## Webpack相关

## loader是什么?有哪些常见的loader?

**loader是模块转换器，用于把模块原内容按照需求转换成新内容。**（**通过使用不同的loader，webpack可以调用外部的脚本或工具，实现对不同格式的文件的处理**，比如说分析转换sass为css，或者把下一代的JS文件（ES6，ES7)转换为现代浏览器兼容的JS文件，对React的开发而言，合适的Loaders可以把React的中用到的JSX文件转换为JS文件。）

babel-loader：把 ES6 转换成 ES5

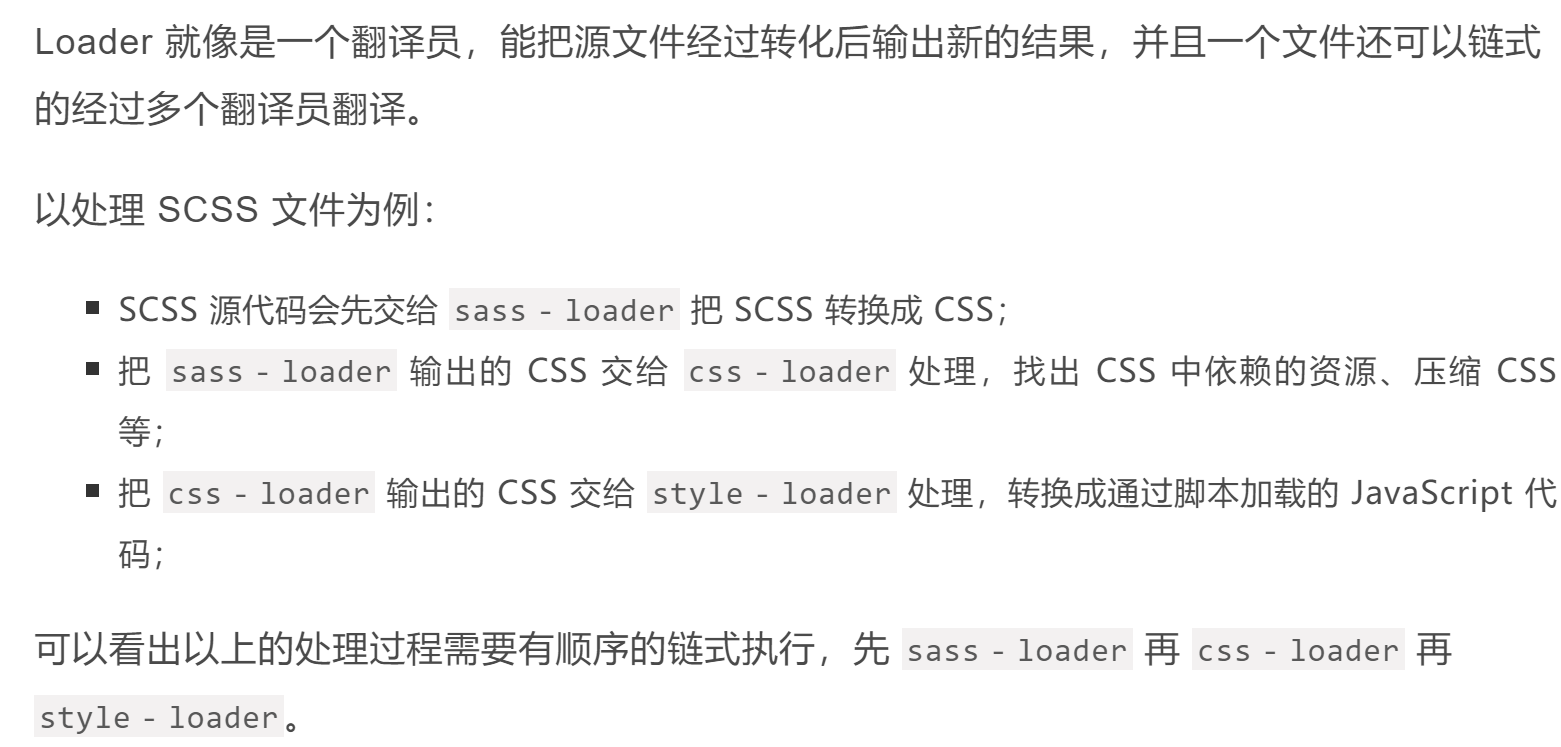
css-loader：**加载 CSS，使用类似**@import 和 url(...)的语法实现require()的功能，**支持模块化、压缩、文件导入等特性**。

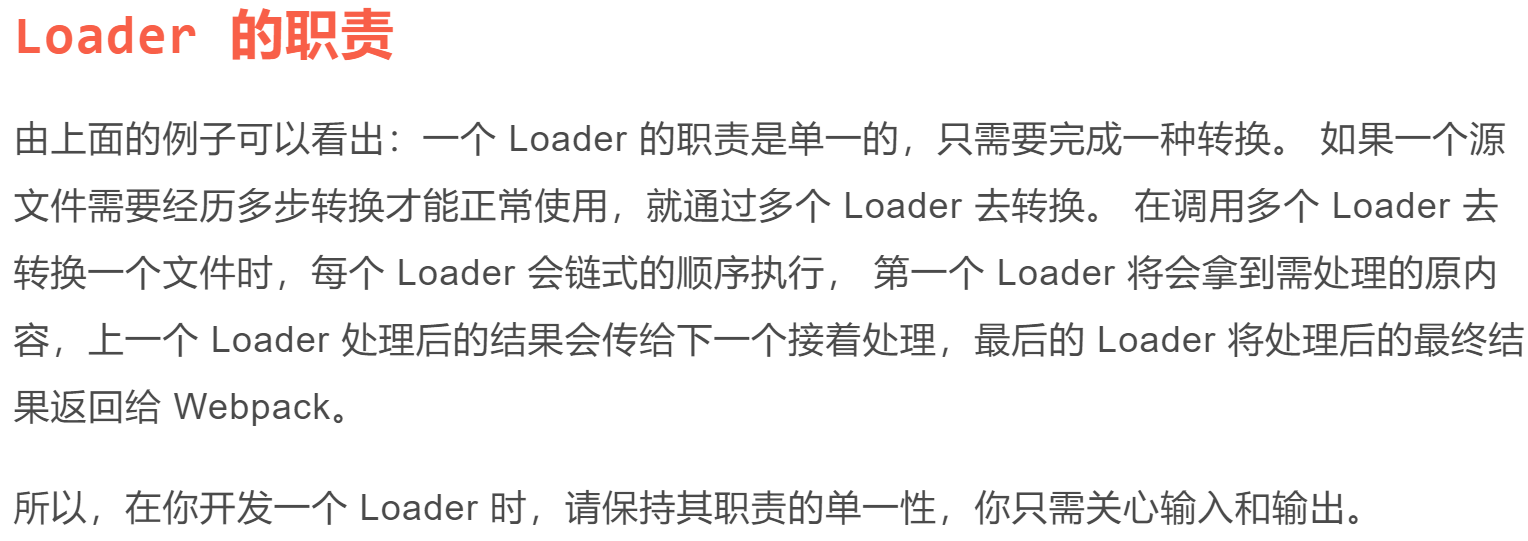
style-loader：把样式插入到DOM中，方法是在head中插入一个style标签，并把样式写入到这个标签的 innerHTML里，**通过JavaScript操作DOM 来添加样式。**（**动态创建 style 标签，将 css 插入到 head 中**，将所有的计算后的样式加入页面中）

