配置等。

* 使⽤环境变量

通过环境变量来区分不同的配置环境，使⽤ webpack-merge 或 env-cmd 这样的库来合并配置。

* 模块化配置

将常⽤的loader、plugins、entry等配置项封装成函数或者模块，然后在主配置⽂件中引⼊。

* 使⽤webpack-merge抽离通⽤配置

检查配置中的重复部分，将它们抽象成共⽤的配置，再使⽤ webpack-merge 来合并多个配置⽂件，保持配置的清晰和可维护性。

* 统⼀管理插件和加载器

如果项⽬中使⽤了⼤量插件和加载器，请考虑将它们的实例化和配置逻辑封装在单独的函数或⽂件中。然后根据不同的环境，直接pick不同的配置即可。可以达到配置的loader和plugin集中管理。

**[Webpack]你用过哪些可以提高效率的插件?【热度: 179]**

webpack-dashboard：可以更友好的展⽰相关打包信息。

webpack-merge：提取公共配置，减少重复配置代码

speed-measure-webpack-plugin：简称SMP，分析出Webpack打包过程中Loader和Plugin的耗时，有助于找到构建过程中的性能瓶颈。

size-plugin：监控资源体积变化，尽早发现问题

HotModuleReplacementPlugin：模块热替换

webpack.ProgressPlugin：打包进度分析

webpack-bundle-analyzer：打包结果分析

friendly-errors-webpack-plugin：代码源码编译报错友好提⽰

**[Webpack] ts编写的库，在使用webpack构建的时候，如何对外提供d.ts【热度:224】**

在TypeScript项目中，当你想要在使用webpack构建时提供类型声明文件（.d.ts），你需要确保你的TypeScript编译器配置正确，并且在构建过程中包含了这些声明文件。

**1.配置 TypeScript 编译选项：**在库项目的根目录下创建或编辑 tsconfig.json 文件，确保编译器配置选项如下：

{

"compilerOptions": {

"declaration": true, // 生成对应的 '.d.ts' 文件

"declarationDir": "types", // 指定生成的声明文件存放目录

"outDir": "lib" // 指定编译后文件的输出目录

// 其他需要的编译选项

},

"include": ["src/\*\*/\*"], // 包含源码的目录

"exclude": ["node\_modules"] // 排除的目录

}

* declaration: 这个选项会告诉 TypeScript 编译器为每个 .ts 文件生成相应的 .d.ts 声明文件。
* declarationDir: 这是指定声明文件的输出目录。

**2.配置 Webpack：**

在我们的 Webpack 配置中（通常是 webpack.config.js），我们需要设置 output 以指向我们的输出目录，同时可能需要使用一些加载器(loader)如 ts-loader 或 babel-loader 来处理 TypeScript 文件。

一个简单的 webpack 配置示例可能如下：

const path = require("path");

module.exports = {

entry: "./src/index.ts", // 入口文件

module: {

rules: [

{

test: /\.tsx?$/,

use: "ts-loader",

exclude: /node\_modules/,

},

],

},

resolve: {

extensions: [".tsx", ".ts", ".js"],

},

output: {

filename: "your-library.js", // 输出文件名

path: path.resolve(\_\_dirname, "lib"), // 输出文件夹

libraryTarget: "umd", // 使库支持各种模块系统

globalObject: "this",

},

};

**3.发布包：**

当你发布你的库时，你需要确保 package.json 文件中包含 types 或 typings 字段指向入口 .d.ts 文件。

{

"name": "your-library",

"version": "1.0.0",

"main": "lib/your-library.js",

"typings": "types/index.d.ts"

// 其他配置项...

}

这告诉使用你库的 TypeScript 用户，在哪里可以找到类型声明文件。

**4.保证类型声明文件的发布：**

如果你的 npm 发布流程排除了 types 目录，你需要更新 .npmignore 文件来确保 .d.ts 文件会被包含在发布的 npm 包中。

完成这些配置后，当你用 webpack 构建并发布你的库时，用户将能够获得与 JavaScript 文件关联的 TypeScript 类型声明，以便在他们的 TypeScript 项目中获得类型检查和智能提示。

**[Webpack]如何提取复用代码给多个entry使用?【热度:292】**

在Webpack中提取源码⾥被多个⼊⼝点复⽤的代码，例如⼀个 utils ⽂件，可以通过配置optimization.splitChunks 来实现。Webpack会将这些频繁复⽤的模块提取出来，打包到⼀个独⽴的chunk中，使得浏览器可以单独缓存这部分代码，并在多个⻚⾯间共享使⽤，优化加载性能。

使⽤ splitChunks 的基本配置如下：



这个配置的含义是：

• chunks: 'all' 指定要优化的chunk类型，这⾥设置为 all 代表所有的chunk，不管是动态还是⾮动态加载的模块。

• cacheGroups 是⼀个对象，⽤于定义缓存组，可以继承和/或覆盖 splitChunks 的任何选项。每个缓存组可以有⾃⼰的配置，将不同的模块提取到不同的⽂件中。

• cacheGroups.commons 定义了⼀个缓存组，专⻔⽤于提取 initial chunk（最初依赖的模块）中被⾄少两个chunk所共享的模块。

• name: 'commons' 为⽣成的⽂件定义了⼀个⾃定义名称。

• minChunks: 2 表⽰模块⾄少被两个⼊⼝点引⽤时，才会被提取。

• minSize: 0 指定模块的最⼩体积是0，即任意⼤⼩的模块都被提取。

这会让任何从 node\_modules ⽬录导⼊，并在⾄少两个⼊⼝点中使⽤的模块，都会被打包到⼀个名为 commons.js 的⽂件中（当然，实际的⽂件名会受到 output 配置的影响，例如是否包含哈希值等）。

正确配置这些参数后， utils 这样的模块就会被⾃动提取并共享，⽽不是在每个⼊⼝点的bundle中重复包含。这样做的好处是，任何更新业务逻辑的时候，只要 utils 没有发⽣变化，⽤户浏览器上已缓存的 commons.js ⽂件就不需要重新下载。

**[Webpack]如何将一些通用的依赖打包成一个独立的bundle【热度: 643】**

在Webpack中，将⼀些通⽤的依赖，如React、ReactDOM、ReactRouter等库和框架，打包成⼀个独⽴的bundle，通常是为了⻓期缓存和减少每次部署更新的下载量。这可以通过"代码分割"(codesplitting)和"优化"(optimization)配置来实现。

以下是Webpack中分离通⽤依赖的⼏个步骤：

1. 使⽤ entry 来定义不同的⼊⼝点:可以通过配置⼀个额外的⼊⼝来创建⼀个只包含通⽤库的bundle，也就是所谓的"vendor"bundle。



2. 使⽤ SplitChunksPlugin :这个插件可以将共享代码分割成不同的chunks，并可以通过配置将其从业务代码中分离出来。在Webpack4及之后的版本中，默认内置了 optimization.splitChunks ，就是这个插件的配置⽅法。



3. 配置 output :虽然不是必须的，你还可以在output中定义 filename 和 chunkFilename ，来控制主⼊⼝和⾮主⼊⼝chunks的⽂件名。



通过这样的配置，Webpack在打包时会⾃动将node\_modules中的依赖和业务代码分离开来，业务代码会被打包到 main chunk中，⽽第三⽅库则会打包到 vendors chunk。

**[Webpack] output配置里面， chunkFilename和filename区别是什么?【热度:210】**

在 Webpack 中的 output 配置对象中，filename 和 chunkFilename 是用来指定输出文件的命名方式的关键属性。它们之间的区别主要涉及到最终生成的 JavaScript 文件的类型。

filename: filename 属性用于指定输出的 bundle 的名称。当你的应用只有一个入口点时，可以直接使用一个固定名称。如果有多个入口点，那么你可以使用占位符来确保每个文件具有唯一的名称，如使用 [name] 来对应每个入口点的名称。filename 主要与入口点相关联的那些文件有关。

chunkFilename: chunkFilename 属性用于指定非入口的 chunk（通常是动态加载的模块）的名称。这些 chunk 文件通常是由于代码分割产生的。当使用如 import() 这样的动态导入语法时，Webpack 会分割代码到新的 chunk 中，这时候 chunkFilename 的命名规则就会被应用。

这意味着如果你有一个动态加载的模块（例如懒加载模块），Webpack 会使用 chunkFilename 的规则来生成这些额外的文件。同样，你也可以在 chunkFilename 中使用占位符来保持文件名的唯一性。常用的占位符有 [id], [name], [chunkhash] 等。

使用这两个属性使得 Webpack 能够区分出入口文件和其他类型的文件，从而允许开发者更好地控制输出资源的命名和缓存。

output: {

filename: "bundle.js"; // 一个固定名称，适用于单入口

// 或者

filename: "[name].bundle.js"; // 使用占位符，适用于多入口

chunkFilename: "[name].chunk.js";

}

**[Webpack] 多入口打包共享模块【热度: 337】**

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"],

},

};

完整案例:

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 in the main chunk.

},

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

},

},

};

**[Webpack]如何使用ts来编写配置文件?【热度:251】**

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

# 如果使用版本低于 v4.7.0 的 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"

}

}

}

**第三种方法**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

}

}

提示

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。

**[webpack]工作原理/内部执行原理【热度:668】**

**1、核心概念**

（1）entry：一个可执行模块或者库的入口。

（2）chunk：多个文件组成一个代码块。可以将可执行的模块和他所依赖的模块组合成一个chunk，这是打包。

（3）loader：文件转换器。例如把es6转为es5，scss转为css等

（4）plugin：扩展webpack功能的插件。在webpack构建的生命周期节点上加入扩展hook，添加功能

**2、webpack构建流程（原理）**

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：

**解析配置文件**：Webpack会读取项目根目录下的Webpack配置文件，解析其中的配置项，并根据配置项构建打包流程。

**解析模块依赖**：Webpack会从entry配置中指定的入口文件开始，递归解析模块之间的依赖关系，并构建模块依赖图谱。

**加载模块**：Webpack会根据模块依赖图谱，加载所有需要打包的模块，通过配置的loader将文件转换成Webpack可识别的模块。

**执行插件**：Webpack会在打包流程中执行一系列插件，插件可以用于完成各种任务，例如生成HTML文件、压缩代码等等。

**输出打包结果**：Webpack会将打包后的代码和资源输出到指定的输出目录，可以使用配置项进行相关设置。

**监听变化**：在开发模式下，Webpack会在代码修改后重新构建打包流程，并将修改后的代码热更新到浏览器中。

**3、简单说**

1. 初始化: 读取配置文件，初始化构建过程，配置 resolve, loaders 等。
2. 确定入口: 根据配置中的 entry 入口开始解析文件。
3. 编译模块: 使用 loader 解析文件，转换模块。
4. 依赖分析: 解析出 require/import 的依赖，形成依赖图。
5. 打包优化: 优化构建结果，例如 tree-shaking，scope hoisting 等。
6. 输出结果: 生成 chunk 文件，并可以输出到各种资源中。

**[Webpack]为何不支持CMD模块化【热度: 255】**

**关键词：webpack 模块化支持**

* CMD 是国内产品， webpack 是国外产品， 而且 CMD 还没有火起来的时候， 就已经被 ESM 替代了
* CMD 是更加懒惰，是依赖就近，延迟执行。也就是说，在模块中需要用到依赖时，才去引入依赖。这和 Webpack 的理念以及模块收集和打包机制不兼容

CMD（Common Module Definition）是一种深受国内前端开发者喜爱的模块定义规范，主要被用在了 Sea.js 这个模块加载器中。CMD 是国内开发者提出的规范，它和 AMD 很相似，但是更符合国内开发者的习惯，需要时可以延迟执行。

Webpack 本身是围绕 NPM 生态和标准化模块格式（如 ES Modules 和 CommonJS）构建的，而 NPM 生态主要使用的是 CommonJS 形式。因此，对于大多数使用 NPM 之 Webpack 的用户来说，这些就足够用了。而 ES Modules 作为 JavaScript 官方的模块系统标准，越来越多地在现代应用中被采用。

面对 CMD，Webpack 的社区并没有广泛地采用或者需要支持这种模块定义。CMD 在模块定义时依赖于具体的 API 和加载时机，这和 Webpack 的理念以及模块收集和打包机制不完全兼容。Webpack 鼓励在编译时就确定模块依赖，而 CMD 更倾向于运行时动态确定。

尽管如此，理论上是可以通过一些插件或 loader 来实现对 CMD 模块的支持的，但是官方并没有集成这样的功能，因为需求没有那么大，同时现有的模块加载机制已经可以满足绝大多数场景的需要。随着前端工程化的深入，标准化的模块定义（如 ES Modules）更加受到青睐，而特定的模块定义（如 CMD）则逐渐被边缘化。因此，Webpack 没有默认支持 CMD，也反映了当前前端模块化开发的趋势和实践。

**[Webpack]支持哪些模块化加载?【热度:154】**

**关键词：webpack 模块化支持**

Webpack 支持以下几种模块化标准：

* ESM (ECMAScript Modules): 这是 JavaScript ES6 中引入的官方标准模块系统。使用 import 和 export 语句来导入和导出模块。
* CommonJS: 主要用于 Node.js，允许使用 require() 来加载模块和 module.exports 来导出模块。
* AMD (Asynchronous Module Definition): 用于异步加载模块，并使用 define 方法来定义模块。

UMD (Universal Module Definition): 结合了 AMD 和 CommonJS 的特点，并支持全局变量定义的方式，使得模块可以在客户端和服务端上运行。

除此之外，Webpack 还可以处理非 JavaScript 文件并将它们视为模块，例如 CSS, LESS, SASS, 图像文件(PNG, JPG, GIF, SVG 等), 字体(OTF, TTF, WOFF, WOFF2, EOT), HTML 以及任何其他类型的文件。这通过使用相应的 loader 来实现，如 style-loader, css-loader, file-loader 等。这些 loader 会将非 JavaScript 文件转换为可以被 Webpack 处理的模块。

**[webpack]的主要配置项有哪些【热度:766】**

Webpack是⼀个现代JavaScript应⽤程序的静态模块打包器。配置⽂件名通常为webpack.config.js ，它提供了⼀种配置Webpack的⽅式。下⾯是⼀些主要的Webpack配置选项：

1. entry:⼊⼝起点(entrypoint)指⽰webpack应该使⽤哪个模块，来作为构建其内部依赖图的开始。可以指定⼀个或多个⼊⼝起点。

2. output:output属性告诉webpack在哪⾥输出它所创建的bundles，以及如何命名这些⽂件，默认值为 ./dist 。

3. module:module属性⽤于决定如何处理项⽬中的不同类型的模块。

◦ rules:配置模块的读取和解析规则，通常⽤来配置loader。

4. resolve:配置模块如何解析。

◦ extensions:⾃动解析确定的扩展，此选项能够使⽤户在引⼊模块时不带扩展。

5. plugins:插件是⽤来扩展webpack功能的。它们会在构建流程中的特定时机注⼊运⾏逻辑来改变构建结果或做你想要的事情。

6. devServer:通过来⾃ webpack-dev-server 的这些选项能够对开发服务器的⾏为进⾏控制。

7. devtool:此选项控制是否⽣成，以及如何⽣成sourcemap。

8. mode:通过设置 development 或 production 之中的⼀个，来为流程提供相应模式下的内置优化。

9. optimization:包含⼀组可⽤来调整构建输出的选项。

◦ splitChunks:配置模块的拆分，可以将公共的依赖模块提取到已有的⼊⼝chunk中，或者提取到⼀个新⽣成的chunk。

◦ runtimeChunk:为每个entry创建⼀个运⾏时⽂件。

10. performance:允许webpack根据某些参数，控制资产和⼊⼝起点的最⼤⼤⼩。

11. externals:防⽌将某些import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。

每个项⽬的具体需求不同，Webpack的配置也会有所不同。这些选项提供了强⼤的配置能⼒和灵活性，可以⽤来定制Webpack的打包、加载和转换⾏为。

**[webpack] optimize配置中，分割代码配置splitChunks怎么使用【热度:546】**

**关键词：webpack 分割代码**

在 webpack 中，splitChunks选项是optimization对象的一个属性，可以用来定义如何分割代码块。默认情况下，webpack 会将所有来自node\_modules的模块分割到一个叫做vendors的 chunk 中，并且将共享或来自异步请求的模块分割成不同的 chunks。通过配置splitChunks选项，你可以控制这些行为，创建更细致的代码分割策略。以下是如何使用splitChunks来优化你的 bundle。

