ES Module使用-原理-包管理工具npm

> ### 作者：小余同学
>
> 为什么没有01，那是因为01是讲谷歌V8引擎的原理，我看完了，但忘记写笔记了，不想重新看一遍写笔记了hh，大致来说也是很重要的一章节
>
> 需要全系列笔记请到2002XiaoYu (小余) (github.com)中自行获取，觉得不错给个star，这是对作者非常大的鼓励

(理解)前端使用模块化的方案解析

> - es modules
>   1. ECMA2015提出模块化规范
>   2. 必须浏览器本身支持es modules才能够使用
> - webpack 模块化打包工具
>   1. 如果浏览器不支持的话我们可以通过webpack进行打包，这样就算浏览器不支持也能够进行使用，因为webpack内部已经做出处理了。后续我们也会写webpack内部是怎么实现的

认识 ES Module

- **JavaScript没有模块化一直是它的痛点，所以才会产生我们前面学习的社区规范：CommonJS、AMD、CMD等，所以在ECMA推出自己的模块化系统时，大家也是兴奋异常**
- **ES Module和CommonJS的模块化有一些不同之处：**
  - 一方面它使用了**import**和**export**关键字
  - 另一方面它采用**编译期的静态分析**，并且也**加入了动态引用的方式**
- **ES Module模块采用export和import关键字来实现模块化：**
  - **export**负责将模块内的内容**导出**；
  - **import**负责从其他模块**导入**内容；
- 了解：采用ES Module将自动采用严格模式：use strict

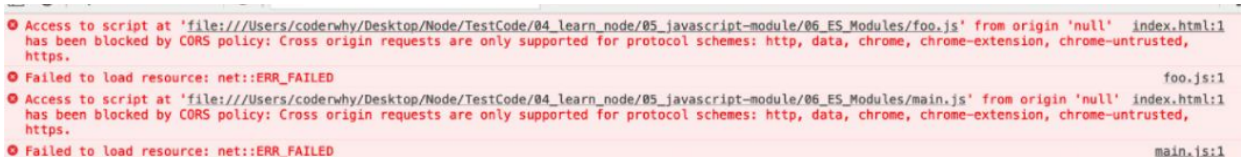
(掌握)ESModule的基本的导入导出

案例代码结构组件

- **这里我在浏览器中演示ES6的模块化开发：**

```
<script src="./modules/foo.js" type="module"></script>
<script src="main.js" type="module"></script>
```

- **如果直接在浏览器中运行代码，会报如下错误：**(不开启本地服务)



- **这个在MDN上面有给出解释：**
  - https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Modules
  - 你需要注意本地测试 — 如果你通过本地加载Html 文件 (比如一个 file:// 路径的文件), 你将会遇到 CORS 错误，因为Javascript 模块安全性需要
  - 你需要通过一个服务器来测试
- **我这里使用的VSCode插件：Live Server**

exports关键字

- **export关键字将一个模块中的变量、函数、类等导出；**
- **我们希望将其他中内容全部导出，它可以有如下的方式：**
  - **方式一**：在语句声明的前面直接加上export关键字
  - **方式二**：将所有需要导出的标识符，放到export后面的 {}中
    - 注意：这里的 {}里面不是ES6的对象字面量的增强写法，{}也不是表示一个对象的
    - 所以： export {name: name}，是错误的写法；
  - **方式三**：导出时给标识符起一个别名
    - 通过as关键字起别名

```
//方式2
//定义的时候就直接导出了，这种方式不能起别名，但有时候反而更加方便
export const name = "小余"

export function sayHello(){
    console.log("你好啊")
}

export class Person{

}
```

```
//方式3
//导出取别名
exports {
    name as fname//要导出本文件的name，取别名为fname
}

//导入
import {fname} from "./xxx"
//为什么这么做？因为导入怕原先变量名跟当前文件的变量名有冲突，而直接修改导入文件的变量名又会突兀又麻烦，所以在导出的时候就采用取别名的方式
```

import关键字

- **import关键字负责从另外一个模块中导入内容**
- **导入内容的方式也有多种：**
  - **方式一**：import {标识符列表} from '模块';

    注意：这里的{}也不是一个对象，里面只是存放导入的标识符列表内容

    ```
    import {} from "./xxx"//常用
    ```

  - **方式二**：导入时给标识符起别名

    通过as关键字起别名

    ```
    //刚刚导出的第二种方式无法在导出的时候起别名，那就可以在导入的时候起别名
    import {name as myName} from "./xxx"//常用
    //这种起别名方式更常用
    ```

  - **方式三**：通过 * 将模块功能放到一个模块功能对象（a module object）上

    ```
    //*就全部导入的意思，as foo就是给*起别名
    import * as foo from "./foo.js"//特殊情况使用
    ```

(掌握)ESModule的导入和导出方式扩展export和import结合使用

- 补充：**export和import可以结合使用**

```
//原写法
import {sum} from "./bar"
export{sum}
//优化写法1，两步省略到一步，随着使用次数的增加，省略的就会越明显，也更简洁(阅读性强)，推荐这种
export { sum as barSum} from "./bar.js"//导出来自./bar.js的sum，并起别名为barSum

//优化写法2，导入多个内容直接超级简写(简洁性强)  有总结工具文档的话使用这种
export * form "./bar.js"//导出来自./bar.js的所有内容
```

- **为什么要这样做呢?**

- 在开发和封装一个功能库时，通常我们希望将暴露的所有接口放到一个文件中(比如放到index.js里面)
- 这样方便指定统一的接口规范，也方便阅读
- 这个时候，我们就可以使用export和import结合使用

