Webpack

- 核心概念
  - entry：以哪个文件作为入口开始打包(分析构建内部依赖)
  - output：打包好的文件输出的目的地以及如何命名
  - loader：处理除js,json的其他资源(如css,img,font等，职责相对单一)
  - plugin：处理一系列更强大的任务(如文件压缩，定义全局变量等)
  - mode：区分开发环境(development)和生产环境(production)
- 开发环境
  - 1. 默认打包
    - 开发环境：webpack --mode=development /src/index.js -o ./dist/bundle.js
    - 生产环境：webpack --mode=production /src/index.js -o ./dist/bundle.js
    - 打包后会产生一个hash码(如73113a4a4f2147ab8dee3)，生产环境会压缩代码，但开发环境和生产环境都能消费ES6模块编译试浏览器缓识别的模块
    - 执行：node ./dist/bundle.js查看运行结果
    - webpack默认只能打包js, json资源，不能处理其他资源(如css, 必须配合对应的loader才能处理)
  - 2. 打包样式
    - 添加webpack配置文件：webpack.config.js(commonjs规范) =
    - 安装style-loader和css-loader: yarn add style-loader css-loader -D
    - 如需要处理其他样式资源(如less), 安装less和less-loader: yarn add less less-loader -D
    - 执行：webpack查看运行结果
    - 不论是css资源还是less资源，都会最终把样式以style的形式插入到head标签
  - 3. 打包html
    - 安装HtmlWebpackPlugin: yarn add html-webpack-plugin -D
    - 更新webpack配置文件：webpack.config.js(引入HtmlWebpackPlugin) =
    - 默认的HtmlWebpack配置在输出目录自动创建一个index.html文件，引入打包的所有资源(js/css)
    - 指定template选项后，复制指定template对应的index.html并自动引入所有打包后的资源(js/css)
    - 执行：webpack查看运行结果
    - 需要注意的是：由于HtmlWebpack会自动引入js/css, 因此在模板index.html中不要手动引入打包后的资源
  - 4. 打包图片
    - 安装file-loader和url-loader: yarn add file-loader url-loader -D
    - 更新webpack配置文件：webpack.config.js(注意：limit选项) =
    - 执行：webpack查看运行结果(自配配置无法处理html中直接引入图片的问题)
    - 安装html-loader: yarn add html-loader -D并更新webpack.config.js =
    - 默认url-loader采用的是es6的模块化解析方式，而html-loader@0.5.5版用的commonjs规范引入图片，解析后会出现[Object Module]的问题
      - 解决方案(2选1)
      - 升级html-loader为1.0.0
      - 关闭url-loader的es6模块解析方式
    - 对图片资源打包，如在css和html同时引入同一张图片，则最终打包文件中只会包含生成一份带hash的图片
  - 5. 打包其他
    - 下载iconfont图标并将iconfont引入
    - 更新webpack配置文件：webpack.config.js(加入对图标字体的处理) =
    - 执行：webpack查看运行结果
  - 6. devServer
    - 为了解决在开发环境每次加入新功能都需要重新构建了能看到更新的问题，加入devServer配置(独立于头核心概念之外)
    - 完成自动化功能(监视文件变化完成自动编译，自动打开浏览器，自动刷新浏览器等)
    - 安装webpack-dev-server: yarn add webpack-dev-server -D
    - 更新webpack配置文件: webpack.config.js(加入devServer选项) =
    - 执行：npx webpack-dev-server查看运行结果(只会在内存中打包，不会在本地输出)
    - 此时加入新代码webpack-dev-server监测文件变化，不需重新构建(就会自动编译)，浏览器就会自动刷新
  - 7. 开发环境基础配置
    - 集成入门知识1-6环节中所有的配置，并归类划分各种资源(js/css/img/font)
    - 更新webpack配置文件: webpack.config.js =
    - 为了能打包后也能引src目录保持一致性，需要修改js/css/html中文件路径指向
    - 执行：webpack查看运行结果
    - 执行：npx webpack-dev-server查看运行结果