**基本配置**

module.exports = {

//...

optimization: {

splitChunks: {

chunks: "all", // 分割所有类型的chunks：初始和动态加载的chunk

},

},

};

在这个配置中，chunks: 'all'指示 webpack 对同步和异步引入的模块都进行分割。webpack 会根据内部的一些默认标准（如模块大小、请求的 chunks 数目等）来决定是否分割一个模块。

**[webpack] optimize 配置有哪些作用【热度:280】**

**关键词：webpack optimize**

Webpack 的 optimize 选项是在指定 Webpack 配置对象时，用于配置优化选项的一个属性。该属性下包含了一系列用于调整构建输出质量和性能的设置。这里是一些 optimize 属性中可能包含的选项：

* splitChunks：这用于代码分割，可以将公共的依赖模块提取到已有的入口 chunk 中，或者产生一个新的 chunk。这可以被用来得到更小的 bundle 体积，优化加载时间，或者更好的缓存利用。
* runtimeChunk：该选项将 Webpack 的运行时代码分割成一个单独的 chunk。使用这个设置有利于长期缓存，并且当你使用多个入口点时推荐使用。
* minimize：当设置为 true 时，Webpack 会启动代码压缩。通常，这会使用 UglifyJSPlugin 来进行 JavaScript 代码的压缩，但现在通常默认使用更现代的工具如 TerserPlugin。
* minimizer：当你想要覆盖默认的压缩工具或者提供额外的压缩工具时使用。
* noEmitOnErrors（早期版本称为 NoEmitOnErrorsPlugin）：启用该选项后，Webpack 编译错误将会导致不生成输出。这确保了不会发出包含错误的 assets。
* concatenateModules（早期版本称为 ModuleConcatenationPlugin）：这个选项会试图找到模块图中可以安全地连接到单一模块的所有模块，来优化结果的体积。
* usedExports（也称为 tree shaking）：该选项用于标记 "tree shaking" 中未被使用的导出，使它们能被压缩工具删除。

在 Webpack 4 及以上版本中，这些优化默认在 mode 被设置为 production 时生效。通过合理地配置这些选项，开发者可以显著提高应用程序的加载和运行性能。这些优化通常包括减少 bundle 的体积和提高代码的运行时效率。在开发模式下，很多优化默认是关闭的，以提供更快的构建速度和更好的调试体验。

**[webpack] mode是做什么用【热度: 475】**

**关键词：webpack mode**

在 webpack 中，mode 属性用来指定当前的构建环境是：development、production 或者是 none。设置 mode 可以使用 webpack 内置的函数，默认值为 production。

mode 属性的主要作用是：根据当前的构建环境，启用 webpack 内置在该环境下推荐的优化。

mode 的具体作用包括：

* development

主要优化了增量构建速度和开发体验。

process.env.NODE\_ENV 的值设为 development。

启用热替换模块（Hot Module Replacement）。

启用开发工具（如调试源码的 source map）以更好地进行调试。

* production

一些处理优化，以提升应用在生产环境的性能。

process.env.NODE\_ENV 的值设为 production。

启用代码压缩（例如 TerserPlugin）。

删除 dead code（通过 Tree Shaking）。

作用域提升等各种性能优化措施。

* none

mode 设置为 none 则不启用任何默认优化选项，process.env.NODE\_ENV 也不会被设置，默认为 production。

使用方法:

module.exports = {

mode: "development", // 'production' 或 'none'

// 其他配置...

};

// 或者，在命令行中使用 --mode 参数：

webpack --mode=production

设置 mode 是告诉 webpack 使用其内部的优化策略，各个模式预定义了一些 webpack 的行为，开发者可以不需要进行详细的配置，也能快速启动一个针对特定环境优化过的构建过程。

**[webpack]打包时hash码是如何生成的【热度: 167】**

**关键词：webpack hash 码的生成**

Webpack 在打包过程中生成 hash 码主要用于缓存和版本管理。主要有三种类型的 hash 码：

hash：是和整个项目的构建相关，只要项目文件有修改，整个项目构建的 hash 值就会更改。这意味着任何一个文件的改动都会影响到整体的 hash 值。

chunkhash：与 webpack 打包的 chunk 有关，不同的 entry 会生成不同的 chunkhash 值。例如，如果你的配置生成了多个 chunk（例如使用了 code splitting），每个 chunk 的更新只会影响到它自身的 chunkhash。

contenthash：根据文件内容来定义 hash，内容不变，则 contenthash 不变。这在使用诸如 CSS 提取到单独文件的插件时特别有用，因此只有当文件的内容实际改变时，浏览器才会重新下载文件。

生成方式：

hash 和 chunkhash 主要是通过某种 hash 算法（默认 MD5）来对文件名或者 chunk 数据进行编码。

contenthash 是通过构建时的 webpack 插件（如 mini-css-extract-plugin）来处理的，它会对文件内容进行 hash。

Hash 码的生成可以被 webpack 配置的 hashFunction，hashDigest，hashDigestLength 等选项影响。例如，你可以选择不同的算法如 SHA256 或者 MD5，以及可以决定 hash 值的长度。

在 webpack 的配置文件中，可以通过如下方式设定 hash:

output: {

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

path: \_\_dirname + '/dist'

}

这会将输出的文件名设置为入口名称加上基于每个 chunk 内容的 hash。在使用 webpack-dev-server 或者 webpack --watch 时，不会生成实际的文件，所以这些 hash 值是在内存中计算并关联的。

**Webpack项目中通过script标签引入资源，在项目中如何处理?【热度: 100】**

在使用 Webpack 打包的项目中，通常资源（如 JavaScript、CSS、图片等）会被 Webpack 处理，因为 Webpack 的设计初衷就是将所有资源视为模块，并进行有效的管理和打包。但有时候可能需要通过<script>标签直接引入资源，这通常有两种情况：

**1.在 HTML 文件中直接引入：**

可以在项目的 HTML 文件中直接使用<script>标签来引入外部资源：

<!-- 若要使用 CDN 上托管的库 -->

<script src="https://cdn.example.com/library.js"></script>

这种方法简单直接，但要记住，由于这些资源不会被 Webpack 处理，它们不会被包含在 Webpack 的依赖图中，并且也不会享受到 Webpack 的各种优化。

**2.使用 Webpack 管理：**

如果想要 Webpack 来处理这些通过<script>引入的资源，可以使用几种插件和加载器：

* html-webpack-plugin可以帮助你生成一个 HTML 文件，并在文件中自动引入 Webpack 打包后的 bundles。
* externals配置允许你将一些依赖排除在 Webpack 打包之外，但还是可以通过require或import引用它们。
* script-loader可以将第三方全局变量注入的库当作模块来加载使用。

例如，使用html-webpack-plugin和externals，你可以将一个库配置为 external，然后通过html-webpack-plugin将其引入：

// webpack.config.js 文件

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

// ...

externals: {

libraryName: "LibraryGlobalVariable",

},

plugins: [

new HtmlWebpackPlugin({

template: "src/index.html",

scriptLoading: "blocking", // 或者 'defer'

}),

],

};

然后，在你的index.html模板文件中可以这样引入资源：

<script src="https://cdn.example.com/library.js"></script>

使用externals的方法能让你在 Webpack 打包的模块代码中用正常的import或require语句来引用那个全局变量：

// 你的 JavaScript 代码文件中

import Library from "libraryName"; // 虽然定义了external，Webpack依然会处理这个import

应根据项目需求和现有的架构来决定使用哪种方法。上述两种方法中，第二种可以更好地利用 Webpack 的功能，第一种则更加简单直接。

**[webpack] webpack-dev-server为何不适用于线上环境?【热度: 88】**

**关键词：webpack-dev-server 作用**

原因有以下几点：

* 性能问题：webpack-dev-server是一个开发服务器，它将编译后的文件保存在内存中，并提供给浏览器访问。这种方式在开发阶段可以提供快速的编译和热模块替换，但在线上环境中，内存中保存的文件无法持久化，每次启动服务器都需要重新编译和打包，这会影响性能，并且占用服务器资源。
* 安全问题：webpack-dev-server并不进行代码压缩和混淆，源代码是以明文形式传输给浏览器的。这将使得攻击者可以很容易地获取到项目的源代码，从而可能导致安全漏洞。
* 缺少文件输出：webpack-dev-server并没有生成最终的打包文件，它只是将打包后的文件保存在内存中，提供给浏览器访问。这样就无法进行资源的版本管理和持久化存储，无法做到长期的缓存和优化。

**[webpack] webpack-dev-server 作用是啥?【热度: 387】**

* **关键词：webpack-dev-server 作用**

webpack-dev-server 是一个开发服务器，它提供了一个快速开发的环境，并且配合Webpack使用。它的作用主要有以下几个方面：

* 自动编译和打包：webpack-dev-server 可以监听源文件的变化，当文件发生改动时，它会自动重新编译和打包，保证开发过程中始终使用最新的代码。
* 热模块替换（Hot Module Replacement）：webpack-dev-server 支持热模块替换，即在开发过程中，当某个模块发生变化时，只会重新编译该模块，而不是整个应用，然后将变更的模块替换到浏览器中，从而实现实时更新，无需手动刷新页面。
* 代理和反向代理：webpack-dev-server 可以配置代理，用于解决前端开发中跨域请求的问题。可以将某些请求转发到其他服务器，或者改变请求的路径。
* 路由处理：webpack-dev-server 也支持将所有请求重定向到特定的 HTML 文件，用于前端单页应用的路由处理，可以通过配置实现 SPA（Single Page Application）的路由。
* 静态文件服务：webpack-dev-server 可以将打包后的文件提供给浏览器访问，可以通过配置指定静态文件的路径，并且通过指定的端口提供服务。

总结来说，webpack-dev-server 提供了一个方便的开发环境，可以实时编译和打包代码，并且支持热模块替换，代理和反向代理，路由处理等功能，提高了开发效率和开发体验。

**webpack-dev-server 有编译和打包的能力？**

webpack-dev-server本身并没有编译和打包的能力，它是使用Webpack来实现编译和打包的。

webpack-dev-server是基于Webpack的一个开发服务器，**它通过监听源文件的变化，自动调用Webpack进行编译和打包，并将打包后的文件提供给浏览器访问**。

这样可以在开发过程中实时更新代码，无需手动进行编译和打包操作。

webpack-dev-server还支持热模块替换等功能，提供了一个方便的开发环境。

但是需要注意的是，webpack-dev-server只适用于开发阶段，它并不会生成最终的打包文件，而是将打包后的文件保存在内存中，提供给浏览器访问。

在真正发布项目时，还需要运行Webpack的打包命令生成最终的打包文件。

**[Webpack]有哪些优化项目的手段?【热度: 1163】**

**关键词：打包优化**

围绕 webpack 做性能优化，分为两个方面：**构建时间优化、构建产物优化**

* 优化构建时间
* 缩小范围
* 文件后缀
* 别名
* 缓存
* 并行构建
* 定向查找第三方模块
* 构建产物
* 压缩 js
* 压缩 css
* 压缩 html
* 压缩图片
* 按需加载
* preload、prefetch
* 代码分割
* tree shaking
* gzip
* 作用域提升

### 构建时间优化

#### 缩小范围

我们在使用 loader 时，可以配置 include、exclude缩小 loader 对文件的搜索范围，以此来提高构建速率。

像 /node\_moudles 目录下的体积辣么大，又是第三方包的存储目录，直接 exclude 掉可以节省一定的时间的。

当然 exclude 和 include 可以一起配置，大部分情况下都是只需要使用 loader 编译 src 目录下的代码

module.exports = {

module: {

rules: [

{

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

// 只解析 src 文件夹下的 ts、tsx、js、jsx 文件

// include 可以是数组，表示多个文件夹下的模块都要解析

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

use: [ 'thread-loader', 'babel-loader'],

//当然也可以配置 exclude，表示 loader 解析时不会编译这部分文件

//同样 exclude 也可以是数组

exclude: /node\_modules/,

}

]

}

}

还需注意一个点就是要确保 loader 的准确性，比如不要使用 less-loader 去解析 css 文件

#### 文件后缀

resolve.extensions 是我们常用的一个配置，他可以在导入语句没有带文件后缀时，可以按照配置的列表，自动补上后缀。我们应该根据我们项目中文件的实际使用情况设置后缀列表，将使用频率高的放在前面、同时后缀列表也要尽可能的少，减少没有必要的匹配。同时，我们在源码中写导入语句的时候，尽量带上后缀，避免查找匹配浪费时间。

module.export = {

resolve: {

// 按照 tsx、ts、jsx、js 的顺序匹配，若没匹配到则报错

extensions: ['.tsx', '.ts', '.jsx', '.js'],

}

}

#### 别名

通过配置 resolve.alias 别名的方式，减少引用文件的路径复杂度

module.exports = {

resolve: {

alias: {

//把 src 文件夹别名为 @

//引入 src 下的文件就可以 import xxx from '@/xxx'

'@': path.join(\_\_dirname, '../src')

}

}

}

// 引入 src 下的某个模块时

import XXX from '@/xxx/xxx.tsx'

#### 缓存

在优化的方案中，缓存也是其中重要的一环。在构建过程中，开启缓存提升二次打包速度。

在项目中，js 文件是占大头的，当项目越来越大时，如果每次都需要去编译 JS 代码，那么构建的速度肯定会很慢的，所以我们可以配置 babel-loader 的缓存配置项 cacheDirectory 来缓存没有变过的 js 代码

module.exports = {

module: {

rules: [

{

test: /.jsx?$/,

use: [

{

loader: 'babel-loader',

options: {

cacheDirectory: true,

},

}

]

}

]

}

}

上面的缓存优化只是针对像 babel-loader 这样可以配置缓存的 loader，那没有缓存配置的 loader 该怎么使用缓存呢，此时需要 cache-loader

module.exports = {

module: {

rules: [

{

test: /.jsx?$/,

use: [

'cache-loader',

"babel-loader"

],

}

]

}

}

编译后同样多一个 /node\_modules/.cache/cache-loader 缓存目录

当然还有一种方式，webpack5直接提供了 cache 配置项，开启后即可缓存

module.exports = {

cache: {

type: 'filesystem'

}

}

编译后会多出 /node\_modules/.cache/webpack 缓存目录

#### 并行构建

首先，运行在Node里的webpack是单线程的，所以一次性只能干一件事，那如果利用电脑的多核优势，也能提高构建速度 ？thread-loader可以开启多进程打包

module.exports = {

module: {

rules: [

{

test: /.jsx?$/,

use: [

// 开启多进程打包。

{

loader: 'thread-loader',

options: {

workers: 3 // 开启 3个 进程

}

},

{

loader: 'babel-loader',

}

]

}

]

}

}

放置在这个 thread-loader 之后的 loader 就会在一个单独的 worker 池(worker pool) 中运行。

每个 worker 都是一个单独的有 600ms 限制的 node.js 进程。同时跨进程的数据交换也会被限制。所以建议仅在耗时的 loader 上使用。若项目文件不算多就不要使用，毕竟开启多个线程也会存在性能开销。

#### 定向查找第三方模块

resolve.modules 配置用于指定 webpack 去哪些目录下寻找第三方模块。默认值是 ['node\_modules']。而在引入模块的时候，会以 node 核心模块 -----> node\_modules ------> node全局模块 的顺序查找模块。

我们通过配置 resolve.modules 指定 webpack 搜索第三方模块的范围，提高构建速率

module.export = {

resolve: {

modules: [path.resolve(\_\_dirname, 'node\_modules')]