> 而且我们专门拿出一个文件用来存放导入的东西的时候，就不用到处翻来翻去的了，使用的都是统一在index.js里面，甚至文件名都可以直接省略掉，而是填入文件夹名，文件夹名就会自动推导去找自身里面的index.js，非常的科学规范

(理解)ESModule的导入和导出结合使用

default用法

- **前面我们学习的导出功能都是有名字的导出（named exports）：**
  - 在导出export时指定了名字
  - 在导入import时需要知道具体的名字

> 其实也可以不指定名字的，后续到webpack的时候再来进行讨论

- **还有一种导出叫做默认导出（default export）**
  - 默认导出export时可以**不需要指定名字**
  - 在**导入时不需要使用 {}**，并且**可以自己来指定名字**
  - 它也方便我们和现有的CommonJS等规范相互操作
- **注意：在一个模块中，只能有一个默认导出（default export）**

```
//xiaoyu文件
function parseLyric(){
    return ["歌词"]
}

const name = "小余"

//1.默认导出
export default parseLyric

//2.定义标识符直接作为默认导出(一个模块只能有一个默认导出，这里主要是为了记录所以才写在一起)
export default function(){//直接默认导出甚至可以不起函数名字，而是由导出的那方去决定名字
    return ["十万八千梦"]
}
```

```
//在另一个文件中导入
import aaaa from "./xiaoyu"//这个aaaa是随便起的名字，由于是默认导出的方式，所以取名是什么在我们导入的时候可以随便起
```

(掌握)ESModule的默认导出和导入

import函数

> 模块要放在js最顶层的，不允许在逻辑代码中编写import导入。
>
> 而且顶层的import引入的路径是不能够拼接的
>
> ```
> //错误做法
> import {name,age} from ("./foo"+".js")
> ```
>
> - 因为顶层模块是最先执行的，优于代码的执行，所以当模块开始执行的时候，字符串是还没有拼起来的，所以会报错

- **通过import加载一个模块，是不可以在其放到逻辑代码中的，比如：**
- **为什么会出现这个情况呢?**
  - 这是因为**ES Module在被JS引擎解析**时，就**必须知道它的依赖关系**
  - 由于**这个时候js代码没有任何的运行**，所以**无法在进行类似于if判断中根据代码的执行情况**
  - 甚至**拼接路径的写法也是错误**的：因为我们必须到运行时能确定path的值
- **但是某些情况下，我们确确实实希望动态的来加载某一个模块：**
  - 如果根据不懂的条件，动态来选择加载模块的路径(就是逻辑成立，我们才导入某个模块)
  - 这个时候我们需要**使用 import()** 函数来动态加载

    √ import函数返回一个Promise，可以通过then获取结果

```
if (true) {
  import sub from './modules/foo.js';
}
```

```
let flag = true;
if (flag) {
  import('./modules/aaa.js').then(aaa => {
    aaa.aaa();
  })
} else {
  import('./modules/bbb.js').then(bbb => {
    bbb.bbb();
  })
}
```

```
const flag = "作者是xiaoYu"
if(flag){//满足逻辑
    // const importPromise = import("./foo.js")
    // importPromise.then(res => {
    //     //import()返回的是一个Promise，所以我们可以这样来决定处理
    // })

    //简写
    import("xiaoyu").then(res =>{
      console.log("你说对了，作者是：",res);
    })
}
```