postcss-loader：配合 autoprefixer 插件，自动生成浏览器兼容性前缀（自动补齐 CSS3 前缀）

file-loader：把图片和字体等本地资源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，大于阈值时返回其 publicPath，小于阈值时将文件转换为base64形式编码 (处理图片和字体)

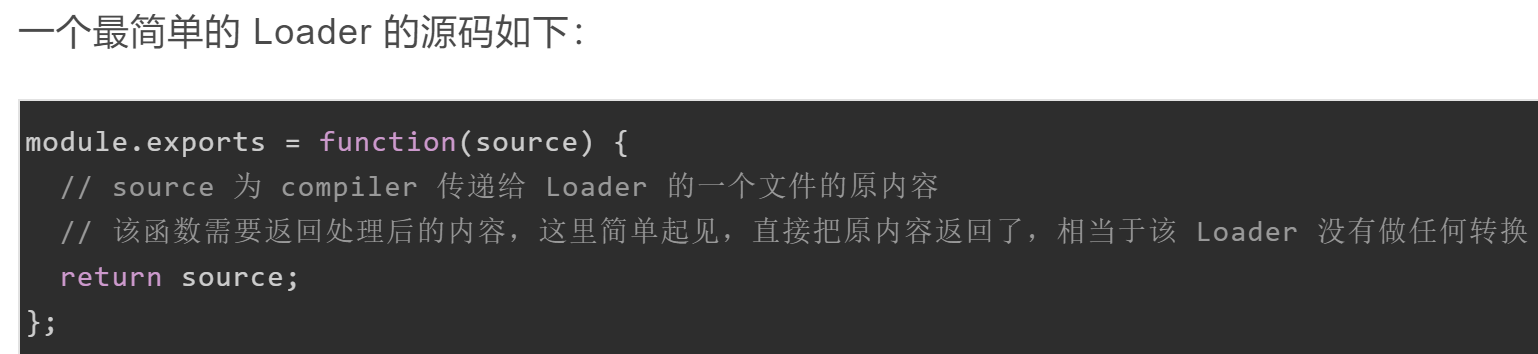




## 怎么编写/开发一个loader?简单描述一下编写loader的思路。

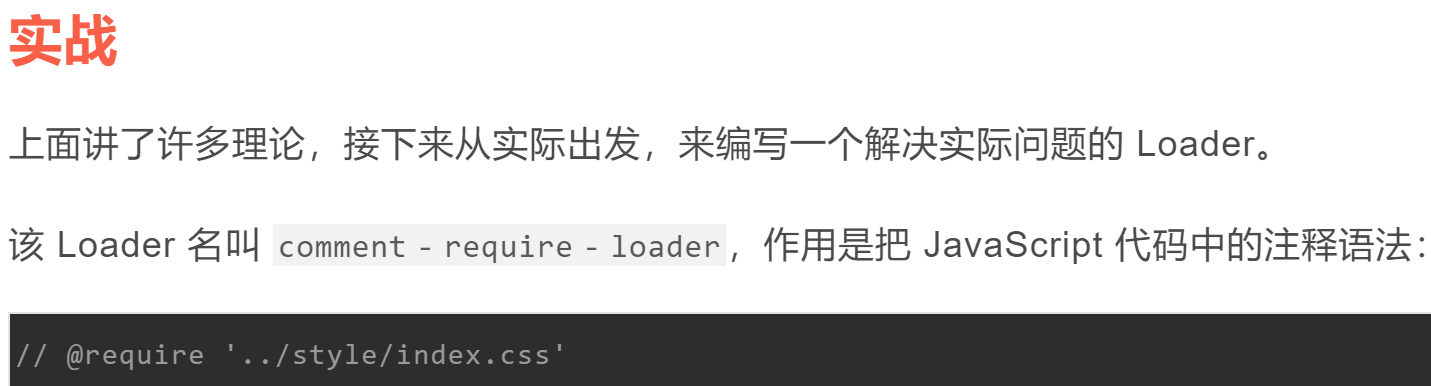
（由于 Webpack 是运行在 Node.js 之上的，）**一个 Loader 其实就是一个导出一个函数的Node.js模块，这个导出的函数就是用来转换源文件的，也就是获得处理前的原内容，对原内容执行处理后，返回处理后的内容。**

**Loader 支持链式调用，一个 Loader 的职责是单一的，只需要完成一种转换。**（如果一个源文件需要经历多步转换才能正常使用，就通过多个 Loader 去转换。在调用多个 Loader 去转换一个文件时，每个 Loader 会链式的顺序执行，第一个 Loader 将会拿到需处理的原内容，上一个 Loader 处理后的结果会传给下一个接着处理，最后的 Loader 将处理后的最终结果返回给Webpack。）



* 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
* **获得 Loader 的 options**：可以通过**loaderUtils**来获得给loader设置的options参数，使用 loader-utils 能够编译 loader 的配置，还可以通过 schema-utils 进行验证。
* **返回其它结果**：如果返回结果只有一个，也可以直接使用 return 返回结果。但是，如果有些情况下还需要返回其他内容，如sourceMap或是AST语法树，这个时候可以借助**webpack提供的api this.callback**
* **同步和异步**：尽可能的异步化 Loader，使用 this.async() 来获取 callback 函数。如果计算量很小，同步也可以。
* **处理二进制数据：**在默认的情况下，Webpack 传给 Loader 的原内容都是 UTF-8 格式编码的字符串。 但有些场景下 Loader 不是处理文本文件，而是处理二进制文件，例如 file-loader，就需要 Webpack 给 Loader 传入二进制格式的数据。此时需要通过设置**module.exports.raw = true**告诉 Webpack 该 Loader 需要二进制数据。
* **缓存加速：**在有些情况下，有些转换操作需要大量计算非常耗时，如果每次构建都重新执行重复的转换操作，构建将会变得非常缓慢。为此，Webpack 会**默认**缓存所有 Loader 的处理结果，也就是说在需要被处理的文件或者其依赖的文件没有发生变化时，是不会重新调用对应的 Loader 去执行转换操作的。直接跳过 rebuild 环节，节省不必要重建带来的开销。
* **加载本地loader**：**Npm link**：可以像使用一个真正的 Npm 模块一样使用本地的 Loader。**ResolveLoader**：**ResolveLoader 用于配置 Webpack 如何寻找 Loader。** 默认情况下只会去 node\_modules 目录下寻找，为了让 Webpack 加载放在本地项目中的 Loader 需要修改 **resolveLoader.modules**。假如本地的 Loader 在项目目录中的 ./loaders/loader-name 中，则需要如下配置：