}

}

### 构建产物优化

#### 压缩 js

webpack5的话通过 terser-webpack-plugin 来压缩 JS，但在配置了 mode: production 时，会默认开启

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

module.exports = {

optimization: {

// 开启压缩

minimize: true,

// 压缩工具

minimizer: [

new TerserPlugin({}),

],

},

}

需要注意一个地方：生产环境会默认配置terser-webpack-plugin，所以如果你还有其它压缩插件使用的话需要将TerserPlugin显示配置或者使用...，否则terser-webpack-plugin会被覆盖。

const TerserPlugin = require("terser-webpack-plugin");

optimization: {

minimize: true,

minimizer: [

new TerserPlugin({}), // 显示配置

// "...", // 或者使用展开符，启用默认插件

// 其它压缩插件

new CssMinimizerPlugin(),

],

},

#### 压缩 css

压缩 css 我们使用 css-minimizer-webpack-plugin

同时，应该把 css 提取成单独的文件，使用 mini-css-extract-plugin

const MiniCssExtractPlugin = require("mini-css-extract-plugin");

const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");

module.exports = {

module: {

rules: [

{

test: /\.css$/,

use: [

// 提取成单独的文件

MiniCssExtractPlugin.loader,

"css-loader"

],

exclude: /node\_modules/,

},

]

},

plugins: [

new MiniCssExtractPlugin({

// 定义输出文件名和目录

filename: "asset/css/main.css",

})

],

optimization: {

minimize: true,

minimizer: [

// 压缩 css

new CssMinimizerPlugin({}),

],

},

}

#### 压缩 html

压缩 html 使用的还是 html-webpack-plugin 插件。该插件支持配置一个 minify 对象，用来配置压缩 html。

module.export = {

plugins: [

new HtmlWebpackPlugin({

// 动态生成 html 文件

template: "./index.html",

minify: {

// 压缩HTML

removeComments: true, // 移除HTML中的注释

collapseWhitespace: true, // 删除空⽩符与换⾏符

minifyCSS: true // 压缩内联css

},

})

]

}

#### 压缩图片

可以通过 image-webpack-loader 来实现

module.exports = {

module: {

rules: [

{

test: /\.(png|jpg|gif|jpeg|webp|svg)$/,

use: [

"file-loader",

{

loader: "image-webpack-loader",

options: {

mozjpeg: {

progressive: true,

},

optipng: {

enabled: false,

},

pngquant: {

quality: [0.65, 0.9],

speed: 4,

},

gifsicle: {

interlaced: false,

},

},

},

],

exclude: /node\_modules/, //排除 node\_modules 目录

},

]

},

}

#### 按需加载

很多时候我们不需要一次性加载所有的JS文件，而应该在不同阶段去加载所需要的代码。

**将路由页面/触发性功能单独打包为一个文件，使用时才加载**，好处是减轻首屏渲染的负担。因为项目功能越多其打包体积越大，导致首屏渲染速度越慢。

实际项目中大部分是对懒加载路由，而懒加载路由可以打包到一个 chunk 里面。比如某个列表页和编辑页它们之间存在相互跳转，如果对它们拆分成两个 import() js 资源加载模块，在跳转过程中视图会出现白屏切换过程。

因为在跳转期间，浏览器会动态创建 script 标签来加载这个 chunk 文件，在这期间，页面是没有任何内容的。

所以一般会把路由懒加载打包到一个 chunk 里面

const List = lazyComponent('list', () => import(/\* webpackChunkName: "list" \*/ '@/pages/list'));

const Edit = lazyComponent('edit', () => import(/\* webpackChunkName: "list" \*/ '@/pages/edit'));

但需要注意一点：**动态导入 import() 一个模块，这个模块就不能再出现被其他模块使用 同步 import 方式导入**。

比如，一个路由模块在注册 <Route /> 时采用动态 import() 导入，但在这个模块对外暴露了一些变量方法供其他子模块使用，在这些子模块中使用了同步 ESModule import 方式引入，这就造成了 动态 import() 的失效。

#### preload、prefetch

对于某些较大的模块，如果点击时再加载，那可能响应的时间反而延长。我们可以使用 prefetch、preload 去加载这些模块

* prefetch：将来可能需要一些模块资源（一般是其他页面的代码），在核心代码加载完成之后带宽空闲的时候再去加载需要用到的模块代码。
* preload：当前核心代码加载期间可能需要模块资源（当前页面需要的但暂时还没使用到的），其是和核心代码文件一起去加载的。

只需要通过魔法注释即可实现，以 prefetch 为例：

document.getElementById('btn1').onclick = function() {

import(

/\* webpackChunkName: "btnChunk" \*/

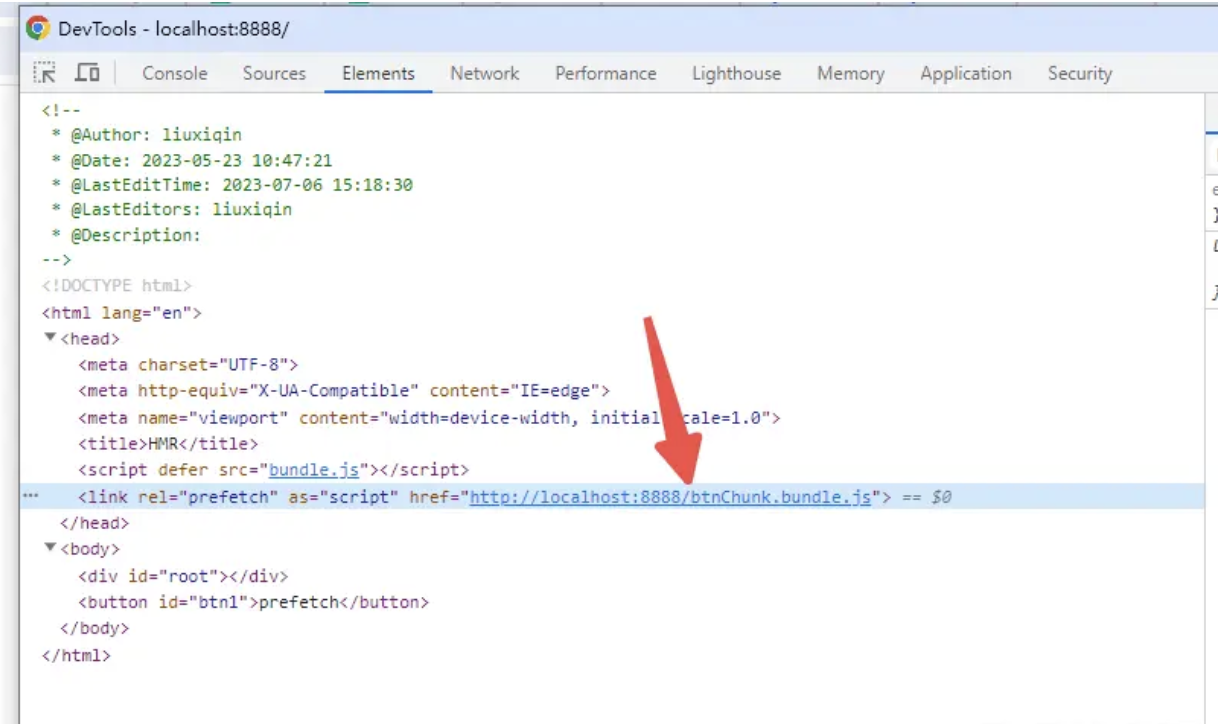
/\* webpackPrefetch: true\*/

'./module1.js'

).then(fn => fn.default());

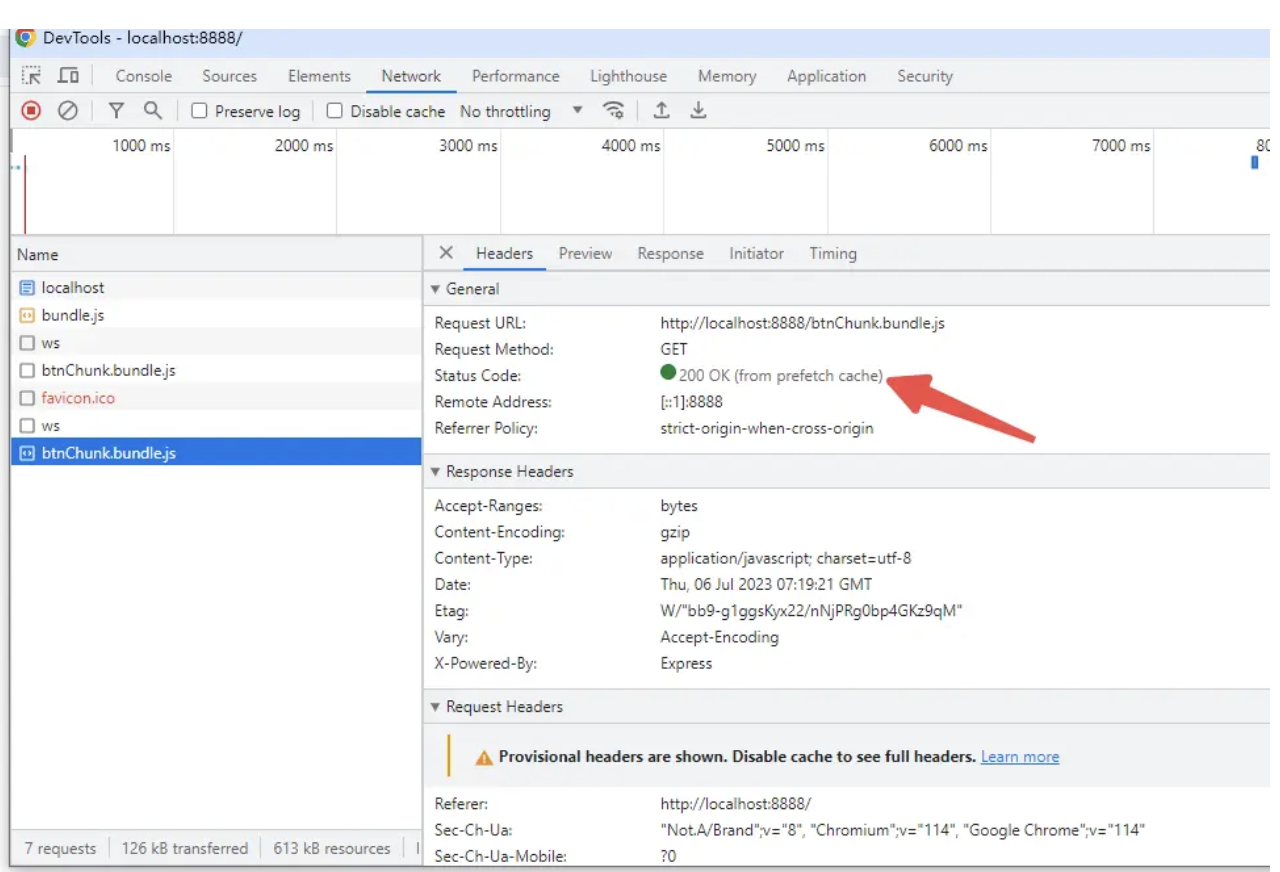
}

这行代码表示在浏览器空闲时加载 module1.js 模块，并且单独拆一个 chunk，叫做 btnChunk



可以看到，在head里面，我们的懒加载模块被直接引入了，并且加上了rel='prefetch'。

这样，页面首次加载的时候，浏览器空闲的会后会提前加载module1.js。当我们点击按钮的时候，会直接从缓存中读取该文件，因此速度非常快。



#### 代码分割

在项目中，一般是使用同一套技术栈和公共资源。**如果每个页面的代码中都有这些公开资源，就会导致资源的浪费**。在每一个页面下都会加载重复的公共资源，一是会浪费用户的流量，二是不利于项目的性能，造成页面加载缓慢，影响用户体验。

一般是把不变的第三方库、一些公共模块（比如 util.js）这些单独拆成一个 chunk，在访问页面的时候，就可以一直使用浏览器缓存中的资源

webpack 里面通过 **splitChunks** 来分割代码

module.exports = {

//...

optimization: {

splitChunks: {

chunks: 'async', // 值有 `all`，`async` 和 `initial`

minSize: 20000, // 生成 chunk 的最小体积（以 bytes 为单位）。

minRemainingSize: 0,

minChunks: 1, // 拆分前必须共享模块的最小 chunks 数。

maxAsyncRequests: 30, // 按需加载时的最大并行请求数。

maxInitialRequests: 30, // 入口点的最大并行请求数。

enforceSizeThreshold: 50000,

cacheGroups: {

defaultVendors: {

test: /[\/]node\_modules[\/]/, //第三方模块拆出来

priority: -10,

reuseExistingChunk: true,

},

util.vendors: {

test: /[\/]utils[\/]/, //公共模块拆出来

minChunks: 2,

priority: -20,

reuseExistingChunk: true,

},

},

},

},

};

#### tree shaking

tree shaking 的原理细节可以看这篇文章：[# webpack tree-shaking解析](https://juejin.cn/post/7246219936594821180)

**tree shaking在生产模式下已经默认开启了**

只是需要注意下面几点：

* 只对ESM生效
* 只能是静态声明和引用的 ES6 模块，不能是动态引入和声明的。
* 只能处理模块级别，不能处理函数级别的冗余。
* 只能处理 JS 相关冗余代码，不能处理 CSS 冗余代码。

而可能样式文件里面有些代码我们也没有使用，我们可以通过purgecss-webpack-plugin 插件来对 css 进行 tree shaking

const path = require("path");

const PurgecssPlugin = require("purgecss-webpack-plugin");

const glob = require("glob"); // 文件匹配模式

module.exports = {

//...

plugins: [

...

new PurgeCSSPlugin({

paths: glob.sync(`${PATH.src}/\*\*/\*`, { nodir: true }),

})

// Add your plugins here

// Learn more about plugins from https://webpack.js.org/configuration/plugins/

],

};

#### gzip

前端除了在打包的时候将无用的代码或者 console、注释剔除之外。我们还可以使用 Gzip 对资源进行进一步压缩。那么浏览器和服务端是如何通信来支持 Gzip 呢？

* 当用户访问 web 站点的时候，会在 request header 中设置 accept-encoding:gzip，表明浏览器是否支持 Gzip。
* 服务器在收到请求后，判断如果需要返回 Gzip 压缩后的文件那么服务器就会先将我们的 JS\CSS 等其他资源文件进行 Gzip 压缩后再传输到客户端，同时将 response headers 设置 content-encoding:gzip。反之，则返回源文件。
* 浏览器在接收到服务器返回的文件后，判断服务端返回的内容是否为压缩过的内容，是的话则进行解压操作。

一般情况下我们并不会让服务器实时 Gzip 压缩，而是利用webpack提前将静态资源进行Gzip 压缩，然后将Gzip 资源放到服务器，当请求需要的时候直接将Gzip 资源发送给客户端。

我们只需要安装 compression-webpack-plugin 并在plugins配置就可以了

const CompressionWebpackPlugin = require("compression-webpack-plugin"); // 需要安装

module.exports = {

plugins: [

new CompressionWebpackPlugin()

]

}

#### 作用域提升

Scope Hoisting 可以让 webpack 打包出来的代码文件体积更小，运行更快。

**在开启 Scope Hoisting后，构建后的代码会按照引入顺序放到一个函数作用域里，通过适当重命名某些变量以防止变量名冲突，从而减少函数声明和内存花销。**

需要注意：Scope Hoisting 需要分析模块之间的依赖关系，所以源码必须采用 ES6 模块化语法

Scope Hoisting 是 webpack 内置功能，只需要在plugins里面使用即可，或者直接开启生产环境也可以让作用域提升生效。

module.exports = {

//方式1

mode: 'production',

//方式2

plugins: [

// 开启 Scope Hoisting 功能

new webpack.optimize.ModuleConcatenationPlugin()

]

}

**[Webpack]全面了解tree shaking【热度: 790】**

**概念**：Tree-shaking又叫摇树优化，是通过静态分析消除JS模块中未使用的代码，减小项目体积。

**原理**：Tree-shaking依赖于ES6的模块机制，因为ES6模块是静态的，编译时就能确定模块的依赖关系。对于非ES6模块的代码或者动态引入的代码，无法被消除掉。

**配置**：Tree-Shaking需要配置optimization选项中的usedExports为true，同时在babel配置中使用babel-preset-env，开启modules选项为false，这样可以保证ES6模块在编译时不会被转换为CommonJS模块。

