webpack

工作流程

- webpack会从配置文件中的entry递归找到所有依赖的模块

webpack的打包文件类型

- bundle
  - 是webpack打包出来的文件
- trunk
  - 是代码块，一个trunk由多个模块组合而成，用于代码的合并和分割
  - 对于打包产物bundle，有些情况下，我们觉得太大了。为了优化性能，比如快速打开首屏，利用缓存等，我们需要对bundle进行以下拆分，对于拆分出来的东西，我们叫它chunk。
- module
  - 是开发中的单个模块

利用webpack优化前端性能

- 压缩代码并删除多余代码，简化代码写法
  - 压缩js文件
    - uglify.js
    - paralleluglify-plugin
  - 压缩css文件
    - cssnano
- 使用CDN
  - 将引用的静态资源路径修改为CDN路径
  - 可以通过output参数和各个loader的publicPath来修改路径
- 提取公共代码
  - 通过externals配置来提取常用库
- 分包**SplitChunk**
  - 使用插件：split-chunks-plugin
  - webpack 将根据以下条件自动拆分 chunks：
    - 新的 chunk 可以被共享，或者模块来自于 `node_modules` 文件夹
    - 新的 chunk 体积大于 20kb（在进行 min+gz 之前的体积）
    - 当按需加载 chunks 时，并行请求的最大数量小于或等于 30
    - 当加载初始化页面时，并发请求的最大数量小于或等于 30
- 开发时优化
  - 配置alias

```
alias: {
  '~': resolve('src'),
  '@': resolve('src'),
  'components': resolve('src/components'),
}
```

  - 配置contentBase
    - 在devServer中

```
contentBase: path.resolve(__dirname, 'public'), // 静态文件目录
```

    - 因为 webpack 在进行打包的时候，对静态文件的处理，例如图片，都是直接 copy 到 dist 目录下面。但是对于本地开发来说，这个过程大费时，也没有必要，所以在设置 contentBase 之后，就直接到对应的静态目录下面去读取文件，而不需对文件做任何移动，节省了时间和性能开销。
  - 解析模块时应该搜索的目录

```
const config = {
  //...
  resolve: {
    modules: [resolve('src'), 'node_modules']
  }
}
```

    - 告诉 webpack 优先 src 目录下查找需要解析的文件，会大大节省查找时间
  - 你有自定义的 Loader 就需要配置一下，不配可能会因为找不到 loader 报错

```
resolveLoader: {
  modules: ['node_modules', resolve('loader')]
},
```

  - 浏览器空闲的时候进行资源的拉取:prefetch

```
// 按需加载
img.addEventListener('click', () => {
    import(/* webpackPrefetch: true */ './desc').then(({ default: element }) => {
console.log(element) document.body.appendChild(element) })
})
```

- 打包速度优化
  - **thread-loader**
    - loader 都会在一个单独的 worker 池（worker pool）中运行
    - 开启多进程打包的工具
  - **babel-loader 开启缓存**

```
{ loader: 'babel-loader', options: { cacheDirectory: true // 启用缓存 } },
```

    - babel 在转译 js 过程中时间开销比价大，将 babel-loader 的执行结果缓存起来，重新打包的时候，直接读取缓存
  - 其他loader开启缓存cache-loader
- 对上线后的优化
  - 配置选项提供了「从输出的 bundle 中排除依赖」的方法
    - externals
  - **noParse**

```
const config = {
  //...
  module: {
    noParse: /jquery|lodash/,
    rules: [...]
  }
};
```

    - 不需要解析依赖的第三方大型类库等，可以通过这个字段进行配置，以提高构建速度
  - 压缩js
    - terser-webpack-plugin
  - CSS
    - 压缩css
      - optimize-css-assets-webpack-plugin
    - **清除无用的 CSS**
      - purgecss-webpack-plugin
  - **Tree-shaking**
    - 删除没有使用的代码，以降低包的体积
    - 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)
    - 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

```
externals表示打包时对应配置的包不打包到最终的bundle.js中

noParse表示不解析配置项的相关依赖(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)

exclude表示不使用loader处理匹配到的文件路径

resolve.extensions 尽可能减少后缀尝试的可能性
resolve.modules 指明第三方模块的绝对路径 (减少不必要的查找)
```

- 借助的工具
  - **构建费时分析**
    - speed-measure-webpack-plugin
  - 可以直观的看到打包结果中，文件的体积大小、各模块依赖关系、文件是够重复等问题，极大的方便我们在进行项目优化的时候，进行问题诊断
    - webpack-bundle-analyzer
- 压缩代码
  - 多进程执行代码压缩，提升构建速度
    - webpack-parallel-uglify-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。
    - mini-css-extract-plugin
- 压缩图片
  - 配置 image-webpack-loader
- babel-loader 开启缓存
  - 充分利用缓存提升二次构建速度

babel内部是如何转的

与vite的比较

- vite
  - 服务器快速启动
    - 将应用中的模块区分为 依赖 和 源码 两类
      - 依赖 大多为在开发时不会变动的纯 JavaScript。比如npm安装的库
        - Vite 将会使用 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。
      - 源码 通常包含一些并非直接是 JavaScript 的文件，需要转换（例如 JSX，CSS 或者 Vue/Svelte 组件），时常会被编辑
        - 并不是所有的源码都需要同时被加载（例如基于路由拆分的代码模块）。
    - Vite 以 原生 ESM 方式提供源码。这实际上是让浏览器接管了打包程序的部分工作
      - 原生ESM：浏览器自身对模块的支持
        - 在使用上，唯一的区别就是需要在script标签上添加一个type="module"的属性来表示这个文件是作为module的方式来运行的。

```
<script type="module">
  import module from './module.js'
</script>
<script nomodule>                优雅降级
  alert('your browsers can not supports es modules! please upgrade it.')
</script>
```

