# 同源策略和跨域

## 同源策略是什么以及为什么要有同源策略：

同源策略（same-origin policy）是浏览器的一种安全策略，它可以防止恶意网站窃取用户信息，保证用户信息安全。所谓同源指的是两个url的**协议**、**域名**和**端口**三者完全相同。**同源策略只允许与本域下的接口交互**，不同源的客户端脚本在没有明确授权的情况下，不能读写对方的资源。如果缺少了同源策略，浏览器很容易受到XSS、CSRF等攻击（回答为什么要有同源策略）。

同源策略限制了以下内容/行为：

* Cookie、LocalStorage、IndexedDB 等存储性内容无法读取
* DOM 节点和JS对象无法获取
* AJAX 请求发送后，结果被浏览器拦截了

但是有三个HTML标签允许跨域加载资源：

* <img src=XXX>
* <link href=XXX>
* <script src=XXX>

## 跨域是什么以及为什么要跨域/会出现跨域：

（首先协议（http）、域名（www.sina.com）、端口（80）三者中有一个不同就叫不同的域。）由于**同源策略**的限制，**当我们从一个域名的网页去请求另一个域名的资源时**，就无法成功获取资源。如果我们要想成功获取资源，那么就要用到跨域。跨域**就是不同的域间相互访问时使用某些方法来突破同源策略带来的限制**。

PS: 协议或者端口的不同，只能通过后台来解决。

## 跨域有哪些实现方法？

**同源策略限制下接口请求的正确打开方式**

### JSONP跨域原理及手写实现

<https://www.cnblogs.com/chiangchou/p/jsonp.html>

JSONP 是 JSON with padding（填充式 JSON 或参数式 JSON）的简写。

JSONP实现跨域请求的原理，简单来说，**就是动态创建<script>标签，然后利用<script>标签的src属性不受同源策略的限制，可以获取*任意服务器*上的脚本并执行这个特点，来跨域获取数据**。

**实现流程：**

* **声明一个回调函数callback**，其**函数名**(如show)会被作为参数值传递给跨域请求的服务器，函数形参为要获取的目标数据(服务器返回的data)。
* **创建一个<script>标签**，把跨域请求的url赋值给script的src,还要在这个地址中向服务器传递刚刚声明的回调函数名（可以通过**问号**传参:?callback=show）。
* 服务器接收到请求后，**解析客户端传过来的 callback 方法，返回该方法的调用，并且把数据作为参数传入该方法**（传递进来的函数名和它需要给你的数据拼接成一个字符串,例如：传递进去的函数名是show，它准备好的数据是show('我不爱你')。）
* **客户再执行服务端返回的方法调用，即调用之前声明的回调函数callback（show），对返回的数据进行操作。**

**源码实现：**





服务端代码：



jsonp优缺点：

优点：简单兼容性好，可用于解决主流浏览器的跨域数据访问的问题。

缺点：

**仅支持get请求（本质上script加载资源就是GET），并且要求返回JavaScript。具有局限性；**

**不安全，可能会遭受XSS攻击。**

### CORS

**CORS全称是跨域资源共享(Cross-Origin Resource Sharing)，是HTML5规范定义的如何跨域访问资源。**（CORS 需要浏览器和服务端同时支持。IE 8 和 9 需要通过 XDomainRequest 来实现。）

**实现 CORS 通信的关键是服务端**，服务端需要设置 **Access-Control-Allow-Origin** 属性**来开启** CORS。 该属性表示哪些域**名**可以访问该服务端的资源，如果设置为通配符\*则表示所有网站都可以访问资源。

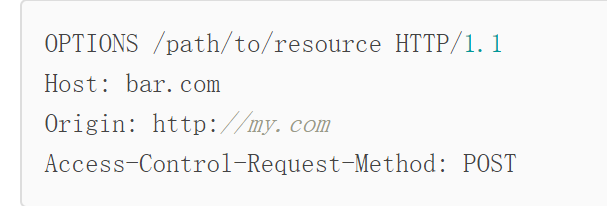
通过CORS解决跨域问题，会在发送请求时出现两种情况，分别为**简单请求**和**非简单/复杂请求**。

**简单请求：**

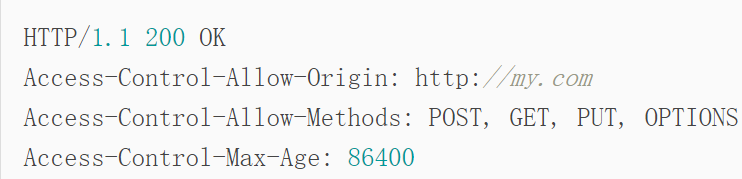


非简单请求：

对于PUT、DELETE以及其他类型如Content-Type为application/json的POST请求，**在发送**AJAX**请求之前**，**浏览器会先发送一个OPTIONS请求**（称为预检测请求），询问目标服务器是否接受跨域请求。



服务器必须响应并明确指出允许的Method：

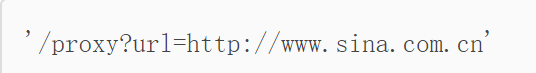


**浏览器确认服务器响应的Access-Control-Allow-Methods头包含将要发送的**AJAX**请求的Method**，才会继续发送AJAX，否则，抛出一个错误。

由于以POST、PUT方式传送JSON格式的数据在REST中很常见，所以要跨域正确处理POST和PUT请求，服务器端必须正确响应OPTIONS请求。

### nginx反向代理

实现原理：**同源策略是浏览器需要遵循的标准，而如果是服务器向服务器请求就无需遵循同源策略**。可以在**同源**域名下架设/搭建一个代理服务器，浏览器负责把跨域请求发送到代理服务器，由代理将跨域请求转发给目标服务器：



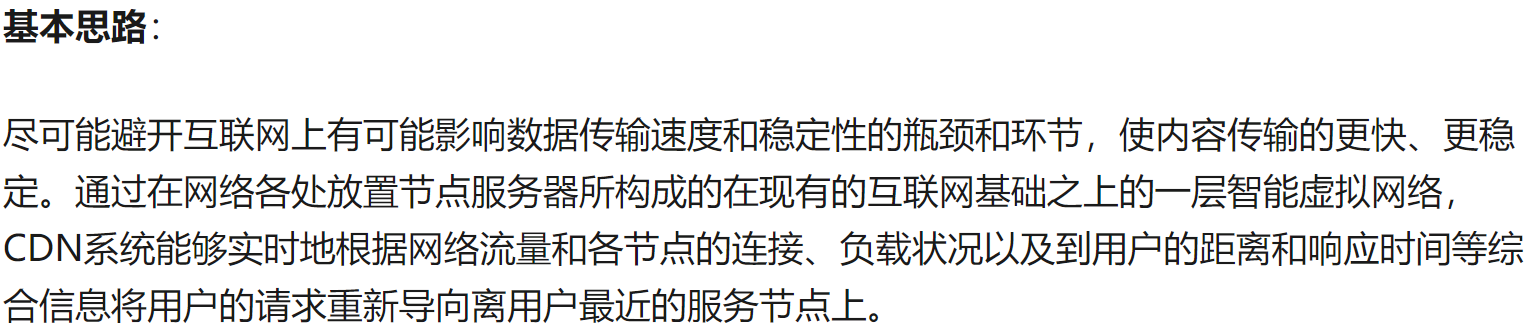
然后代理服务器拿到目标服务器响应的数据，并将响应转发给客户端，这样就遵守了浏览器的同源策略。

