## webpack打包流程，文件监听原理以及与Vite的区别

打包流程：

1. 初始化参数：从配置文件和Shell语句中读取并合并参数，加载用户配置的plugins得出最终的参数。
2. 用上一步得到的参数初始化Compiler对象，加载所有配置的插件，执行对象的run方法开始执行编译。
3. 从入口文件（entry）开始解析，并且找到其导入的依赖模块，对不同模块文件使用对应的Loader进行编译，最终转为Javascript代码。然后通过acorn库生成模块代码的AST语法树，通过语法树就可以分析这个模块是否还有依赖的模块，然后递归解析，
4. 整个过程中webpack会通过发布订阅模式，向外抛出一些hooks，而webpack的插件即可通过监听这些关键的事件节点，执行插件任务进而达到干预输出结果的目的。
5. 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，这步是可以修改输出内容的最后机会；在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。

文件监听原理：

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后通知监听 者。

与vite的区别：

webpack在启动本地开发服务器的时候，会遍历所有依赖文件进行一次完全的打包， 并将bundle打包文件存在内存中（不生成文件）供浏览器调用。

Vite的本地开发服务是基于ES module，用esbuild预构建依赖，esbuild使用 Go 编写，并且比以 JavaScript编写的打包器预构建依赖快 10-100 倍（现代浏览 器基本已经全部支持了import/export语法了）。

Vite会依据当前路由文件，找到文件中的依ES module件通过import引入，相当 于路由懒加载，因此速度更快。

Vite的生产打包使用的是Rollup，Rollup仅支持ES module，使用 CommonJS 和 AMD需要安装对应插件，Rollup将小块代码编译成大块复杂的代码，不支持模块按 需加载（webpack支持）。



## webpack配置

### 导出形式

1. 对象
2. 函数（参数一是环境参数，参数二是传递给webpack的配置项）
3. Pormise（延迟加载）
4. 数组（导出多种配置）

### **context**

基础目录路径。

### entry

构建入口文件。

### mode

区分环境：development、production和none。

开发环境：需要更快的构建速度，需要打印debug信息，需要 live reload 或hot reload 功能，需要sourcemap方便定位问题。

生产环境：需要更小的包体积，代码压缩，tree-shaking，需要进行代码分割，需要压缩图片体积。

在webpack中，可以通过mode选项为webpack指定一些默认的配置：

<https://segmentfault.com/a/1190000013712229>

### output

输出文件配置。

### module

决定如何处理项目中的不同类型的模块，为不同的资源配置不同的loader等。

#### loader

webpack只能理解JavaScript和JSON文件，这是webpack开箱可用的自带能力。 loader让webpack能够去处理其他类型的文件，并将它们转换为有效模块，以供应 用程序使用，以及被添加到依赖图中。

常用loader：

1. css-loader:加载 CSS，支持模块化、压缩、文件导入等特性。
2. 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。
3. 处理CSS中的图片或文字资源时，常用的两种loader是[file-loader](https://link.jianshu.com/?t=https://doc.webpack-china.org/loaders/file-loader/)或者[url-loader](https://link.jianshu.com/?t=https://doc.webpack-china.org/loaders/url-loader/)，两者的主要差异在于：url-loader可以设置图片大小限制，当图片超过限制时，其表现行为等同于file-loader，而当图片不超过限制时，则会将图片以[base64](https://so.csdn.net/so/search?q=base64&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/ma_jiang/article/details/_blank)的形式打包进CSS文件，以减少请求次数。
4. webpack5新增资源模块(asset module)，允许使用资源文件（字体，图标等）而无需配置额外的loader。
5. babel-loader：可以把es6等新标准的语法转化为原生js进行打包，可以在根目录下新建.babelrc.js文件进行补充配置，包括一些最新的语法标准（例如@装饰器）都需要安装插件并在.babelrc.js中配置。
6. ts-loader：ts代码转为js代码。
7. vue-loader：加载 Vue.js单文件组件。
8. eslint-loader/tslint-loader：通过ESLint检查JavaScript/TypeScript 代码。

### resolve

确定编译范围等。

### optimization

根据选择的 [mode](https://webpack.docschina.org/concepts/mode/) 来执行不同的优化。

### plugins

plugins选项用于以各种方式自定义webpack构建过程。

与Loader 用于转换特定类型的文件不同，插件（Plugin）可以贯穿webpack 打包的生 命周期，执行不同的任务，实现一些webpack不能实现的功能。

### devServer

配置本地服务。

### cache

缓存生成的webpack模块和chunk，来改善构建速度。

### devtool

此选项控制是否生成，以及如何生成source map。

开发环境推荐配置为eval-cheap-module-source-map，特点是:

1. 快速（eval）。
2. 包含 dataUrl形式的SourceMap 文件。
3. 可以在编译后的代码中定位到错误所在行信息（module）。
4. 不需要定位列信息，打包速度较快（cheap）。

生产环境： hidden-source-map

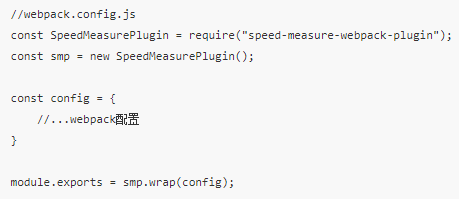
### hash命名



## webpack性能优化

### 性能量化工具

speed-measure-webpack-plugin插件可以测量各个插件和loader所花费的时间。



### 优化构建速度

主要策略：减少文件解析（noParse，ignorePlugin，externals，dllPlugin）、缓存（cache-loader，loader-option-cacheDirectory）、多进程构建（happypack，thread-loader）。

#### HMR

热更新，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。

原理：

1. webpack-dev-middleware调用webpack的api对文件系统watch，当文件发生改变后，webpack重新对文件进行编译打包，然后保存到内存中，不生成文件。
2. [sockjs](https://links.jianshu.com/go?to=(https://github.com/sockjs/sockjs-client)" \t "https://www.jianshu.com/p/_blank)在服务端和浏览器端建立了一个webSocket长连接，devServer通知浏览器端文件发生改变（compile的done事件），当compile完成后，webpack-dev-server通过\_sendStatus 方法将编译打包后的新模块hash值发送到浏览器端。
3. 浏览器端的bundle.js（打包文件，被webpack-dev-server/client变更过），接收到消息后判断是不是热更新，不是的话直接刷新浏览器，是的话webpack-dev-server/client发送webpackHotUpdate消息给webpack/hot/dev-server，调用 webpack/lib/HotModuleReplacement.runtime（简称HMR runtime）中的check方法，检测是否有新的更新。
4. 在check过程中会调用webpack/lib/JsonpMainTemplate.runtime（简称jsonp runtime）中的方法通过jsonp获取新的代码。

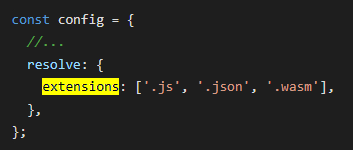
#### 缩小编译范围，减少不必要的编译工作，即 modules、mainFields、noParse、includes、exclude、alias 全部用起来。

##### resolve

modules指定webpack解析模块时应该搜索的目录。

mainFields指定字段作为入口文件描述字段，减少搜索步骤。

extensions定义的后缀查找文件，使用import等访问文件时可以忽略文件名后缀，webpack会根据extensions定义的后缀查找文件。



##### module

module.rules: includes、exclude指定和排除loader的处理范围，exclude 的优先级高于 include。

module.noParse:如果一些第三方模块没有AMD/CommonJS（import, require, define）依赖其他包，可以使用noParse来标识这个模块，这样Webpack会引入这些模块，但是不进行转化和解析，从而提升Webpack的构建性能，例如：jquery、lodash。

#### Happypack、webpack-parallel-uglify-plugin和thread-loader

多进程打包。

#### cache-loader

目前大部分loader都提供cache配置项。

（如果不支持）一些性能开销较大的 loader 之前添加cache-loader，将结果缓存中磁盘中。默认保存在node\_modueles/.cache/cache-loade目录下。

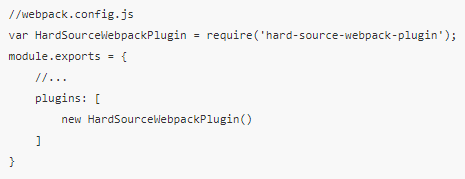
webpack5自带cache缓存功能。



#### hardSourceWebpackPlugin

HardSourceWebpackPlugin为模块提供中间缓存，缓存默认的存放路径是: node\_modules/.cache/hard-source。

配置hard-source-webpack-plugin，首次构建时间没有太大变化，但是第二次开始，构建时间大约可以节约80%。



Webpack5不再允许使用该插件，使用cache属性。

#### dllPlugin（抽离模块，建议使用hardSourceWebpackPlugin或cache代替）

如果所有的JS文件都打成一个JS文件，会导致最终生成的JS文件很大，这个时候，我们就要考虑拆分bundles，需要新建一个webpack.dll.config.js文件。