    - 经历这7步，一个基本的webpack开发环境配置就完成了
- 生产环境
  - 1. 打包提取css到单独文件
    - 为了解决开发环境中css最终打包到bundle.js中夹杂着的问题(js体积变大，页面启动发生闪动)，引入MiniCssExtractPlugin
    - 安装mini-css-extract-plugin: yarn add mini-css-extract-plugin -D
    - 更新webpack配置文件: webpack.config.js(注意: 代替style-loader) =
    - 执行：webpack查看运行结果
    - 与style-loader不同的是: MiniCssExtractPlugin.loader提取bundle.js中的css为单独文件并以link标签添加到head标签
  - 2. css浏览器兼容性处理
    - 为了解决css部分特性在各大浏览器不兼容的问题，引入PostCSS
    - 安装postcss-loader和autoprefixer: yarn add postcss-loader autoprefixer -D
    - 添加postcss配置文件: postcss.config.js =
    - 更新webpack配置文件: webpack.config.js =
    - 执行：webpack查看运行结果
  - 3. css压缩
    - 为了解决css打包后体积变大的问题，引入OptimizeCssAssetsWebpackPlugin
    - 安装optimize-css-assets-webpack-plugin: yarn add optimize-css-assets-webpack-plugin -D
    - 更新webpack配置文件: webpack.config.js =
    - 执行：webpack查看运行结果
  - 4. 利用eslint进行js语法检查
    - 为了解决js源代码的书写规范问题，引入ESLint
    - 安装eslint和eslint-loader: yarn add eslint eslint-loader -D
    - 安装eslint-plugin-import和eslint-config-airbnb-base: yarn add eslint-plugin-import eslint-config-airbnb-base -D
    - 增加eslint配置文件: .eslintrc.js =
    - 可使用eslint-disable-line, eslint-disable-next-line, eslint-disable禁用对某些代码的检测
    - 更新webpack配置文件: webpack.config.js =
    - 执行：webpack查看运行结果
  - 5. 利用babel进行js兼容性处理
    - 为了解决高级es6+规范的js代码在各大浏览器的兼容性，引入Babel
    - 安装babel-loader, @babel/core和@babel/preset-env: yarn add babel-loader @babel/core @babel/preset-env -D
    - 增加babel配置文件: babel.config.js =
    - 更新webpack配置文件: webpack.config.js =
    - 执行：webpack查看运行结果(在浏览器中测试)
    - 目前的兼容性处理只是针对最基础的兼容性问题，无法处理其他高级语法(如Promise等)，需引入Polyfill
    - 安装@babel/polyfill: yarn add @babel/polyfill (注意: 此处没有加-D选项)
    - 在源码中引入polyfill: import '@babel/polyfill' // 优先其他用引入
    - 全部引入polyfill会造成js打包体积大大，且包含了很多未使用的兼容性代码，因此需要按需加载polyfill，需引入core-js
    - 安装core-js: yarn add core-js (注意: 此处没有加-D选项)
    - 更新babel配置文件: babel.config.js =
    - 由于已添加按需加载，因此在源码中注释掉原那段polyfill: import '@babel/polyfill'，同时移除@babel/polyfill: yarn remove @babel/polyfill
  - 6. js和html压缩
    - 更新webpack配置文件: webpack.config.js(修改mode: production) =
    - 执行：webpack查看运行结果(js代码将会自动压缩)
    - 更新webpack配置文件: webpack.config.js(修改html-webpack-plugin插件选项) =
    - 需要注意的是，对于html压缩，html-webpack-plugin@4.0.0后将不需再配置minify+collapseWhitespace和removeComments选项，只需要配置mode为production即可
  - 7. 生产环境基础配置