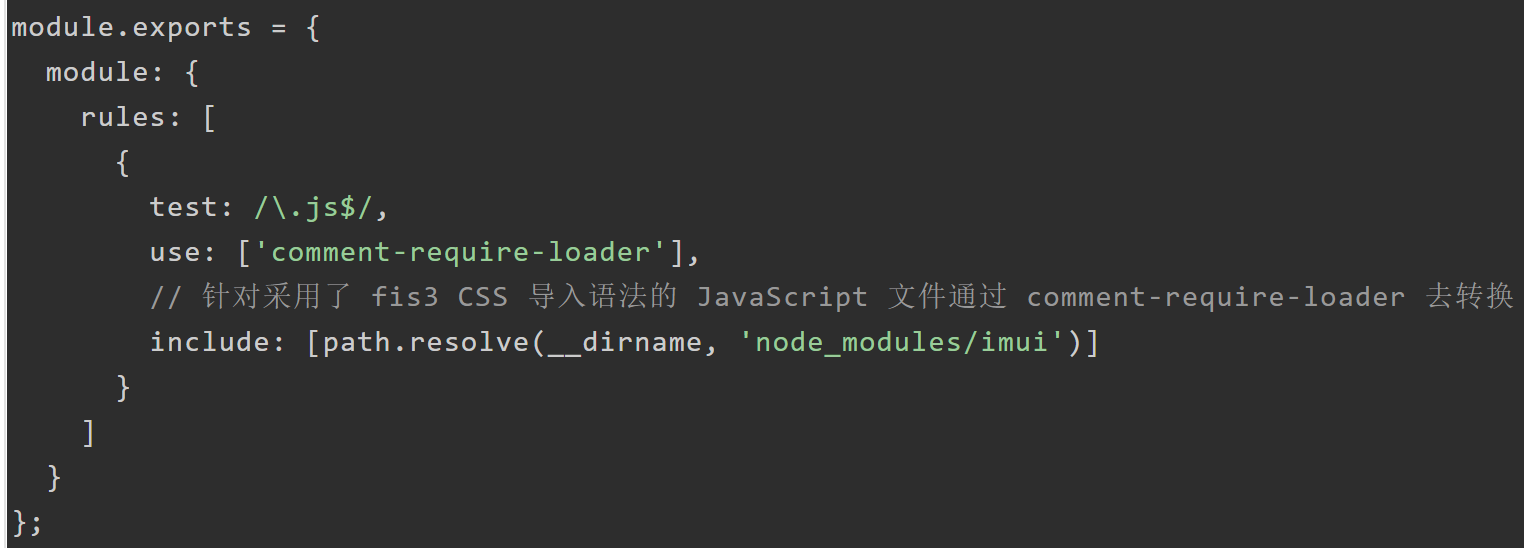






该 Loader 的使用场景是去正确加载针对 Fis3 编写的 JavaScript，这些 JavaScript 中存在通过注释的方式加载依赖的 CSS 文件。

该 Loader 的使用方法如下：





## plugin是什么？有哪些常见的plugin

plugin插件，是用来拓展webpack的功能的，在 Webpack 构建流程中的特定时机会广播出对应的事件，插件可以监听这些事件的发生，在特定时机做对应的处理。

html-webpack-plugin：**简化 HTML 文件创建** (依赖于 html-loader)。依据一个简单的html模板，自动生成一个引用你打包后的JS文件的新的html。

clean-webpack-plugin: 清理目录。

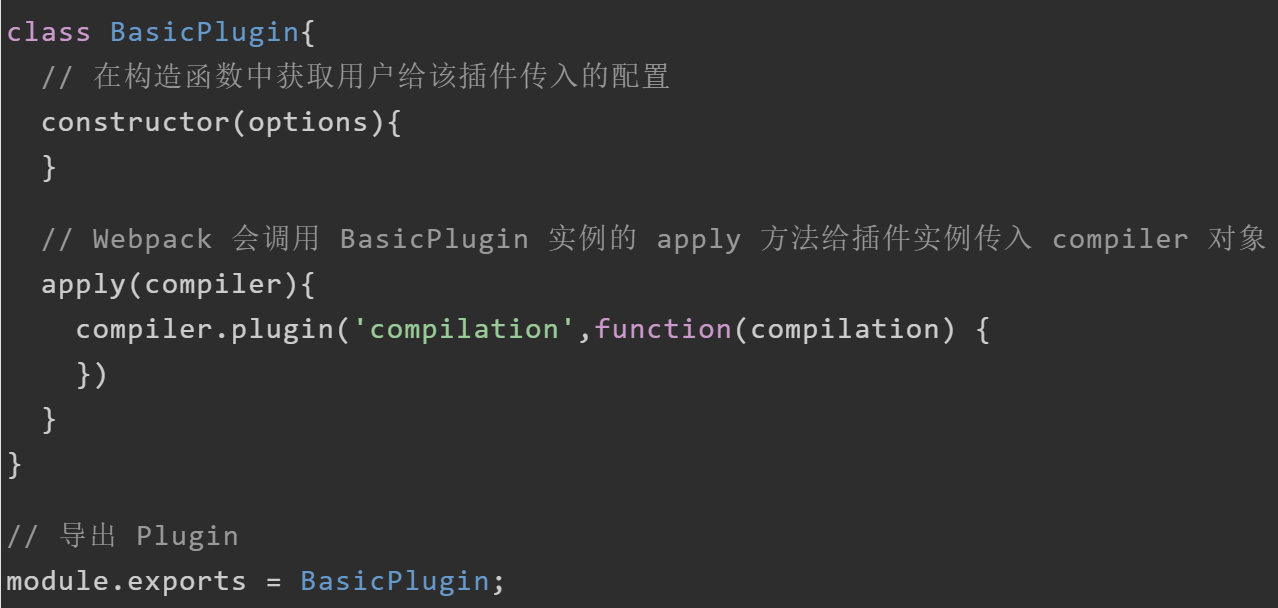
copy-webpack-plugin：静态资源拷贝。（将本地已有的js,css文件拷贝至打包的输出目录中）

mini-css-extract-plugin：抽离样式文件，将CSS文件单独打包为独立文件，支持按需加载。

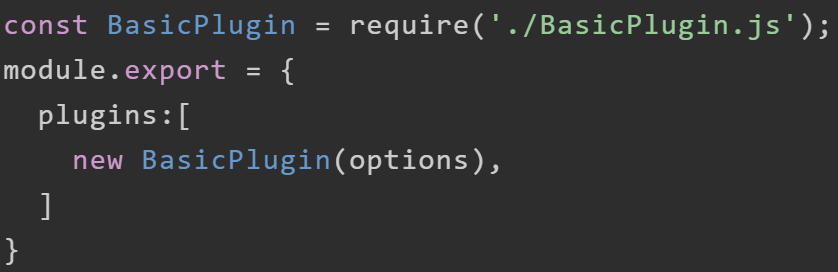
## 编写plugin？

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。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。

