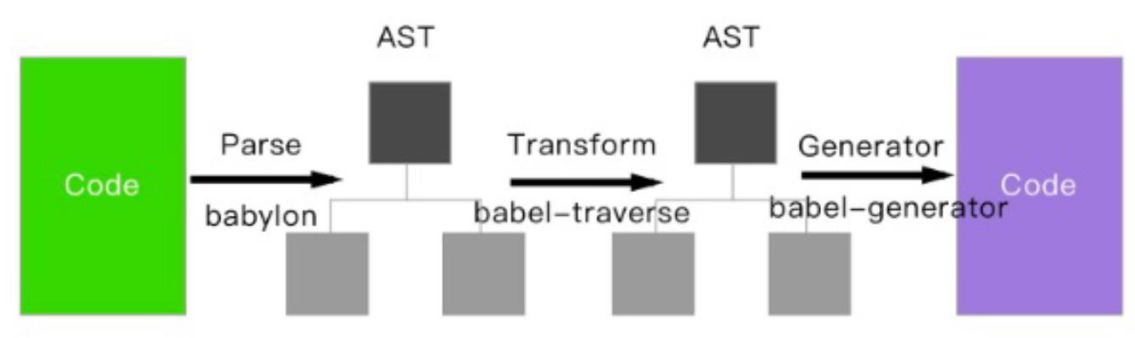
## Babel的原理是什么?

babel 的转译过程也分为三个阶段，这三步具体是：

* 解析 Parse: 将代码解析⽣成抽象语法树( 即AST )，即词法分析与语法分析的过程
* 转换 Transform: 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
* ⽣成 Generate: 将变换后的 AST 再转换为 JS 代码, 使⽤到的模块是 babel-generator

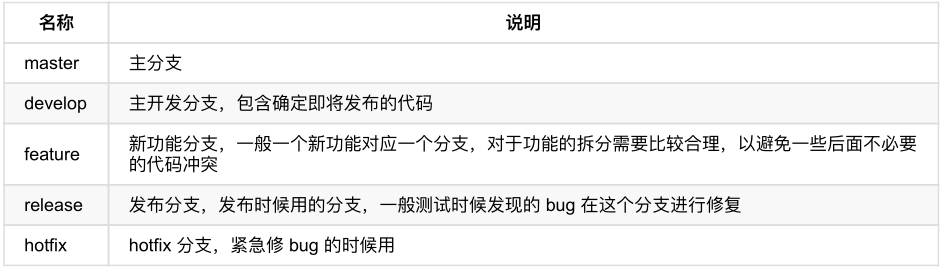


## 如何写⼀个babel插件?

**Babel解析成AST，然后插件更改AST，最后由Babel输出代码**

## 你的git⼯作流是怎样的?

GitFlow 是由 Vincent Driessen 提出的⼀个 git操作流程标准。包含如下⼏个关键分⽀：



GitFlow 的优势有如下⼏点：

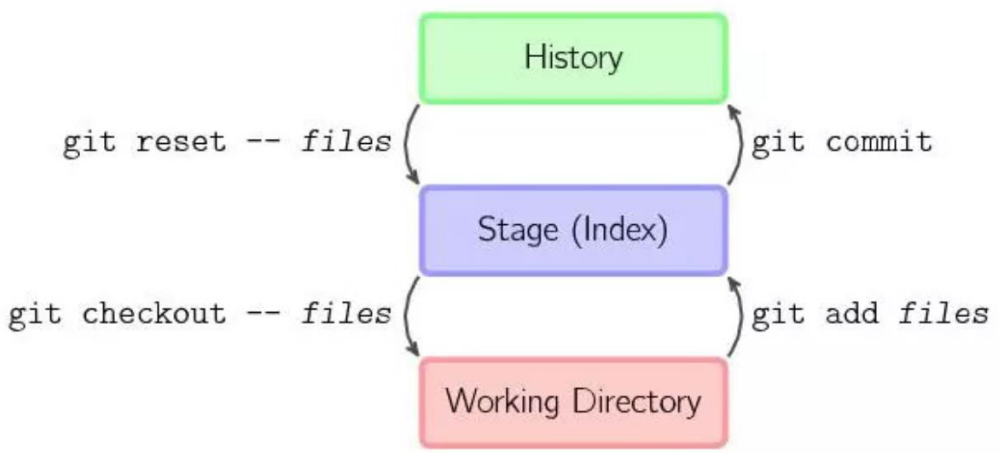
* 并⾏开发：GitFlow 可以很⽅便的实现并⾏开发：每个新功能都会建⽴⼀个新的 feature 分⽀，从⽽和已经完成的功能隔离开来，⽽且只有在新功能完成开发的情况下，其对应的 feature 分⽀才会合并到主开发分⽀上（也就是我们经常说的 develop 分⽀）。另外，如果你正在开发某个功能，同时⼜有⼀个新的功能需要开发，你只需要提交当前 feature 的代码，然后创建另外⼀个 feature 分⽀并完成新功能开发。然后再切回之前的 feature 分⽀即可继续完成之前功能的开发。
* 协作开发：GitFlow 还⽀持多⼈协同开发，因为每个 feature 分⽀上改动的代码都只是为了让某个新的 feature可以独⽴运⾏。同时我们也很容易知道每个⼈都在⼲啥。
* 发布阶段：当⼀个新 feature 开发完成的时候，它会被合并到 develop 分⽀，这个分⽀主要⽤来暂时保存那些还没有发布的内容，所以如果需要再开发新的 feature ，我们只需要从 develop 分⽀创建新分⽀，即可包含所有已经完成的 feature 。
* ⽀持紧急修复：GitFlow 还包含了 hotfix 分⽀。这种类型的分⽀是从某个已经发布的 tag 上创建出来并做⼀个紧 急的修复，⽽且这个紧急修复只影响这个已经发布的 tag，⽽不会影响到你正在开发的新 feature 。

