前端模块化webpack

模块化的演进过程

- CommonJs

  暴露模块：module.exports = value或exports.xxx= value

  引入模块：require(xxx),如果是第三方模块，xxx为模块名；如果是自定义模块，xxx为模块文件路径

  加载机制：输入的是被输出的值的拷贝。也就是说，一旦输出一个值，模块内部的变化就影响不到这个值

  主要应用：服务器端，模块的加载是运行时**同步加载**的

- AMD

  定义暴露模块：

  ```
  //定义有依赖的模块
  define(['module1', 'module2'], function(m1, m2){
     return 模块
  })
  ```

  引入模块：

  ```
  require(['module1', 'module2'], function(m1, m2){
     使用m1/m2
  })
  ```

  AMD模块定义的方法非常清晰，不会污染全局环境，能够清楚地显示依赖关系。AMD模式可以用于浏览器环境，并且允许非同步加载模块，也可以根据需要动态加载模块

- CMD

  主要应用：专门用于浏览器端

  加载机制：模块的加载是异步的，模块使用时才会加载执行

  CMD规范整合了CommonJS和AMD规范的特点

- ES Modules

  编译时加载

前端模块化规范的最佳实践方式基本实现了统一

- 在Node.js环境中，遵循CommonJS规范来阻止模块
- 在浏览器环境中，遵循**ESModules规范**（当前最主流的前端模块化标准）

前端模块化打包工具

当前市场主流工具：**Webpack**、**Parcel**、**Rollup**

Webpack

注意⚠️：配置文件是运行在Nodejs环境下的

Loader机制

Loader机制是webpack最核心的机制，loader就是一个加载器，用来加载处理不同的文件类型/特殊类型资源

Any sorece——>Loader1——>Loader2——>Loader3——>Javascript Code

插件机制

目的：增强Webpack在项目自动化构建方面的能力

CleanWebpackPlugin：清除上一次的打包结果，避免文件堆积

HtmlWebpackPlugin：将根目录的index.html也打进包中

CopyWebpackPlugin：复制一些固定存在的文件

webpack在整个打包过程中：

- 通过Loader处理特殊类型资源的加载，例如加载样式、图片
- 通过plugin实现各种自动化的构建任务，例如自动压缩、自动发布

工作原理剖析

- webpack cli启动打包流程
- 载入webpack核心模块，创建Compiler对象
- 使用Compiler对象开始编译整个项目
- 从入口文件开始，解析模块依赖，形成依赖关系树
- 递归依赖树，将每个模块交给对应的Loader处理
- 合并Loader处理完的结果，将打包结果输出到dist目录

webpack-dev-server

是Webpack官方推出的一款开发工具，提供了一个开发服务器，并且将自动编译和自动刷新浏览器等一系列对开发友好的功能全部集成在一起

作用：提高本地开发效率、模块热替换

工作流程：开始——>启动HTTP服务——>Webpack构建——>监听文件变化

注意⚠️：Webpack构建——>内存——>HTTP Server（为了提高工作效率，webpack并没有将打包结果写入磁盘中，而是暂时存放在内存当中，降低磁盘读取的时间消耗）

配置SourceMap

webpack支持sourceMap

举例

- inline-source-map模式

与普通的source-map相同，只不过这种模式下SourceMap文件不是以物理文件存在，而是以data Urls的方式出现在代码中

- hidden-source-map模式

在开发工具中看不到SourceMap的效果，但是确实是生成了的

- nosources-source-map模式

可以看到错误出现的行列信息，但是点进去看不到源码

模式名称的规律

- cheap代表只展示错误位置的行信息，不展示列信息
- module代表解析出来的源码是没有经过Loader加工的

使用建议

- 开发环境：推荐使用cheap-module-source-map，因为大多数情况下我们需要调试loader转换前的源代码，且通常编写代码每行不会超过80个字符，所以定位到行即可，而且省略行信息还可以提升构建速度
- 生产环境：不使用，因为sourcemap会暴露源代码到生产环境，不安全，如果实在是对自己的代码没有信心，可以使用nosources-source-map模式，这样出现错误可以定位到源码位置，但是不至于暴露源码

高级特性

Tree shaking

- 本质是借助 ES module 的静态分析能力来消除无用的 js 代码的
- 并不是指webpack中的某一个配置选项，而是一组功能搭配使用过后实现的效果，这组功能在生产模式下都会自动启用，所以**使用生产模式打包就会有Tree-shaking的效果**
- 其他环境如何手动开启Tree shaking

  usedExports：true // 打包结果中只导出外部用到的成员

  minimize：true // 压缩打包结果
- 注意⚠️：Tree shaking的使用前提是ES Module，也就是说最终交给Webpack打包的代码，必须是使用ES Module的方式来组织的模块。但是在我们使用babel-loader的时候，会将代码中的ES Module转换为Common JS，这时Tree shaking可能会失效，什么情况下不会导致失效呢？最新版的babel-loader已经自动关闭了对ES Module的转换插件。为了确保babel-loader不会影响Tree shaking，我们可以将babel-loader的配置presets的modules：false