一个最基础的 Plugin 的代码是这样的：

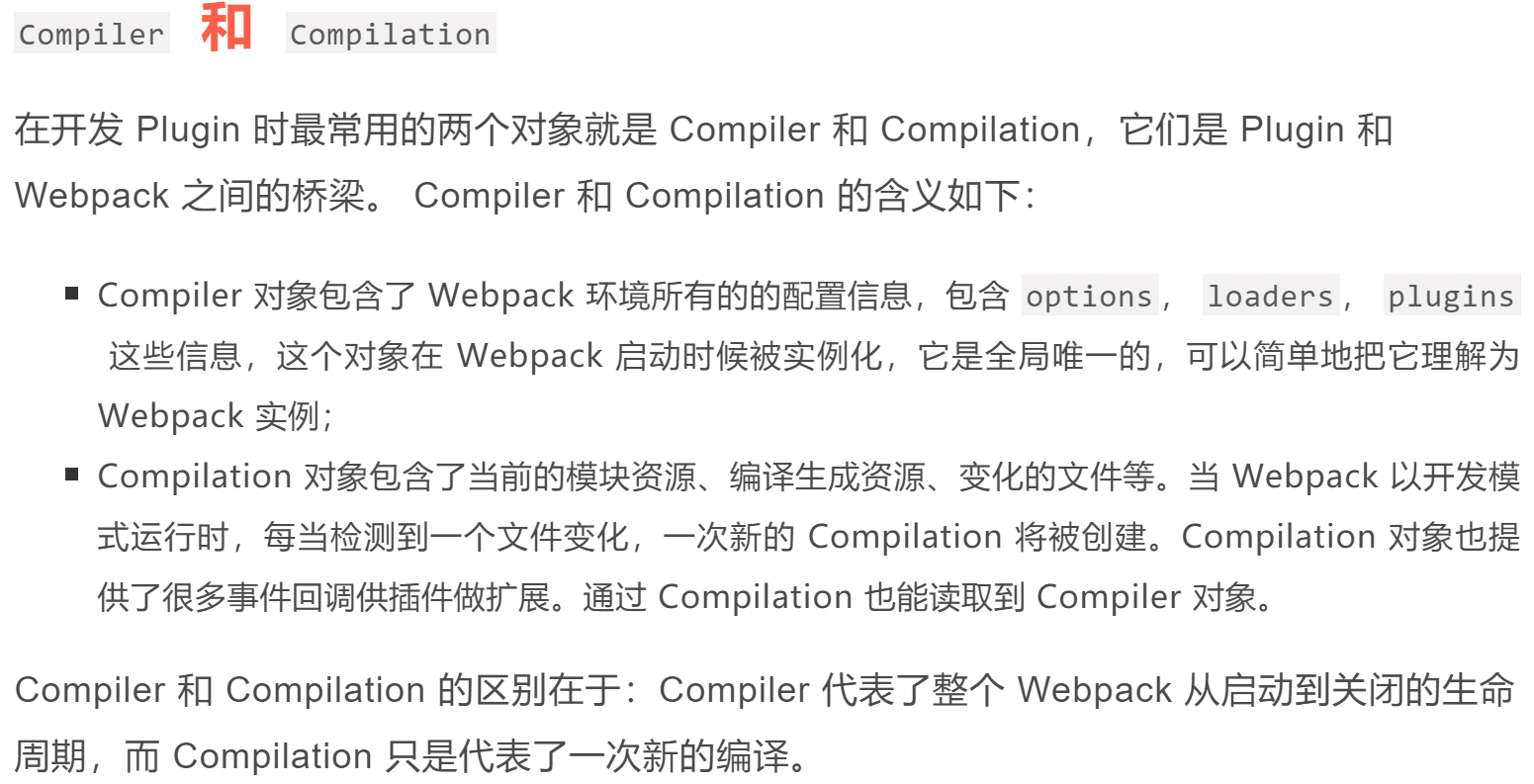


在使用这个 Plugin 时，相关配置代码如下：



Webpack 启动后，在读取配置的过程中会先执行new BasicPlugin(options)实例化一个插件BasicPlugin。 在初始化 compiler 对象后，再调用插件实例的apply方法给插件实例传入 compiler 对象(basicPlugin.apply(compiler))。 插件实例在获取到 compiler 对象后，就可以通过 compiler.plugin(事件名称,回调函数) 监听到 Webpack 广播出来的事件。 并且可以通过compiler对象去操作 Webpack。