## git reset、git revert 和 git checkout 有什么区别?

这个问题同样也需要先了解 git 仓库的三个组成部分：⼯作区（Working Directory）、暂存区（Stage）和历史记录区 （History）。

* ⼯作区：在 git 管理下的正常⽬录都算是⼯作区，我们平时的编辑⼯作都是在⼯作区完成
* 暂存区：临时区域。⾥⾯存放将要提交⽂件的快照
* 历史记录区：git commit 后的记录区

三个区的转换关系以及转换所使⽤的命令：



git reset、git revert 和 git checkout的共同点：⽤来撤销代码仓库中的某些更改。

然后是不同点：

⾸先，从 commit 层⾯来说：

* git reset 可以将⼀个分⽀的末端指向之前的⼀个 commit。然后再下次 git 执⾏垃圾回收的时候，会把这个 commit之后的 commit 都扔掉。git reset 还⽀持三种标记，⽤来标记 reset 指令影响的范围：
  + --mixed：会影响到暂存区和历史记录区。也是默认选项
  + --soft：只影响历史记录区
  + --hard：影响⼯作区、暂存区和历史记录区

注意：因为 git reset 是直接删除 commit 记录，从⽽会影响到其他开发⼈员的分⽀，所以不要在公共分⽀（⽐如develop）做这个操作。

* git checkout 可以将 HEAD 移到⼀个新的分⽀，并更新⼯作⽬录。因为可能会覆盖本地的修改，所以执⾏这个指令之前，你需要 stash 或者 commit 暂存区和⼯作区的更改。
* git revert 和 git reset 的⽬的是⼀样的，但是做法不同，它会以创建新的 commit 的⽅式来撤销 commit，这样能保 留之前的 commit 历史，⽐较安全。另外，同样因为可能会覆盖本地的修改，所以执⾏这个指令之前，你需要stash 或者 commit 暂存区和⼯作区的更改。

然后，从⽂件层⾯来说：

* git reset 只是把⽂件从历史记录区拿到暂存区，不影响⼯作区的内容，⽽且不⽀持 --mixed、--soft 和 --hard。
* git checkout 则是把⽂件从历史记录拿到⼯作区，不影响暂存区的内容。
* git revert 不⽀持⽂件层⾯的操作。

## webpack、rollup、parcel优劣？

* webpack适⽤于⼤型复杂的前端站点构建: webpack有强⼤的loader和插件⽣态,打包后的⽂件实际上就是⼀个⽴即执⾏函数，这个⽴即执⾏函数接收⼀个参数，这个参数是模块对象，键为各个模块的路径，值为模块内容。⽴即执行函数内部则处理模块之间的引⽤，执⾏模块等,这种情况更适合⽂件依赖复杂的应⽤开发.
* rollup适⽤于基础库的打包，如vue、d3等: Rollup 就是将各个模块打包进⼀个⽂件中，并且通过 Tree-shaking 来删除⽆⽤的代码,可以最⼤程度上降低代码体积,但是rollup没有webpack如此多的的如代码分割、按需加载等⾼级功能，其更聚焦于库的打包，因此更适合库的开发.
* parcel适⽤于简单的实验性项⽬: 他可以满⾜低⻔槛的快速看到效果,但是⽣态差、报错信息不够全⾯都是他的硬 伤，除了⼀些玩具项⽬或者实验项⽬不建议使⽤

