## webpack配制说明

Webpack运行在node.js环境下，它的配置文件webpack.config.js遵循CommonJS规范，最终export出一个json对象。

webpack.config.js基础配置说明：

1、entry：指定了模块的入口，它让源文件加入构建流程中被webpack控制。

2、output：配置输出文件的存放位置、文件名、文件基础路径publicPath。

3、module：配置各种类型文件的解析规则,比如说.vue文件、.js文件、.css文件的loader等。

4、resolve：配置alias(别名)，或者定义寻找模块的规则。

5、plugins：配置扩展插件，扩展webpack的更多功能。

6、devServer：实现本地http服务等。

## 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' },

],

},

],

},

};

## 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'

})

]

}

## 说一说Loader和Plugin的区别？

Loader 本质就是一个函数，在该函数中对接收到的内容进行转换，返回转换后的结果。 因为 Webpack 只认识 JavaScript，所以 Loader 就成了翻译官，对其他类型的资源进行转译的预处理工作。

Plugin 就是插件，基于事件流框架 Tapable，插件可以扩展 Webpack 的功能，在 Webpack 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，在合适的时机通过 Webpack 提供的 API 改变输出结果。

Loader 在 module.rules 中配置，作为模块的解析规则，类型为数组。每一项都是一个 Object，内部包含了 test(类型文件)、loader、options (参数)等属性。

Plugin 在 plugins 中单独配置，类型为数组，每一项是一个 Plugin 的实例，参数都通过构造函数传入。

## Webpack构建流程简单说一下

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。

### 简单说

1. 初始化：启动构建，读取与合并配置参数，加载 Plugin，实例化 Compiler
2. 编译：从 Entry 出发，针对每个 Module 串行调用对应的 Loader 去翻译文件的内容，再找到该 Module 依赖的 Module，递归地进行编译处理
3. 输出：将编译后的 Module 组合成 Chunk，将 Chunk 转换成文件，输出到文件系统中

## 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。

## 使用webpack开发时，你用过哪些可以提高效率的插件？

(注重实际，用户的体验还是要从小抓起的)

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

## source map是什么？生产环境怎么用？

**Source Map概念**

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原理**：Source Map包含了源代码和编译后的代码之间的映射关系，通常是一个JSON文件，它包含了每行代码的映射信息，例如源文件路径、行号、列号等。当浏览器执行编译后的代码时，它会通过Source Map将执行位置映射回源代码的位置，从而使得开发者可以直接在源代码中进行调试。

**怎么配置Source Map**：在Webpack中，可以使用devtool配置选项来生成Source Map。常用的选项有：

eval：生成每个模块的eval代码，并且模块执行完后，eval代码被执行。这种方式速度很快，但是不适合生产环境。

source-map：生成独立的source-map文件，适合生产环境，但是会增加构建时间和文件大小。

cheap-source-map：生成source-map，但是不包含列信息，适合大型项目。

cheap-module-source-map：生成source-map，同时会将loader的sourcemap也加入进来。

## 了解过Tree-shaking吗？

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。

## 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。

## 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。

## 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。

## 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 }}

## 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。

## 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]'

}

}]

}]

}

}

## 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。

## 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

## 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

## 聊一聊Babel原理吧

大多数JavaScript Parser遵循 estree 规范，Babel 最初基于 acorn 项目(轻量级现代 JavaScript 解析器) Babel大概分为三大部分：

解析：将代码转换成 AST

词法分析：将代码(字符串)分割为token流，即语法单元成的数组

语法分析：分析token流(上面生成的数组)并生成 AST

转换：访问 AST 的节点进行变换操作生产新的 AST

[Taro](https://link.juejin.cn?target=https://github.com/NervJS/taro/blob/master/packages/taro-transformer-wx/src/index.ts%23L15" \o "https://github.com/NervJS/taro/blob/master/packages/taro-transformer-wx/src/index.ts#L15" \t "/Users/gdj/Documents\\x/_blank)就是利用 babel 完成的小程序语法转换

生成：以新的 AST 为基础生成代码

## webpack工作原理

### 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）

从启动构建到输出结果一系列过程：

（1）初始化参数：解析webpack配置参数，合并shell传入和webpack.config.js文件配置的参数，形成最后的配置结果。

（2）开始编译：上一步得到的参数初始化compiler对象，注册所有配置的插件，插件监听webpack构建生命周期的事件节点，做出相应的反应，执行对象的 run 方法开始执行编译。

（3）确定入口：从配置的entry入口，开始解析文件构建AST语法树，找出依赖，递归下去。

（4）编译模块：递归中根据文件类型和loader配置，调用所有配置的loader对文件进行转换，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理。

（5）完成模块编译并输出：递归完事后，得到每个文件结果，包含每个模块以及他们之间的依赖关系，根据entry配置生成代码块chunk。

（6）输出完成：输出所有的chunk到文件系统。

注意：在构建生命周期中有一系列插件在做合适的时机做合适事情，比如UglifyPlugin会在loader转换递归完对结果使用UglifyJs压缩覆盖之前的结果

## webpack的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减小项目体积

去除不必要的插件

去除devtool选项

分离css

使用webpack.optimize.UglifyJsPlugin插件压缩混淆[js](//www.360doc.cn/outlink.html?url=http://lib.csdn.net/base/javascript)代码