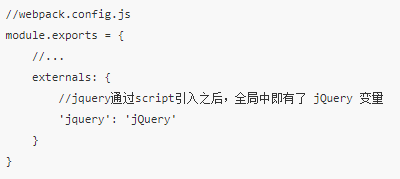
DllPlugin 和 可以实现拆分bundles，并且可以大大提升构建速度，DllPlugin和DLLReferencePlugin都是webpack的内置模块。

使用 dllPlugin将不会频繁更新的库进行编译，当这些依赖的版本没有变化时，就不需要重新编译，只用于开发环境。

#### ignorePlugin

webpack的内置插件，作用是忽略第三方包指定目录打包，例如moment包的其他语言包等。

#### externals

我们可以将一些JS文件存储在CDN上(减少Webpack打包出来的js体积)，在index.html中通过<script>标签引入。仍然可以通过import的方式去引用(如import $ from 'jquery')：

### 优化构建结果

主要策略：压缩文件（js，css）、cdn分担加载压力（externals）、清除无用代码（tree-shaking，ignorePlugin）。

#### 压缩js

uglifyjs-webpack-plugin、uglifyjs-webpack-plugin

[webpack](https://so.csdn.net/so/search?q=webpack&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_15601471/article/details/_blank) 在v4.26.0将默认的压缩插件从uglifyjs-webpack-plugin改成了teaser-webpack-plugin，直接引用即可，如果需要自定义配置则仍需下载。

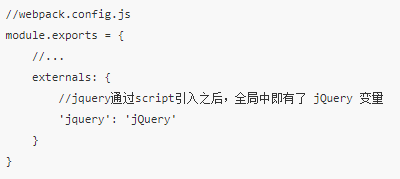
#### 压缩css

optimize-css-assets-webpack-plugin, css-minimizer-webpack-plugin

#### 清除无用css

purgecss-webpack-plugin

#### externals

我们可以将一些JS文件存储在CDN上(减少Webpack打包出来的js体积)，在index.html中通过<script>标签引入。仍然可以通过import的方式去引用(如import $ from 'jquery')：

#### ignorePlugin

webpack的内置插件，作用是忽略第三方包指定目录打包，例如moment包的其他语言包等。

#### tree-shaking

如果使用ES6的import语法，那么在生产环境下，会自动移除没有使用到的代码。

按需打包，如果所有代码都不包含副作用，我们就可以简单地将sideEffects属性标记为false来告知webpack，它可以安全地删除未用到的export导出。

配置位置：package.json或者是module.rules。在.bablerc 里面配置model：false，也可在生产环境下默认开启。

#### Scope Hoisting

即作用域提升，原理是将多个模块放在同一个作用域下，并重命名防止命名冲突，通过这种方式可以减少函数声明和内存开销。webpack 默认支持，在生产环境下默认开启；只支持es6代码。

### 优化运行体验（提升首屏的加载速度）

主要策略：js与css并行加载（extract-text-webpack-plugin、mini-css-extract-plugin）、抽离公共代码（splictChunks）、清除无用代码（tree-shaking）。

#### extract-text-webpack-plugin、mini-css-extract-plugin

如果不使用插件，css将被打包到style标签中；使用插件分离css到link加载css文 件，使js和css并行加载。

mini-css-extract-plugin相比于extract-text-webpack-plugin的特点：

异步加载，不重复编译，性能更好，更容易使用；只针对CSS ,目前缺失HMR功能。

#### splitChunks

公共模块抽取,利⽤浏览器缓存可以⻓期缓存这些⽆需频繁变动的公共代码，避免了重复下载。代码分割的三种方式：entry、import和splitChunks。

splitChunks是根据cacheGroups去拆分模块的，cacheGroups会继承splitChunks的配置。

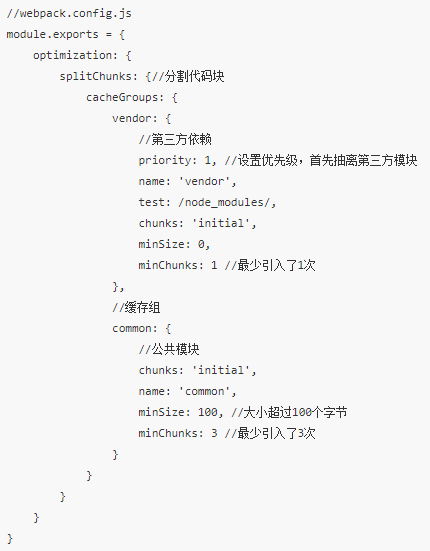
chunks：拆分模块的范围。

minSize：被拆分的最小体积

minChunks：被拆分的最引用次数

maxAsyncRequests：在拆分chunk后，import异步加载的数不超过指定的值。

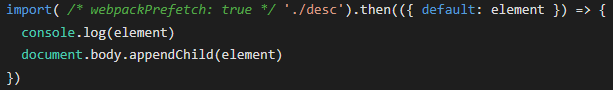
maxIntialRequests：在拆分chunk后，需要请求的初始chunk数量不超过指定的值。