### tree-shaking 原理

Webpack 的 Tree Shaking 主要是用来消除未被使用的代码，以减小最终打包文件的体积。其原理如下：

#### 一、静态分析

##### 模块依赖分析：

Webpack 在构建过程中，会对项目中的模块进行依赖分析。它会解析每个模块的内容，确定模块之间的导入和导出关系。

通过分析，可以构建出一个模块依赖图，展示了各个模块之间的引用关系。

##### 识别未使用的代码：

基于模块依赖图，Webpack 可以确定哪些模块被实际使用，哪些模块未被使用。

对于 JavaScript 模块，它可以识别出未被调用的函数、未被访问的变量等。对于其他资源文件，如 CSS 和图片，也可以根据引用情况判断是否被使用。

#### 二、代码优化

##### 消除未使用的代码：

一旦识别出未使用的代码，Webpack 会在打包过程中将这些代码从最终的输出文件中移除。

这可以显著减小打包文件的大小，提高应用的加载速度和性能。

##### 作用域分析：

在消除未使用的代码时，Webpack 还会进行作用域分析。它会确保在移除代码的过程中，不会影响到实际使用的代码的正确性。

例如，如果一个函数在一个模块中未被使用，但在另一个模块中被间接引用，Webpack 会谨慎处理，避免错误地移除该函数。

#### 三、实现条件

##### ES2015 模块语法：

Tree Shaking 主要依赖于 ES2015 模块语法（import和export）。这种模块语法是静态的，使得 Webpack 能够在编译时确定模块的导入和导出关系。

相比之下，CommonJS 模块语法（require和module.exports）是动态的，难以在编译时进行准确的分析。

##### 支持的模块类型：

Webpack 不仅可以对 JavaScript 模块进行 Tree Shaking，还可以对一些其他类型的模块进行处理，如 CSS 模块（通过特定的加载器和插件）。

对于不同类型的模块，Webpack 可能会使用不同的技术和策略来实现 Tree Shaking。

总之，Webpack 的 Tree Shaking 通过静态分析模块依赖关系，识别并消除未使用的代码，从而优化打包文件的大小和性能。它依赖于 ES2015 模块语法和准确的模块依赖分析，同时需要注意一些实现条件和潜在的问题。

### webpack如何实现treeShaking

webpack实现tree shaking的原理是**基于ES6模块化语法的静态特性**。

在编译阶段，Webpack会根据模块的依赖关系，通过AST（抽象语法树）进行静态分析，识别出那些代码块（函数、变量、对象等）被引用并且使用了。然后将这些代码块打包输出到最终的打包文件中。在这个过程中，Webpack会自动将未被引用的代码块进行剔除，这个过程就是tree shaking。

具体来说，当Webpack在打包时遇到一个ES6模块导入语句（import），它会自动去加载这个模块并分析其导出对象。然后它会分析项目中哪些导出对象被引用了。如果一个导出对象没有被引用，那么Webpack会直接把它从最终的代码中剔除掉。

需要注意的是，tree shaking只对ES6模块生效，对于CommonJS等其他模块化规范，由于其动态加载特性，无法在静态分析阶段确定哪些代码块被引用，因此无法进行tree shaking。

另外，为了使Webpack能够正确识别和剔除未引用的代码块，开发者也需要做出一定的努力，例如将代码编写为纯函数的形式，避免使用全局变量等副作用等。

### webpack 如何做 tree shaking

Webpack通过tree shaking技术实现了JavaScript代码的优化和精简。Tree shaking是指通过静态代码分析，识别和移除未被使用的代码（被称为"dead code"），从而减小最终打包后的文件大小。

下面是Webpack如何进行tree shaking的步骤：

* 代码标记：在代码中使用ES6模块化语法（如import和export）来明确指定模块的依赖关系。
* 代码解析：Webpack会解析整个代码，并构建一个依赖图谱，记录模块之间的依赖关系。
* 标记未使用代码：在构建依赖图谱的过程中，Webpack会标记那些未被使用的模块，以及这些模块中的未被使用的函数、类、变量等。
* 无副作用标记：Webpack还会检查模块的副作用（例如对全局变量的修改、网络请求等），并将没有副作用的代码视为可安全移除的。
* 未使用代码移除：在代码打包阶段，Webpack会根据标记的未使用代码信息，从最终的打包结果中移除这些未被使用的代码。

通过tree shaking，Webpack可以减小打包后的文件大小，提高应用的加载速度和性能。但要注意的是，tree shaking只对ES6模块化的代码有效，对于CommonJS模块化的代码则无法进行优化。另外，有些代码可能由于复杂的依赖关系无法被正确地标记为未使用，这就需要开发者自己进行配置或使用其他工具进行优化。

### webpack 处理 tree shaking 配置

要在Webpack中配置tree shaking，需要进行以下步骤：

1.在webpack.config.js文件中，将mode设置为production，这会启用Webpack的优化功能，包括tree shaking。

module.exports = {

mode: 'production',

// 其他配置...

};

2.确保你的代码使用了ES6模块化语法（使用import和export），以便Webpack能够正确地进行静态代码分析。

3.确保你的代码库中没有副作用。Webpack会假设没有副作用的代码可以安全地移除。如果你的代码确实有副作用，可以在webpack配置文件中的optimization选项中设置sideEffects为false来告诉Webpack可以安全地进行tree shaking。

module.exports = {

mode: 'production',

optimization: {

sideEffects: false,

},

// 其他配置...

};

### 了解一下副作用

在计算机科学中，副作用是指函数或代码的执行对除了返回一个值之外的程序状态产生了可观察的变化。换句话说，副作用是指对外部环境产生了影响或产生了可观察的行为。

以下是一些常见的副作用示例：

* 修改全局变量或外部状态：函数修改了全局变量或外部状态，例如修改了一个共享的数组、对象或文件等。
* 发送网络请求：函数通过网络发送了一个HTTP请求，这会触发网络交互并产生副作用。
* 修改函数参数：函数修改了传入的参数值，这会影响函数外部的变量。
* 控制台打印：函数在执行过程中使用了console.log()或其他打印语句，这会在控制台中产生可观察到的输出。
* 异步操作：函数中包含了异步操作，例如定时器、Promise或通过回调函数实现的异步操作。

### 如何申明代码是有副作用

某一些代码是是需要禁止被清理掉， 这个时候该如何处理呢？

有几个办法：

#### 方法一：在配置文件中指定副作用

在Webpack配置文件中，可以使用sideEffects选项来指定哪些文件或模块具有副作用，不允许清理。sideEffects接受一个正则表达式、一个文件名或一个数组。例如：

module.exports = {

//...

optimization: {

usedExports: true

},

mode: 'production',

sideEffects: ["./src/some-module.js"]

};

在上面的例子中，sideEffects数组中的./src/some-module.js文件将会被标记为具有副作用，不会被清理。

请注意，为了使sideEffects选项生效，你需要在配置文件中启用optimization.usedExports选项，并将mode设置为production。

#### 方法二：package.json 中配置 sideEffects 属性

可以在package.json文件中使用sideEffects字段来申明哪些文件或模块具有副作用，不允许被清理。

1.如果将sideEffects设置为布尔值false，表示所有导入的文件都被认为没有副作用，可以被tree shaking清理。这在大多数情况下是默认的行为。

{

"name": "my-package",

"version": "1.0.0",

"sideEffects": false

}

2.如果设置为布尔值true，表示所有导入的文件都被认为有副作用，不会被tree shaking清理。

{

"name": "my-package",

"version": "1.0.0",

"sideEffects": true

}

3.如果将sideEffects设置为一个数组，数组中的每个元素可以是一个字符串或一个正则表达式，表示具有副作用的文件或模块。

{

"name": "my-package",

"version": "1.0.0",

"sideEffects": [

"./src/some-module.js",

"/\.css$/"

]

}

在上述示例中，./src/some-module.js文件和所有以.css结尾的文件都被认为有副作用，不会被tree shaking清理。

### **如果我某一个文件配置了 sideEffects 申明该文件有副作用， 但是我又想清理其中的某个函**数

魔法中的魔法注释: **/\*#\_\_PURE\_\_\*/**

通过上面的知识， 我们知道了， 如果是有如果被 sideEffects 申明了副作用的文件， 是不会被 tree shaking 清理掉的，但是也有例外。

/\*#\_\_PURE\_\_\*/这个注释的作用是告诉Webpack或Babel等构建工具，这一行代码是纯粹的，没有副作用，并且可以安全地进行tree shaking（摇树优化）。

对于一些库或框架，可能会有一些函数或类被导出，但实际上很少被使用，为了让构建工具知道这些代码可以被删除，可以在导出语句上添加/\*#\_\_PURE\_\_\*/注释。

例如，假设 src/myModule.js 文件有下面的代码：

export /\*#\_\_PURE\_\_\*/ function add(a, b) {

return a + b;

}

export function subtract(a, b) {

return a - b;

}

且 webpack 已经将 src/myModule.js 申明为了有副作用文件

module.exports = {

// ...

optimization: {

sideEffects: ["./src/myModule.js"],

},

};

虽然通过 sideEffects 配置申明了 ./src/myModule.js 文件是有副作用的，但是由于 add 方法前面有 /\*#\_\_PURE\_\_\*/ 注释标记，意味着这个方法被标记为纯函数，该方法是没有副作用。

因此最终通过 /\*#\_\_PURE\_\_\*/ 注释标记的 add 方法依然可以被 Webpack 的 Tree Shaking 清理。

### commonjs 模块就真的不能被 tree shaking 了？

Webpack 曾经不进行对 CommonJs 导出和 require() 调用时的导出使用分析。

Webpack 5 增加了对一些 CommonJs 构造的支持，允许消除未使用的 CommonJs 导出，并从 require() 调用中跟踪引用的导出名称。

支持以下构造：

* exports|this|module.exports.xxx = ...
* exports|this|module.exports = require("...") (reexport)
* exports|this|module.exports.xxx = require("...").xxx (reexport)
* Object.defineProperty(exports|this|module.exports, "xxx", ...)
* require("abc").xxx
* require("abc").xxx()
* 从 ESM 导入
* require() 一个 ESM 模块
* 被标记的导出类型 (对非严格 ESM 导入做特殊处理):
* Object.defineProperty(exports|this|module.exports, "\_\_esModule", { value: true|!0 })
* exports|this|module.exports.\_\_esModule = true|!0

当检测到不可分析的代码时，webpack 会放弃，并且完全不跟踪这些模块的导出信息（出于性能考虑）。

## [webpack] tree-shaking 在什么情况下会失效？【热度: 171】

**关键词：tree shaking 失效**

在以下情况下，webpack 的 tree-shaking 可能会失效：

* 使用了 sideEffects 属性：在 webpack 的配置文件中，如果设置了 sideEffects: true，则 webpack 会假设所有模块都有副作用，因此不会进行 tree-shaking。这通常用于避免某些模块被误标记为无用代码而被删除。
* 动态导入：如果你使用了动态导入（例如使用了 import() 或 require.ensure()），webpack 无法静态分析模块的导入和导出，因此无法进行 tree-shaking。
* 使用了 commonjs 模块语法：如果你的代码中使用了 commonjs 模块语法（例如使用了 require() 或 module.exports），webpack 无法进行静态分析，因此无法进行 tree-shaking。
* 未使用 ES6 模块语法：tree-shaking 只能对 ES6 模块语法进行优化，如果你的代码中没有使用 ES6 模块语法，webpack 将无法进行 tree-shaking。
* 模块被动态引用或条件引用：如果模块的引用方式是动态的（例如在循环或条件语句中引用），或者通过字符串拼接来引用模块，webpack 无法确定哪些模块会被引用，因此无法进行 tree-shaking。
* 使用了副作用的代码：如果你的代码中包含有副作用的代码（例如在模块的顶级作用域中执行了一些操作），webpack 无法确定哪些代码是无用的，因此无法进行 tree-shaking。

需要注意的是，即使 tree-shaking 可能会失效，webpack 仍然会进行其他优化，例如代码压缩和代码分割等。同时，你可以通过设置 mode 为 production，来启用 webpack 的优化功能，包括 tree-shaking。

## 在你的项目中，使用过哪些webpack loader,说一下他们的作用【热度: 361】

babel-loader 将 ES6+ 代码转换为 ES5 代码，以便在旧版浏览器中运行。

css-loader 解析 CSS 文件，处理 CSS 中的依赖关系，并将 CSS 转换为 JS 模块。

style-loader 将 CSS 代码以内联的方式注入到 HTML 页面中。

file-loader 处理文件资源（如图片、字体等），将文件复制到输出目录，并返回文件路径。

url-loader 与 file-loader 类似，但可以根据文件大小将文件转换为 Data URL（base64 格式）或文件路径。

sass-loader 解析 Sass/SCSS 文件，并将其转换为 CSS 代码。

less-loader 解析 Less 文件，并将其转换为 CSS 代码。

postcss-loader 使用 PostCSS 处理 CSS，可以进行自动添加前缀、压缩、CSS Modules 等操作。

ts-loader 将 TypeScript 代码转换为 JavaScript 代码。

eslint-loader 在构建过程中使用 ESLint 进行代码检查。

stylelint-webpack-plugin 在构建过程中使用 Stylelint 进行 CSS/SCSS 代码检查。

vue-loader 解析 Vue 单文件组件（.vue 文件），并将其转换为 JavaScript 代码。

image-webpack-loader 优化图片资源，包括压缩、转换格式等操作。

html-loader 解析 HTML 文件，处理其中的引用资源（如图片、字体等），并返回处理后的 HTML 代码。

markdown-loader 将 Markdown 文件转换为 HTML 代码。

json-loader 解析 JSON 文件，并返回解析后的 JavaScript 对象。

eslint-loader 在构建过程中使用 ESLint 进行代码检查。

tslint-loader 在构建过程中使用 TSLint 进行 TypeScript 代码检查。

prettier-loader 在构建过程中使用 Prettier 进行代码格式化。

stylelint-webpack-plugin 在构建过程中使用 Stylelint 进行 CSS/SCSS 代码检查。

mini-css-extract-plugin 提取 CSS 代码到单独的文件，而不是内联到 JavaScript 代码中。

optimize-css-assets-webpack-plugin 压缩 CSS 代码。

terser-webpack-plugin 压缩 JavaScript 代码。

**在webpack 中，通常用于css提取的工具是什么?【热度: 69】**

在webpack中，通常**使⽤ mini-css-extract-plugin 来提取CSS**。它是⼀个独⽴的插件，可以将CSS从JavaScript⽂件中提取出来，⽣成独⽴的CSS⽂件。

**使⽤ mini-css-extract-plugin 可以优化代码分离和缓存，以及提⾼加载速度。**

通过配置webpack的插件选项，可以将 mini-css-extract-plugin 添加到webpack构建流程中。在配置中，需要将该插件实例化，并指定输出的CSS⽂件名和路径。

⼀旦配置完成并运⾏webpack构建， mini-css-extract-plugin 就会将CSS提取到独⽴的⽂件中，⽽不是将其嵌⼊到JavaScript⽂件中。

## 浏览器本身是不支持模块化的, webpack是如何通过文件打包，让浏览器可以读取到前端各个模块的代码的?【热度:1,153】

**关键词：webpack模块化、浏览器模块化支持**

浏览器本身不支持模块化的特性，无法直接读取和执行模块化的代码。Webpack通过使用一种称为"**模块化打包**"的方式，**将模块化的代码转换为浏览器可以执行的形式**。

Webpack使用了一个称为**"模块系统"的机制**，通过对模块的依赖关系进行分析，将所有依赖的模块打包为一个或多个包含所有依赖关系的文件。这些打包后的文件被称为“打包产物”或“bundle”。

在打包过程中，Webpack会根据配置文件中的入口点（entry point）来确定应用程序的起始模块。然后，它将从该模块开始递归地解析所有的依赖关系，包括其他模块或文件。Webpack通过识别模块之间的依赖关系，将它们合并到一个或多个打包产物中。

