- 包管理工具
  - npm
    - install 原理
      - npm从registry仓库中下载包时，其实是下载一个压缩包
        这个压缩包有对应的配置文件来记录压缩包的对应信息
      - 1 、执行了npm i webpack -D时，会先去查找配置文件
        2 、看配置文件有没有对应下载记录，有则会根据配置文件进而找到已经下载过的压缩包，解压到我们对应的node_modules文件夹中
        3 、没有下载记录则去registry仓库进行下载(版本一致会直接从缓存中获取，版本不一致则下载新内容)
      - 极大优化下载流程，缓解本地带宽、提高下载使用效率、降低registry仓库的高并发压力
      - package.lock.json
        1 、会在我们第一次安装库时，自动生成，该文件记录了对应所有的详细版本，关键记录来自构建依赖关系
        2 、第一次比如下载 axios，他会依赖其他库，会构建一条完整的关系链并记录好，再去下载
        3 、存在锁文件，后续使用npm install进行安装时，都会通过锁文件去安装一致的版本(查找缓存)，找到则直接从本地获取，没找到则取registry仓库下载，最终解压缩到node_modules文件夹
        4 、下载版本不一致(例如我们明确下载更新版本)，则重新构建依赖关系，去仓库下载压缩包解压缩到node_modules，然后重新构建lock锁文件
      - 查看缓存 npm get cache
        - content-v
          - 用于存储项目依赖项或缓存文件，这样做的好处是可以避免不同版本或内容的依赖之间产生冲突，并确保每个缓存文件是唯一的且不会覆盖其他版本的文件
        - index-v5
          - 多层的哈希文件夹
          - index-v5是一个索引目录，记录content-v2的一个索引或者说是位置，也就是name + version + integrity(完整校验值)的一个哈希值。如果lock锁文件内的这三者和index-v5能够对上，就会去content-v2找到我们缓存的那个文件
    - install 问题
      - 低版本install 问题
        - 包依赖嵌套问题
          - 公共的依赖，无法复用，占用磁盘空间大
        - 文件路径问题
          - windows 的文件路径最长是 260 多个字符，这样嵌套是会超过 windows 路径的长度限制
      - npmv3以上
        - 也采用了铺平的方式
          - 都通过 lock 文件来锁定文件版本
        - 作用：锁定依赖的具体版本，确保每次安装依赖时都能获得完全一致的依赖树（包括间接依赖）。
        - 但我现在开发的项目没有提交 lock 文件...
          可能是老项目变动不频繁
    - npx
      - npm5.2之后自带的一个命令
        解决了在项目开发过程中，频繁需要安装全局工具包所带来的困扰
      - 1 、全局安装 webpack5，项目里使用webpack4
        2 、当在命令行执行 webpack，他回去当前目录找，但是项目里的 webpack 是安装在 node_modules/bin下的，找不到就会用全局的
        - 解决办法1 、 npm run 脚本
          npm run 实质就是去 node_modules/bin下面去找可执行命里，自然就用的局部的
        - 解决办法 2 、 npx
          npx替代npm，查找顺序会和scripts属性一样，先从当前node_modules文件夹下寻找
      - 举个例子
        - bin文件夹下，存在众多可执行命令，其中有三种不同后缀的命令，不管是webpack还是vite或者其他工具都一样
          1、.cmd 文件用于 Windows 的 CMD
          2 、.ps1 文件用于 Windows 的 PowerShell
          3 、而没有后缀的文件 是为 Linux、macOS 等 Unix 系统设计的可执行脚本，通常通过 Shell（如 Bash）直接执行
        - 1 、npm run命令首先在当前项目的 node_modules/.bin 目录中查找 运行文件的路径 的可执行命令
          2 、如果在项目的 node_modules/.bin 中没有找到，它会尝试在全局的 node_modules 目录中查找
          3 、如果全局目录中也没有，npm 会继续在系统的环境变量中查找
          4 、如果这些位置都没有找到，npm 会报错，指出无法找到命令
      - npm run 原理
      - npm link
      - 发布包
        - dependencies、devDependencies、peerDependencies
          - dependencies 生产依赖
            - 安装时包默认会将包添加到生产依赖，这是因为放在生产依赖下一定不会出问题，顶多就是构建时候多一些包
            - npm install时，dependencies和devDependencies都会被安装，
              但是如果在生产环境下（例如运行npm install –only=prod或设置NODE_ENV=production环境变量时），devDependencies不会被安装
          - devDependencies 开发依赖 | 生产依赖
            - //开发环境依赖设置
              npm install xxx –save-dev//全写
              npm install xxx –D//简写
          - peerDependencies对等依赖
            - 我们依赖的一个包，它必须是以另外一个宿主包为前提的，设置该项依赖来自peerDependencies属性