## webpack中有哪些常⻅的Loader？

* 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
* url-loader：和 file-loader 类似，但是能在⽂件很⼩的情况下以 base64 的⽅式把⽂件内容注⼊到代码中去
* source-map-loader：加载额外的 Source Map ⽂件，以⽅便断点调试
* image-loader：加载并且压缩图⽚⽂件
* babel-loader：把 ES6 转换成 ES5
* 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。
* eslint-loader：通过 ESLint 检查 JavaScript 代码

## webpack中有哪些常⻅的Plugin？

* define-plugin：定义环境变量
* html-webpack-plugin：简化html⽂件创建
* uglifyjs-webpack-plugin：通过 UglifyES 压缩 ES6 代码
* webpack-parallel-uglify-plugin: 多核压缩,提⾼压缩速度
* webpack-bundle-analyzer: 可视化webpack输出⽂件的体积
* mini-css-extract-plugin: CSS提取到单独的⽂件中,⽀持按需加载

## 分别介绍webpack中bundle，chunk，module是什么？

* bundle：是由webpack打包出来的⽂件
* chunk：代码块，⼀个chunk由多个模块组合⽽成，⽤于代码的合并和分割
* module：是开发中的单个模块，在webpack的世界，⼀切皆模块，⼀个模块对应⼀个⽂件，webpack会从配置的entry中递归开始找出所有依赖的模块

## webpack与grunt、gulp的不同？

Grunt、Gulp是基于任务运⾏的⼯具：

它们会⾃动执⾏指定的任务，就像流⽔线，把资源放上去然后通过不同插件进⾏加⼯，它们包含活跃的社区，丰富的插 件，能⽅便的打造各种⼯作流。

Webpack是基于模块化打包的⼯具:

⾃动化处理模块,webpack把⼀切当成模块，当 webpack 处理应⽤程序时，它会递归地构建⼀个依赖关系图 (dependency graph)，其中包含应⽤程序需要的每个模块，然后将所有这些模块打包成⼀个或多个 bundle。

因此这是完全不同的两类⼯具,⽽现在主流的⽅式是⽤npm script代替Grunt、Gulp,npm script同样可以打造任务流.

## webpack中Loader和Plugin的不同

不同的作⽤:

* Loader直译为"加载器"。Webpack将⼀切⽂件视为模块，但是webpack原⽣是只能解析js⽂件，如果想将其他⽂件 也打包的话，就会⽤到 loader 。 所以Loader的作⽤是让webpack拥有了加载和解析⾮JavaScript⽂件的能⼒。
* Plugin直译为"插件"。Plugin可以扩展webpack的功能，让webpack具有更多的灵活性。 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。

不同的⽤法:

* Loader在 module.rules 中配置，也就是说他作为模块的解析规则⽽存在。 类型为数组，每⼀项都是⼀ 个 Object ，⾥⾯描述了对于什么类型的⽂件（ test ），使⽤什么加载( loader )和使⽤的参数（ options ）
* 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的构建流程是什么