Webpack还会对打包产物进行一系列的优化，例如代码压缩、拆分和按需加载等，以提高应用程序的性能和加载速度。

当浏览器加载打包产物时，它会执行其中包含的代码，并通过模块系统来解决模块之间的依赖关系。Webpack为浏览器提供了一个称为"运行时"的文件，它是Webpack打包产物的一部分，负责处理模块之间的依赖关系，并提供加载和执行模块的功能。

通过Webpack的打包过程，浏览器可以读取和解析打包产物中的模块化代码，并顺利执行应用程序的逻辑。这样，开发者可以在编写代码时使用模块化的语法，而无需担心浏览器的兼容性问题。

## webpack5 Module Federation(模块联邦)了解多少

概念

Webpack 5 的 Module Federation 是一项功能强大的功能，它**允许将 JavaScript 应用程序拆分成独立的模块，并在不同的 Webpack 构建中共享这些模块**。它解决了多个独立应用程序之间共享代码的问题，使得实现微前端架构变得更加容易。

Module Federation 可以将一个应用程序拆分成多个独立的子应用，每个子应用都可以被独立开发、部署和运行。每个子应用都可以通过配置指定需要共享的模块，然后将这些共享模块以动态方式加载到其他子应用中使用，而无需将这些模块打包进每个子应用的构建文件中。

Module Federation 的核心概念是 "容器"（Container）和 "远程"（Remote）。容器是一个主应用程序，它可以加载和渲染其他子应用程序，而远程是一个独立的子应用程序，它提供了一些模块供其他子应用程序使用。

Module Federation 提供了一种简单的方式来定义远程模块，并在容器中引用这些远程模块。容器可以从远程加载模块，并通过配置将这些模块暴露给其他子应用程序。这样，子应用程序可以通过远程加载和使用容器中的模块，实现了模块的共享和动态加载。

Module Federation 在实现微前端架构时非常有用，可以将多个独立开发的子应用程序组合成一个整体，并实现共享模块和资源的灵活管理。它提供了一种解决方案，让多个团队可以独立开发和部署自己的子应用程序，同时又能够共享代码和资源，提高开发效率和整体性能。

Webpack 5 的 Module Federation 是一项用于实现微前端架构的功能，它可以将 JavaScript 应用程序拆分成独立的子应用程序，并通过动态加载和共享模块的方式实现子应用程序之间的交互和共享。

## source map了解多少【热度: 396】

在开发过程中，我们经常需要对编译后的代码进行调试，但是编译后的代码往往很难阅读和理解。Source Map（源映射）是一种文件格式，它可以将编译后的代码映射回源代码。通过使用Source Map，我们可以在浏览器中直接调试源代码，而不需要在编译后的代码中进行调试。

比如Vue项目，跑在浏览器里的代码其实并不是你写的.Vue文件，而是经过编译后的。可是平时调试的时候，我们写的代码位置却能和浏览器控制台对应上。

而帮我们做这个事情的，就是Source Map。

### Source Map（源映射）作用

Source Map（源映射）是一种文件，用于将压缩、混淆或编译后的代码映射回原始的源代码，以便在调试过程中能够直接查看和调试源代码。它提供了压缩文件和源文件之间的映射关系，包括每个压缩文件中的代码位置、原始文件的路径和行号等信息。

Source Map的主要作用如下：

调试：在开发过程中，源代码经常会被压缩、合并或转换为其他形式的代码，这使得在调试时直接查看和调试源代码变得困难。Source Map提供了一种方式，通过将压缩代码映射回源代码，开发者可以在调试器中直接查看和调试原始的、易于理解的源代码。

错误追踪：当发生错误或异常时，浏览器或运行环境会提供错误信息，其中包含了压缩后的代码行号和列号。Source Map可以将这些行号和列号映射回源代码的行号和列号，帮助开发者定位和追踪错误。

性能分析：Source Map可以提供压缩文件中每个代码片段对应的原始文件位置信息，这对于性能分析工具来说非常有用。性能分析工具可以使用Source Map来将性能数据映射回源代码，以便更准确地分析和优化代码性能。

Source Map的原理是通过在压缩文件中添加特定的注释或者生成独立的.map文件来存储映射关系。在调试过程中，浏览器或调试器会读取Source Map，并根据其中的映射关系将压缩代码中的行号、列号等信息映射回源代码的对应位置。

### Source Map（源映射）实现原理

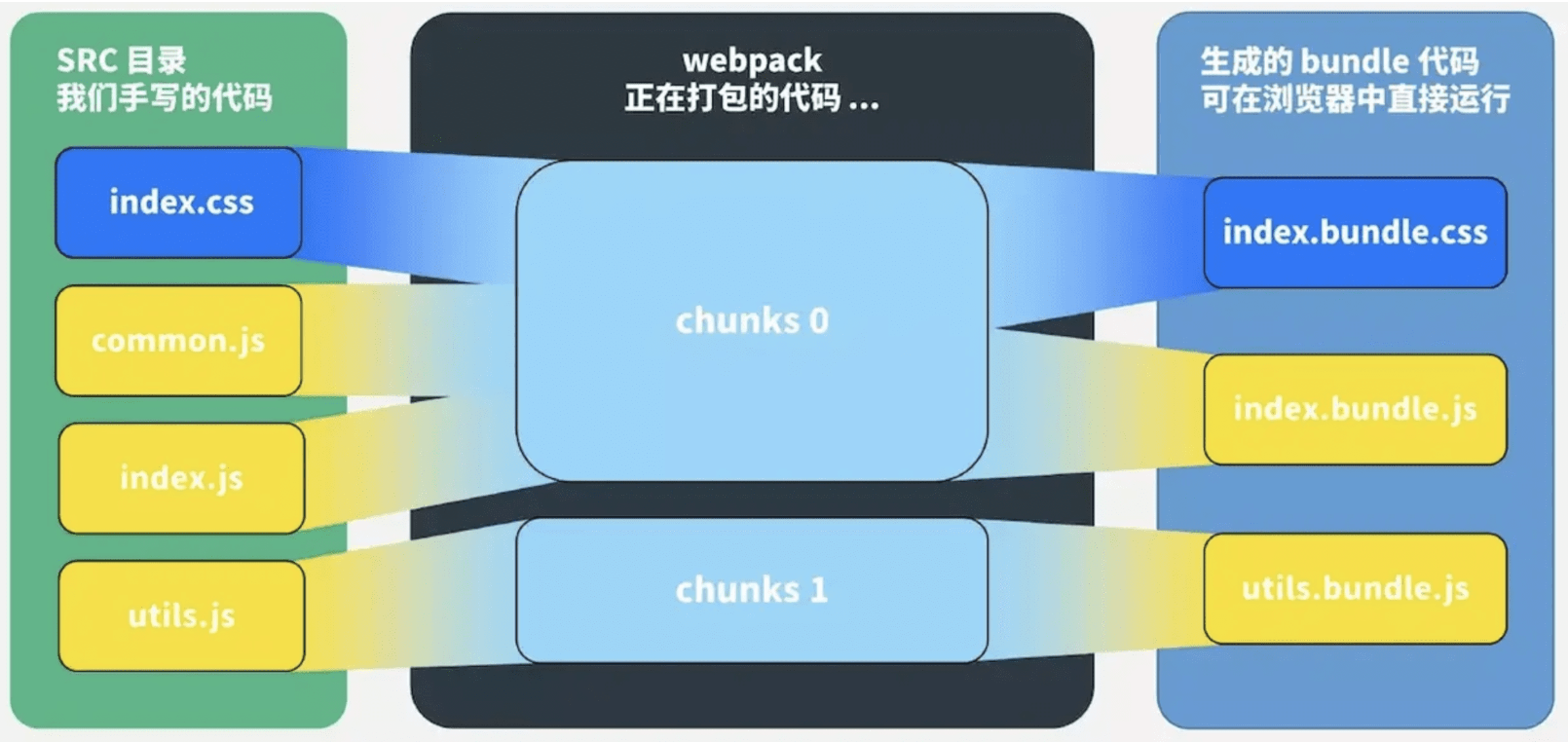
Source Map 的实现原理可以简单描述如下：

* 生成 Source Map：在代码的压缩、混淆或编译过程中，生成器会创建一个 Source Map 对象，并收集相关的映射信息。这些信息包括原始文件路径、行号、列号以及对应的压缩文件中的位置信息。
* 生成编码字符串：将收集到的映射信息使用 VLQ（Variable Length Quantity）编码进行压缩，将数字转换为可变长度的 Base64 编码字符串。VLQ 编码能够通过特定的规则将数字转换为可变长度的字符串，以减小 Source Map 的体积。
* 关联 Source Map：在生成的压缩文件中，通过注释或独立的 .map 文件将 Source Map 关联到压缩文件。注释方式可以通过特定的注释语法将编码字符串直接嵌入到压缩文件中，而独立的 .map 文件则将编码字符串保存在一个独立的文件中。
* 调试时使用 Source Map：在调试过程中，当开发者需要查看或调试源代码时，浏览器或调试工具会加载关联的 Source Map 文件，根据映射关系将压缩文件中的位置信息映射回源代码的对应位置。

通过这种方式，Source Map 实现了将压缩后的代码映射回原始源代码的功能，使得在调试、错误追踪和性能分析时能够更方便地操作和理解源代码。实际上，Source Map 的实现会有更多的细节和规范，但以上是其基本的实现原理概述。

## [webpack] module、chunk 、bundle的区别【热度:136】

首先上图



* 手写下一个一个的文件，它们无论是 ESM 还是 commonJS 或是 AMD，他们都是 module
* 当我们写的 module 源文件传到 webpack 进行打包时，webpack 会根据文件引用关系生成 chunk 文件，webpack 会对这个 chunk 文件进行一些操作
* webpack 处理好 chunk 文件后，最后会输出 bundle 文件，这个 bundle 文件包含了经过加载和编译的最终源文件，可以直接在浏览器中运行。

一般来说一个 chunk 对应一个 bundle，比如上图中的 utils.js -> chunks 1 -> utils.bundle.js

但也有例外，比如说上图中，我就用 MiniCssExtractPlugin 从 chunks 0 中抽离出了 index.bundle.css 文件

### 总结：

module，chunk 和 bundle 其实就是同一份逻辑代码在不同转换场景下的取了三个名字：

我们直接写出来的是 module，webpack 处理时是 chunk，最后生成浏览器可以直接运行的 bundle。

## [webpack]分包的方式有哪些?

在 Webpack 中，可以通过以下方式进行分包

* 通过 entry 属性指定入口文件，在打包时，Webpack 会按照入口文件生成一个 chunk，每个 chunk 包含了一组代码块，最终生成一个或多个 bundle。
* 通过 SplitChunksPlugin 插件对公共依赖进行分割。该插件会把公共依赖提取出来，形成一个或多个独立的 chunk，以便在多个 bundle 中共享。
* 使用动态导入（Dynamic Import）技术进行按需加载。在代码中使用 import() 或 require.ensure()，Webpack 会将这些代码块按照配置的策略进行分割，生成一个或多个独立的 chunk。
* 使用 DllPlugin 插件将一些不经常变化的代码提取出来，形成一个动态链接库（DLL）。在打包时，可以直接引用这个 DLL，而不必重复打包。
* 通过 externals 属性将一些模块声明为外部依赖。在打包时，Webpack 会跳过这些模块的打包过程，而是在运行时从外部环境中加载。

### SplitChunksPlugin 是怎么对公共依赖进行分割的 ?

Webpack的SplitChunksPlugin插件是用来对公共依赖进行分割的，其原理是将公共模块提取出来，形成一个或多个共享块，并在需要时动态加载。这个插件会分析模块之间的依赖关系，将具有相同引用模块的代码块进行提取，以便于实现缓存和更快的加载。

SplitChunksPlugin的默认配置包括以下三个块：

* 通过异步加载（异步块）生成的代码块
* 与初始块（即入口点）共享的代码块
* 在两个或更多块之间共享的代码块

其中，第二个块是最常见的，它包括从多个入口点引用的公共模块。我们可以通过以下示例来说明：

// webpack.config.js

module.exports = {

entry: {

main: './src/main.js',

other: './src/other.js'

},

optimization: {

splitChunks: {

cacheGroups: {

commons: {

test: /[\\/]node\_modules[\\/]/,

name: 'vendors',

chunks: 'all'

}

}

}

}

};

在上面的配置中，我们定义了两个入口点main和other，并将SplitChunksPlugin配置为将从node\_modules目录中引用的代码提取到名为vendors的代码块中。这个代码块将被引用到我们的入口点中，以便于减少加载时间和提高性能。

当我们打包时，Webpack将分析这两个入口点，并发现它们都引用了node\_modules目录下的依赖，于是将这些依赖提取到vendors代码块中。我们可以在main.js和other.js中看到这样的代码：

import 'lodash'; // 公共模块被引用

通过这种方式，Webpack能够将公共代码提取到单独的代码块中，以便于更高效地加载和缓存。

### webpack DllPlugin 是如何提取依赖的？

Webpack DllPlugin 是用于将某些库（例如第三方库）提前编译和打包，以便在开发和生产构建过程中减少依赖项的分析时间，提高打包速度的插件。

具体来说，DllPlugin 分为两个步骤：第一步是生成一个包含库的映射文件（manifest.json），第二步是在开发或生产构建中使用该映射文件来引入预编译的库。

DllPlugin 的基本原理是将库单独编译成一个或多个 dll 文件（动态链接库），同时生成一个 manifest.json 文件记录这些库文件的名称、路径等信息。在开发或生产构建过程中，通过引用 manifest.json 文件，将已经预编译好的库文件引入到项目中。这样可以避免在每次打包时都去分析和打包第三方库，提高构建速度。

下面是一个简单的例子：

在 webpack.config.dll.js 中定义打包的库：

const path = require('path');

const webpack = require('webpack');

module.exports = {

mode: 'production',

entry: {

vendor: ['react', 'react-dom', 'lodash']

},

output: {

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

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

library: '[name]'

},

plugins: [

new webpack.DllPlugin({

name: '[name]',

path: path.join(\_\_dirname, 'dist', '[name].manifest.json')

})

]

};

然后运行 webpack --config webpack.config.dll.js，会生成 vendor.dll.js 和 vendor.manifest.json 文件。

在开发或生产环境中，通过引入生成的 manifest.json 文件，将预编译好的库文件引入项目中：

const path = require('path');

const webpack = require('webpack');

module.exports = {

mode: 'production',

entry: {

app: './src/index.js'

},

output: {

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

filename: '[name].js'

},

plugins: [

new webpack.DllReferencePlugin({

manifest: require('./dist/vendor.manifest.json')

})

]

};

这样，在打包过程中，webpack 会自动将 vendor.dll.js 中包含的第三方库从项目中分离出来，而不需要重复打包和分析这些库。

## [webpack] externals 作用是啥?

webpack 中的 externals 配置项**用于指定在打包时需要排除掉的模块**，这些模块会被视为外部依赖，即不会被打包进最终的输出文件中，而是通过其他方式引入。

使用 externals 配置项可以使得打包后的代码文件更小，同时也可以在运行时从外部获取依赖，例如通过 CDN、全局变量或者通过 require 的方式等。

举个例子，假设我们需要在项目中引入 jquery 库，但我们并不想在打包的过程中将其打包进最终的输出文件中，而是从外部引入。我们可以通过以下的配置来实现：

module.exports = {

// ...

externals: {

jquery: 'jQuery'

}

};

这里的 externals 配置项告诉 webpack 在打包时忽略 jquery 模块的引用，而在代码运行时，我们需要手动将 jquery 通过 script 标签引入，并将其暴露在全局变量 jQuery 下，例如：

<script src="https://cdn.bootcdn.net/ajax/libs/jquery/3.6.0/jquery.min.js"></script>

<script>

window.jQuery = jQuery;

</script>

这样在代码中引入 jquery 模块时，webpack 就会将其作为外部依赖进行处理，而不是将其打包进输出文件中。

需要注意的是，使用 externals 配置项需要谨慎，因为如果在运行时无法正确获取到指定的外部依赖，就会导致代码运行出错。

## [Webpack] webpack热更新原理是什么?

Hot Module Replacement，简称HMR，无需完全刷新整个页面的同时，更新模块。HMR的好处，在日常开发工作中体会颇深：节省宝贵的开发时间、提升开发体验。

刷新我们一般分为两种：

* 一种是页面刷新，不保留页面状态，就是简单粗暴，直接window.location.reload()。
* 另一种是基于WDS (Webpack-dev-server)的模块热替换，只需要局部刷新页面上发生变化的模块，同时可以保留当前的页面状态，比如复选框的选中状态、输入框的输入等。

HMR作为一个Webpack内置的功能，可以通过HotModuleReplacementPlugin或--hot开启。

### 那么，HMR到底是怎么实现热更新的呢？

1. webpack-dev-server启动本地服务

2. 修改webpack.config.js的entry配置

3. 监听webpack编译结束

4. webpack监听文件变化

5. 浏览器接收到热更新的通知

6. HotModuleReplacementPlugin

7. moudle.hot.check 开始热更新

8. hotApply 热更新模块替换

①删除过期的模块，就是需要替换的模块

②将新的模块添加到 modules 中

③通过\_\_webpack\_require\_\_执行相关模块的代码

详细见 [[Webpack] webpack热更新原理是什么？](https://github.com/pro-collection/interview-question/issues/267)

## [webpack]异步加载原理是啥

* 使用动态 import: 使用 ES6 的 import() 语法，动态加载模块。

import('./path/to/module')

.then(module => {

// do something with module

})

.catch(error => {

// handle error

});

* 使用 require.ensure: 异步加载模块并将其放置到指定的 chunk 中。

require.ensure(['./path/to/module'], function(require) {

const module = require('./path/to/module');

// do something with module

});

* 使用 bundle-loader: 将模块放置到一个单独的文件中，按需加载。

const load = require('bundle-loader!./path/to/module');

load(function(module) {

// do something with module

});

* 使用webpack的require.ensure API

require.ensure([], function(require) {

// require dependencies

var foo = require("./foo");

// ...

