Vite 详细介绍

Vite、Webpack 编译方式有啥区别?

- **Webpack：打包编译**
  - 从入口文件开始递归解析所有模块的依赖关系，然后对这些模块进行编译、打包。即：不管浏览器是否 有请求某个模块，都会对所有模块打包。
    - 缺点：项目规模越大，启动时间就越长。
  - 热更新：增量编译！需要对发生变化的模块及其依赖进行编译。不过项目越大，依赖链越大，重新编译范围越大，导致**热更新速度随着项目规模的增大而明显变慢**。
- **Vite：按需编译**
  - 只有当浏览器通过import语句请求某个模块时，Vite 才会对该模块进行编译处理，然后返回给浏览器。
    - 优点：避免了对未被请求模块的无效计算，极大地加快了开发服务器的启动速度。
  - 热更新：仅对修改的单个模块进行编译，不处理依赖链，更新速度与项目规模关联不大，

一、Vite 有什么优缺点?

Vite 的核心优势：**开发阶段的极速体验**

> 按需编译（启动快、热更新快）、零配置支持 TS/CSS/JSX、原生支持ESM/import()/async、生产环境基于Rollup打包 / 自动压缩资源

优点：

- **开发体验极致高效**
  - **按需编译**：浏览器请求时才处理模块，避免无效计算。需要基于esbuild进行转译（TypeScript → JavaScript、JSX/TSX → JavaScript）
  - **启动速度极快**：无需预打包，开发服务器毫秒级启动，大型项目优势明显（相比 Webpack 快 10-100 倍）。
  - **热更新实时响应**：仅重新编译修改的模块，更新速度不受项目规模影响（通常 < 100ms）。
- **零配置开箱即用**：内置对 JS、CSS、TypeScript、JSX、静态资源的支持，无需手动配置 loader 或 plugin，基础项目可直接启动
- 原生支持现代特性
  - 直接利用浏览器 ESM 能力，支持 import() 动态导入、顶层 await 等语法。
  - `用 esbuild 处理 TypeScript/JSX（比 Babel 快 10–100 倍）`，编译效率极高。
- **生产构建优化充分**
  - **基于 Rollup 打包，自动实现代码分割、Tree-Shaking、压缩等优化，产物体积通常比 Webpack 更小**（Rollup 对 Tree-Shaking 更高效）。
  - **自动处理资源压缩、代码分割等**，无需额外配置。

缺点：

- 依赖浏览器原生 ESM 支持：
  - 开发环境中对 IE 等旧浏览器兼容性差（生产环境可生成相关兼容代码）。
  - 若项目中有大量非 ESM 模块，预构建环节可能会有兼容性问题。
- 生产构建灵活性差：基于 Rollup 打包，对于多入口、复杂代码分割策略的支持没 Webpack 灵活。
- 插件生态不如 Webpack 成熟，主流框架(Vue、React)插件完善，小众需求插件支持度不足。

Vite适合追求极致开发体验的团队、中小型项目或快速原型开发、基于 ESM 规范的项目。

疑问❓：vite不需要打包?

答：在开发环境下无需预打包（这里的不打包仅仅是针对 业务代码），生产环境仍会进行打包优化。

1. 开发环境（不打包，基于原生 ESM）
   - 依赖预构建：
     - 通过 esbuild **预构建**第三方依赖(如：React、Vue）。 预构建的好处： ① 解决兼容性问题，如将**不同格式**的模块统一转换为 **ESM 格式**； ② 加快首次加载速度：预构建结果**缓存**到 `node_modules/.vite 目录`，当浏览器首次请求相关模块时，可直接从缓存中取，仅在**依赖变更时重新构建**。 ③ **合并多个依赖模块为单个文件**，减少浏览器请求次数。
     -
   - 开发服务器：
     - 启动一个基于 Koa 的本地服务器，过 WebSocket 监听文件变化，当文件修改时，仅重新编译该模块。
     - HMR 模块替换：利用 ESM 的动态导入特性
2. 生产环境（基于 Rollup 打包） 执行打包流程，将所有模块合并为优化后的静态资源，然后输出可直接部署的 dist 目录。常见的优化方案：
   - 代码压缩（使用 esbuild 或 terser）。
   - Tree-Shaking 移除未使用代码。
   - 代码分割（按路由或组件拆分 chunk）。
   - 处理 CSS 提取、图片优化等资源转换。