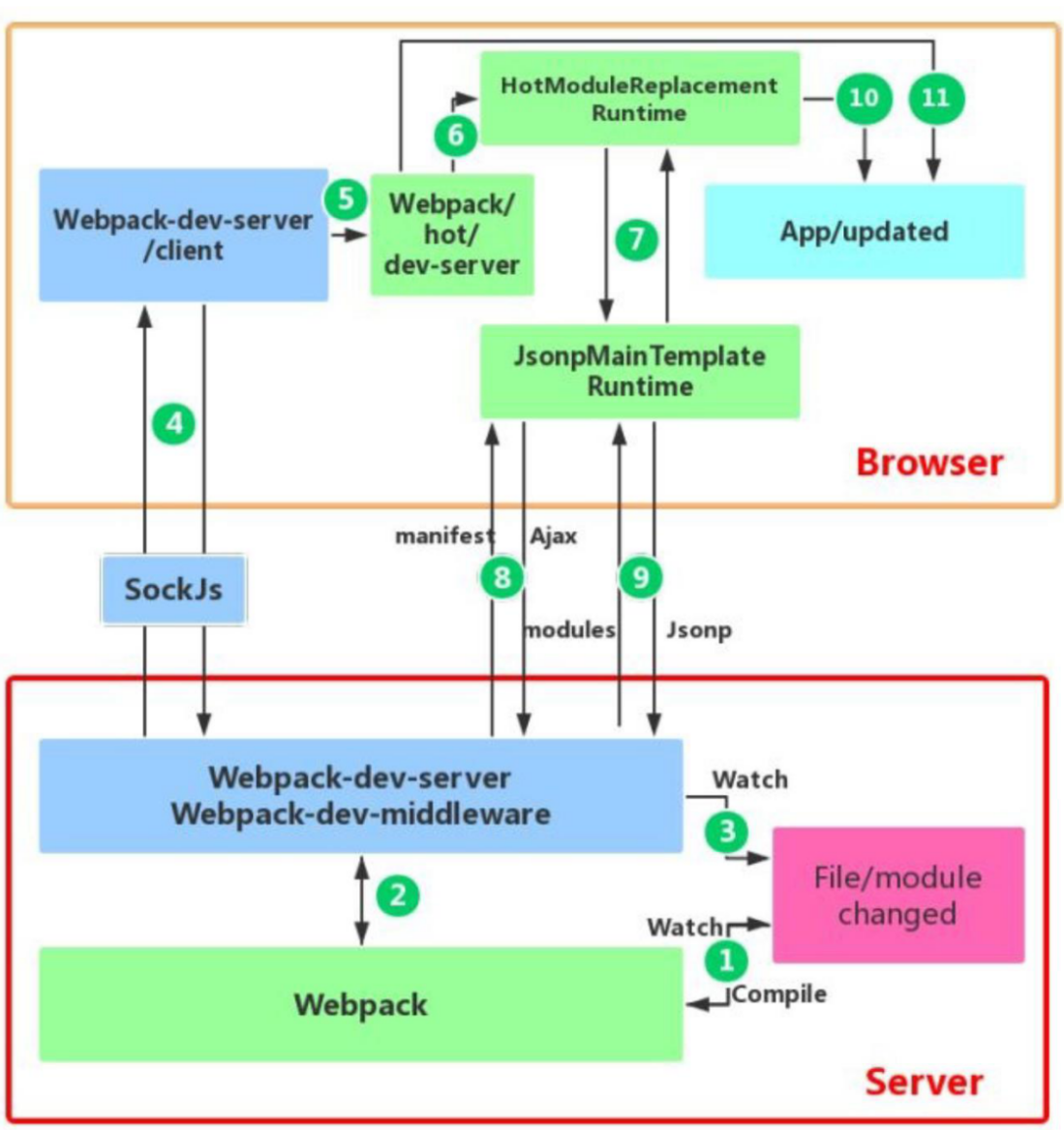
Webpack 的运⾏流程是⼀个串⾏的过程，从启动到结束会依次执⾏以下流程：

1. 初始化参数：从配置⽂件和 Shell 语句中读取与合并参数，得出最终的参数；
2. 开始编译：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. 确定⼊⼝：根据配置中的 entry 找出所有的⼊⼝⽂件；
4. 编译模块：从⼊⼝⽂件出发，调⽤所有配置的 Loader 对模块进⾏翻译，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：在经过第4步使⽤ Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的 依赖关系；
6. 输出资源：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；
7. 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，把⽂件内容写⼊到⽂件系统。 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。