需要注意的细节：



补充：compilation对象，它继承于compiler，所以能拿到一切compiler的内容

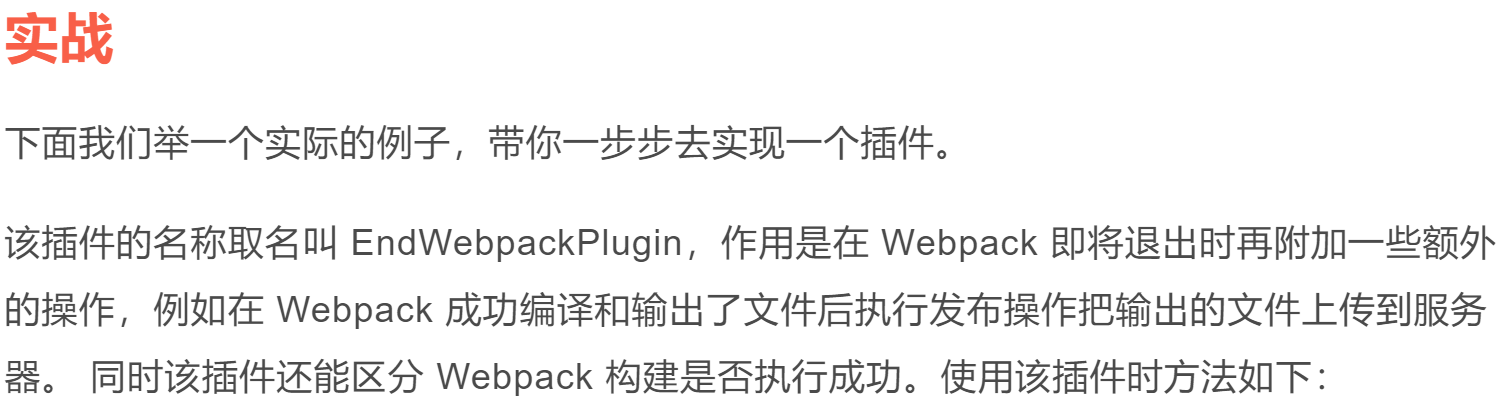


Webpack 的事件流机制应用了观察者模式，和 Node.js 中的 EventEmitter 非常相似。**Compiler 和 Compilation 都继承自 Tapable，可以直接在 Compiler 和 Compilation 对象上广播和监听事件**，方法如下：

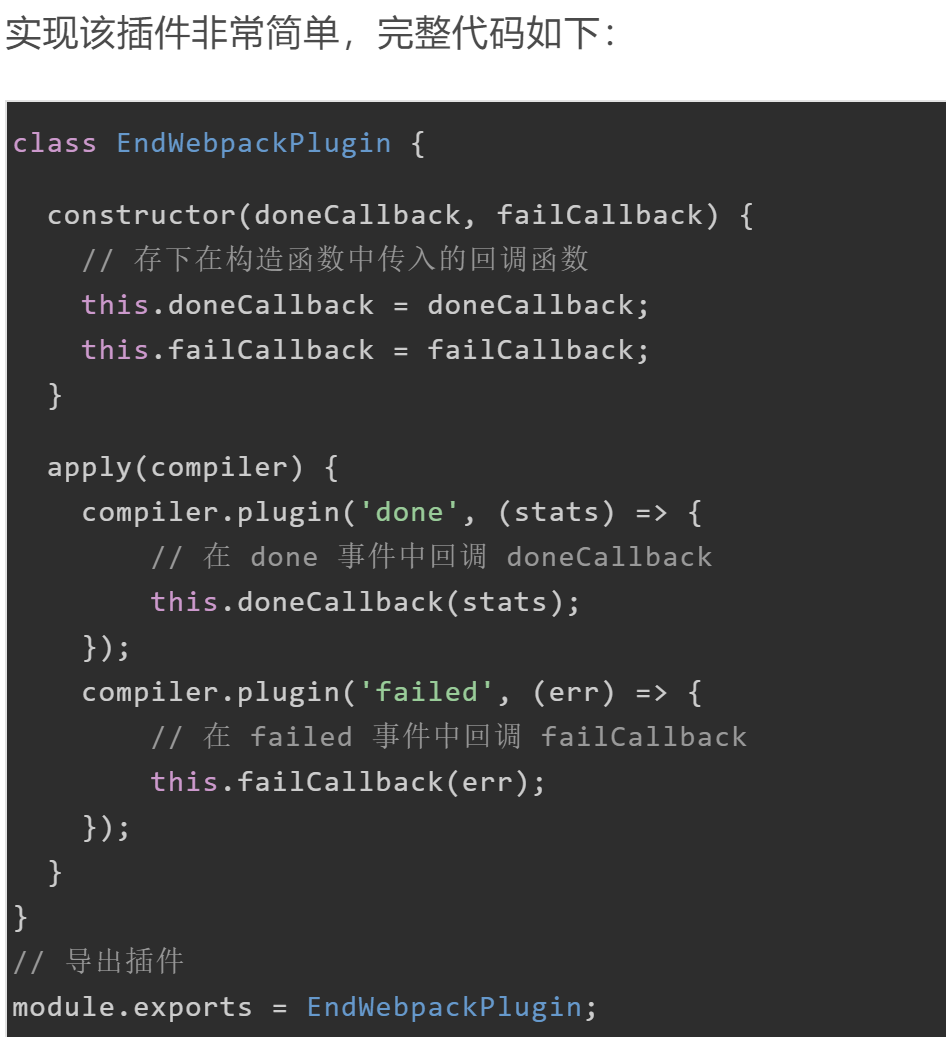




即异步的事件需要在插件处理完任务时调用回调函数通知 Webpack 进入下一个流程，不然会卡住。







## chunk是什么？

代码块，**一个 Chunk 由多个模块组合而成**，是由webpack提供的代码分割形成的，用于实现按需加载。（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）

## loader和plugin的区别？（必问）

loader是在打包构建过程中用来处理源文件的（TypeScript,JSX，Sass，Less..），一次处理一个。因为webpack本身只能打包CommonJS规范的js文件，对于其他类型资源，例如css，图片等，是没有办法加载的，这就需要对应的loader充当翻译官，**对这些资源进行转译的预处理工作**。loader只专注于转换源文件，处理单一文件的输入输出

plugin用于扩展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，在特定的时机执行相应的任务（通过 Webpack 提供的 API ）改变打包的输出结果。**plugin并不直接操作单个文件，而对整个构建过程起作用**。比如mini-css-extract-plugin，可以将所有文件中的css抽离到一个独立的文件中。（这样样式就不会随着组件加载而加载了。）