通过这种方式我们**只需要修改nginx的配置**即可解决跨域问题，但它的麻烦之处在于**需要服务器端额外做开发**。

# CDN相关

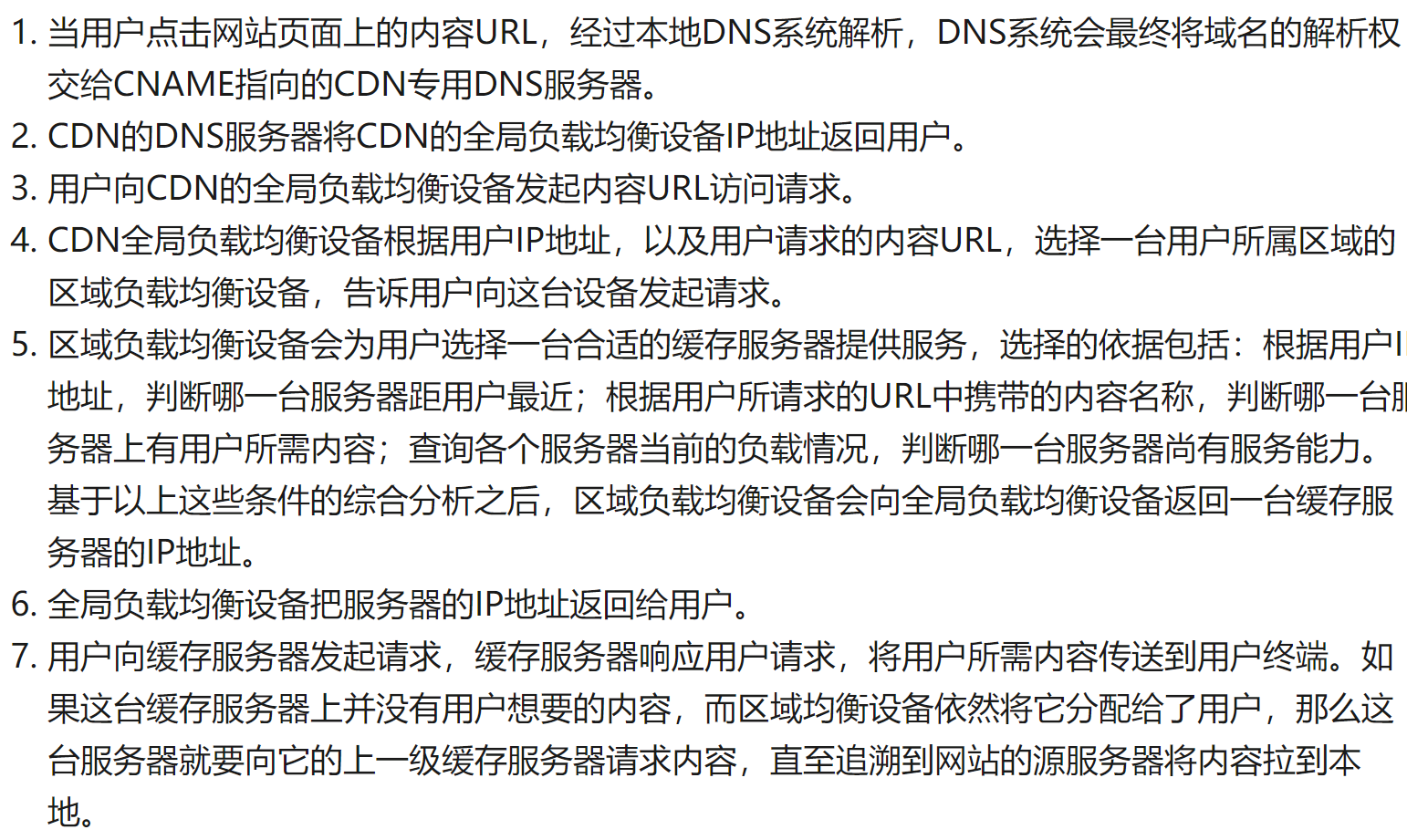
## CDN的概念及作用

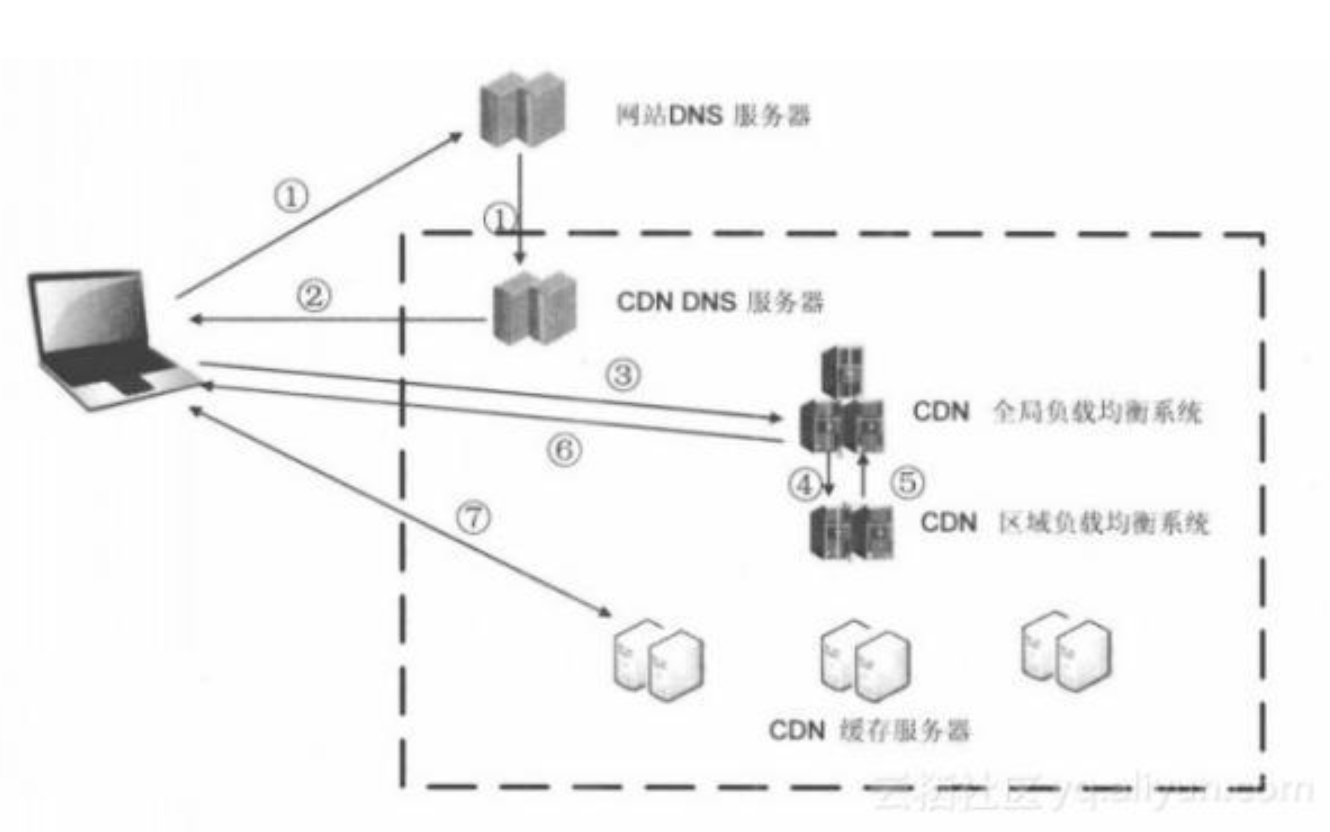
CDN全称:**Content Delivery Network**或Content Distribute Network，即**内容分发网络**。CDN作用：**将源站内容分发到离用户最近的节点上，使用户可以就近取得所需内容，提高用户访问网站的响应速度和成功率。**（解决因分布、带宽、服务器性能带来的访问延迟问题，适用于站点加速、点播、直播等场景。）



## CDN工作原理/流程

**基础架构**：最简单的CDN网络由**一个DNS服务器**和**几台缓存服务器**组成：





简单来说就是，**通过dns服务器来实现优质节点的选择，通过缓存来减少源站的压力**。

# Webpack相关

## 0 关于开发环境(development)和生产环境(production)

首先这两个环境下都必然要经过打包构建这一步的，但这两个环境下的构建目标存在着巨大差异

* **开发环境**：需要**完整**的 **source map** 方便调试，还需要**一个有**着 **live reloading(实时重新加载，**即监听开发时代码的修改，并让浏览器自动刷新显示修改后的结果) 或 **hot module replacement(热模块替换)** 能力的 **localhost server。**总的来说就是为了方便开发调试。