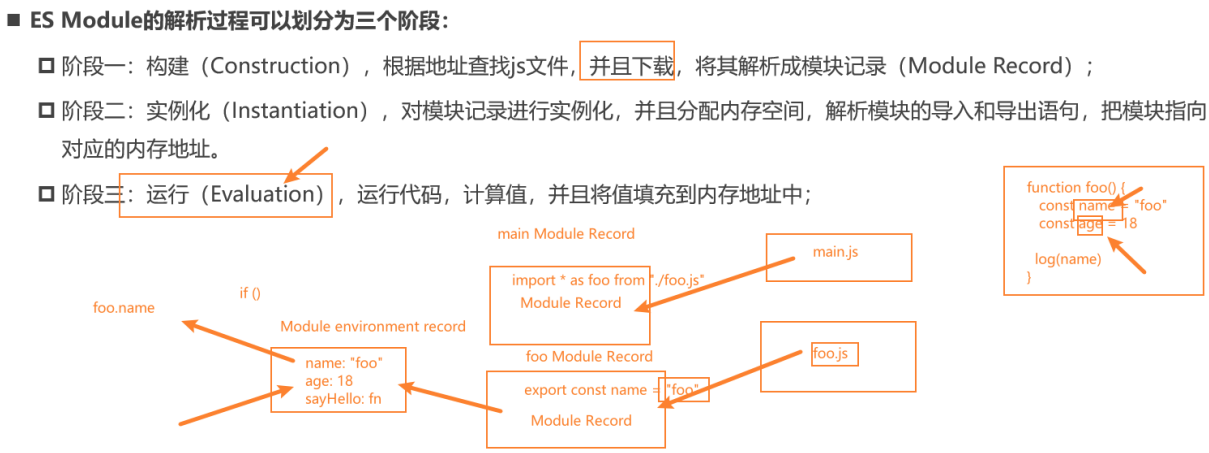
import meta

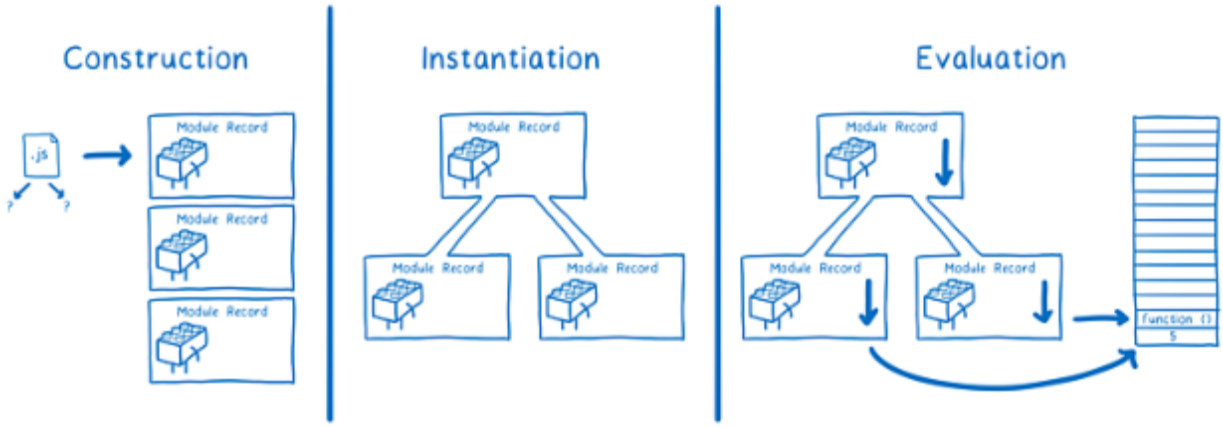
- **import.meta是一个给JavaScript模块暴露特定上下文的元数据属性的对象。**
  - 它包含了这个模块的信息，比如说这个模块的URL
  - 在ES11（ES2020）中新增的特性

(理解)ESModule的解析过程和原理

ES Module的解析流程

- **ES Module是如何被浏览器解析并且让模块之间可以相互引用的呢?**
  - https://hacks.mozilla.org/2018/03/es-modules-a-cartoon-deep-dive/
- **ES Module的解析过程可以划分为三个阶段:**
  - 阶段一：构建（Construction），根据地址查找js文件，并且下载，将其解析成模块记录（Module Record）
  - 阶段二：实例化（Instantiation），对模块记录进行实例化，并且分配内存空间，解析模块的导入和导出语句，把模块指向对应的内存地址(**模块环境记录Module environment record**)
  - 阶段三：运行（Evaluation），运行代码，计算值，并且将值填充到内存地址中

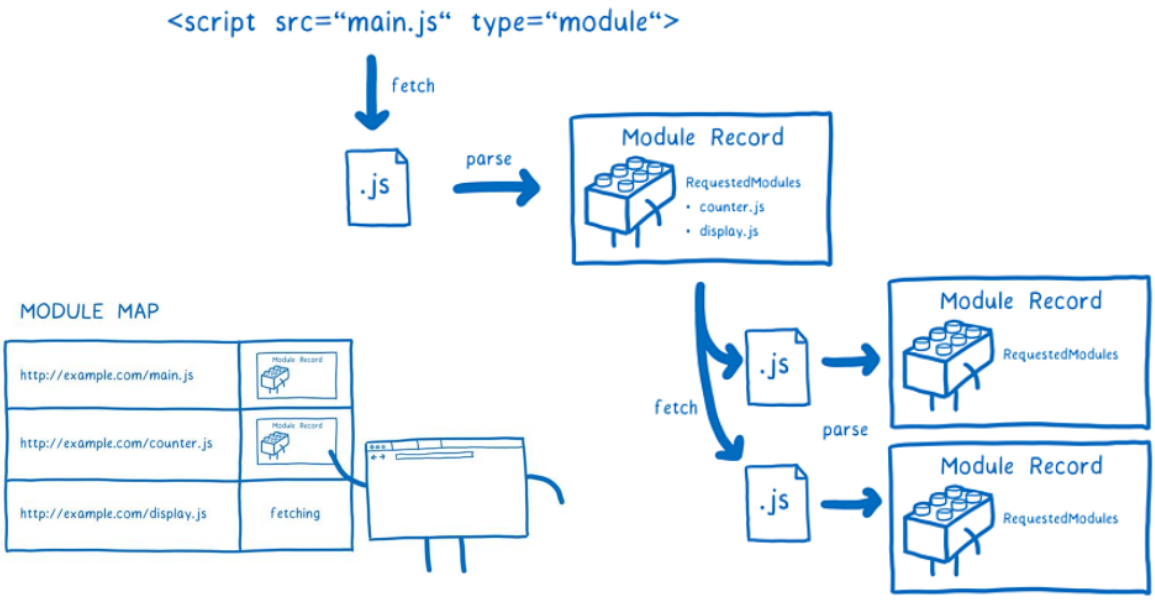




阶段一：构建阶段

1. 首先从服务器下载下来js文件，然后script的src进行fetch(获取)，使用type="module"当作模块去解析
2. 解析成Module Record，然后在这中途这个js文件解析出来的里面引入了counter.js和display.js文件，然后就又从去fetch(获取)下来，然后解析(parse)生成两个Mudule Record
3. 然后还会继续检查这两个有没有继续引入，有的话就重复上面的操作，没有的话就结束