## Webpack构建流程

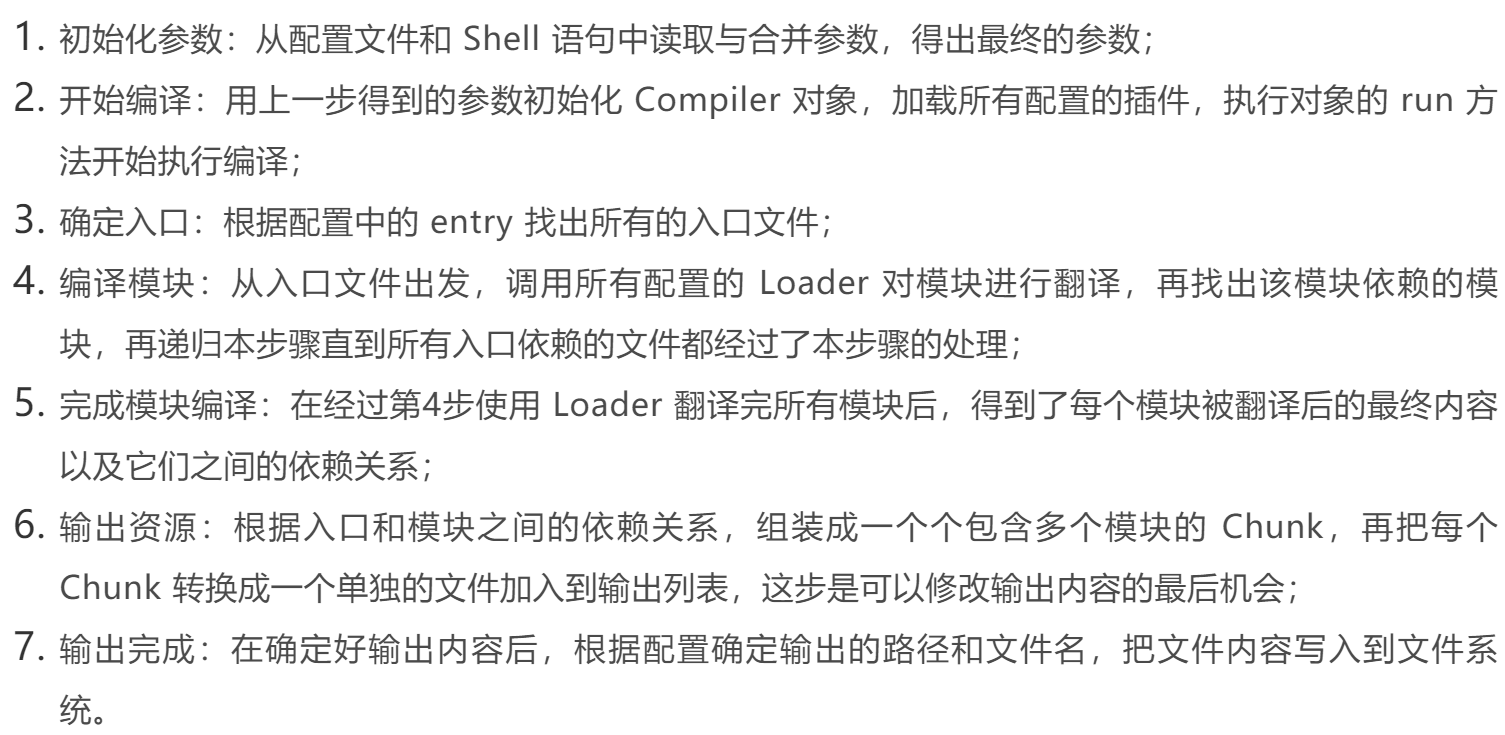
Webpack的运行流程是一个**串行**的过程，从启动到结束可以概括为以下三步：

* + **初始化**：**启动构建**，读取与合并配置参数，加载插件，实例化 Compiler
  + **编译**：从入口文件Entry 出发，针对每个模块串行**调用**对应的 Loader翻译文件的内容，再找到该模块依赖的模块，**递归地进行编译处理**。
  + **输出**：将**编译后的模块组合**成 Chunk，再把每个Chunk 转换成单独的文件添加到输出目录中。

在以上过程中，Webpack 会在特定的时间点广播出特定的事件，**插件在监听到感兴趣的事件后会执行特定的逻辑**，并且插件可以调用 Webpack 提供的 API **改变 Webpack 的运行结果**。

在输出阶段已经得到了各个模块经过转换后的结果和其依赖关系，并且把**相关模块组合在一起形成一个个 Chunk**。 在输出阶段会根据 Chunk 的类型，使用对应的模版生成最终要要输出的文件内容。

具体：



## webpack打包原理

**webpack打包，最基本的实现方式，是将所有的模块代码放到一个数组里，通过数组索引来引用不同的模块。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。**



可以发现**入口entry.js的代码是放在数组索引0的位置**，其它a.js和b.js的代码分别放在了数组索引1和2的位置，而**webpack引用的时候，主要通过\_\_webpack\_require\_\_的方法引用不同索引的模块**。