});

* 使用webpack的import动态导入

import("./dynamic-module.js").then(module => {

// do something with module

});

### 动态加载的原理

在 Webpack 中，异步加载组件的原理是利用动态导入（Dynamic import）特性。使用动态导入可以将模块的加载从编译时刻延迟到运行时刻。

具体来说，当 Webpack 打包代码时，遇到动态导入语句时不会将其打包进入主文件，而是将其单独打包为一个新的文件。在运行时，当代码需要加载该组件时，会通过网络请求动态加载该文件。

这样做的好处是可以减小主文件的体积，从而加快页面的加载速度，并且也可以提高代码的灵活性和可维护性。同时，Webpack 还可以对动态加载的文件进行代码分割和按需加载，进一步优化页面的性能。

在使用动态导入时，需要注意一些细节。例如，在支持动态导入的浏览器中，需要使用 import() 函数进行动态导入；而在不支持动态导入的浏览器中，需要使用 Webpack 提供的 require.ensure 或 require.include 等方法进行模块的异步加载。同时，还需要注意动态导入的兼容性和性能问题。

## [webpack]核心库- tapable的设计思路与实现原理是什么?

### Webpack Tapable 的设计思路

Webpack Tapable 的设计思路主要基于观察者模式（Observer Pattern）和发布-订阅模式（Publish-Subscribe Pattern），用于解耦各个插件之间的依赖关系，让插件能够独立作用于特定的钩子（Hook），从而实现可扩展性和灵活性。

具体来说，Tapable 采用了钩子（Hook）的概念，每个钩子对应一组事件，Webpack 在不同的时刻触发这些钩子，插件可以注册自己的事件处理函数到对应的钩子上，以实现各种功能。

为了避免插件之间的耦合，Tapable 将事件处理函数按照钩子类型分为同步钩子（Sync Hook）、异步钩子（Async Hook）、单向异步钩子（Async Parallel Hook）和多向异步钩子（Async Series Hook）四种类型。这样，不同类型的钩子对应着不同的事件处理顺序和调用方式，插件在注册自己的事件处理函数时，可以选择不同的钩子类型来适应不同的应用场景。

除此之外，Tapable 还提供了一些辅助方法和工具函数，用于方便地创建和管理钩子、向钩子注册事件处理函数、调用钩子的事件处理函数等。这些工具函数的设计思路也遵循了解耦、简单易用的原则，为插件开发提供了很大的便利性。

Webpack Tapable 的设计思路

Webpack Tapable 的设计思路主要基于观察者模式（Observer Pattern）和发布-订阅模式（Publish-Subscribe Pattern），用于解耦各个插件之间的依赖关系，让插件能够独立作用于特定的钩子（Hook），从而实现可扩展性和灵活性。

具体来说，Tapable 采用了钩子（Hook）的概念，每个钩子对应一组事件，Webpack 在不同的时刻触发这些钩子，插件可以注册自己的事件处理函数到对应的钩子上，以实现各种功能。

为了避免插件之间的耦合，Tapable 将事件处理函数按照钩子类型分为同步钩子（Sync Hook）、异步钩子（Async Hook）、单向异步钩子（Async Parallel Hook）和多向异步钩子（Async Series Hook）四种类型。这样，不同类型的钩子对应着不同的事件处理顺序和调用方式，插件在注册自己的事件处理函数时，可以选择不同的钩子类型来适应不同的应用场景。

除此之外，Tapable 还提供了一些辅助方法和工具函数，用于方便地创建和管理钩子、向钩子注册事件处理函数、调用钩子的事件处理函数等。这些工具函数的设计思路也遵循了解耦、简单易用的原则，为插件开发提供了很大的便利性。

### Tapable 的使用

Webpack Tapable 的使用分为三个步骤：

1.定义一个新的 Tapable 实例：在 Webpack 插件中定义一个新的 Tapable 实例，并定义需要监听的事件。

const { SyncHook } = require('tapable');

class MyPlugin {

constructor() {

this.hooks = {

beforeRun: new SyncHook(['compiler']),

done: new SyncHook(['stats'])

};

}

apply(compiler) {

this.hooks.beforeRun.tap('MyPlugin', compiler => {

console.log('Webpack is starting to run...');

});

this.hooks.done.tap('MyPlugin', stats => {

console.log('Webpack has finished running.');

});

}

}

2.触发事件：在 Webpack 的编译过程中，调用 Tapable 实例的触发方法，触发事件。

compiler.hooks.beforeRun.call(compiler);

// Webpack is starting to run...

compiler.run((err, stats) => {

if (err) {

console.error(err);

return;

}

console.log(stats);

compiler.hooks.done.call(stats);

// Webpack has finished running.

});

3.注册插件：在 Webpack 的配置文件中，将插件实例注册到 Webpack 中。

const MyPlugin = require('./my-plugin');

module.exports = {

plugins: [new MyPlugin()]

};

以上是使用 Tapable 的基本流程，通过 Tapable 可以监听到编译过程中的各个事件，并对编译过程进行修改，从而实现各种插件。以下是一些常见的 Tapable 类型和用法：

* SyncHook：同步 Hook，按照注册的顺序同步执行所有回调函数。

const { SyncHook } = require('tapable');

const hook = new SyncHook(['arg1', 'arg2']);

hook.tap('MyPlugin', (arg1, arg2) => {

console.log(`Hook is triggered with arguments: ${arg1}, ${arg2}`);

});

hook.tap('MyPlugin', (arg1, arg2) => {

console.log('Second callback is called');

});

hook.call('Hello', 'world');

// Hook is triggered with arguments: Hello, world

// Second callback is called

* AsyncParallelHook：异步 Hook，按照注册的顺序异步执行所有回调函数，不关心回调函数的返回值。

const { AsyncParallelHook } = require('tapable');

const hook = new AsyncParallelHook(['arg1', 'arg2']);

hook.tap('MyPlugin', (arg1, arg2, callback) => {

setTimeout(() => {

console.log(`Hook is triggered with arguments: ${arg1}, ${arg2}`);

callback();

}, 1000);

});

hook.tap('MyPlugin', (arg1, arg2, callback) => {

setTimeout(() => {

console.log('Second callback is called');

callback();

}, 500)

})

### Tapable 是如何实现的？代码简单实现一下？

Webpack Tapable 是基于发布-订阅模式的一个插件系统，它提供了一组钩子函数，让插件可以在相应的时机执行自己的逻辑。

下面是一个简单的自定义 Tapable 的实现：

class Tapable {

constructor() {

this.hooks = {};

}

// 注册事件监听函数

tap(name, callback) {

if (!this.hooks[name]) {

this.hooks[name] = [];

}

this.hooks[name].push(callback);

}

// 触发事件

call(name, ...args) {

const callbacks = this.hooks[name];

if (callbacks && callbacks.length) {

callbacks.forEach((callback) => callback(...args));

}

}

}

在这个例子中，我们定义了一个 Tapable 类，它有一个 hooks 对象属性，用于存储各个事件对应的监听函数。然后我们定义了 tap 方法，用于注册事件监听函数，以及 call 方法，用于触发事件。

下面是一个使用自定义 Tapable 的例子：

const tapable = new Tapable();

tapable.tap('event1', (arg1, arg2) => {

console.log('event1 is triggered with arguments:', arg1, arg2);

});

tapable.tap('event2', (arg1, arg2) => {

console.log('event2 is triggered with arguments:', arg1, arg2);

});

tapable.call('event1', 'hello', 'world');

tapable.call('event2', 'foo', 'bar');

在这个例子中，我们定义了两个事件 event1 和 event2，并为它们注册了监听函数。当我们调用 call 方法触发事件时，注册的监听函数就会依次执行。

这个自定义 Tapable 的实现虽然简单，但它体现了 Tapable 的设计思路和核心功能。在实际使用中，Webpack 的 Tapable 提供了更多的功能和钩子，可以满足不同场景的需求。

## [webpack]构建流程是怎么样的?

webpack 的运行流程是一个串行的过程，它的工作流程就是将各个插件串联起来

在运行过程中会广播事件，插件只需要监听它所关心的事件，就能加入到这条webpack机制中，去改变webpack的运作，使得整个系统扩展性良好

**从启动到结束会依次执行以下三大步骤：**

* 初始化流程：从配置文件和 Shell 语句中读取与合并参数，并初始化需要使用的插件和配置插件等执行环境所需要的参数
* 编译构建流程：从 Entry 发出，针对每个 Module 串行调用对应的 Loader 去翻译文件内容，再找到该 Module 依赖的 Module，递归地进行编译处理
* 输出流程：对编译后的 Module 组合成 Chunk，把 Chunk 转换成文件，输出到文件系统

### 初始化流程

从配置文件和 Shell 语句中读取与合并参数，得出最终的参数

配置文件默认下为webpack.config.js，也或者通过命令的形式指定配置文件，主要作用是用于激活webpack的加载项和插件

webpack 将 webpack.config.js 中的各个配置项拷贝到 options 对象中，并加载用户配置的 plugins

完成上述步骤之后，则开始初始化Compiler编译对象，该对象掌控者webpack声明周期，不执行具体的任务，只是进行一些调度工作

### 编译构建流程

根据配置中的 entry 找出所有的入口文件

module.exports = {

entry: './src/file.js'

}

初始化完成后会调用Compiler的run来真正启动webpack编译构建流程，主要流程如下：

* compile 开始编译
* make 从入口点分析模块及其依赖的模块，创建这些模块对象
* build-module 构建模块
* seal 封装构建结果
* emit 把各个chunk输出到结果文件

### 输出流程

#### seal 输出资源

seal方法主要是要生成chunks，对chunks进行一系列的优化操作，并生成要输出的代码.

webpack 中的 chunk ，可以理解为配置在 entry 中的模块，或者是动态引入的模块.

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，再把每个 Chunk 转换成一个单独的文件加入到输出列表.

#### emit 输出完成

在确定好输出内容后，根据配置确定输出的路径和文件名

output: {

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

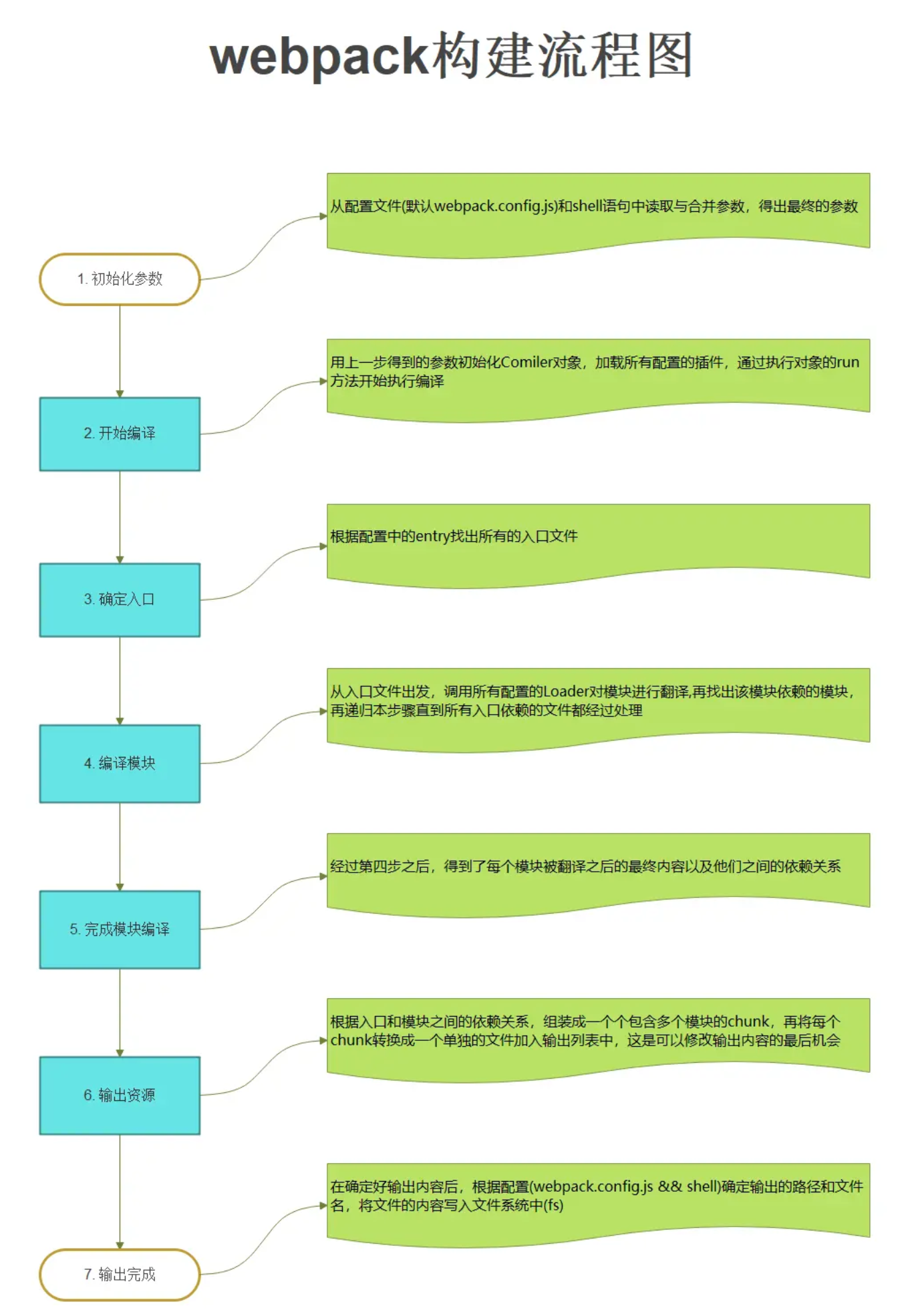
filename: '[name].js'

}

在 Compiler 开始生成文件前，钩子 emit 会被执行，这是我们修改最终文件的最后一个机会

从而webpack整个打包过程则结束了

### 总结



## [webpack]如果解决重复引用node\_modules 里面的不同版本的包

**1.使用 npm 或者 yarn 的工具进行依赖的版本控制，尽量避免引用不同版本的同一个依赖库。**在 package.json 文件中使用 "^"、"~"、">=" 等方式指定依赖版本，可以有效减少不同版本的包冲突问题。

**2.使用 webpack 的 resolve.alias 配置选项**，将需要共享的模块指定到一个目录下，然后在其它模块中使用别名引用该模块。例如，将需要共享的模块指定到 src/shared 目录下，然后在其它模块中使用别名 @shared 引用该模块，这样就可以保证在不同模块中引用相同的依赖库。