二、Vite、Webpack 的区别：

维度	Webpack	Vite
底层原理	基于「打包器（bundler）」：将所有模块预打包成 bundle 后再提供给浏览器。	基于「原生 ESM + 构建器」： 开发时不打包，利用浏览器原生 import 加载模块； 生产环境用 Rollup 打包。
开发启动速度	较慢。需递归解析所有依赖并打包成 bundle，项目越大启动越慢（大型项目可能需要数十秒）。	极快。无需预打包，**启动时仅启动服务器，毫秒级响应**（大型项目通常只需 1-2 秒）。
热更新（HMR）	**需处理依赖链**：修改一个模块时，需重新编译该模块及依赖它的所有上游模块，更新速度随项目规模下降。	精准更新：仅重新编译修改的单个模块，通过 WebSocket 通知浏览器直接替换，更新速度稳定（通常 < 100ms）。

维度	Webpack	Vite
配置复杂度	较高。需手动配置 loader（处理不同类型文件）和 plugin（扩展功能），基础配置文件可能需要数十行代码。	极低。内置对 JS、CSS、TS、JSX 等的支持，零配置即可启动项目，复杂需求也只需少量配置。
模块处理方式	支持多种模块规范（ESM、CommonJS、AMD 等），但需通过 babel 等工具转译后打包。	优先支持 ESM，开发时直接传给浏览器原生解析；对 CommonJS 模块（如 node_modules 中的依赖）会预构建为 ESM。
生产构建工具	自身负责生产打包，通过 webpack.optimize 等配置优化产物。	开发时用原生 ESM，生产环境基于 Rollup 打包（Rollup 对 Tree-Shaking 和代码分割的处理更高效）。
适用场景	复杂项目（多入口、特殊模块转换、深度定制构建流程）、需兼容旧浏览器的项目。	中小型现代项目（Vue/React/TypeScript）、追求开发效率的场景、基于 ESM 的项目。
生态成熟度	生态极丰富，插件和 loader 覆盖几乎所有需求，适合处理边缘场景。	生态快速成长，官方插件覆盖主流框架（Vue/React），但部分小众需求可能缺乏成熟插件。

三、vite常见配置

Vite 的配置通过项目根目录下的`vite.config.js 文件`进行，使用 **defineConfig** 函数来定义配置对象。

Vite **内置对 TypeScript 的支持**，可直接创建 .ts 文件即可使用，无需额外插件。

```
import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import react from '@vitejs/plugin-react'

export default defineConfig({
  base:'/',  // 公共基础路径，部署时的路径!
  // base修改后，由 JS 引入的资源 URL，CSS 中的 url() 引用以及 .html 文件中引用的资源在构建过程中都会自动调整，以适配此选项。

  plugins: [
    vue(),  // 支持 Vue 单文件组件（.vue）
    react(),  // 支持 React 的 JSX 和热更新等功能
  ],

  // 开发服务器配置（server），只在本地生效；构建生产环境（vite build）时会完全忽略该配置。
  server: {
    open: true
    port: 3000,
    proxy: {
      '/api': {
        target: 'http://localhost:8080', // 目标服务器地址
        changeOrigin: true, // 开启跨域
        rewrite: (path) => path.replace(/^\/api/, '') // 重写路径
      }
```

```
      }
    }
  })
```

Vite 内置了对静态资源的基本处理能力，开箱即用。

Webpack 处理静态资源相对繁琐，需要手动配置各种 loader 和 plugin；但 Webpack 的优势在于其高度灵活的配置体系，可以针对各种复杂的需求进行定制化处理。

1. server跨域解决：

- 你的前端项目（本地开发）：运行在 `http://localhost:3000`（域名 A）
- 你需要请求的接口：部署在 `https://test-api.example.com`（域名 B）