MODULE MAP：映射关系

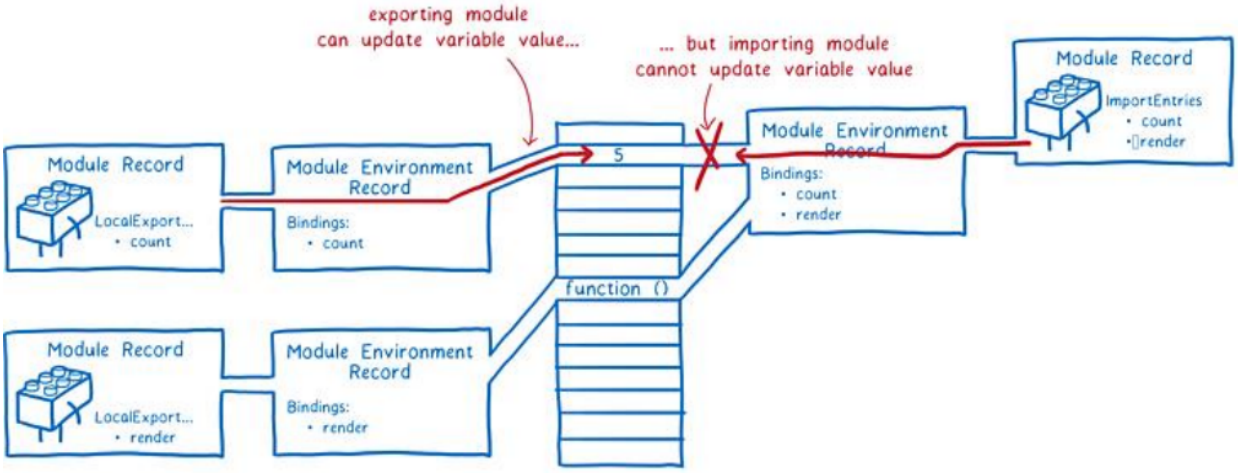


阶段二和三：实例化阶段 - 求值阶段

1. 查看模块Module Record有没有对这个模块进行导出东西，也就是LocalExport
2. 有导出就会针对导出的东西生成Module Environment Record(模块环境记录)，这里面就记录着我导出的东西，比如图中的count或者render
3. 然后对count或者render赋值具体值5或者是函数
4. 然后我们这时候再获取这值就有具体的信息了

(模块环境记录)：不是由我们创建出来的，而是由浏览器(JS引擎)生成的，这也就是说我们下面这个不是对象

```
export{//这个括号不是对象，而是导出的特殊语法，是为了告诉JS引擎我们想要导出哪些内容
  xxx
}
```



包管理工具详解npm、yarn、cnpm、npx、pnpm

(了解)包管理工具-内容概述

1. npm包管理工具
2. package配置文件
3. npm install原理
4. yarn，cnpm，npx
5. 发布自己的开发包
6. pnpm使用和原理

(掌握)包管理工具-代码共享和npm基本操作

代码共享方案

- 我们已经学习了在JavaScript中可以通过模块化的方式将代码划分成一个个小的结构：
  - 在以后的开发中我们就可以通过模块化的方式来封装自己的代码，并且**封装成一个工具**
  - 这个工具我们可以让同事通过**导入的方式来使用**，甚至你可以**分享给世界各地的程序员来使用**
- **如果我们分享给世界上所有的程序员使用，有哪些方式呢?**
- 方式一：**上传到GitHub上、其他程序员通过GitHub下载我们的代码手动的引用**
  - 缺点是大家**必须知道你的代码GitHub的地址**，并且**从GitHub上手动下载**

    - 需要在**自己的项目中手动的引用**，并且**管理相关的依赖**
    - 不需要**使用的时候**，需要**手动来删除相关的依赖**
    - 当遇到**版本升级或者切换**时，需要**重复上面的操作**
- 显然，上面的方式是有效的，但是这种传统的方式非常麻烦，并且容易出错；
- 方式二：**使用一个专业的工具来管理我们的代码**
    - 我们**通过工具将代码发布到特定的位置**(npm (npmjs.com))
    - 其他程序员直接通过工具来**安装、升级、删除我们的工具代码**
    - 专业的工具就是npm，yarn，cnpm，npx，pnpm这类
- 显然，通过第二种方式我们可以**更好的管理自己的工具包**，其他人也可以更好的**使用我们的工具包**

包管理工具npm

- **包管理工具npm：**
    - Node Package Manager，也就是**Node包管理器**；
    - 但是目前已经不仅仅是Node包管理器了，在**前端项目**中我们也在使用它来管理依赖的包
    - 比如vue、vue-router、vuex、express、koa、react、react-dom、axios、babel、webpack等等
- **如何下载和安装npm工具呢?**
    - npm属于node的一个管理工具，所以我们需要先安装Node
    - node管理工具：https://nodejs.org/en/，安装Node的过程会自动安装npm工具
- **npm管理的包可以在哪里查看、搜索呢?**
    - https://www.npmjs.org/(上面也有写一遍)
    - 或者看你要下载那个包的官网，例如dayjs：Day.js中文网 (fenxianglu.cn)
    - 在GitHub上找包名
- **npm管理的包存放在哪里呢?**
    - 我们发布自己的包其实是**发布到registry**上面的
    - 当我们安装一个包时其实是**从registry上面下载的包**

3.24小时整集内容

(掌握)项目的配置文件

npm的配置文件(包工具的使用)