    - 集成性能优化1-6环节中所有的配置(提取css到单独文件,css兼容性处理, 压缩js/css/html, eslint代码检查, babel代码兼容性处理)
    - 搭建对应的js/css/img/font存放对应的资源文件
    - 更新webpack配置文件: webpack.config.js =
    - 需要特别注意的是：目前由于os已单独分离相对于开发环境基础配置来说，打包后会多出一个css文件夹，会出现图片引用路径问题，解决方法是：为MiniCssExtractWebpackPlugin添加publicPath选项 =
    - 执行：webpack查看运行结果
    - 经历这7步，一个基本的webpack生产环境配置就完成了
- 性能优化
  - 1. HMR热加载(开发环境)
    - 执行：npx webpack-dev-server, 此时更改任何模块(js/css)都会将整个应用被重新打包而使页面刷新
    - 更新webpack配置文件: webpack.config.js(开启devServer的hot选项) =
    - 此时已经开启HMR(Hot Module Replacement), 首先更改任何的css/less/scss文件，会发现页面局部刷新(style-loader内部已经支持HMR)
    - 继续更改任何的js代码，会发现页面仍然被整体刷新(js默认不支持HMR)，但可添加代码支持
    - 在入口js(index.js)加入需要支持热加载的模块，在四调代码中执行该模块对应的功能，这样就启用热加载了(但入口js本身不支持热加载) =
    - 至于是否需要对html文件开启热加载功能，可以实现但没有必要，因为热加载只针对同一类型的多个模块发生改变时只单独打包改变的文件而使页面局部刷新，而index.html只有一个所以没必要支持，实在要支持可以将index.html加入到webpack的入口配置(entry)中
  - 2. sourcemap(源代码映射)
    - 源代码映射技术，解决打包文件错误调试问题，主要从打包速度和错误友好性两方面来对开发环境和生产环境分别考虑。
    - 更新webpack配置文件: webpack.config.js(devtool选项) =
    - 格式：[inline-|hidden-|eval-][nosources-][cheap-[module-]]source-map
    - inline-source-map 将source-map只包生成一个内联嵌入到bundle.js中, 打包速度更快，能准确提供源代码错误信息和错误位置行，列均提供
    - eval-source-map 将source-map对每个模块都会生成一个内联嵌入到bundle.js中，打包速度最快，能准确提供源代码错误信息和错误位置行，列均提供
    - hidden-source-map 将source-map单独生成一个外部bundle.js.map文件，只能提供构建后代码的错误位置，无法提供源代码的任何错误位置(隐藏源代码)
    - nosources-source-map 将source-map单独生成一个外部bundle.js.map文件, 只告知出现错误了，但不提供任何构建后和源代码的错误信息(隐藏源代码和构建代码)
    - cheap-source-map 将source-map单独生成一个外部bundle.js.map文件, 能准确提供源代码错误信息和错误位置(但仅限于错误所在行)
    - cheap-module-source-map 基本与cheap-source-map功能一致，提示相对友好一些，会将其他模块(loader)的source-map也加入进来
    - 从打包速度来看，eval > inline > cheap > ...,
从提示友好性来看，source map > cheap module > cheap > ...,
从安全性来看，nosources > hidden > ...
    - 开发环境：选用eval-source-map(推荐) | eval-cheap-module-source-map(打包速度快，提示更友好);
生产环境：选用souce-map(推荐) | nosouces-source-map(安全性更高，但无法快速定位错误)
  - 3. oneOf(生产环境)
    - 正常来说，每个源文件都会经过loader的rules规则来匹配对应的打包规则(如loader会在开发环境下对图片压缩，但每个rules都会走一遍这次文件也会走js的规则检验)，这样耗费性能。为了解决生产环境打包构建速度，引入OneOf
    - 更新webpack配置文件: webpack.config.js(更新module选项) =
    - 但值得注意的是: oneOf只能针对同一类资源的唯一规则生效，对js资源来说，既要eslint-loader，也要babel-loader，因此将eslint-loader移出oneOf(单独作为一个rule来进行处理
    - 执行：webpack查看运行结果
  - 4. 缓存(生产环境)
    - 更新webpack配置文件: webpack.config.js(开启babel构建时缓存) =
    - 如果此时代码发生变动(js/css)并使用webpack重新构建，再次刷新浏览器发现不会发生任何变化，因为此时静态资源文件已被缓存(第二次从缓存中读取)，这将带来此时新的特性无法快速更新的问题