**关于webpack-dev-server(自动编译打包(？？)，页面自动刷新（socket）):**

**监听源文件的改变并实时编译、打包（还不是很懂），然后通过websocket通信通知浏览器进行页面刷新**



**通过建立websocket实现服务端和客户端的双向通讯**，当**我们的服务端发生变化时可以通知客户端进行页面的刷新。**

实现的方式主要有两种iframe mode和inline mode

* **生产环境**：关注点在于压缩 bundle、更轻量的 source map、资源优化等，通过这些优化方式**改善加载时间**。

## 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**。）

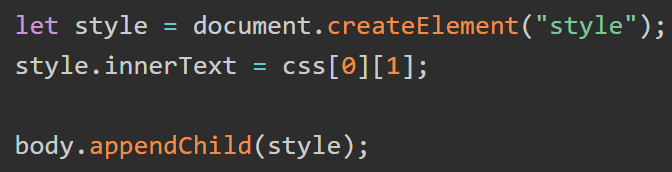
babel-loader：把 ES6 转换成 ES5

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

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）

style-loader：**动态创建 style 标签**，通过DOM操作去加载CSS，将css-loader解析后的内容挂载到html页面当中（**将 css 插入到 head 中**，将所有计算后的样式加入页面中）

即：

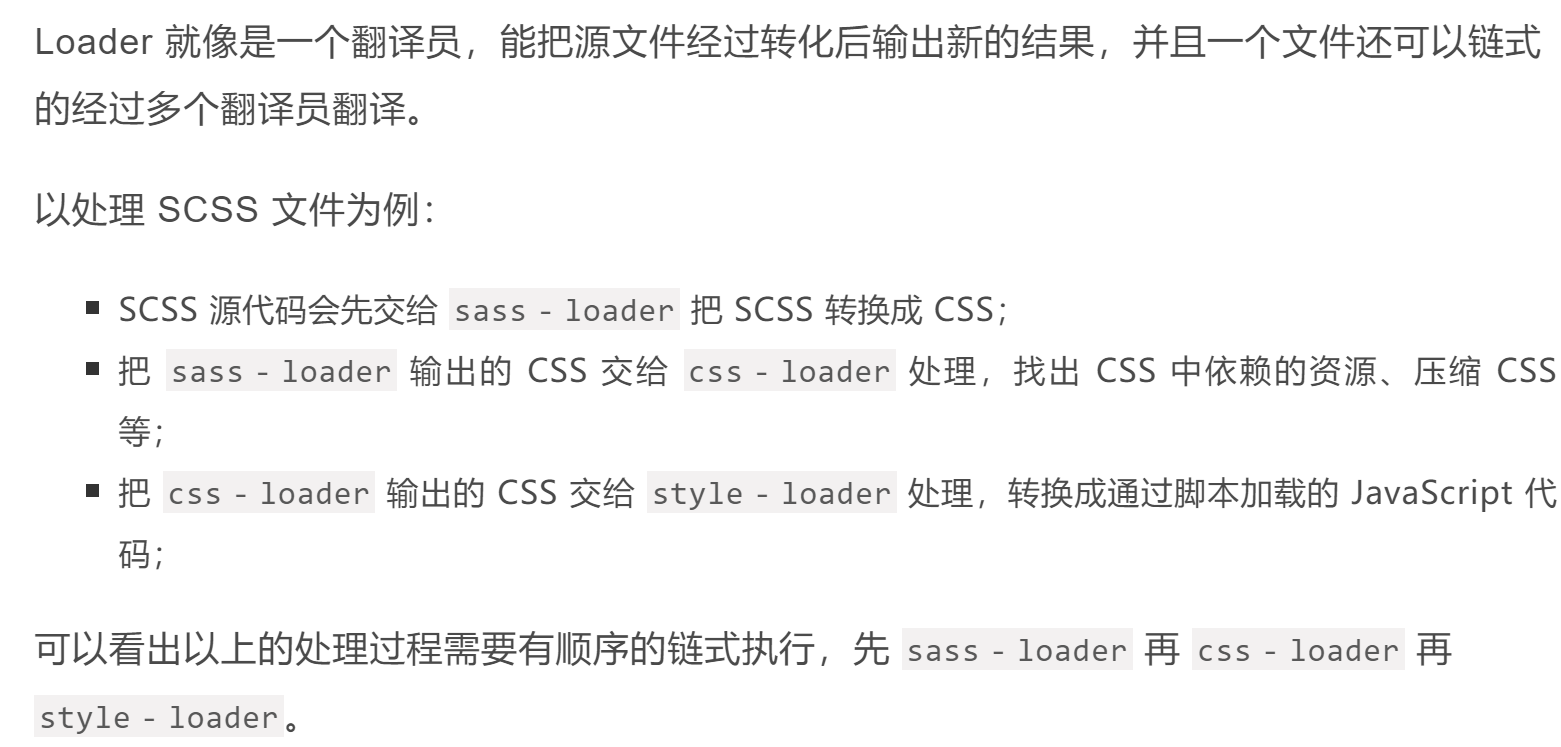


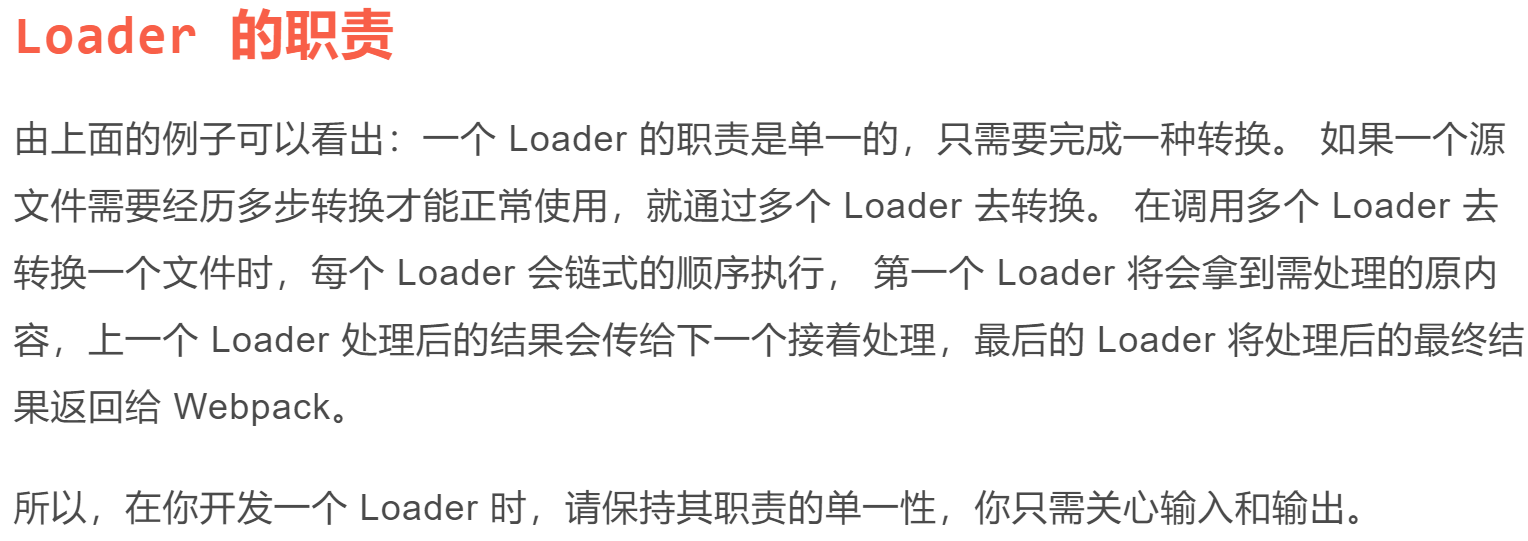
PS:关于css-loader和style-loader的区别见：

<https://www.jianshu.com/p/d2470f719fee>

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 来引用输出的文件 (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Node.js模块，这个模块导出一个函数，这个函数就是用来转换源文件的，也就是获得需要转换的原内容，对原内容执行转换后，返回转换后的内容。在转换过程中，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。）**