前端页面（域名 A）和接口（域名 B）不同源，因此接口请求会报错！

Vite 的 server.proxy 配置会在本地启动一个“代理服务器”（和前端页面同域 localhost）。具体过程：

- 前端 → 代理服务器

  > 同源请求”（都是 http://localhost:3000），不会拦截。

- 代理服务器 → 目标接口服务器

  > “服务器之间的通信”，服务器没有跨域限制（跨域限制是浏览器特有的安全机制）

- 代理服务器 → 前端

`changeOrigin: true`：本地代理服务器在转发请求时，会 修改请求里的Origin字段，让其和目标服务器同域。因为有的服务器为了防止 CSRF 攻击（跨站请求伪造），可能会做同源验证。不过该行为并不是强制的，是服务器主动配置的。

如果目标服务器主动做了 Origin 校验，即：服务器会配置一个可信列表，只有在可信列表中的请求 它才会响应。这时，如果上述的代理服务器 不修改 Origin的话，目标服务器就会因为 “同源的 主动校验逻辑” 拒绝这个请求！

2. CSS处理：Vite 内置对 CSS 的支持，无需额外配置

- 直接导入 CSS 文件：`import './style.css'`
- 支持 CSS 模块：命名为 `xxx.module.css`，导入后自动转为**模块化对象**(避免样式冲突)。
- 支持**预处理器（Sass/Less）**：*安装对应依赖*即可直接使用

```
npm install sass  # 支持 .scss/.sass 文件
```

3. 静态资源处理

- 直接在代码中导入静态资源（图片、字体等），Vite 会自动处理。`import logo from './assets/logo.png'`
- public 目录的资源 会被原样复制到输出目录(dist)，可通过根路径（'/'）访问

4. 常用插件推荐

- vite-plugin-pwa：添加 PWA 支持
- vite-plugin-compression：生产环境资源压缩（gzip/brotli）
- unplugin-auto-import：自动导入 API（如 Vue 的 ref、reactive）
- vite-plugin-html：动态修改 HTML 内容（如注入环境变量）

5. JS压缩：

生产环境默认使用 esbuild 进行压缩，速度快。 如果需要更高级的压缩功能，如使用 terser 压缩 JavaScript，可以通过 build.minify 和 build.minifyOptions 进行配置。

```
export default defineConfig({
  build: {
    minify: 'terser',
    minifyOptions: {
      // 可传入 terser 相关配置
    }
  }
})
```

疑问❓：Vite的图片资源啥的会压缩吗?

- 开发环境下默认不压缩，直接通过原生 ESM 机制按需加载。

  > 开发阶段的核心目标是**快速响应代码变更（热更新）**，压缩图片只会增加时间，影响开发效率。

- 生产环境下会使用 esbuild 对图片进行基础压缩。若想实现更强的压缩效果(尺寸、格式、体积)，可通过插件```vite-plugin-image-optimizer 实现。

四、Public目录：

参考：网站部署目录、域名

用于存放那些**不需要参与构建过程的静态资源**。例如：网站的图标。

- Vite 中的 public 目录：
  - 资源原样复制：放在该目录下的资源，在项目构建时，会被**原样复制**到最终的输出目录（默认是dist目录），不会经过任何构建处理（如压缩、哈希重命名等）。
  - 可以直接通过 **根路径（/）** 来访问这些资源
- Webpack 中模拟 public 目录功能：
  Webpack 本身没有原生的public目录机制，需借助`html-webpack-plugin`和`copy-webpack-plugin`插件来实现：
  - copy-webpack-plugin：负责**将指定目录（可自行命名，但功能类似public）下的文件/文件夹，复制到 Webpack 的输出目录中**
  - html-webpack-plugin：用于生成 HTML 文件，并自动注入打包后的 JavaScript 和 CSS 文件，可以在 HTML 模板文件中按照“根路径”的方式引用从public目录复制过来的资源。

疑问❓：**部署一个网站后，首页访问的是public目录?**
答：不是，Vite 等工具的「部署流程」有一个关键前提：**先执行构建命令（npm run build），生成可直接部署的「产物目录」（默认是 dist 文件夹）**，最终上传到服务器的是这个 dist 目录，而非源码中的 public 目录。这是生产环境的情况。

> 若是开发环境，通过 `http://localhost:5173` 访问，本质是 Vite 开发服务器直接读取 `public/index.html`作为入口；

注意📣：**index.html 在引用 public 中的文件时，需使用根路径（以 / 开头）**，因为 public目录中的文件不参与构建打包 是直接复制到 dist 根目录的