    - 因为资源是否缓存是根据文件名是否发生化来决定，因此可以代码发生变化了动态修改文件名(添加hash后缀)
    - 更新webpack配置文件: webpack.config.js(修改js/css输出文件名规则) =
    - 为了能测试缓存是否生效，首先安装express：yarn add express并新增server.js同时启动 node server.js =
    - 执行：webpack重看运行结果，发现文件名已被添加hash后缀(每次webpack构建都会产生一个hash值，这里取前8位)
    - 此时刷新浏览器发现改动编码被更新，但由于一旦代码有变动每次构建都会产生的新的hash值，导致浏览器缓存js/css缓存同时失效，因此需要采用chunkhash来解决，再次更新webpack配置文件: webpack.config.js并运行webpack
    - 此时发现构建完成后，js/css的缓存hash不再是webpack构建完成后产生的hash值了，但是引入js/css缓存的一个chunk值(此hash值仍然一样，这又会引起另一个问题，就是如果只有js/css其中一个发生变动，另一个的缓存也会跟着变化而失效。
    - 为了应对js/css文件单独发生变化后每次可以重新构建而浏览器缓存又不会失效，此时需要使用contenthash(根据文件内容来决定是否更新)，再次更新webpack配置文件: webpack.config.js并运行webpack =
    - 为了能解决在打包过程中，去掉一些从未引用过的代码(js/css)，webpack提供tree shaking"摇摆"功能，但必须有两个前提条件：1. 满足ES6模块化规范；2. 开启production环境(mode: production)
  - 5. Tree Shaking(生产环境)
    - 新增一个模块calc.js并在入口index.js中引入(只引入其中的add方法)，使用webpack重新构建 =
    - 构建完成后，打开dist/js/bundle.js搜索"+"可以查到，而"*"却无法找到，此时证明tree shaking已生效(mul方法未引用，因此去掉代码)
    - 注意：由于webpack不同版本对tree shaking支持程度不一，可能会造成有些副作用的功能失效(比如css,@babel/polyfill，因为这些资源并未在js代码中直接使用)
    - 为了测试副作用失效的问题，更新package.json(添加sideEffects: false)再次打包后运行会发现css已不再生效(因为被tree shaking了)，所以可将sideEffects配置为不想被tree shaking的模块 =
    - 由于webpack打包构建后只产生一个bundle.js，随着开发功能越来越多而复杂，导致bundle.js打包后非常巨大，也不能实现按需加载，因此需要引入Code Split代码分割技术。
  - 6. Code Split(生产环境)
    - 代码分割有以下几种处理方式：多入口打包(MPA)，分支拆分(Split Chunks)，动态引入(import)
    - 更新webpack配置文件: webpack.config.js(修改entry和output/filename) =
    - 运行: webpack查看dist/js发现拥有了打包文件(使用filename重命名了)
    - 为了测试分支拆分配置，引入jquery: yarn add 'jquery' 并在入口index.js中引入，此时直接打包，会发现bundle.js中已包含jquery的代码，体积陡然变大
    - 更新webpack配置文件: webpack.config.js(修改optimization) =
    - 为了支持动态引入配置，安装babel-eslint和@babel/plugin-syntax-dynamic-import: yarn add babel-eslint @babel/plugin-syntax-dynamic-import -D
    - 更新babel.config.js, .eslintrc.js和index.js(注意魔法注释语法) =
    - 更新webpack配置文件: webpack.config.js(将改为单入口) =
    - 这时如果更改calc模块(index模块保持不变)重新打包，会发现两个模块都产生了contenthash值有导致index模块也被重新打包，其原因是index打包文件中包含了calc的contenthash值，因此需要将calc的contenthash移出去
    - 更新webpack配置文件: webpack.config.js(修改optimization) =
  - 7. 懒加载和预加载(生产环境)
    - 为了支持js模块懒加载(不同于图片懒加载), 有些js模块可能不是应用一启动就开始加载，首是要通过某种操作之后才需要加载，引入懒加载(lazy load)技术
    - 更新index.html, .eslintrc.js和index.js =
    - 运行：webpack发现calc模块并未一开始就加载而是点击按钮之后才加载，实现上也是利用了动态引入(import)的特性