**另外，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。）



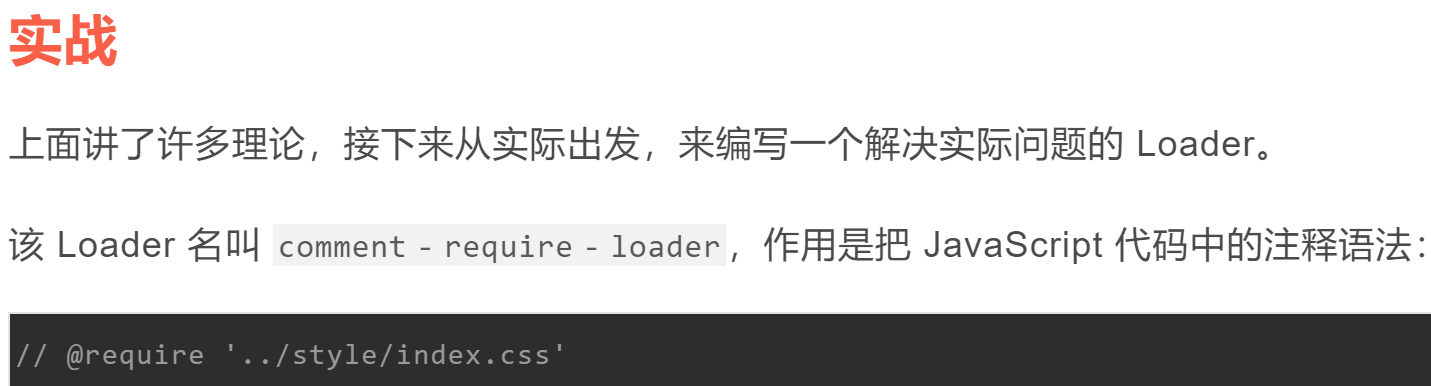
* **获得 Loader 的 options：可以通过loaderUtils来获得给loader设置的options参数**，使用 loader-utils 能够编译 loader 的配置，还可以通过 schema-utils 进行验证。
* **返回其它结果**：如果返回结果只有一个，也可以直接使用 return 返回结果。但是，如果有些情况**下还需要返回其他内容，如sourceMap**或是AST语法树，这个时候可以借助**webpack提供的api this.callback**
* **同步和异步**：如果 Loader的转换是异步的**，需要使用 this.async() 来获取 callback 函数，通过callback返回异步执行后的结果**。
* **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处理的是二进制文件，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：



也有说是resolve的：resolve.modules 配置 webpack 去哪些目录下寻找第三方模块



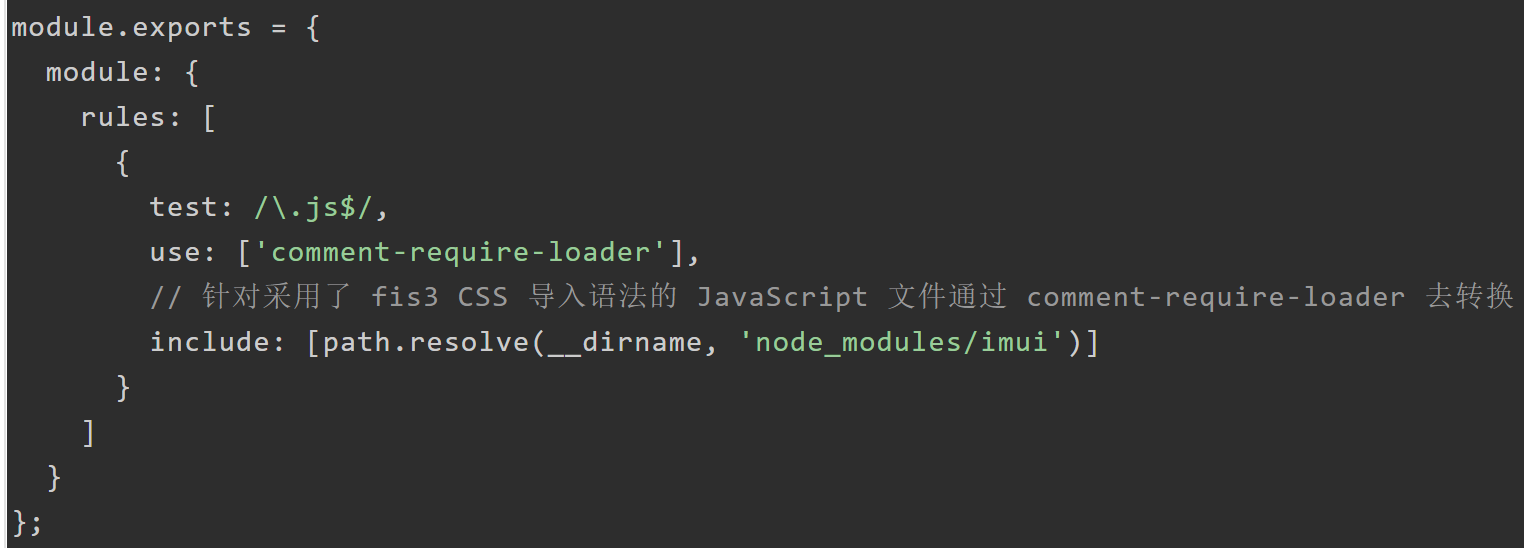






该 Loader 的使用场景是去正确加载针对 Fis3 编写的 JavaScript，这些 JavaScript 中存在通过注释的方式加载依赖的 CSS 文件。

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，在特定时机做出对应的处理。

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。

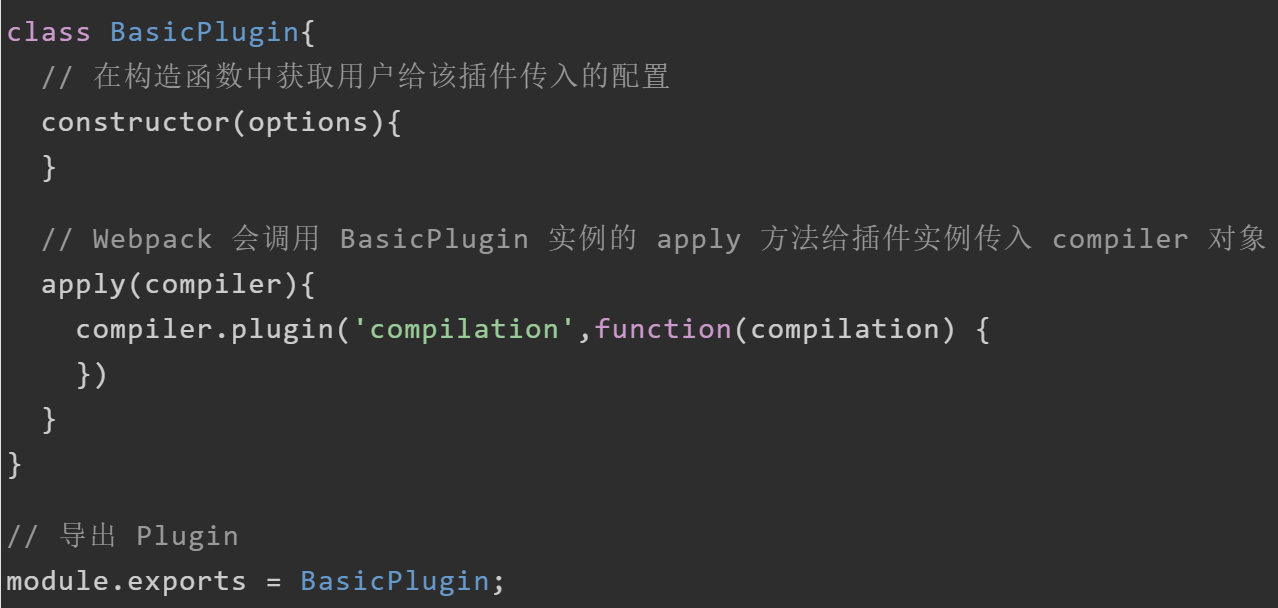
copy-webpack-plugin：静态资源拷贝。（将本地已有的不需要webpack编译的js,css文件拷贝至打包的输出目录中）