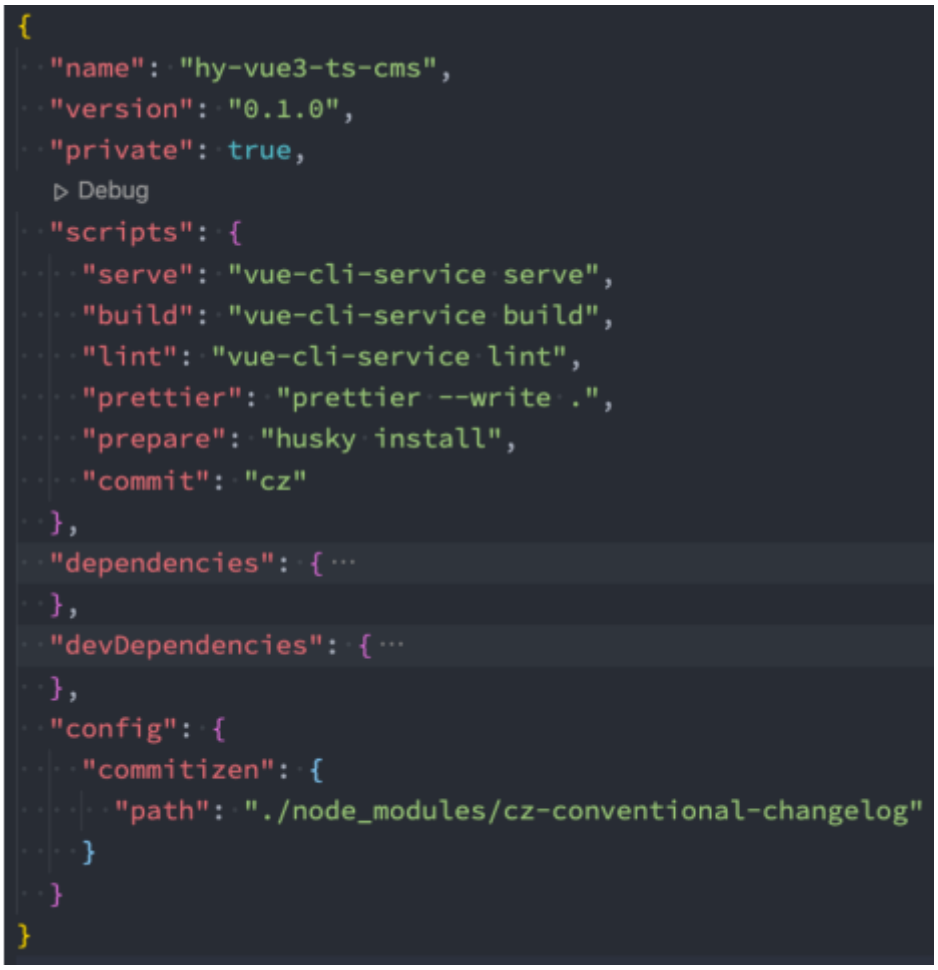
- **那么对于一个项目来说，我们如何使用npm来管理这么多包呢?**
    - 事实上，我们每一个项目都会有一个对应的**配置文件**，无论是前端项目（Vue、React）还是后端项目（Node）
    - 这个**配置文件会记录着你项目的名称、版本号、项目描述**等
    - 也会记录着你项目所依赖的**其他库的信息**和**依赖库的版本号**
- **这个配置文件就是package.json**

- **那么这个配置文件如何得到呢?**
  - 方式一：手**动从零创建项目**，npm init –y
  - 方式二：**通过脚手架创建项目**，脚手架会帮助我们生成package.json，并且里面有相关的配置(都配置好了)

常见的配置文件

- npm init #创建时填写信息
  npm init -y # 所有信息使用默认的(比较简单)
- 

- Vue CLI4创建的Vue3项目(Vue的脚手架)



- create-react-app创建的react17项目(react的脚手架)

(掌握)项目配置文件-基础字段

常见的属性

- **必须填写的属性：name、version**

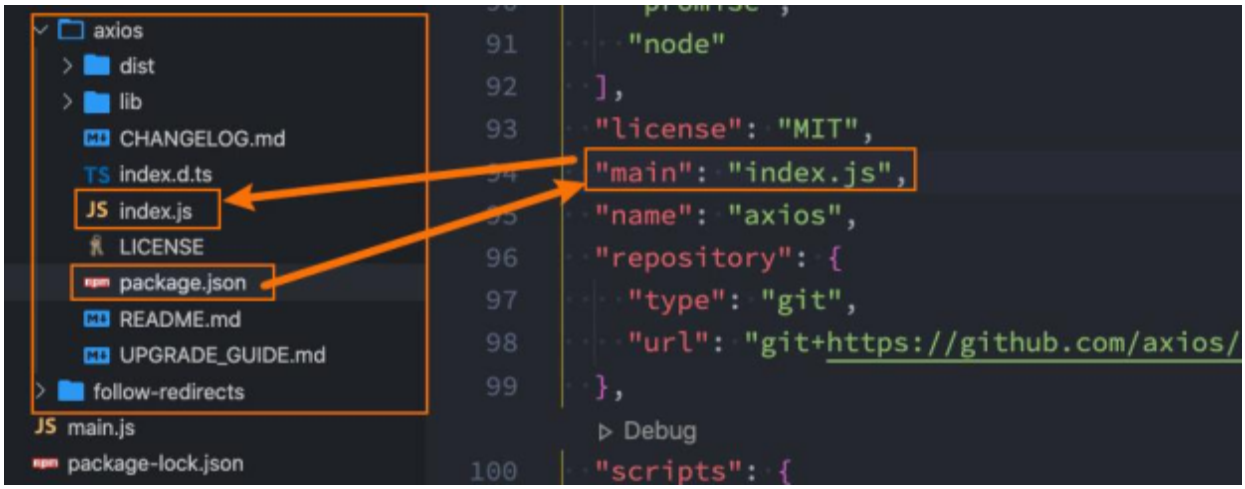
属性	意思
name	项目的名称
version	当前项目的版本号
description	描述信息，很多时候是作为项目的基本描述
author	作者相关信息(发布时用到)
license	开源协议(发布时用到)

- **private属性：**
  - **private**属性记录当前的项目是否是私有的；
  - **当值为true时，npm是不能发布它的**，这是防止私有项目或模块发布出去的方式
- **main属性：**
  - **设置程序的入口**

√ 比如我们使用axios模块 const axios = require('axios');

√ 如果有main属性，实际上是找到对应的main属性查找文件的(就不会默认找到index.js、json、node文件去了)

- 就引入的时候直接输入main的名字，而不是完整路径了(更加简洁，也是基本的操作)



(掌握)项目配置文件-项目依赖

- **scripts属性**
  - **scripts属性用于配置一些脚本命令**，以**键值对的形式**存在；
  - 配置后我们可以通过 **npm run** 命令的key来执行这个命令
  - npm start和npm run start的区别是什么?