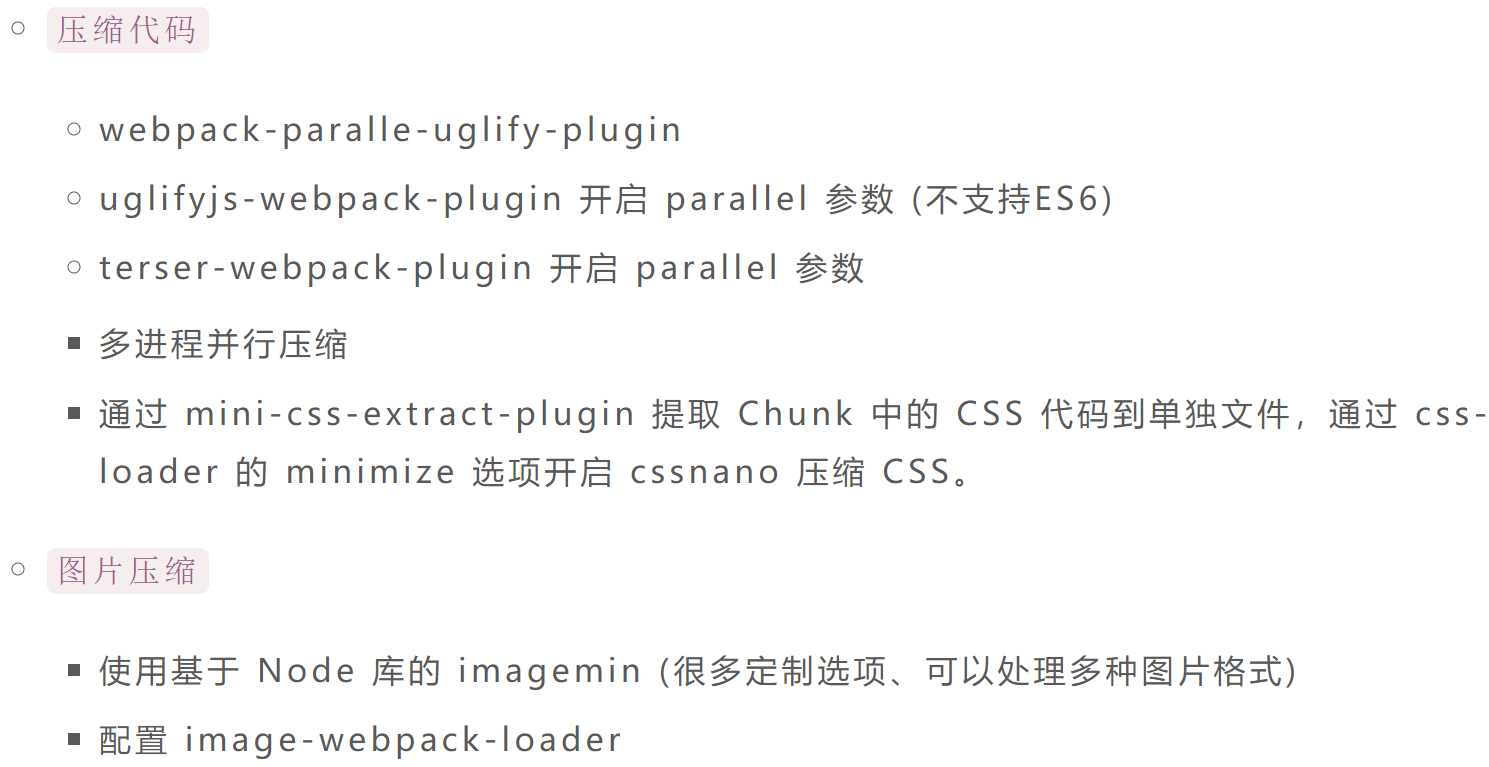
拓展：

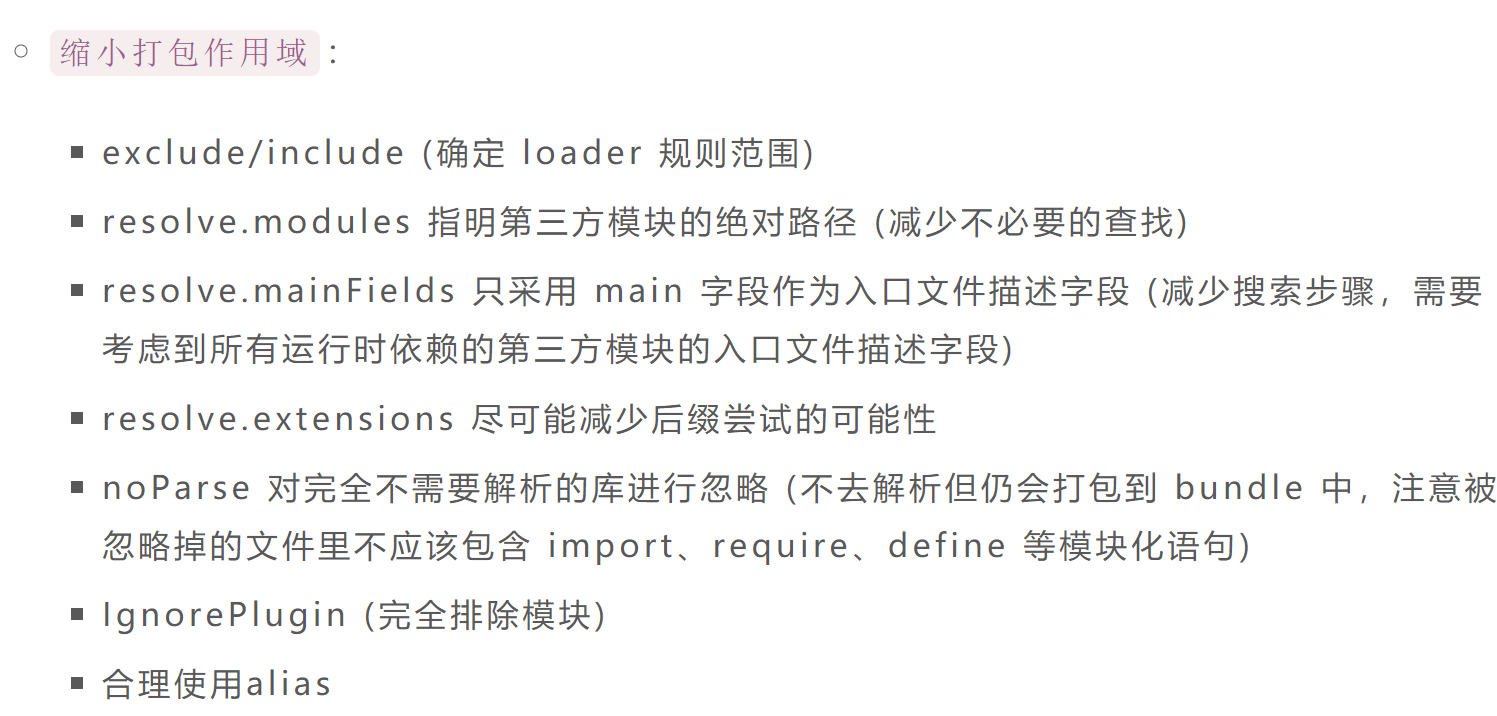


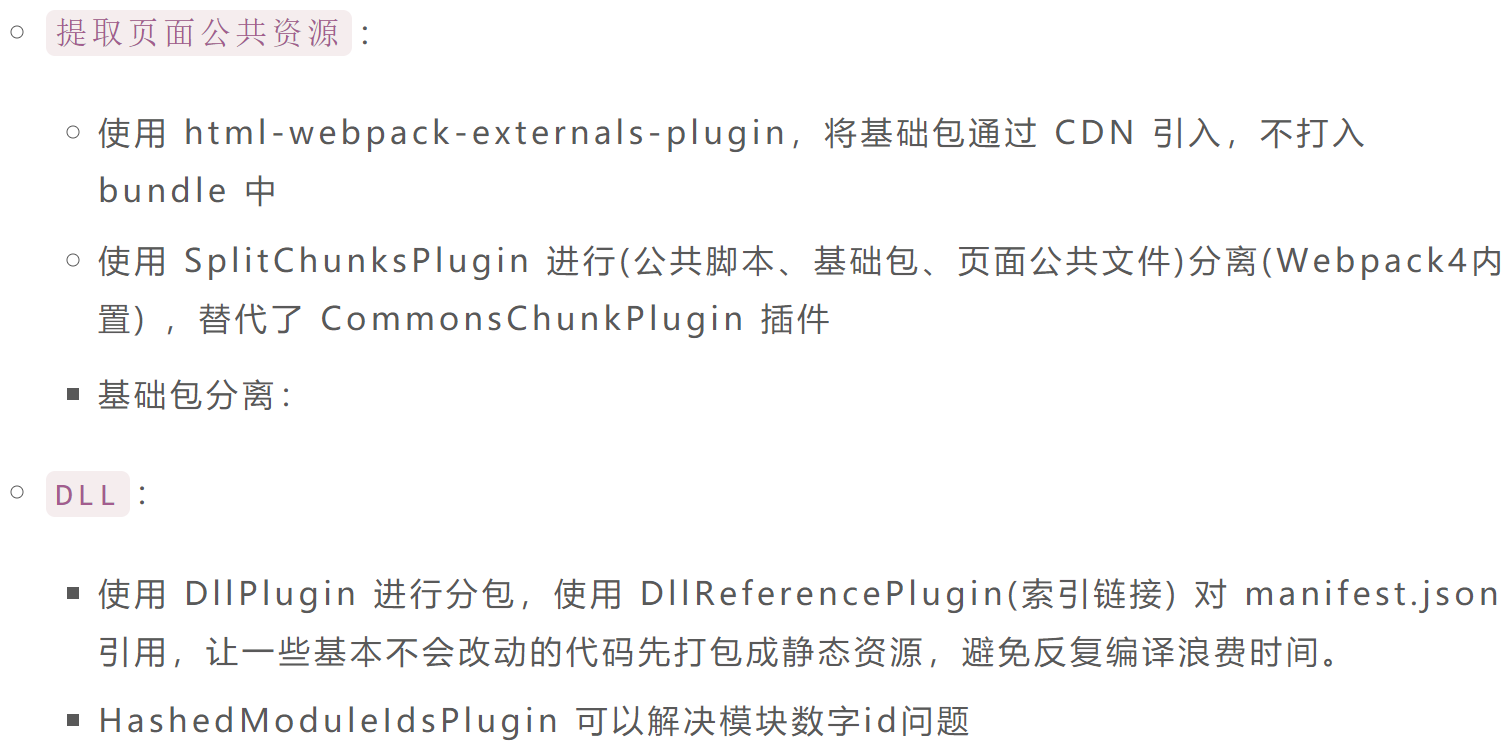
## webpack打包后文件体积过大怎么办？

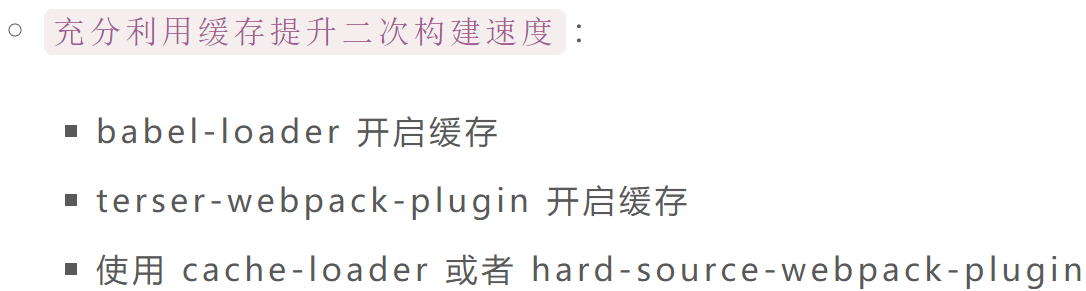
* 代码分割，异步（按需）加载模块；
* 代码压缩；
* 提取第三方库（使用cdn或者vender）；
* 去除不必要的插件；
* 去除devtool选项；
* dllplugin等等。

## 如何优化 Webpack 的构建速度？









## webapck热更新原理？

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去更新后的文件，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。