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。）

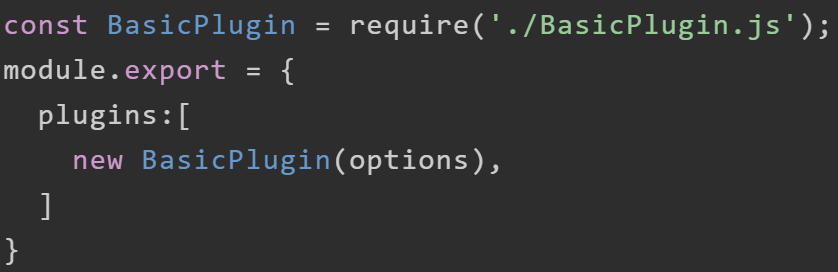
## 编写plugin？

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的打包结果。）

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

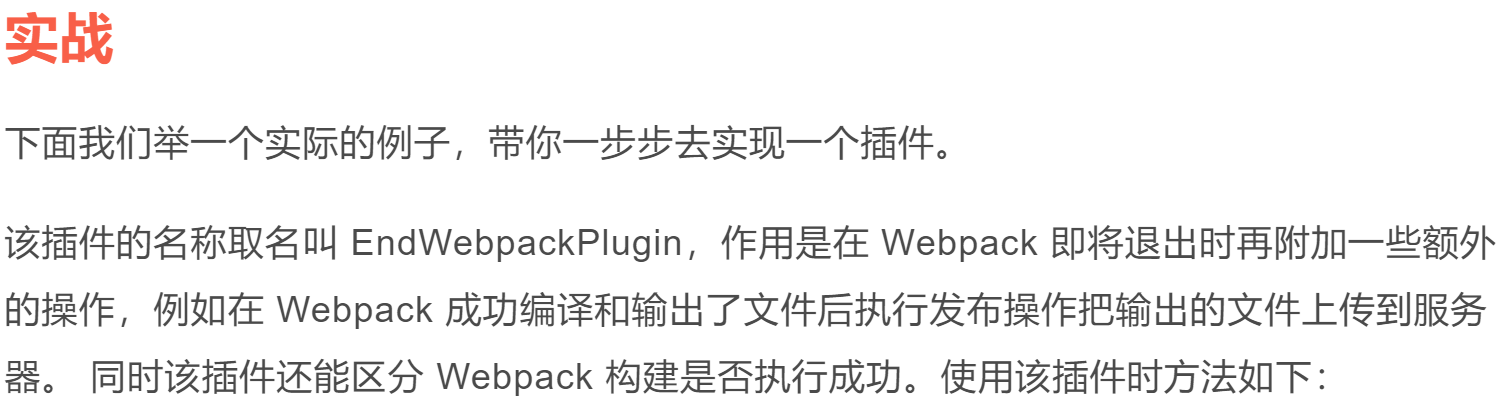


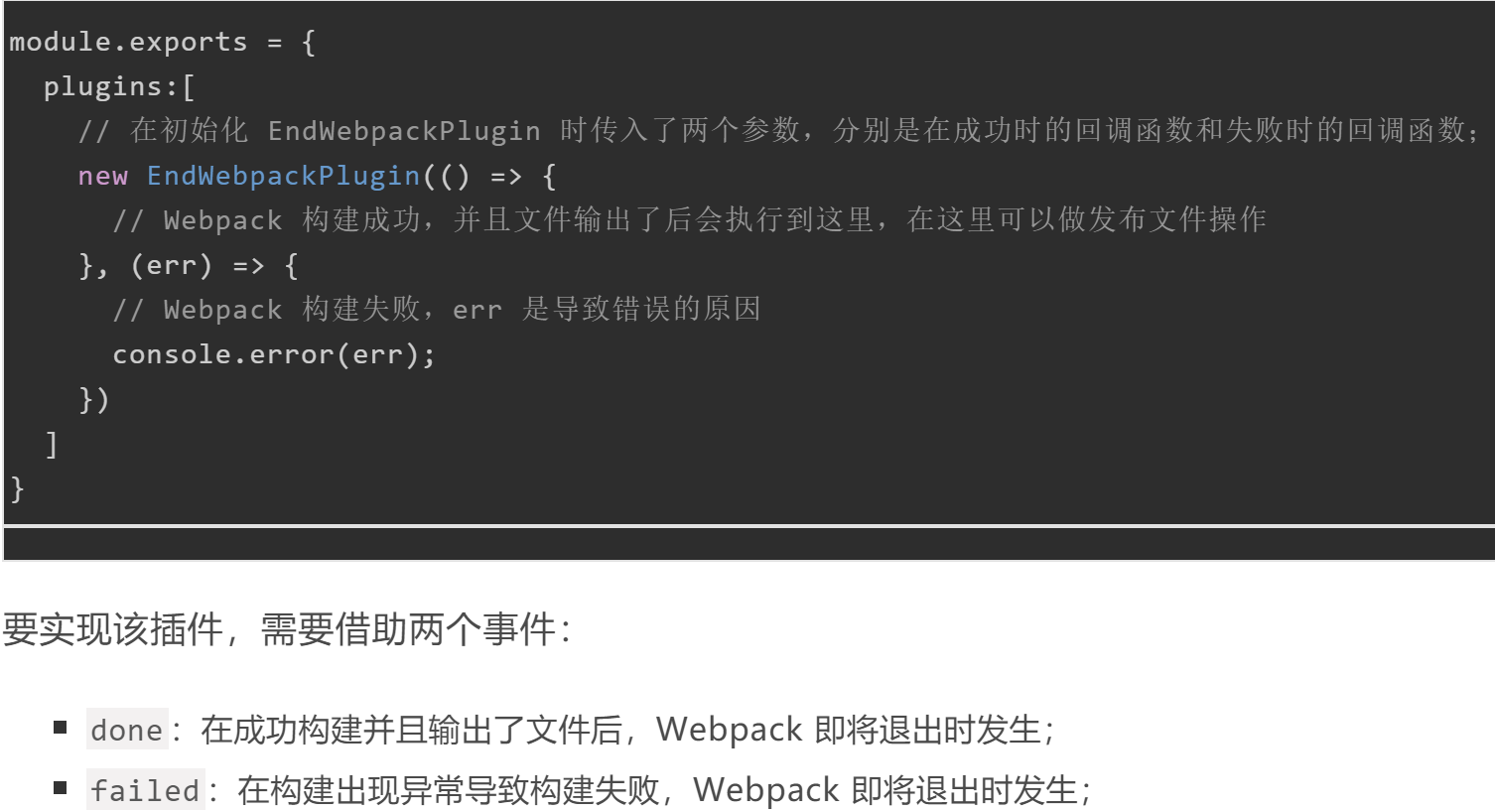
Webpack 的事件流机制应用了观察者模式，和 Node.js 中的 EventEmitter 非常相似。**Compiler 和 Compilation 都继承自Tapable，可以直接在 Compiler 和 Compilation 对象上广播和监听事件**，方法如下：