**3.使用 webpack 的 ProvidePlugin 插件**，将需要共享的模块注入到全局作用域中，这样就可以在不同模块中共享相同的依赖库。

**4.使用 webpack 的 resolve.modules 配置选项**，将 node\_modules 目录移动到项目根目录之外，然后在 resolve.modules 中添加该目录的绝对路径，这样就可以解决不同模块中引用相同依赖库不同版本的问题。

## [webpack] loader和plugin有啥区别

在Webpack中，Loader和Plugin是两个不同的概念，它们的作用和使用方式也有所不同。

**Loader用于对源代码文件进行转换和处理，而Plugin用于对Webpack的编译过程进行扩展和增强。**

### Loader

Loader是Webpack中的一个核心概念，它用于处理源代码文件，将它们转换成Webpack可处理的模块。Webpack在处理代码模块的过程中，会根据模块的类型来选择相应的Loader进行处理，例如，处理CSS文件需要使用css-loader，处理图片需要使用file-loader等。使用Loader可以实现代码转换、文件处理、代码压缩等功能。

Loader的使用方式是在Webpack的配置文件中定义module.rules属性，它是一个数组，每个元素是一个对象，用于描述如何处理特定类型的文件。一个Loader对象通常包括以下几个属性：

* test：用于匹配需要处理的文件类型，通常是一个正则表达式。
* use：指定需要使用的Loader，可以是一个字符串或一个数组，数组中的每个元素都是一个Loader。
* exclude/include：指定需要排除/包含的文件夹。

### Plugin

Plugin是Webpack中的另一个核心概念，它用于扩展Webpack的功能。Plugin可以用于执行任意类型的任务，例如，生成HTML文件、压缩代码、提取公共代码等。使用Plugin可以实现Webpack无法处理的复杂任务。

Plugin的使用方式是在Webpack的配置文件中定义plugins属性，它是一个数组，每个元素是一个Plugin实例。Plugin通常包括以下几个方法：

* apply：用于安装插件，接收一个compiler对象作为参数。
* 一些Webpack钩子函数的实现。

### 表格对比他们之间的差异和适用范围

| 区别 | Loader | Plugin |
| --- | --- | --- |
| 输入/输出 | 输入文件，输出处理后的文件 | 可以在Webpack构建过程中处理输出结果或做额外处理 |
| 使用方式 | 在模块加载时直接调用 | 在Webpack配置中进行配置 |
| 功能 | 用于处理某些类型的文件 | 可以处理打包过程的各个环节 |
| 实现方式 | 导出一个函数 | 导出一个类 |
| 配置方式 | 在Webpack配置中使用 | 在Webpack配置中使用 |
| 作用 | 转换文件或模块 | 对整个构建过程进行自定义操作 |
| 适用场景 | 处理各种类型的文件，如css、图片等 | 执行比较复杂的操作，如代码压缩、代码分割等 |
| 使用方式 | 需要在Webpack中明确的指定 | 无法单独使用，必须在Webpack中明确的指定使用 |
| 作用对象 | 针对每一个文件进行处理 | 针对整个构建过程进行处理 |

### 总结

**Loader**主要用于**针对单个文件进行处理**，可以根据不同文件类型来选择对应的Loader。

**Plugin**则是**针对整个构建过程进行自定义操作**，比如代码压缩、分离CSS文件、创建HTML文件等。

## [webpack] webpack是如何给web应用注入环境变量的,原理是啥

Webpack 可以通过 DefinePlugin 插件给 web 应用注入环境变量。该插件会在编译过程中替换掉代码中指定的变量，以实现在运行时替换成环境变量的值。

在 webpack 的配置文件中，需要先引入该插件，然后将需要注入的环境变量通过该插件进行配置。例如：

const webpack = require('webpack');

module.exports = {

// 其他配置

plugins: [

new webpack.DefinePlugin({

'process.env': {

NODE\_ENV: JSON.stringify('production')

},

API\_URL: JSON.stringify('http://api.example.com')

})

]

};

上述配置中，定义了两个需要注入的变量，分别是 process.env.NODE\_ENV 和 API\_URL。其中，process.env.NODE\_ENV 是一个常用的环境变量，用来标识当前是开发环境还是生产环境；API\_URL 是一个自定义的环境变量，用来存储 API 的地址。

在代码中使用这些环境变量时，只需要直接引用即可：

if (process.env.NODE\_ENV === 'production') {

console.log('当前为生产环境');

}

fetch(API\_URL + '/users')

.then(response => response.json())

.then(data => console.log(data));

Webpack 在编译时会将这些变量替换成对应的值，例如：

if ('production' === 'production') {

console.log('当前为生产环境');

}

fetch('http://api.example.com' + '/users')

.then(response => response.json())

.then(data => console.log(data));

通过这种方式，我们就可以在代码中方便地使用环境变量，同时保证了在不同环境下都能正确地使用相应的变量值。

## [Webpack] chunk 是什么概念，介绍一下？【热度: 1,100】

在Webpack中，Chunk（代码块）是指Webpack在构建过程中生成的一个或多个独立的文件，它包含了一组相关的模块。每个Chunk都有一个唯一的标识符，可以通过该标识符来访问和加载对应的Chunk。

Webpack根据指定的入口文件和依赖关系图来确定需要生成哪些Chunk。入口文件是Webpack构建的起点，而依赖关系图描述了每个模块之间的依赖关系。Webpack根据这些信息将模块分割为不同的代码块，并生成相应的Chunk。

**Chunk的主要作用是实现代码的分割和按需加载**。通过将代码拆分为多个Chunk，Webpack可以进行按需加载，只有在需要时才加载对应的Chunk，从而减少了初始加载的大小和时间。这样可以提高应用程序的性能和加载速度。

Webpack提供了多种代码分割的方式，包括使用入口配置、使用动态导入语法（如import()）和使用Webpack插件（如SplitChunksPlugin）。这些方式可以帮助开发者将代码分割为不同的Chunk，并根据实际需求进行配置和优化。

具体而言，Webpack的代码分割机制通过两种方式来创建chunk：

静态代码分割（Static Code Splitting）：在Webpack配置中使用splitChunks或optimization.splitChunks选项，可以将第三方库、公共模块或重复的模块分割成独立的chunk。这些chunk可以在多个入口文件之间共享，从而减少重复加载的代码。

动态代码分割（Dynamic Code Splitting）：通过使用动态导入（Dynamic Import）语法，可以将应用程序的不同部分分割成独立的chunk。例如，在React中可以使用React.lazy()函数和Suspense组件来实现动态导入和渲染。

分割成的chunk可以通过Webpack的内置功能（如代码分割插件、懒加载等）实现按需加载，即当需要时才加载对应的chunk，从而减少初始加载时间并提高网页性能。

通过使用chunk，Webpack可以自动将代码分割成更小的部分，实现按需加载和并行加载，从而提高应用程序的性能和用户体验。

## [Webpack] 如何配置多入口应用， 且区分公共依赖的？【热度: 124】

1.在Webpack配置文件中，使用entry属性指定多个入口文件，并为每个入口文件命名一个唯一的键名。例如：

module.exports = {

entry: {

app1: './src/app1.js',

app2: './src/app2.js'

},

// 其他配置项...

};

上面的配置指定了两个入口文件app1.js和app2.js，并为它们分别指定了键名app1和app2。

2.使用SplitChunks插件进行公共依赖的提取。在Webpack配置文件中添加以下配置：

module.exports = {

// 其他配置项...

optimization: {

splitChunks: {

cacheGroups: {

commons: {

name: 'commons',

chunks: 'all',

minChunks: 2

}

}

}

}

};

上面的配置中，我们使用optimization.splitChunks.cacheGroups选项配置了一个名为commons的缓存组。该缓存组将对公共依赖进行提取，name属性指定了提取后文件的名称，chunks属性指定了提取的范围为所有类型的块（入口文件和异步加载的块），minChunks属性指定了至少被引用两次的模块才会被提取为公共依赖。

3.添加output配置，指定打包后文件的输出路径和文件名。例如：

module.exports = {

// 其他配置项...

output: {

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

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

}

};

上面的配置中，使用[name]占位符来动态生成根据入口文件的键名生成对应的文件名。

通过以上配置，Webpack将会根据指定的多个入口文件进行打包，并在打包过程中自动提取公共依赖为一个独立的文件。例如，假设app1.js和app2.js都引用了lodash库，那么在打包后的结果中，lodash库将会被提取为commons.bundle.js文件，而app1.js和app2.js则分别生成对应的app1.bundle.js和app2.bundle.js。

### 追问 上面的配置， 最终会输出几个文件？

根据上述的打包配置，最终将会输出3个文件。假设配置的多入口应用有两个入口文件app1.js和app2.js，并且两个入口文件都引用了lodash库作为公共依赖。

根据上述的配置，Webpack将会进行以下操作：

1. 根据entry配置，将app1.js和app2.js作为入口文件进行打包。
2. 遇到公共依赖lodash库时，使用SplitChunks插件将其提取为独立的文件commons.bundle.js。
3. 根据output配置，将app1.js打包后生成app1.bundle.js，将app2.js打包后生成app2.bundle.js，将commons.bundle.js生成commons.bundle.js。
4. 最终，在输出路径下将会生成3个文件：app1.bundle.js、app2.bundle.js和commons.bundle.js。

## [Webpack] 如何打包运行时 chunk ， 且在项目工程中， 如何去加载这个运行时 chunk ?【热度: 421】

Webpack打包运行时chunk的方式可以通过optimization.runtimeChunk选项来配置。下面是一个示例的配置：

module.exports = {

// ...

optimization: {

runtimeChunk: 'single',

},

};

上述配置中，通过设置optimization.runtimeChunk为'single'，将会把所有的webpack运行时代码打包为一个单独的chunk。

在项目工程中加载运行时chunk有两种方式：

1.通过script标签加载：可以使用HtmlWebpackPlugin插件来自动将运行时chunk添加到 HTML 文件中。在webpack配置文件中添加以下配置：

const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {

// ...

plugins: [

new HtmlWebpackPlugin({

// ...

chunks: ['runtime', 'app'],

}),

],

};

上述配置中，chunks选项指定了要加载的chunk，包括运行时chunk（'runtime'）和其他的业务代码chunk（'app'）。最终生成的HTML文件会自动引入这些chunk。

2.通过import语句动态加载：可以使用动态导入的方式来加载运行时chunk。在需要加载运行时chunk的地方，使用以下代码：

import(/\* webpackChunkName: "runtime" \*/ './path/to/runtime').then((runtime) => {

// 运行时chunk加载完成后的逻辑

});

上述代码中，通过import()函数动态加载运行时chunk，通过webpackChunkName注释指定要加载的chunk名称（这里是'runtime'）。加载完成后，可以进行相关逻辑处理。

总结：Webpack可以通过optimization.runtimeChunk选项配置打包运行时chunk，可以通过script标签加载或者使用动态导入的方式来加载运行时chunk。

### 追问 如果只想把某几个文件打包成运行时加载， 该如何处理呢？

如果你想将某几个文件打包成运行时加载，可以使用Webpack的entry配置和import()语法来实现。

首先，在Webpack的配置文件中，将这几个文件指定为单独的entry点。例如：

module.exports = {

// ...

entry: {

main: './src/main.js',

runtime: './src/runtime.js',

},

};

上述配置中，main.js是业务代码的入口文件，runtime.js是你想要打包成运行时加载的文件。

然后，在你的业务代码中，通过import()动态导入这些文件。例如：

function loadRuntime() {

return import('./runtime.js');

}

// 使用动态导入的方式加载运行时文件

loadRuntime().then(runtime => {

// 运行时文件加载完成后的逻辑

});

使用import()会返回一个Promise，可以通过.then()来处理文件加载完成后的逻辑。

最后，使用Webpack进行打包时，会根据配置的entry点和import()语法自动将这几个文件打包成运行时加载的模块。运行时模块会在需要时动态加载并执行。

注意：在使用import()动态导入文件时，需要确保你的环境支持Promise和动态导入语法。

### 作为上面回复的补充

除了 entry 的方式可以处理自己申明的 runtime 文件以外， 还可以直接在 import('xx') 的时候申明；

例如：

import(/\* webpackChunkName: "runtime" \*/ './path/to/runtime').then((runtime) => {

// 运行时chunk加载完成后的逻辑

});

上面的方式， 可以在也可以达到同样的效果， 只是在 import 的时候申明runtime文件名称而已

## [webpack] 是如何处理 commonjs 模块的文件， 使其编译结果能被浏览器使用。【热度: 210】

以下是详细讲解 Webpack 如何将 CommonJS 模块转换为浏览器可以执行的 JavaScript 文件：

### 一、模块解析阶段

#### 1.识别 CommonJS 模块：

* Webpack 从项目的入口文件（通常在配置中指定，如entry: './src/index.js'）开始扫描代码。当遇到require('some-module')这样的语句时，Webpack 识别出这是一个 CommonJS 模块的引入。
* 它会记录下这个模块的依赖关系，以便后续处理。

#### 2.构建依赖图：

* 对于每个引入的模块，Webpack 会继续深入解析该模块内部的依赖关系，递归地构建出一个完整的模块依赖图。
* 例如，如果模块 A 引入了模块 B 和模块 C，而模块 B 又引入了模块 D，那么 Webpack 会构建出一个反映这些依赖关系的有向无环图。

### 二、模块转换阶段

#### 1.处理require函数：

* 在浏览器环境中，没有原生的require函数。Webpack 会将 CommonJS 中的require函数转换为一种在浏览器中可行的模块加载方式。
* 通常，Webpack 会使用一种称为“模块加载器”的机制。在打包后的文件中，会生成一个模块加载函数，这个函数负责在运行时加载和执行模块。
* 例如，对于const moduleB = require('moduleB')这样的语句，Webpack 可能会将其转换为类似于以下的代码：

// 假设模块加载函数名为 \_\_webpack\_require\_\_

const moduleB = \_\_webpack\_require\_\_("./moduleB.js");

#### 2.处理module.exports和exports：

* CommonJS 使用module.exports或exports来导出模块的内容。Webpack 会将这些导出的内容转换为在浏览器中可访问的形式。
* 如果一个模块使用module.exports = { someFunction: () => {} }这样的方式导出，Webpack 可能会将其转换为：

// 假设模块的 ID 为 1

\_\_webpack\_module.exports\_\_ = { someFunction: () => {} };

* 然后，在加载这个模块时，可以通过模块加载函数获取到这个导出的对象：

const module = \_\_webpack\_require\_\_(1);

console.log(module.someFunction());

#### 3.代码优化和转换：

Webpack 还会进行一系列的代码优化和转换操作。例如：

* Tree Shaking：去除未使用的代码，减小文件大小。如果一个模块中的某个函数从未被其他模块引用，Webpack 会在打包过程中去除这个函数的代码。
* 代码压缩：减小输出文件的大小，提高加载速度。Webpack 可以使用工具如 UglifyJS 或 Terser 对代码进行压缩。
* 模块合并：如果多个模块具有相同的依赖，Webpack 可能会将这些模块合并在一起，减少重复的代码加载。

### 三、输出打包文件阶段

#### 1.生成浏览器可执行的文件：

* 经过模块转换和优化后，Webpack 会将所有的模块及其依赖关系打包成一个或多个文件。这些文件通常包含了模块加载函数和所有模块的代码。
* 打包后的文件可以直接在浏览器中通过<script>标签引入。例如：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