    √ 它们是等价的

    √ 对于常用的 start、 test、stop、restart可以省略掉run直接通过 npm start等方式运行(你自己定义的就不能省略run)

```
  "scripts": {
    "start":"运行文件的路径"
  }

//运行快捷键 npm run start
```

- **dependencies属性**
  - dependencies属性是**指定无论开发环境还是生产环境都需要依赖的包**
  - 通常是**我们项目实际开发用到的一些库模块vue、vuex、vue-router、react、react-dom、axios等等**
    - 打包npm run build还有vue代码是因为我们还需要依赖vue来操作DOM
  - 与之对应的是devDependencies
- **devDependencies属性**
  - 一些包在**生产环境是不需要的，比如webpack、babel等**(打包完就不需要了)
  - 这个时候我们会**通过 npm install webpack --save-dev**，将它安装到devDependencies属性中
  - devDependencies = development Dependencies(**开发依赖**)

```
//开发环境依赖
npm install xxx --save-dev//全写
npm install xxx --D//简写
```

- **peerDependencies属性**
  - 还有一种项目依赖关系是**对等依赖**，也就是**你依赖的一个包，它必须是以另外一个宿主包为前提**的
  - 比如element-plus是依赖于vue3的，ant design是依赖于react、react-dom

```
//在element-plus中就能看到
"perrDependencies":{
    "vue":"版本"
}//这表示必须有Vue才能够使用
```

(了解)项目配置文件字段-版本管理和其他属性

依赖的版本管理

- **我们会发现安装的依赖版本出现：^2.0.3或~2.0.3，这是什么意思呢?**
- **npm的包通常需要遵从semver版本规范：**
  - semver：https://semver.org/lang/zh-CN/
  - npm semver：https://docs.npmjs.com/misc/semver
- **semver版本规范是X.Y.Z：**
  - **X主版本号（major）**：当你做了不兼容的 API 修改（可能不兼容之前的版本）
  - **Y次版本号（minor）**：当你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）
  - **Z修订号（patch）**：当你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）
- **我们这里解释一下 ^和~的区别：**
  - **x.y.z**：表示一个**明确的版本号**
  - **^x.y.z**：表示**x是保持不变**的，**y和z永远安装最新的版本**
  - **~x.y.z**：表示**x和y保持不变**的，**z永远安装最新的版本**

少用的属性

- **engines属性**
  - engines属性**用于指定Node和NPM的版本号**
  - 在安装的过程中，**会先检查对应的引擎版本，如果不符合就会报错**
  - 事实上也可以指定所在的操作系统 "os" : ["darwin", "linux"]，只是很少用到
- **browserslist属性**
  - 用于配置打包后的JavaScript浏览器的兼容情况，参考

- 否则我们需要手动的添加polyfills来让支持某些语法
- 也就是说它是为webpack等打包工具服务的一个属性（这里不是详细讲解webpack等工具的工作原理，所以不再给出详情）

(掌握)npm安装包的细节补充

npm install 命令

- **安装npm包分两种情况：**
  - **全局安装（global install）**： npm install webpack -g
  - **项目（局部）安装（local install）**： npm install webpack
- **全局安装**
  - 全局安装是直接将某个包安装到全局：
  - 比如全局安装yarn：

```
npm install yarn -g
```

- **但是很多人对全局安装有一些误会：**
  - 通常使用**npm全局安装的包都是一些工具包**：yarn、webpack等
  - **并不是类似于 axios、express、koa等库文件**
  - 所以全局安装了之后并不能让我们在所有的项目中使用 axios等库

项目安装

- **项目安装会在当前目录下生成一个 node_modules 文件夹，我们之前讲解require查找顺序时有讲解过这个包在什么情况下被查找**
- **局部安装分为开发时依赖和生产时依赖：**

```
//默认安装开发环境
npm install axios
npm i axios
//开发依赖
npm install webpack --save-dev
npm install webpack -D
npm i webpack -D
//根据package.json中的依赖包
npm install
```

(理解)package.json文件的作用

package-lock.json

- **package-lock.json文件解析：**

package-lock.json文件属性	意思
name	项目名称
version	项目的版本
lockfileVersion	lock文件的版本
requires	使用requires来跟踪模块的依赖关系
dependencies	项目的依赖(当前包对其他包的依赖)

- 当前项目依赖axios，但是axios依赖follow-redireacts

axios中的属性	意思
version	表示实际安装的axios的版本
resolved	用来记录下载的地址，registry仓库中的位置

axios中的属性	意思
requires(需要)/dependencies	记录当前模块的依赖
integrity(完整)	用来从缓存中获取索引，再通过索引去获取压缩包文件