    - 但懒加载可能有时要到运行时才加载，当加载的资源(如弹框，tooltip等)还够多时可能会引起页面卡顿，因此可预先将资源加载过来，引入预加载(prefetch)技术(存在兼容性问题，有些浏览器并不一定支持)
    - 更新index.js(魔法注释中 webpackChunkName:'calc', webpackPrefetch: true */) =
    - 运行: webpack发现calc模块也同index模块一并载入了，当点击按钮就会立即执行(不会有卡顿的可能)，需要重点说明的是，calc模块一定是在页面其他资源ready之后，才会利用浏览器的空闲时间开始加载(不是一开始就会加载)
    - 为了能让应用可以像app一样可离线访问，提高更好的用户体验，引入PWA(Progressive Web App)技术，但存在一些兼容性问题，也可作为一种提升运行性能和改善用户体验的渠道。
  - 8. PWA(生产环境)
    - 安装workbox-webpack-plugin: yarn add workbox-webpack-plugin -D
    - 更新webpack配置文件: webpack.config.js(创建ServiceWorker) =
    - 由于ServiceWorker只能运行在服务器上，因此新增server.js同时启动: node server.js =
    - 打开浏览器时选择network选项工具栏(disable cache之后的下拉选项)并设置为Offline状态，此时发现页面仍然是可以正常访问
  - 9. 多进程打包(生产环境)
    - 随着开发功能越来越多而复杂，为了加快构建速度，引入多进程打包(一般仅对于js资源作处理)技术，但需要谨慎开启多进程打包(进程开启与通信还是有开销的)
    - 安装thread-loader: yarn add thread-loader -D
    - 更新webpack配置文件: webpack.config.js(为js模块添加thread-loader) =
    - 因此可能针对应用中常复杂的情况下，thread-loader才会发挥很大的作用，对于非常简单的应用，没有必要开启多进程打包，同样不仅仅只针对js资源，其他资源也可以使用
  - 10. 外置模块external(生产环境)
    - 有时应用会引入第三方库(如jQuery), 而不想这些库被webpack打包, 可将这些第三方库放到external(外置模块)中
    - 更新webpack配置文件: webpack.config.js(新增external)并运行: webpack查看结果 =
    - 值得注意的是，因为第三方库被忽略打包，如果在应用中使用的话必须提前在index.html中提前引入(一般以cdn方式) =
    - 除了externals技术，有时如果第三方库我们也需要打包(如vue, react等)应用本身还是分离打包，将同一类型的第三方包打包在一起(如vue, vue-router, vuex等)，引入dll动态链接库技术
    - 为了能与externals进行对比，另外安装vuex,vuex-router: yarn add vue vuex vue-router
    - 为了更好的代码压缩性能，安装terser-webpack-plugin: yarn add terser-webpack-plugin -D
    - 既然要将第三方库单独打包，则同时需要单独的webpack配置文件: webpack.dll.config.js(引入webpack.DllPlugin) =
    - 运行: webpack --config webpack.dll.config.js进行第三方打包文件(先先webpack执行)
  - 11. 动态链接库dll(生产环境)
    - 更新webpack配置文件: webpack.config.js(引入webpack.DllReferencePlugin)并运行: webpack查看结果(此时会报错)
    - 由于第三方库不参与应用一起打包了，那启动应用就找不到对应的第三方库，还需要安装add-asset-html-webpack-plugin: yarn add add-asset-html-webpack-plugin -D
    - 更新webpack配置文件: webpack.config.js(引入AddAssetHtmlWebpackPlugin)并运行: webpack查看结果(此时正常) =
  - 12. 总结
    - 开发环境
      - 构建速度
        - HMR
      - 代码调试
        - eval-source-map
    - 生产环境
      - 构建速度
        - souce-map
        - oneOf
        - babel缓存
        - 多进程打包
        - externals
        - dll
      - 文件缓存
        - hash
        - chunkhash
        - contenthash
      - 运行性能
        - tree shaking
        - code split
        - lazy load
        - prefetch
        - pwa