## webpack的热更新是如何做到的？说明其原理？

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。

原理：



⾸先要知道server端和client端都做了处理⼯作

1. 第⼀步，在 webpack 的 watch 模式下，⽂件系统中某⼀个⽂件发⽣修改，webpack 监听到⽂件变化，根据配置⽂件对模块重新编译打包，并将打包后的代码通过简单的 JavaScript 对象保存在内存中。
2. 第⼆步是 webpack-dev-server 和 webpack 之间的接⼝交互，⽽在这⼀步，主要是 dev-server 的中间件 webpack-dev-middleware 和 webpack 之间的交互，webpack-dev-middleware 调⽤ webpack 暴露的 API对代码变化进⾏监控，并且告诉 webpack，将代码打包到内存中。
3. 第三步是 webpack-dev-server 对⽂件变化的⼀个监控，这⼀步不同于第⼀步，并不是监控代码变化重新打包。当我们在配置⽂件中配置了devServer.watchContentBase 为 true 的时候，Server 会监听这些配置⽂件夹中静态⽂件的变化，变化后会通知浏览器端对应⽤进⾏ live reload。注意，这⼉是浏览器刷新，和 HMR 是两个概念。
4. 第四步也是 webpack-dev-server 代码的⼯作，该步骤主要是通过 sockjs（webpack-dev-server 的依赖）在浏览器端和服务端之间建⽴⼀个 websocket ⻓连接，将 webpack 编译打包的各个阶段的状态信息告知浏览器端，同时也包括第三步中 Server 监听静态⽂件变化的信息。浏览器端根据这些 socket 消息进⾏不同的操作。当然服务端传递的最主要信息还是新模块的 hash 值，后⾯的步骤根据这⼀ hash 值来进⾏模块热替换。
5. webpack-dev-server/client 端并不能够请求更新的代码，也不会执⾏热更模块操作，⽽把这些⼯作⼜交回给了webpack，webpack/hot/dev-server 的⼯作就是根据 webpack-dev-server/client 传给它的信息以及 dev-server 的配置决定是刷新浏览器呢还是进⾏模块热更新。当然如果仅仅是刷新浏览器，也就没有后⾯那些步骤了。
6. HotModuleReplacement.runtime 是客户端 HMR 的中枢，它接收到上⼀步传递给他的新模块的 hash 值，它通过JsonpMainTemplate.runtime 向 server 端发送 Ajax 请求，服务端返回⼀个 json，该 json 包含了所有要更新的模块的 hash 值，获取到更新列表后，该模块再次通过 jsonp 请求，获取到最新的模块代码。这就是上图中 7、8、9 步骤。
7. ⽽第 10 步是决定 HMR 成功与否的关键步骤，在该步骤中，HotModulePlugin 将会对新旧模块进⾏对⽐，决定是否更新模块，在决定更新模块后，检查模块之间的依赖关系，更新模块的同时更新模块间的依赖引⽤。
8. 最后⼀步，当 HMR 失败后，回退到 live reload 操作，也就是进⾏浏览器刷新来获取最新打包代码。

## 如何提⾼webpack的构建速度？

1. 多⼊⼝情况下，使⽤ CommonsChunkPlugin 来提取公共代码
2. 通过 externals 配置来提取常⽤库
3. 利⽤ DllPlugin 和 DllReferencePlugin 预编译资源模块 通过 DllPlugin 来对那些我们引⽤但是绝对不会修改的npm包来进⾏预编译，再通过 DllReferencePlugin 将预编译的模块加载进来。
4. 使⽤ Happypack 实现多线程加速编译
5. 使⽤ webpack-uglify-parallel 来提升 uglifyPlugin 的压缩速度。 原理上 webpack-uglify-parallel 采⽤了多核并⾏压缩来提升压缩速度
6. 使⽤ Tree-shaking 和 Scope Hoisting 来剔除多余代码

## 如何提⾼webpack的打包速度/减少打包时间?

**优化 Loader**

对于 Loader 来说，影响打包效率首当其冲必属 Babel 了。因为 Babel 会将代码转为字符串生成 AST，然后对 AST 继续进行转变最后再生成新的代码，项目越大，转换代码越多，效率就越低。当然了，我们是有办法优化的。

首先我们可以优化 Loader 的文件搜索范围

module.exports = {

module: {

rules: [

{

// js 文件才使用 babel

test: /\.js$/,

loader: 'babel-loader',

// 只在 src 文件夹下查找

include: [resolve('src')],

// 不会去查找的路径

exclude: /node\_modules/

}

]

}

}

对于 Babel 来说，我们肯定是希望只作用在 JS 代码上的，然后 node\_modules 中使用的代码都是编译过的，所以我们也完全没有必要再去处理一遍。

当然这样做还不够，我们还可以将 Babel 编译过的文件缓存起来，下次只需要编译更改过的代码文件即可，这样可以大幅度加快打包时间

loader: 'babel-loader?cacheDirectory=true'

**HappyPack**

受限于 Node 是单线程运行的，所以 Webpack 在打包的过程中也是单线程的，特别是在执行 Loader 的时候，长时间编译的任务很多，这样就会导致等待的情况。

HappyPack 可以将 Loader 的同步执行转换为并行的，这样就能充分利用系统资源来加快打包效率了

module: {

loaders: [

{

test: /\.js$/,

include: [resolve('src')],

exclude: /node\_modules/,

// id 后面的内容对应下面

loader: 'happypack/loader?id=happybabel'

}

]