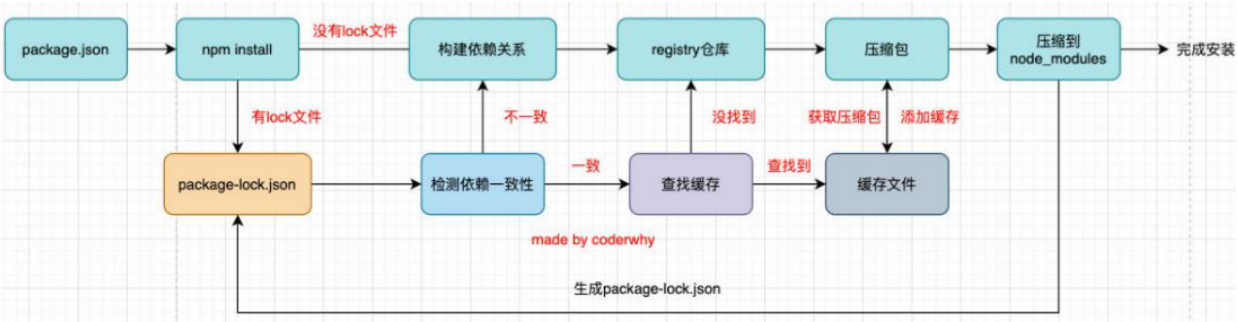
- 这个文件我们是不会手动去修改他的，但是作为理解还是有必要的

(理解)npm install的安装原理

npm install 原理

- **很多同学之前应该已经会了 npm install，但是你是否思考过它的内部原理呢?**
  - 执行 **npm install它背后帮助我们完成了什么操作**?
  - 我们会发现还有一个称之为p**ackage-lock.json的文件**，它的作用是什么?
  - 从npm5开始，**npm支持缓存策略（来自yarn的压力），缓存有什么作用**呢?
- **这是一幅的根据 npm install 的原理图:**

> 压缩到node_modules写错了，是将压缩包解压到node_modules



npm install 原理图解析

- **npm install会检测是有package-lock.json文件:**
  - **没有lock文件**
    1. 分析依赖关系，这是因为我们可能包会依赖其他的包，并且多个包之间会产生相同依赖的情况;
    2. 从registry仓库中下载压缩包（如果我们设置了镜像，那么会从镜像服务器下载压缩包）;
    3. 获取到压缩包后会对压缩包进行缓存（从npm5开始有的），下次再下载的时候就从缓存里面拿(现在npm已经是8版本了)
    4. 将压缩包解压到项目的node_modules文件夹中（前面我们讲过，require的查找顺序会在该包下面查找)
  - **有lock文件**
    1. 检测lock中包的版本是否和package.json中一致（会按照semver版本规范检测)
       - 不一致，那么会重新构建依赖关系，直接会走顶层的流程

2. 一致的情况下，会去优先查找缓存
    - 没有找到，会从registry仓库下载，直接走顶层流程；
3. 查找到，会获取缓存中的压缩文件，并且将压缩文件解压到node_modules文件夹中；

(理解)npm其他命令的解析和查看

npm其他命令

- **我们这里再介绍几个比较常用的：**
- **卸载某个依赖包：**

```
npm uninstall package
npm uninstall package --save-dev
npm uninstall package -D
```

- **强制重新build**

```
npm rebuild
```

- **清除缓存**

```
npm cache clean
```

- **npm的命令其实是非常多的：**
    - https://docs.npmjs.com/cli-documentation/cli
    - 更多的命令，可以根据需要查阅官方文档

yarn工具

- **另一个node包管理工具yarn：**
    - yarn是由**Facebook、Google、Exponent 和 Tilde** 联合推出了一个新的 **JS 包管理工具**
    - yarn 是为了弥补 **早期npm 的一些缺陷**而出现的
    - 早期的npm存在很多的缺陷，比如**安装依赖速度很慢、版本依赖混乱**等等一系列的问题
    - 虽然从npm5版本开始，进行了很多的升级和改进，但是依然很多人喜欢使用yarn

Npm	Yarn
npm install	yarn install
npm install [package]	yarn add [package]
npm install --save [package]	yarn add [package]
npm install --save-dev [package]	yarn add [package] [--dev/-D]
npm rebuild	yarn install --force
npm uninstall [package]	yarn remove [package]
npm uninstall --save [package]	yarn remove [package]
npm uninstall --save-dev [package]	yarn remove [package]
npm uninstall --save-optional [package]	yarn remove [package]
npm cache clean	yarn cache clean
rm -rf node_modules && npm install	yarn upgrade

(了解)cnpm和淘宝镜像的理解

cnpm工具

- **由于一些特殊的原因，某些情况下我们没办法很好的从 https://registry.npmjs.org下载下来一些需要的包。**
- **查看npm镜像：**

```
npm config get registry # npm config get registry
```

- **我们可以直接设置npm的镜像：**

```
npm config set registry https://registry.npm.taobao.org
```

- **但是对于大多数人来说（比如我），并不希望将npm镜像修改了：**
  - 第一，**不太希望随意修改npm原本从官方下来包的渠道**
  - 第二，**担心某天淘宝的镜像挂了或者不维护了，又要改来改去**
- **这个时候，我们可以使用cnpm，并且将cnpm设置为淘宝的镜像：**

```
npm install -g cnpm --registry=https://registry.npmmirror.com/
cnpm config get registry # https://registry.npmmirror.com/
//可能会变的，比如我现在这里的镜像地址就跟coderwhy的不一样，因为淘宝镜像迁移了
```

(掌握)npx的使用和scripts创建脚本

npx工具

- **npx是npm5.2之后自带的一个命令**
  - npx的作用非常多，但是比较常见的是使用它来调用项目中的某个模块的指令
- **我们以webpack为例：**
  - 全局安装的是webpack5.1.3
  - 项目安装的是webpack3.6.0

- **如果我在终端执行 webpack --version使用的是哪一个命令呢?**
  - 显示结果会是 webpack 5.1.3，事实上使用的是全局的，为什么呢?
  - 原因非常简单，在当前目录下找不到webpack时，就会去全局找，并且执行命令
- **如何解决这个问题呢?**