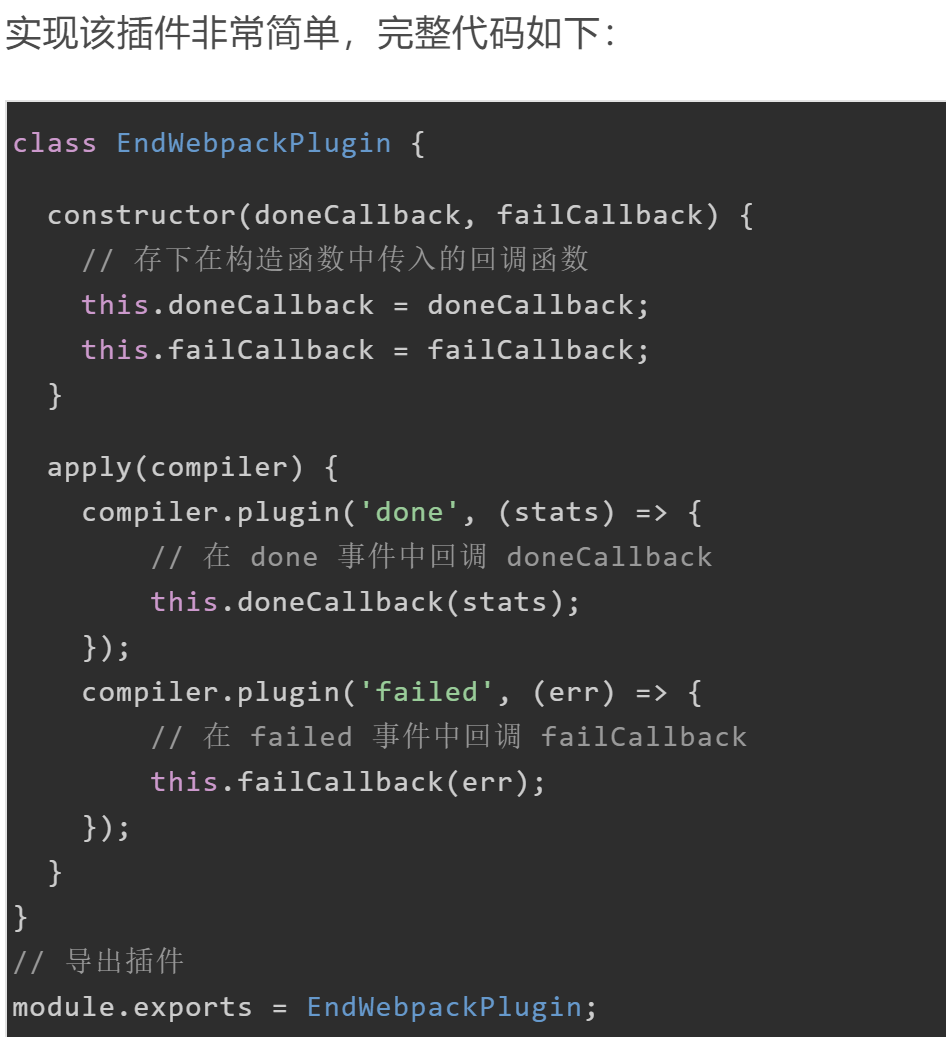




即**异步的事件需要在插件处理完任务时调用回调函数通知 Webpack 进入下一个流程，不然会卡住。**







## 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。）

## 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）

## Webpack构建流程

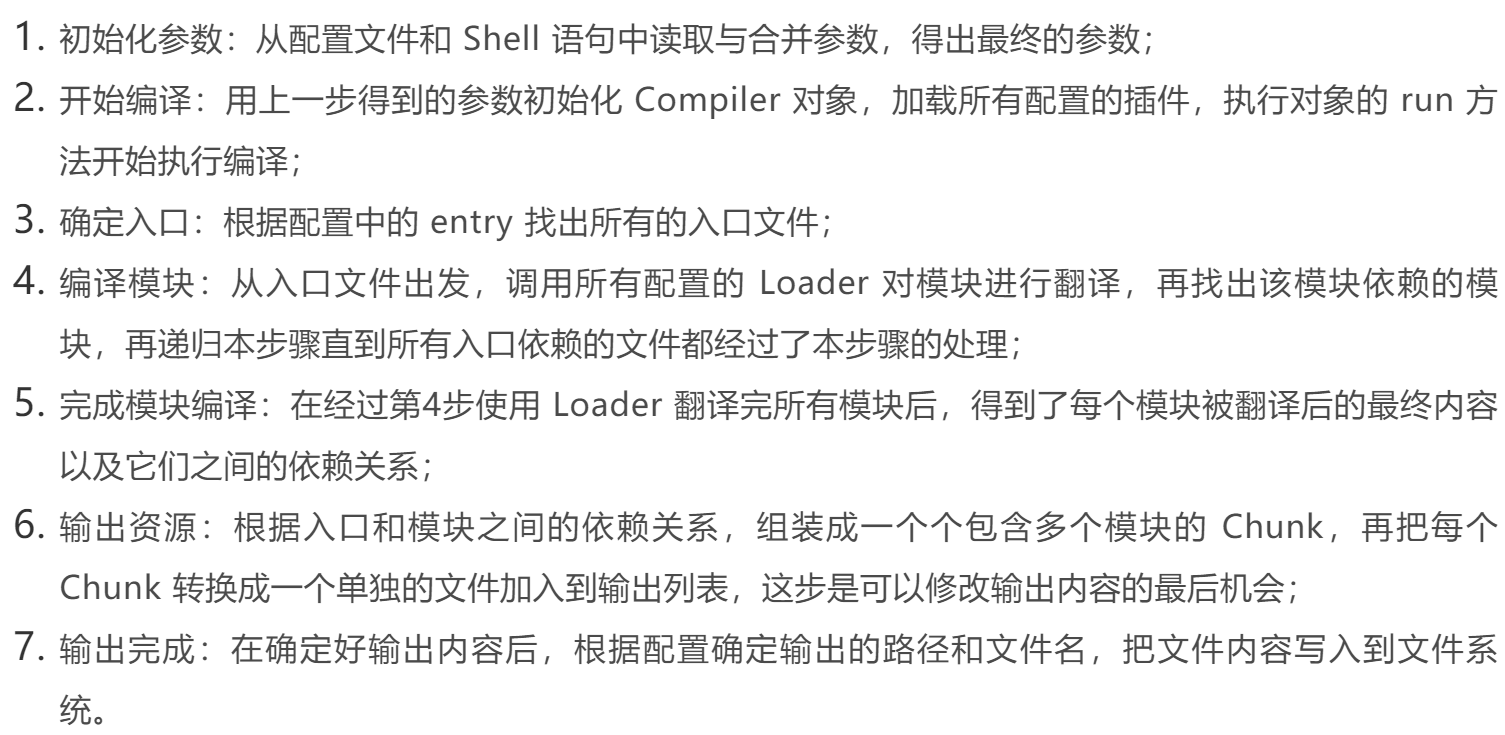
Webpack的运行流程是一个**串行**的过程，从启动到结束可以概括为以下三步：

* + **初始化**：**启动构建**，读取与合并配置参数，加载所有plugin，**初始化** 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 一个个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。**



在这个包含所有模块的数组中，通常把**入口entry.js的代码是放在索引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\_\_方法/函数引用不同索引的模块**。

拓展：



## 如何优化 Webpack 的构建速度？（减小打包体积）

暂时背诵：

### 压缩代码和图片：

* **通过uglifyjs-webpack-plugin 配置 parallel参数开启多进程压缩js代码**
* mini-css-extract-plugin 抽取CSS 代码到单独文件并通过optimize-css-assets-webpack-plugin进行压缩
* 配置 image-webpack-loader压缩图片