},

plugins: [

new HappyPack({

id: 'happybabel',

loaders: ['babel-loader?cacheDirectory'],

// 开启 4 个线程

threads: 4

})

]

**DllPlugin**

DllPlugin 可以将特定的类库提前打包然后引入。这种方式可以极大的减少打包类库的次数，只有当类库更新版本才有需要重新打包，并且也实现了将公共代码抽离成单独文件的优化方案。

接下来我们就来学习如何使用 DllPlugin

// 单独配置在一个文件中

// webpack.dll.conf.js

const path = require('path')

const webpack = require('webpack')

module.exports = {

entry: {

// 想统一打包的类库

vendor: ['react']

},

output: {

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

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

library: '[name]-[hash]'

},

plugins: [

new webpack.DllPlugin({

// name 必须和 output.library 一致

name: '[name]-[hash]',

// 该属性需要与 DllReferencePlugin 中一致

context: \_\_dirname,

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

})

]

}

然后我们需要执行这个配置文件生成依赖文件，接下来我们需要使用 DllReferencePlugin 将依赖文件引入项目中

// webpack.conf.js

module.exports = {

// ...省略其他配置

plugins: [

new webpack.DllReferencePlugin({

context: \_\_dirname,

// manifest 就是之前打包出来的 json 文件

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

})

]

}

**代码压缩**

在 Webpack3 中，我们一般使用 UglifyJS 来压缩代码，但是这个是单线程运行的，为了加快效率，我们可以使用 webpack-parallel-uglify-plugin 来并行运行 UglifyJS，从而提高效率。

在 Webpack4 中，我们就不需要以上这些操作了，只需要将 mode 设置为 production 就可以默认开启以上功能。代码压缩也是我们必做的性能优化方案，当然我们不止可以压缩 JS 代码，还可以压缩 HTML、CSS 代码，并且在压缩 JS 代码的过程中，我们还可以通过配置实现比如删除 console.log 这类代码的功能。

**一些小的优化点**

我们还可以通过一些小的优化点来加快打包速度

* resolve.extensions：用来表明文件后缀列表，默认查找顺序是 ['.js', '.json']，如果你的导入文件没有添加后缀就会按照这个顺序查找文件。我们应该尽可能减少后缀列表长度，然后将出现频率高的后缀排在前面
* resolve.alias：可以通过别名的方式来映射一个路径，能让 Webpack 更快找到路径
* module.noParse：如果你确定一个文件下没有其他依赖，就可以使用该属性让 Webpack 不扫描该文件，这种方式对于大型的类库很有帮助

## 如何⽤webpack来优化前端性能？

⽤webpack优化前端性能是指优化webpack的输出结果，让打包的最终结果在浏览器运⾏快速⾼效。

* 压缩代码:删除多余的代码、注释、简化代码的写法等等⽅式。可以利⽤webpack的 UglifyJsPlugin 和 ParallelUglifyPlugin 来压缩JS⽂件， 利⽤ cssnano （css-loader?minimize）来压缩css
* 利⽤CDN加速: 在构建过程中，将引⽤的静态资源路径修改为CDN上对应的路径。可以利⽤webpack对于 output 参数和各loader的 publicPath 参数来修改资源路径
* Tree Shaking: 将代码中永远不会⾛到的⽚段删除掉。可以通过在启动webpack时追加参数 --optimize-minimize 来实现
* Code Splitting: 将代码按路由维度或者组件分块(chunk),这样做到按需加载,同时可以充分利⽤浏览器缓存
* 提取公共第三⽅库: SplitChunksPlugin插件来进⾏公共模块抽取,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

## 是否写过webpack的Loader和Plugin？描述⼀下编写loader或plugin的思路？

Loader像⼀个"翻译官"把读到的源⽂件内容转义成新的⽂件内容，并且每个Loader通过链式操作，将源⽂件⼀步步翻译成想要的样⼦。

编写Loader时要遵循单⼀原则，每个Loader只做⼀种"转义"⼯作。 每个Loader的拿到的是源⽂件内容（ source ），可以通过返回值的⽅式将处理后的内容输出，也可以调⽤ this.callback() ⽅法，将内容返回给webpack。 还可以通过this.async() ⽣成⼀个 callback 函数，再⽤这个callback将处理后的内容输出出去。 此外 webpack 还为开发者准备了开发loader的⼯具函数集—— loader-utils 。

相对于Loader⽽⾔，Plugin的编写就灵活了许多。 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，在合适的时机通过 Webpack 提供的 API 改变输出结果。