具体参考：<https://zhuanlan.zhihu.com/p/30669007>

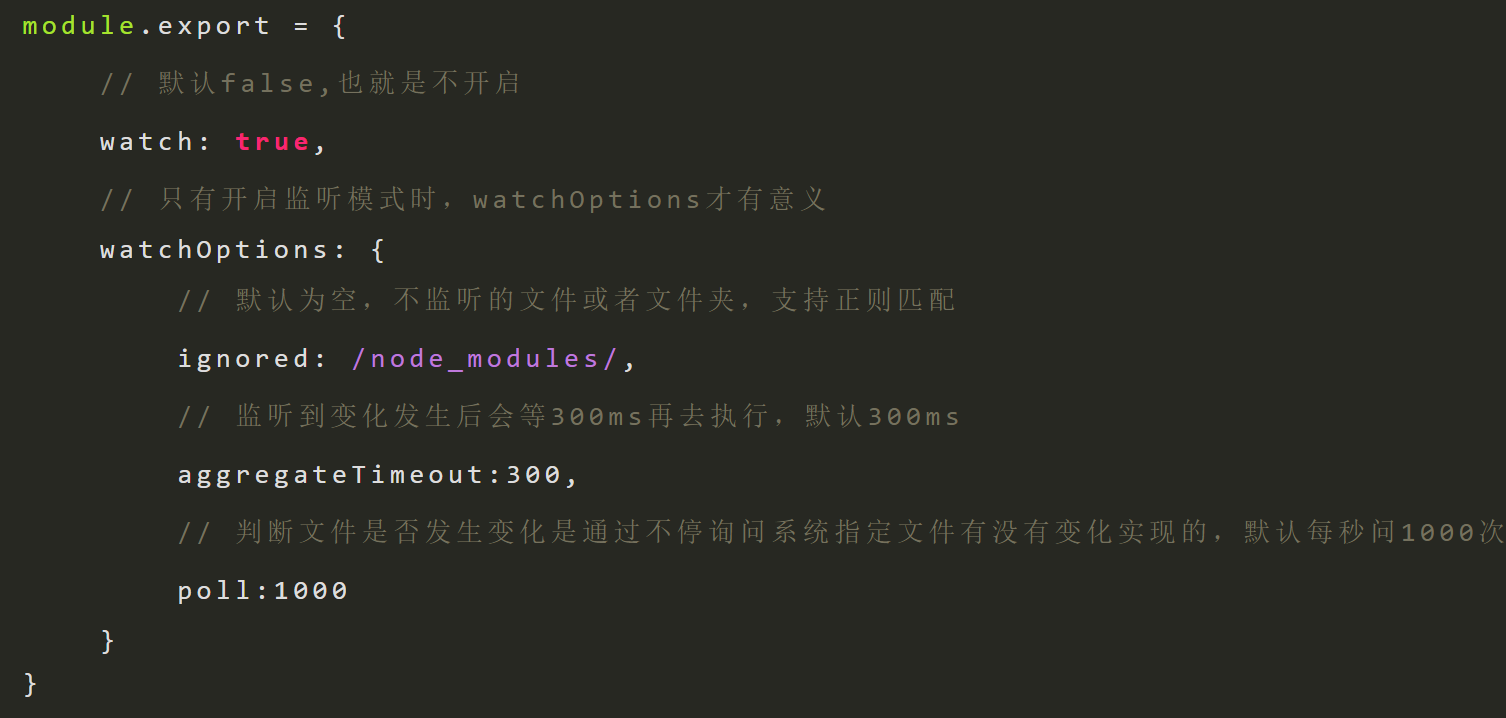
## source map是什么？生产环境怎么用？

**source map 是将编译、打包、压缩后的代码映射回源代码的过程。打包压缩后的代码不具备良好的可读性，想要调试源码就需要 soucre map。**

避免在生产中使用 inline- 和 eval-，因为它们会**增加 bundle 体积大小**，并降低整体性能。

## 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，并不会立刻告诉监听者，而是先缓存起来，等 aggregateTimeout 后再执行。



## 占位