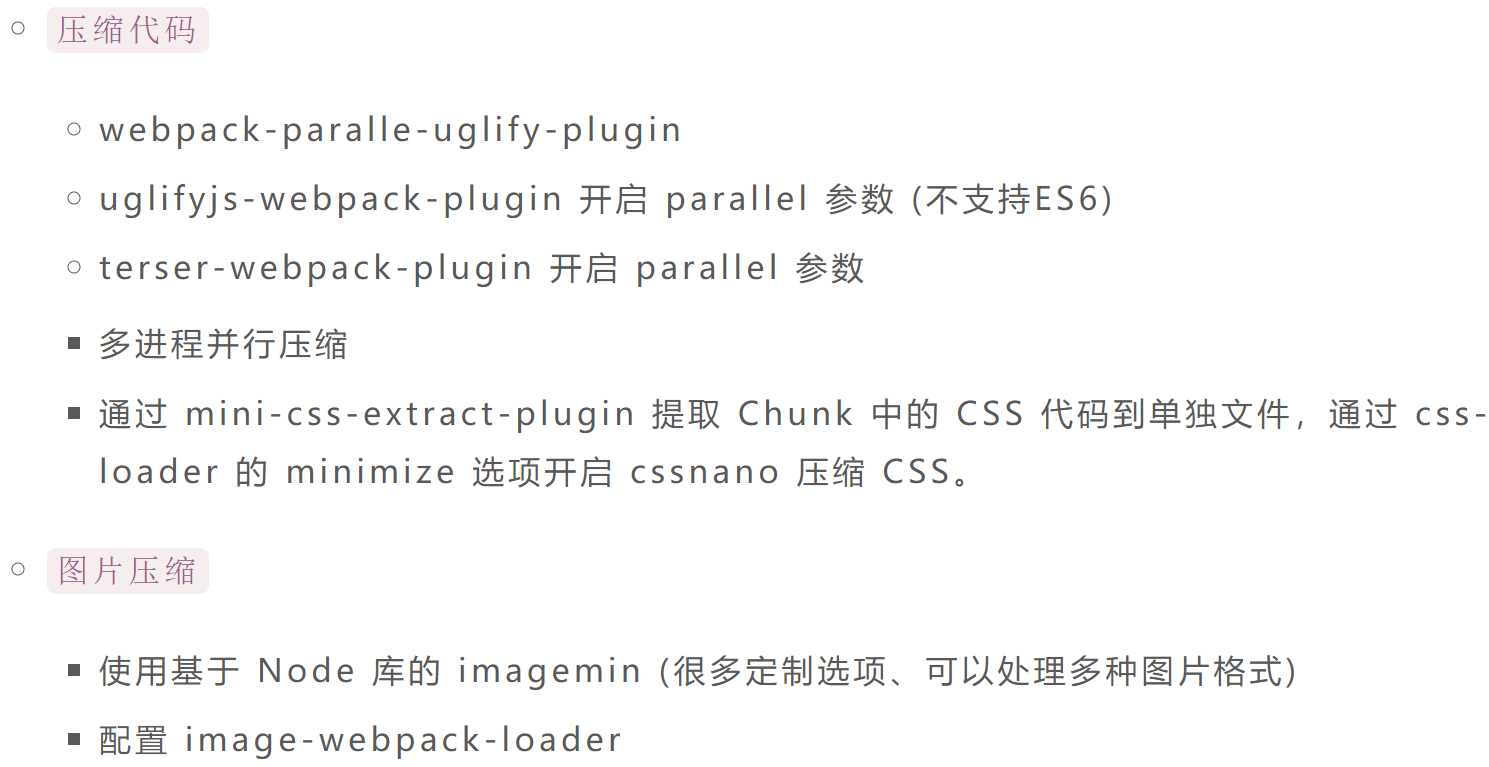
### 缩小打包作用域

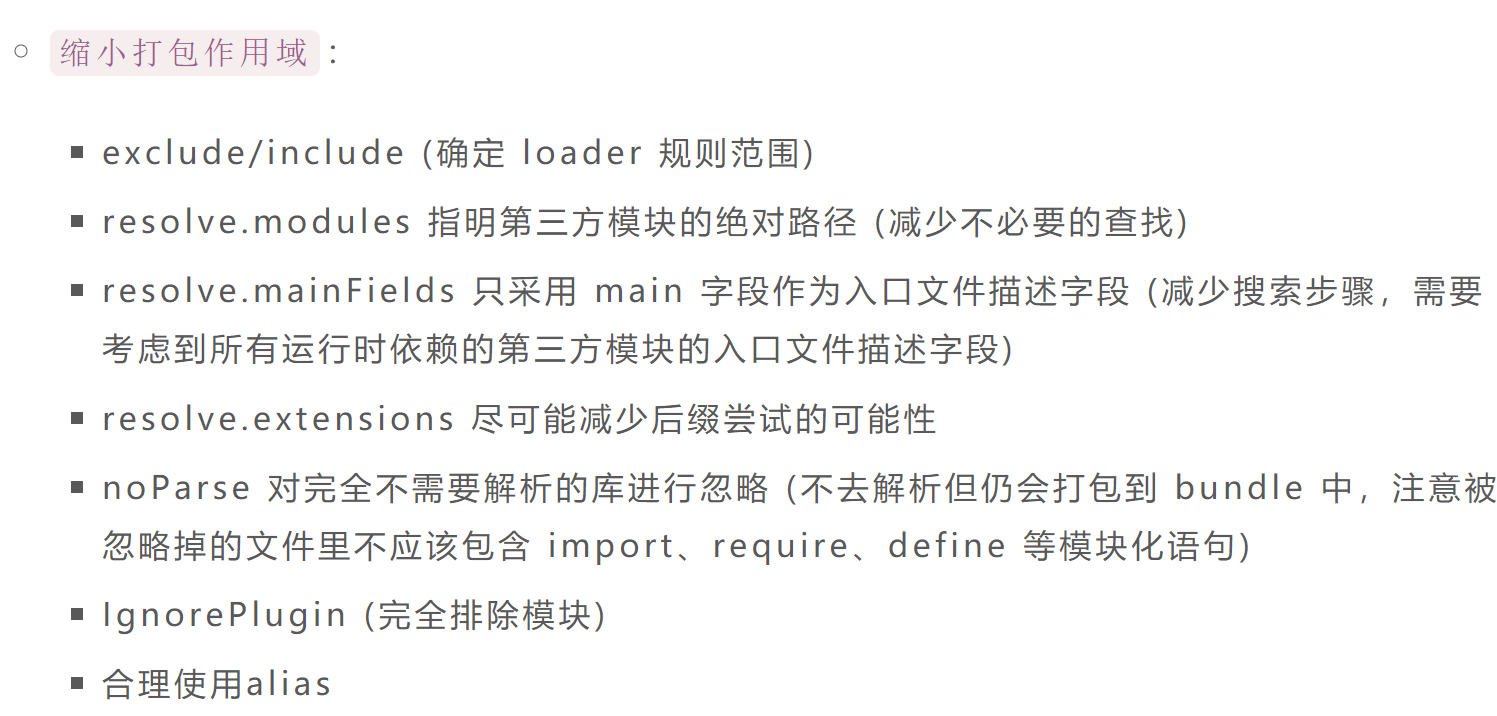
* **指定exclude/include (明确 loader 作用范围**)
* （配置**noParse** 属性来忽略完全不需要解析的库(不去解析但仍会打包到 bundle 中，注意被忽略掉的文件里不应该包含 import、require、define 等模块化语句)）
* **IgnorePlugin忽略第三方包指定目录**（如moment (2.24.0版本) 会将所有本地化内容和核心功能一起打包，我们就可以使用 IgnorePlugin 在打包时**忽略本地化内容**。）