可以借助webpack-bundle-analyzer 查看一下是哪些包的体积较大。

#### prefetch&preload

prefetch(预获取)：浏览器空闲的时候进行资源的拉取



preload(预加载)：提前加载后面会用到的关键资源（因为会提前拉取资源，如果不是特殊需要，谨慎使用）。



## antd组件按需加载 js部分：tree-shaking + import {}

css部分：babel-plugin-import

import { DatePicker } from 'antd' ==>

import DatePicker from 'antd/es/date-picker'; // 加载 JS

import 'antd/es/date-picker/style/css'; // 加载 CSS

## 手写Loader&Plugin

#### 自定义loader

自定义loader其实是一个函数。这个函数不能是一个箭头函数，因为它需要继承this，如果需要使用webpack的方法，那么需要使用webpack的this方法，比如this.version etc这个函数接收一个参数source，source代表匹配到文件的值（模块内容），是一个字符串。





实现了一个替换模块中匹配内容的loader。

#### 自定义plugin

1. 自定义plugin一个JavaScript命名函数，在插件函数的prototype上定义一个apply方法（webpack在打包前会去调用所有plugin的apply方法）。
2. compiler暴露了和Webpack整个生命周期相关的钩子（通过compiler.hooks[hookName]调用），自定义plugin需要指定一个绑定到webpack自身的事件钩子，处理webpack内部实例的特定数据，异步的事件完成后调用webpack提供的回调通知webpack进入下一个流程，不然会卡住。

· compiler.options 可以访问本次启动 webpack 时候所有的配置文件，包括但不限 于loaders、entry、output、plugin等等完整配置信息。

· compiler.inputFileSystem 和 compiler.outputFileSystem 可以进行文件操作， 相当于 NodeJs 中的 fs。

· compiler.hooks 可以注册 tapable 的不同种类 Hook, 从而可以在 compiler 生 命周期中植入不同的逻辑。

1. compilation暴露了与模块和依赖有关的粒度更小的事件钩子。

·compilation.modules 可以访问所有模块，打包的每一个文件都是一个模块。

·compilation.chunks chunk即多个 modules 组成而来的一个代码块。入口文件引入的资源组成一个chunk,通过代码分割的模块又是另外的chunk。

·compilation.assets 可以访问本次打包生成所有文件的结果。

·compilation.hooks 可以注册 tapable 的不同种类 hook,用于 compilation编译模块阶段进行逻辑添加以及修改。

1. 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。





## Babel原理

1. 解析：将代码转换成AST。

（1）词法分析：将代码(字符串)分割为token流，即语法单元成的数组。

（2）语法分析：分析token流(上面生成的数组)并生成AST。

2. 转换：转换：对于AST进⾏变换⼀系列的操作，babel 接受得到 AST 并通过babel-traverse对其进⾏遍历，在此过程中进⾏添加、更新及移除 等操作。Taro就是利用babel完成的小程序语法转换。

1. 生成：以新的AST为基础生成代码。

AST语法树：抽象语法树，是对源代码的树形结构描述。

babel的主要编译流程是parse 是把源码转成AST，transform是对AST做增删改， generate是打印AST成目标代码并生成sourcemap。

## Babel配置

babel-loader仅仅是识别匹配文件和接受对应参数的函数。

babel-core是babel最核心的一个编译库，他可以将我们的代码进行词法分析--语法分 析--语义分析过程从而生成AST抽象语法树，从而对于“这棵树”的操作之后再通过编 译称为新的代码。

babel-preset-env告诉babel需要以为什么样的规则进行代码转换，语法层面的转化 preset-env完全可以胜任，但对一些内置方法和构造函数无能为力。

对低版本代码转换有两个方式：

1. @babel/preset-env + @babel/polyfill实现对方法以及api模块低版本代码的支持， 原理是全局对象上添加属性以及直接修改内置对象的Prototype上添加方，会造成 变量污染（适合于自己的应用）。
2. @babel/preset-env + @babel/plugin-transform-runtime 按需加载的解决方案，在需要的地方引入需要的api或方法即可。 @babel/plugin-transform-runtime解决了@babel/runtime需要手动引入 和重复生成冗余代码的问题（适合开发第三方类库）。

详见：<https://segmentfault.com/a/1190000020237790>

Babel > 7.4.0:

preset-env包含了各类api和方法的转换，不再推荐使用@babel/polyfill，应选择 core-js使用。