</head>

<body>

<script src="bundle.js"></script>

</body>

</html>

* 当浏览器加载这个文件时，模块加载函数会开始执行，根据需要动态地加载和执行各个模块。

## [webpack] 开发中使用的是 esm 和 webpack, 是否可以直接调用 commonjs 模块化的三方包【热度: 359】

**关键词：模块化混用**

在使用 Webpack 作为构建工具的项目中，ESM (ECMAScript Modules) 和 CommonJS 模块系统**可以混用**。

Webpack 提供了对两种模块化标准的支持，能够理解和处理它们之间的差异，使得这两种不同类型的模块可以在同一个项目中共存。

### 混用时的考虑因素

虽然 ESM 和 CommonJS 可以混用，但是还需要注意一些关键的点：

* 导入方式：当你从一个 ESM 模块中导入 CommonJS 模块时，导入的内容会被当成默认导出处理。这意味着，即使 CommonJS 模块导出了多个成员，你也需要以默认导入的方式来获取整个 exports 对象。

// CommonJS 模块

module.exports = {

foo: "bar",

baz: "qux",

};

// 在 ESM 中导入 CommonJS 模块

import cjsModule from "./cjs-module";

console.log(cjsModule.foo); // 输出: 'bar'

* 导出方式：在一个 CommonJS 模块中，你可以通过 require 导入一个 ESM，但是这通常需要额外配置，因为 ESM 默认是异步加载的。此外，ESM 模块的导出在被 CommonJS 模块通过 require 导入时，必须访问它们的 default 属性或使用 import() 异步导入语句。
* 动态与静态：ESM 是静态的，这意味着你不能动态地导入或导出模块，所有的导入和导出必须在模块的顶层发生。CommonJS 模块是动态的，允许你将 require 语句放在代码的任何位置。由于这种静态与动态的区别，混用时要考虑代码组织和模块加载顺序。
* 构建与树摇（Tree Shaking）：Webpack 可以对 ESM 进行树摇优化，以去除未使用的导出。这可以减少最终构建包的大小。由于 CommonJS 模块是动态的，它们不完全支持树摇。如果在意最终包的大小，优先使用 ESM 来编写新模块可能会更好。

### 总结

在 Webpack 下，ESM 和 CommonJS 可以混用，但是混用时需要注意导入导出的细微差别，以及可能对构建优化产生的影响。理解这些模块系统之间的差异，并合理组织代码，可以充分利用 Webpack 提供的灵活性和功能，编写高效且可维护的现代 JavaScript 应用。

## [webpack] 如何优化产物大小【热度: 521】

**关键词：webpack 产物大小优化**

在使用 Webpack 进行项目构建时，减少包体积是提升加载速度、改善用户体验的关键措施之一。以下是一些通用的方法和技巧来减小构建结果的包体积：

### 1. 使用 Tree Shaking

Tree Shaking 是一个通过清除未引用代码（dead-code）的过程，可以有效减少最终包的大小。确保你的代码使用 ES6 模块语法（import 和 export），因为这允许 Webpack 更容易地识别并删除未被使用的代码。

在 webpack.config.js 中设置 mode 为 production 可自动启用 Tree Shaking。

### 2. 启用压缩(Uglification)

Webpack 通过压缩输出文件来减小包大小，如删除未使用的代码、缩短变量名等。确保在生产环境中启用了 UglifyJS 插件或 TerserPlugin。

const TerserPlugin = require("terser-webpack-plugin");

module.exports = {

optimization: {

minimize: true,

minimizer: [

new TerserPlugin({

/\* 附加选项 \*/

}),

],

},

};

### 3. 代码分割(Code Splitting)

通过代码分割，你可以把代码分成多个 bundle，然后按需加载，从而减少初始加载时间。Webpack 提供了多种分割代码的方式，最常见的是动态导入（Dynamic Imports）。

import(/\* webpackChunkName: "my-chunk-name" \*/ "path/to/myModule").then((module) => {

// 使用module

});

### 4. 使用 Externals 减轻体积

通过配置 externals 选项，可以阻止 Webpack 将某些 import 的包打包到 bundle 中，而是在运行时(runtime)再去从外部获取这些扩展依赖。

module.exports = {

externals: {

jquery: "jQuery",

},

};

### 5. 利用缓存(Caching)

使用 [contenthash] 替换 [hash] 或 [chunkhash] 来为输出文件命名，这确保了只有当文件内容改变时，文件名称才会改变，可以更好地利用浏览器缓存。

output: {

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

}

### 6. 移除未使用的 CSS

使用 PurgeCSS 或purify-css等工具检查你的 CSS 文件，自动去除未使用的 CSS，可以极大地压缩 CSS 的体积。

const PurgecssPlugin = require("purgecss-webpack-plugin");

### 7. 优化图片

使用image-webpack-loader等图片压缩插件，可以减小图片文件的体积。

module: {

rules: [

{

test: /\.(png|svg|jpg|jpeg|gif)$/i,

use: [

'file-loader',

{

loader: 'image-webpack-loader',

options: {

// 配置选项

},

},

],

},

],

}

### 8. 使用动态 Polyfills

只为那些实际需要它们的浏览器提供 polyfills，而不是所有浏览器都提供。

以上方法和技巧可以根据项目的具体需求和情况灵活使用，有的方法可能会对构建和重构现有代码产生较大影响，因此在采用前应评估其必要性和影响。

### 9. 高版本浏览器直接使用 es6 代码

将代码编译（或者说不编译）为 ES6（ECMAScript 2015）或更高版本的 JavaScript 代码，确实可以减少产物体积。

## [webpack] 代码分割中，让所有的外部依赖打成一个包，源码 source 打成一个包，该如何配置【热度: 352】

**关键词：webpack 代码分割**

为了实现你的需求，即将所有外部依赖（node\_modules 中的依赖）打包成一个单独的包，而你自己的源码打包成另一个包，可以通过配置 Webpack 的 optimization.splitChunks 选项来实现。下面是具体的实施方案：

**1. 编辑 webpack.config.js**

在你的 webpack.config.js 配置文件中，找到或添加 optimization 部分，并在 splitChunks 中配置如下：

module.exports = {

// ...其他配置

optimization: {

runtimeChunk: "single", // 为 webpack 运行时代码创建一个额外的包

splitChunks: {

cacheGroups: {

vendor: {

// 定义一个缓存组用以分离外部依赖

test: /[\\/]node\_modules[\\/]/, // 检索 node\_modules 目录下的模块

name: "vendors", // 分离后的包名称

chunks: "all", // 对所有模块生效

},

source: {

// 我们可以通过添加另一个缓存组来实现源码的分离（如果需要）

test: /[\\/]src[\\/]/, // 检索 src 目录

name: "source",

chunks: "all",

},

},

},

},

};

**解释**

* runtimeChunk: 'single' 创建一个运行时文件，管理模块化交互，比如加载和解析模块。
* 在splitChunks.cacheGroups 中定义了两个缓存组:
* vendor：这个缓存组的目标是将来自 node\_modules 目录的所有代码移动到命名为 vendors 的包中。它通过 test 属性来匹配 node\_modules 目录下的模块。
* source：这个部分是为了演示如何单独将 src 目录下的源代码打包成一个文件。这不是必须的，因为默认情况下，Webpack 会将未被上述规则匹配到的模块（即你的源代码）打包到主包中。

## [webpack] 多个 loader 对同一个资源进行处理， 他们之间如何通信？【热度: 134】

关键词：webpack loader 通信

在 webpack 中，loader 之间传递数据的常见方式是通过资源文件（即要处理的源文件本身）的内容。每个 loader 接收上一个 loader 的处理结果作为输入，并提供自己的输出给下一个 loader。这种方式适用于大多数使用场景。然而，在某些情况下，loader 需要在它们之间共享额外的状态或数据，而不仅仅是文件内容。对于这种需求，webpack 提供了一种机制，允许 loader 之间共享数据。

### 使用 this.data

在 webpack 4 及以后的版本中，一个 loader 可以利用它的 this.data 属性来共享会话数据。这个属性是特定于当前 loader 运行实例的，可以在 loader 的 pitch 阶段和正常的加载阶段之间共享数据。

// pitch 阶段

module.exports.pitch = function (remainingRequest, precedingRequest, data) {

data.sharedValue = "Hello from pitch phase";

};

在上面的代码片段中，pitch 方法设置了 data.sharedValue。这个 pitch 方法是可选的，它在 loader 处理资源之前执行。data 对象会从 pitch 阶段传递到正常的加载阶段，从而可以在后者中访问之前设置的共享值。

// 正常的加载阶段

module.exports = function (content) {

const callback = this.async();

const sharedValue = this.data.sharedValue;

// 这里可以根据 sharedValue 来处理 content

console.log(sharedValue); // 将输出 "Hello from pitch phase"

callback(null, content);

};

### 使用自定义属性

一些特定的 loader 实现可能通过向源文件内容附加额外的信息来实现间接的通信。例如，一个 loader 可以在文件内容的末尾追加一些注释或者特殊标记，然后下一个 loader 可以读取这些注释或标记来获取必要的信息。然而，这种方法是高度依赖上下文且难以维护的，不推荐在实际项目中使用。

### 注意事项

当使用一种方法在 loader 之间共享数据时，请注意数据的共享是在每个模块的构建过程中进行的，这些数据是特定于当前处理中的资源文件的。通过这种方式共享的数据不应该包含敏感信息，也不应该用于在不同模块或不同构建之间共享全局状态。

理解这些机制以及如何在 loader 之间正确共享数据是创建高效可维护 webpack 构建流程的关键。

## [webpack] 手写 webpack loader 有哪些重要 api 与注意事项？【热度: 718】

关键词：手写 webpack loader

在开发一个 webpack loader 时，除了理解 loader 的基本概念和功能之外，还有一些重要的 API 和注意事项是必需了解的。这些能够帮助你更高效地编写和调试 loader。

### 重要 API

* this.callback:

在 loader 函数内部，this.callback 是一个允许 loader 异步返回结果的函数。你可以通过 this.callback(err, content, sourceMap, meta) 来传递错误或返回结果。

* this.async:

调用 this.async 会返回一个 callback 函数，你可以在异步操作完成后通过这个函数返回结果。如果 loader 要进行异步处理，这个方法非常有用。

* this.loaders:

this.loaders 是一个包含所有需要应用到当前处理文件的 loaders 的数组，当前 loader 的信息也包含在内。

* this.resourcePath 和 this.resourceQuery:

这两个属性提供了当前正在处理的资源文件的路径和查询字符串。

* this.data:

在 loader 的 pitch 阶段和普通阶段之间共享数据的自由对象。

* Loader Utils (loader-utils):

loader-utils 提供了一些实用的工具函数，比如 getOptions(this) 用于获取 loader 配置项。

### 注意事项

#### 使用异步 API 处理异步任务:

对于需要进行异步操作的 loader，应使用 this.async 来获取异步 callback 函数，而不是直接返回内容。

#### 保持 loader 的简单:

按照最佳实践，每个 loader 只做一件事情。这让 loader 链更加灵活和可维护。

#### 避免使用箭头函数:

在编写 loader 时，避免使用箭头函数来声明 loader 函数，因为箭头函数会绑定父作用域的 this，而你需要访问 webpack 传递给 loader 函数的 this 上下文。

#### 处理异常:

在处理资源的过程中，如果遇到错误，应该使用 this.emitError 方法或通过 this.callback 函数的第一个参数传递错误。

#### 缓存:

除非有特定的理由，否则避免关闭 loader 的缓存。webpack 默认会缓存 loader 的结果，以提升构建性能。

#### 资源映射（Source Maps）:

如果你的 loader 转换源内容，生成新的源内容，应当生成新的 source map。然后，使用 this.callback 来返回更新后的代码和对应的 source map。

#### 通信:

如果有多个 loader 对同一个资源进行处理，它们之间可以通过 this.data 来共享数据。

掌握并妥当使用上述 API 和注意事项，将帮助你开发出高效、健壮且易于维护的 webpack loader。

## [webpack] 手写 loader: 我要在方法调用的时候， 上报调用源文件地址【热度: 117】

**关键词：手写 webpack loader**

要在方法调用时上报调用源文件的地址，并且希望通过 webpack 编译时来实现，你可以通过编写一个自定义的 webpack loader 来操作源代码，为特定的方法调用插入上报的代码。自定义 loader 本质上是一个函数，该函数接收源码作为输入，对源码进行处理后返回新的源码。

**步骤 1: 设计你的上报逻辑**

首先明确你想要上报的信息和上报的方式。比如，你可能想要在方法调用时，插入一个上报函数调用，该函数包含当前文件的路径和文件名。

**步骤 2: 创建自定义 Loader**

你可以开始编写你的 loader。假设你有一个上报函数 reportFunction(filePath)，你希望自动为所有 targetMethod() 调用注入这个上报函数。

**步骤 3: 在 webpack 配置中使用你的 Loader**

在你的 webpack.config.js 文件中，添加一个 module.rules 条目，以确定哪些文件应该通过你的 loader 处理：

module.exports = {

module: {

rules: [

{

test: /\.js$/, // 匹配 JavaScript 文件

use: [

{

loader: "path/to/your/report-loader.js", // 使用自定义 loader 的路径

},

],

},

],

},

};

确保将 loader 属性设置为你自定义 loader 文件的路径。

### 注意事项

* 正则表达式: 我在例子中使用的正则表达式非常简单，只匹配特定形式的方法调用。根据你的需要，可能要编写更复杂的正则表达式或使用其他方法（比如抽象语法树解析库，如 Babel）来更准确地识别和修改代码。
* 安全性: 自动修改源代码会带来风险，确保你的匹配和替换逻辑不会导致代码中出现意外的改变。
* 性能: 增加自定义 loader 可能会影响构建的速度，特别是匹配和修改逻辑比较复杂的时候。

编写和测试好你的 loader 后，就可以集成到你的项目中，通过 webpack 构建过程中自动执行所需的代码注入了。

## [webpack] 构建过程中， 是如何将我们 es6 代码 编译为 es5【热度: 420】

**关键词：es6 编译为 es5**

Webpack 本身是一个模块打包器（bundler），它并不直接负责将 ES6 代码编译为 ES5 代码。Webpack 的主要功能是将项目中的所有模块（JavaScript、图片、CSS 等）打包成一个或多个 bundle，以供浏览器加载。然而，Webpack 可以通过加载器（loaders）和插件（plugins）来扩展其功能，实现代码的转换和编译。

### 将 ES6 代码编译为 ES5 的过程通常涉及以下几个步骤：

#### Babel 转换：

Babel 是一个流行的 JavaScript 编译器，可以将 ES6+ 代码转换为向后兼容的 JavaScript 版本，即 ES5。Webpack 可以与 Babel 配合使用，通过 Babel Loader 来实现代码的转换。

#### Loader 配置：

在 Webpack 配置中，你可以指定使用 Babel Loader 来处理 .js 文件。当 Webpack 处理 JavaScript 文件时，Babel Loader 会被调用，并将 ES6 代码转换为 ES5。

#### Babel 预设：

Babel 使用预设（presets）来定义转换规则。@babel/preset-env 是一个常用的预设，它会自动配置 Babel 以兼容目标浏览器的版本。

#### Polyfills：

为了支持旧浏览器，Babel 还可以引入 polyfills，这些是提供现代 JavaScript 特性的第三方代码片段。例如，core-js 和 regenerator-runtime 是两个常用的 polyfill 库。

#### 转换过程：

* 解析：Babel 首先解析 ES6 代码为 AST（抽象语法树）。
* 转换：然后，Babel 遍历 AST，将 ES6 语法转换为 ES5 语法。
* 生成：最后，Babel 生成新的 ES5 代码。

#### Tree Shaking：

Webpack 支持 Tree Shaking，这是一种去除未使用代码的技术，可以进一步减小最终打包文件的大小。

#### 代码分割：

Webpack 还可以进行代码分割，将代码拆分成多个 chunk，按需加载，提高应用的加载性能。

#### 优化：

Webpack 还提供了多种优化选项，如压缩代码、合并模块等，以减小最终输出文件的大小。

### 总结

Webpack 本身不直接编译 ES6 代码为 ES5，\*\*而是通过与 Babel 等工具的配合使用，实现代码的转换和编译。Webpack 的主要作用是模块打包，而 Babel 负责将现代 JavaScript 代码转换为广泛兼容的格式。\*\*通过合理配置 Webpack 和 Babel，可以高效地构建和优化前端应用。