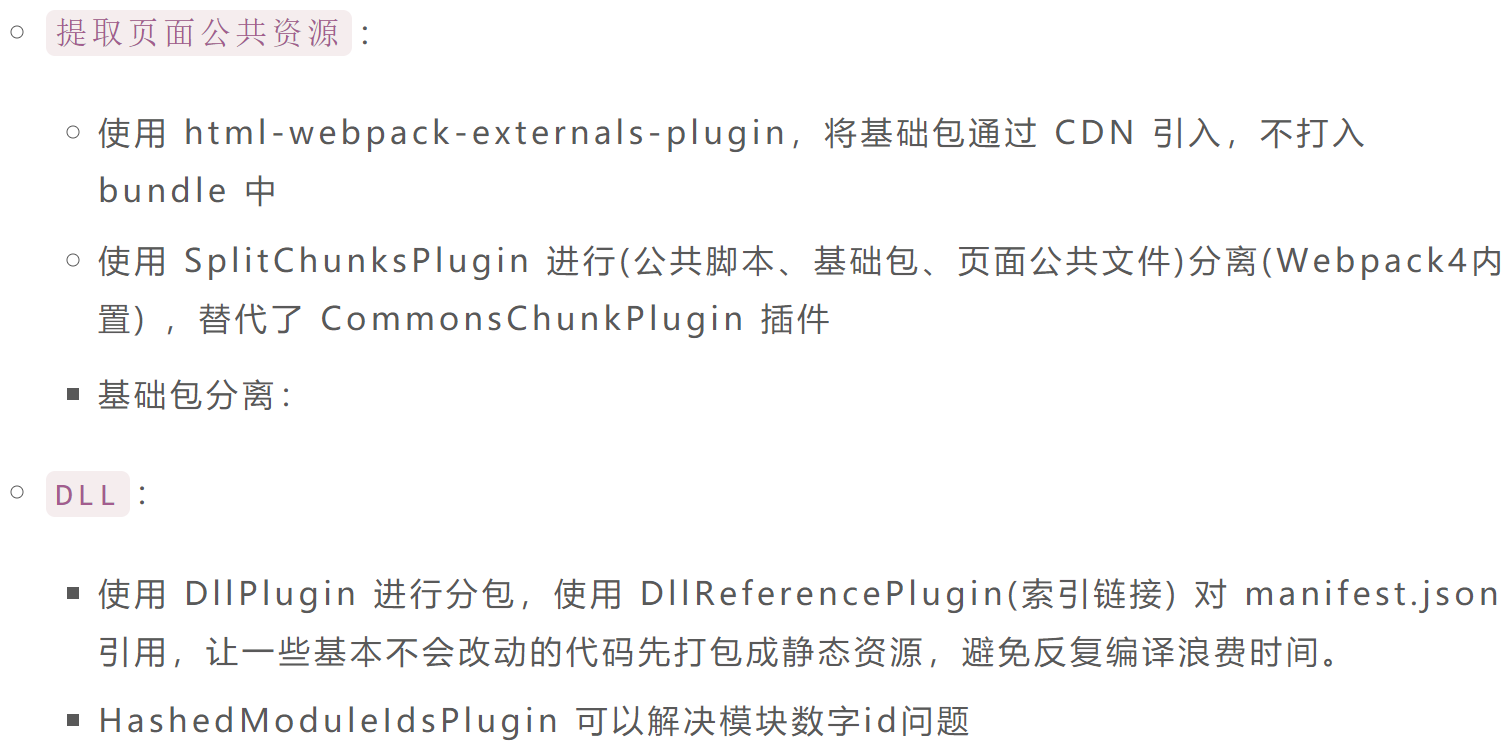
### 提取页面公共资源

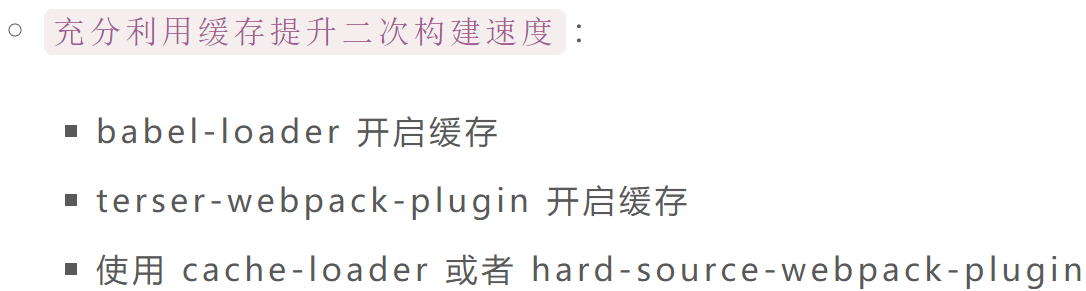
* **将基础包（如jQuery）通过 CDN 引入**，配置 externals属性，webpack不对其打包(但仍然可以通过 import的方式去引用(如 import $ from 'jquery'))
* 配置**optimization.splitChunks**将公共模块（公共脚本、基础包、页面公共文件）抽离出来，**单独打包**。**公共代码只需要下载一次就缓存起来了**，避免了重复下载

### 使用 DllPlugin 进行分包，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）。（如react、react-dom）









## 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）；**
* **dllplugin**等等。
* 去除不必要的插件；
* **去除devtool选项**；

## 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用新变更的模块替换旧的模块。**

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>

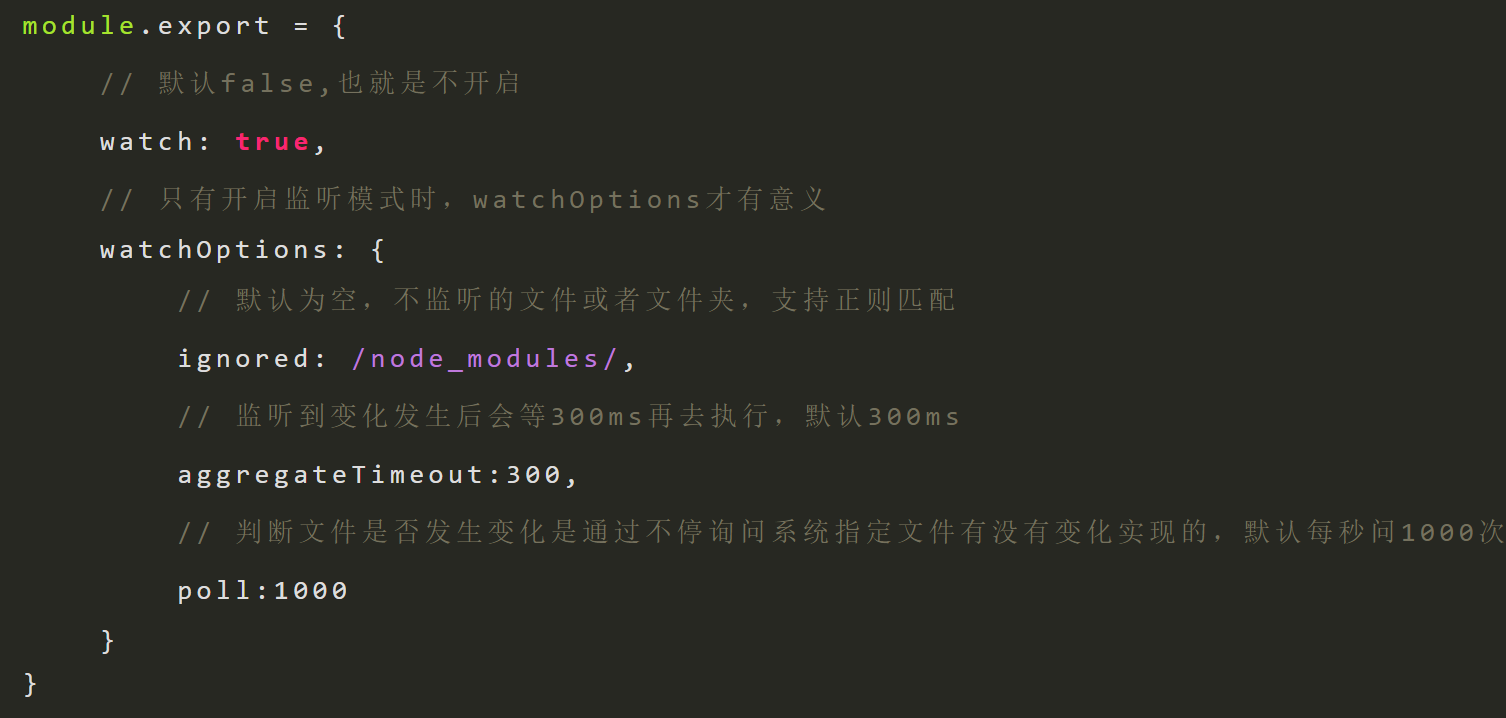
## source map是什么？生产环境怎么用？

**source map 是将编译、打包、压缩后的代码映射回源代码的过程，主要是为了方便调试。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