            - 可以理解为，如果安装 **element-plus**，而没有手动安装对应的 Vue，在 npm 7 及以上版本中，npm 会自动尝试安装符合要求的 Vue 3，并且会尝试解决版本冲突
            - 对于像 React 或 Vue 这样的框架，如果每个插件或者 UI 组件都把 React 或 Vue 作为自己的依赖项，那么在项目中会出现多个版本的框架副本，这不仅浪费空间，还可能导致版本冲突
              使用 peerDependencies 可以确保只有一个版本的宿主包（例如 Vue 或 React）存在，被依赖的包就被称为宿主包
  - 依赖版本 ^2.0.3或~2.0.3
    - 遵从semver版本规范
    - semver版本规范是X.Y.Z：
      1 、X主版本号（major）：做了不兼容的 API 修改（可能不兼容之前的版本，也称为破坏性更新）
      2 、Y次版本号（minor）：做了向下兼容的功能性新增（新功能增加，但是兼容之前的版本）
      3 、Z修订号（patch）：做了向下兼容的问题修正（没有新功能，修复了之前版本的bug）
    - ^和~的区别：
      1 、x.y.z：表示一个明确的版本号，
      2 、^x.y.z：表示x是保持不变的，y和z永远安装最新的版本
      3 、~x.y.z：表示x和y保持不变的，z永远安装最新的版本
    - ^的弹性空间比~更大，但后者更为稳定，从配置中存在的比例，可以预估一下对稳定性的需求有多高
      主版本号为 0 的特殊情况，表示该软件仍在初始开发阶段，API 可能频繁发生变化且不稳定，最好不使用
  - 文件系统
    - 软连接（符号连接）
      - 一类特殊的文件，其包含有一条以绝对路径或者相对路径的形式指向其它文件或者目录的引用，类似快捷方式(但实际并不是快捷方式)
      - 软链接是保存着某一个文件的路径，本身并没有内容，可以理解为这取出存放路径之后，就只是一个空壳，因此软链接的文件基本上不占内存，文件是0字节大小
    - 硬连接
      - 多个文件名引用
      - 一种指向文件物理数据块的指针。对于硬链接来说，文件在磁盘中的物理数据会被
        1 、默认情况下，用户不能访问磁盘
        2 、需要操作系统的文件系统进行交互，通过文件寻址的方式找到真实数据
        3 、这就是硬链接
    - 复制，硬软链接区别
      - 文件复制
        - //cmd终端输入命令
          window: copy foo.js foo_copy.js
          macos : cp foo.js foo_copy.js
        - 会生成一份新的、独立的文件（或文件夹），它们在磁盘上有各自的存储空间，互不影响。你改动其中一个，另一个不会变
      - 建立硬链接
        - window: mklink /H aaa.js bbb.js//创建硬链接的命令，aaa.js跟bbb.js建立起硬链接
          macos : ln aaa.js bbb.js
        - /H 参数用于指定创建的是一个硬链接（Hard Link），默认情况下，如果不使用 /H 参数，mklink 会创建一个符号链接（软链接）
        - 对aaa.js的改变会同时在bbb.js中生效，其中同步间隔大概在0.5-1s左右，这是数据读取并生效的时间
      - 建立软链接
        - 文件的显示中可以看出看到左下角有一个斜箭头，这是软链接的标志
        - 打开的软链接文件，不是一串地址，是正常文件内容，因为操作系统会帮我们自动解析
        - 一旦我们将bbb.js文件删除，那么aaa.js文件就打不开了也找不到了(无法读取)
  - pnpm
    - 如何解决幽灵依赖和嵌套以及相同依赖问题
      - 相同依赖
        - npm 或 Yarn 时，如果有 100 个项目，并且所有项目都有一个相同的依赖包，那么， 在硬盘上就需要保存 100 份该相同依赖包的副本
        - 使用 pnpm，依赖包将被 存放在一个统一的位置，因此：
          1 、如果对同一依赖包使用相同的版本，那么磁盘上只有这个依赖包的一份文件
          2 、如果对同一依赖包需要使用不同的版本，则仅有 版本之间不同的文件会被存储起来
          3 、项目安装软件包时， 其包含的所有文件都会硬链接到此位置，而不会占用 额外的硬盘空间
          4 、这样就可以共享相同依赖
      - 幽灵依赖，嵌套
        - 项目安装依赖时，pnpm 并不会将这些依赖的完整包复制到每个项目的 node_modules 中，而是通过一系列的硬链接和软链接来管理依赖的文件结构
        - node_modules文件夹下的bar@1.0.0软链接到.pnpm内的硬链接所在处，而该硬链接所在处会链接到真实数据所在，也就是.pnpm store之后磁盘内的真实数据
        - 所有下载的包统一存储在全局内容存储（store）中，通常是目录 .pnpm-store
        - 相当于一个中转站，接受项目的软连接然后通过硬链接去找真实的硬盘中的数据
        - pnpm7.0之后，统一的存储位置进行了更改：<pnpm home directory>/store
        - 非常重要的store命令是prune（修剪）：从store中删除当前未被引用的包来释放store的空间
        - 这样可以保持 pnpm 全局存储的整洁，避免存储过多不必要的数据
          假设在项目中更新了某个依赖包版本，而旧版本的包已经不再被任何项目引用。这时候就可以运行 pnpm store prune，删除这个旧版本的包
  - yarn
    - 铺平在同一层节约npm低版本install问题
      - 有的包还会有嵌套
        - 一个包是可能有多个版本的，提升只能提升一个，所以后面再遇到相同包的不同版本，依然还是用嵌套的方式
      - 铺平的问题
        - 幽灵依赖，也就是你明明没有声明在 dependencies 里的依赖，但在代码里却可以 require 进来
        - 如果哪天上层包删掉了，那么引入的下层包就回找不到而报错
        - 也存在浪费磁盘空间的问题