提取第三方库 dll

按需打包

提取公共组件

gzip压缩

优化打包结果的核心目标就是让打出来的**包体积更小**。

**打包体积分析**：使用webpack-bundle-analyzer来分析，一般脚手架里直接运行命令行就能生成打包体积图，很方便，然后可以根据包体积能定向优化。

**代码压缩**：使用UglifyJsPlugin、MiniCssExtractPlugin等插件来对JS代码和CSS代码进行压缩，减小代码体积，实际开发中一般脚手架也会默认有压缩的配置。

**使用懒加载**：可以使用Webpack的动态导入功能，实现懒加载，在需要时再加载代码块。这可以缩短首屏加载时间，提升体验。

**开启gzip**：使用compression-webpack-plugin插件，生成额外的gzip静态文件，然后部署时再开启Nginx的gzip即可。

使用**splitChunks提取公共代码**，在脚手架中一般是默认开启的。

**分离第三方库**：将第三方库从应用程序代码中分离出来，单独打包，这样可以提高缓存效率并减小应用程序代码的大小。

**开启Tree Shaking**，对于Vue和React项目，一般是默认开启Tree Shaking的，我们在编写代码时尽量使用ES模块化语法，就可以了。

## 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：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。

## 如何优化 Webpack 的构建速度？

1. 使用高版本的 Webpack 和 Node.js
2. 多进程/多实例构建：HappyPack(不维护了)、thread-loader
3. 压缩代码

多进程并行压缩

webpack-paralle-uglify-plugin

uglifyjs-webpack-plugin 开启 parallel 参数 (不支持ES6)

terser-webpack-plugin 开启 parallel 参数

通过 mini-css-extract-plugin 提取 Chunk 中的 CSS 代码到单独文件，通过 css-loader 的 minimize 选项开启 cssnano 压缩 CSS。

1. 图片压缩

使用基于 Node 库的 imagemin (很多定制选项、可以处理多种图片格式)

配置 image-webpack-loader

1. 缩小打包作用域：

exclude/include (确定 loader 规则范围)

resolve.modules 指明第三方模块的绝对路径 (减少不必要的查找)

resolve.mainFields 只采用 main 字段作为入口文件描述字段 (减少搜索步骤，需要考虑到所有运行时依赖的第三方模块的入口文件描述字段)

resolve.extensions 尽可能减少后缀尝试的可能性

noParse 对完全不需要解析的库进行忽略 (不去解析但仍会打包到 bundle 中，注意被忽略掉的文件里不应该包含 import、require、define 等模块化语句)

IgnorePlugin (完全排除模块)

合理使用alias

1. 提取页面公共资源：

基础包分离：

使用 html-webpack-externals-plugin，将基础包通过 CDN 引入，不打入 bundle 中

使用 SplitChunksPlugin 进行(公共脚本、基础包、页面公共文件)分离(Webpack4内置) ，替代了 CommonsChunkPlugin 插件

1. DLL：

使用 DllPlugin 进行分包，使用 DllReferencePlugin(索引链接) 对 manifest.json 引用，让一些基本不会改动的代码先打包成静态资源，避免反复编译浪费时间。

HashedModuleIdsPlugin 可以解决模块数字id问题

1. 充分利用缓存提升二次构建速度：

babel-loader 开启缓存

terser-webpack-plugin 开启缓存

使用 cache-loader 或者 hard-source-webpack-plugin

1. Tree shaking

打包过程中检测工程中没有引用过的模块并进行标记，在资源压缩时将它们从最终的bundle中去掉(只能对ES6 Modlue生效) 开发中尽可能使用ES6 Module的模块，提高tree shaking效率

禁用 babel-loader 的模块依赖解析，否则 Webpack 接收到的就都是转换过的 CommonJS 形式的模块，无法进行 tree-shaking

使用 PurifyCSS(不在维护) 或者 uncss 去除无用 CSS 代码

purgecss-webpack-plugin 和 mini-css-extract-plugin配合使用(建议)

1. Scope hoisting

构建后的代码会存在大量闭包，造成体积增大，运行代码时创建的函数作用域变多，内存开销变大。Scope hoisting 将所有模块的代码按照引用顺序放在一个函数作用域里，然后适当的重命名一些变量以防止变量名冲突

必须是ES6的语法，因为有很多第三方库仍采用 CommonJS 语法，为了充分发挥 Scope hoisting 的作用，需要配置 mainFields 对第三方模块优先采用 jsnext:main 中指向的ES6模块化语法

1. 动态Polyfill

建议采用 polyfill-service 只给用户返回需要的polyfill，社区维护。 (部分国内奇葩浏览器UA可能无法识别，但可以降级返回所需全部polyfill)

## 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。

## 讲一下你对模块联邦的理解？

模块联邦是实现多个项目之间共享代码的机制。

举个例子，假设我们有一个微前端应用，其中包含了一个商品管理应用和一个订单管理应用，这两个应用都需要使用到同一个UI组件库。

为了避免重复的代码，我们可以将UI组件库拆分成一个独立的子应用作为模块提供方，然后通过模块联邦的方式在商品管理应用和订单管理应用中动态加载该组件库。

在模块提供方里配置ModuleFederationPlugin：

const ModuleFederationPlugin = require('webpack/lib/container/ModuleFederationPlugin');

module.exports = {

// ...

plugins: [

new ModuleFederationPlugin({

name: 'app1', // 应用名

filename: 'remoteEntry.js',

exposes: { // 需要共享的模块和对应的路径

'./Button': './src/components/Button',

},

shared: ['react', 'react-dom'], // 共享的第三方库

}),

],

};

然后在模块调用方里配置ModuleFederationPlugin：

const ModuleFederationPlugin = require('webpack/lib/container/ModuleFederationPlugin');

module.exports = {

// ...

plugins: [

new ModuleFederationPlugin({

name: 'app2', // 调用方应用名

filename: 'remoteEntry.js',

remotes: {

app1: 'app1@http://localhost:3001/remoteEntry.js', // 模块提供方的路径

},

shared: ['react', 'react-dom'], // 共享的第三方库

}),

],

};