      - 根据情景动态导入代码，即只在当前屏幕上实际使用时才会被处理。
        - Vite 只需要在浏览器请求源码时进行转换并按需提供源码。
  - 更新速度快
    - HMR（热更新）是在原生 ESM 上执行的
      - 当编辑一个文件时，Vite 只需要精确地使已编辑的模块与其最近的 HMR 边界之间的链失活
    - 源码模块会进行写上缓存，依赖模块会进行强缓存
      - 由于现代浏览器本身就支持ES Module，会自动向依赖的Module发出请求
    - Rollup 在应用打包方面更加成熟和灵活。使用Rollup打包
  - 对 TypeScript、JSX、CSS 等支持开箱即用。
- 比较
  - webpack会先打包，然后启动开发服务器
    - 由于vite在启动的时候不需要打包，也就意味着不需要分析模块的依赖、不需要编译，因此启动速度非常快
      - 当浏览器请求某个模块时，再根据需要对模块内容进行编译。这种按需动态编译的方式，极大的缩减了编译时间，项目越复杂、模块越多，vite的优势越明显
  - 在HMR（热更新）方面，当改动了一个模块后，vite仅需让浏览器重新请求该模块即可，不像webpack那样需要把该模块的相关依赖模块全部编译一次，效率更高。
  - vite的主要优势在开发阶段
- vite缺点
  - 生态不及webpack，加载器、插件不够丰富
  - 生产环境esbuild构建对于css和代码分割不够友好
  - 没被大规模重度使用，会隐藏一些问题

是一个打包模块化的JS工具

- 在webpack中，一切文件皆模块
- 通过loader转换文件，通过plugin注入钩子
- 最后输出有多个模块组合成的文件
- webpack所做的事情就是分析项目结构，找到JS模块以及其他浏览器不能直接运行的拓展语言，例如sass,TS等，并将其打包成合适的格式，以供浏览器使用

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。
  - 告诉webpack如何如何转化处理某一类型的文件，并引入到打包文件当中
  - 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
        - 描述了对于什么类型的文件，使用什么加载，使用什么参数
  - Loader 的执行顺序是固定从后往前，即按 `css-loader --> style-loader` 的顺序执行
  - 编写loader
    - 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
    - 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
  - 常用的loader
    - babel-loader
      - 吧ES6转成ES5
    - vue-loader
      - 找到.vue文件，将template、css、js提取出来交给相应的loader进行处理
    - sass-loader
    - css-loader
      - 加载CSS，支持模块化，压缩，文件导入
    - 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
    - style-loader
      - 将处理好的 css 通过 style 标签的形式添加到页面上
    - eslint-loader
      - 通过ESlint检查JS代码
    - file-loader
      - 把文件输出到一个文件夹中，在代码中通过相对URL的方式引入文件
    - 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
      - 生成sourcemap，方便debug
    - image-loader
      - 加载并压缩图片
- 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。
  - 用来自定义webpack打包过程的插件，插件含有aplay方法，可以参与到webpack打包的各个流程
  - 在 plugins 中单独配置
    - 每一项是一个plugin实例，参数通过构造函数传入
  - 常用的plugin
    - html-webpack-plugin
      - 简化 HTML 文件创建 (依赖于 html-loader)
    - HotModuleReplacementPlugin
      - 模块热替换
    - compression-webpack-plugin
      - 把生成的文件进行压缩
    - vue-skeleton-webpack-plugin
      - 骨架屏
    - clean-webpack-plugin
      - **自动清空打包目录**
    - 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

构建流程

- 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
- 加载loader,plugin,并执行run方法
  - 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
- 确定入口
  - 根据配置中的 entry 找出所有的入口文件
- loader编译模块
  - 从入口文件出发，调用配置的 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
- 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
- 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
- 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

https://juejin.cn/post/6844904094281236487#heading-2

treeShaking原理

- ES Module 输出的是值的引用，而 CommonJS 输出的是值的拷贝
- ES Module 是编译时执行，而 CommonJS 模块是在运行时加载
- 所以 ES Module 最大的特点就是静态化，在编译时就能确定模块的依赖关系，以及输入和输出的值，这意味着什么？意味着模块的依赖关系是确定的，和运行时的状态无关，可以进行可靠的静态分析，正是基于这个基础，才使得 Tree-Shaking 成为可能，这也是为什么 rollup 和 Webpack 2 都要用 ES6 Module 语法才能支持 Tree-Shaking。
- Tree-Shaking 实现的大体思路：借助 ES6 模块语法的静态结构，通过编译阶段的静态分析，找到没有引入的模块并打上标记，然后在压缩阶段利用像 uglify-js 这样的压缩工具删除这些没有用到的代码。

热更新原理

- 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。
- webpack-dev-server 在浏览器端和服务端之间建立一个 websocket 长连接，将 webpack 编译打包的各个阶段的状态信息以及静态文件变化信息告知浏览器端，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。
- 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。
- 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。

babel原理

- 解析：将代码转换成 AST
  - 词法分析：将代码(字符串)分割为语法单元的数组
    - 语法单元
      - 空白
      - 注释
      - 字符串
      - 数字
      - 标识符
      - 运算符
      - 括号
  - 语法分析：分析token流(上面生成的数组)并生成 AST
    - 语义分析的过程又是个遍历语法单元的过程，不过相比较而言更复杂，因为分词过程中，每个语法单元都是独立平铺的，而语法分析中，语句和表达式会以树状的结构互相包含。针对这种情况我们可以用栈，也可以用递归来实现。
- 转换：访问 AST 的节点进行变换操作生产新的 AST
- 生成：以新的 AST 为基础生成代码