局部命令的执行

- **那么如何使用项目（局部）的webpack，常见的是两种方式:**
  - **方式一**：明确查找到node_module下面的webpack
  - **方式二**：在 scripts定义脚本，来执行webpack
- **方式一：在终端中使用如下命令（在项目根目录下）**

```
./node_modules/.bin/webpack --version
```

- **方式二：修改package.json中的scripts**

```
"scripts": {
"webpack": "webpack --version"
}//终端输入就可以变成npm run webpack = webpack --version
```

- **方式三：使用npx**

```
npx webpack --version//现在已经集成了，会先从局部开始找
```

- npx的原理非常简单，它会到当前目录的node_modules/.bin目录下查找对应的命令

(理解)npm发布自己的包和使用包的过程

- **注册npm账号:**
  - https://www.npmjs.com/
  - 选择sign up
- **在命令行登录:**

```
npm login
```

- **修改package.json**
- **发布到npm registry上**

```
npm publish
```

- **更新仓库:**
  - 修改版本号(最好符合semver规范)
  - 重新发布
- **删除发布的包:**

```
npm unpublish
```

- **让发布的包过期：**

```
npm deprecate
```

(理解)npx命令使用说明和查找问题思路

> 刚刚在终端(node_modules/webpack5.xx0x(.bin))敲出webpack --version的时候，如果版本不一致，显示的是那个版本呢？局部版本还是全局的？
>
> - 查找出来是子目录下的局部版本，是由于webpack做出的处理(这个比较特殊，像yarn/babel/lessc之类的都是从全局开始找，会从当前目录往上找而不是往下子目录找)
> - 辩证的思考

(理解)原包管理工具的痛点和pnpm的介绍

什么是pnpm呢?

- **什么是pnpm呢？我们来看一下官方的解释：**
  - pnpm：我们可以理解成是**performant npm**缩写；

- pnpm 是一种 JavaScript 包管理工具，它具有以下优点：

1. 空间效率高：pnpm 使用一个共享的本地存储库来存储所有依赖项，而不是将每个依赖项都安装在项目中。这意味着在多个项目之间共享依赖项时，pnpm 可以极大地减少磁盘空间的使用。(**最重要**)
2. 安装速度快：由于 pnmp 只需要安装每个依赖项一次，并且可以在不同的项目之间共享这些依赖项，因此它通常比其他包管理工具的安装速度更快。
3. 版本控制清晰：pnpm 可以管理项目依赖项的版本，这使得开发人员能够更轻松地了解哪些依赖项被安装在项目中，以及这些依赖项的版本是什么。

4. 更容易管理依赖项：pnpm 允许开发人员更轻松地管理项目依赖项，因为它可以为每个项目自动创建一个锁文件，该锁文件包含有关每个依赖项及其版本的详细信息。
5. 对于 monorepo（单个代码库存储多个项目）特别有用：pnpm 可以更轻松地管理多个项目之间共享的依赖项，因此它在 monorepo 中特别有用。

> 总之，pnpm 可以提供更高效的包管理体验，尤其是对于具有多个项目或使用大量共享依赖项的项目。

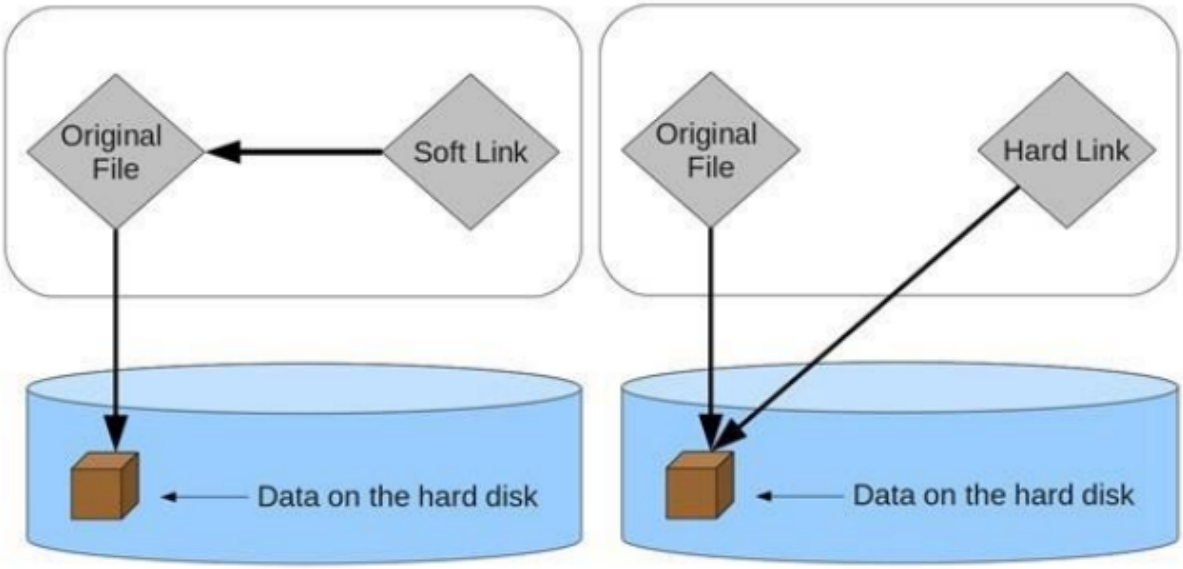
- **哪些公司在用呢?**
  - 包括**Vue在内的很多公司或者开源项目的包管理工具都切换到了pnpm**

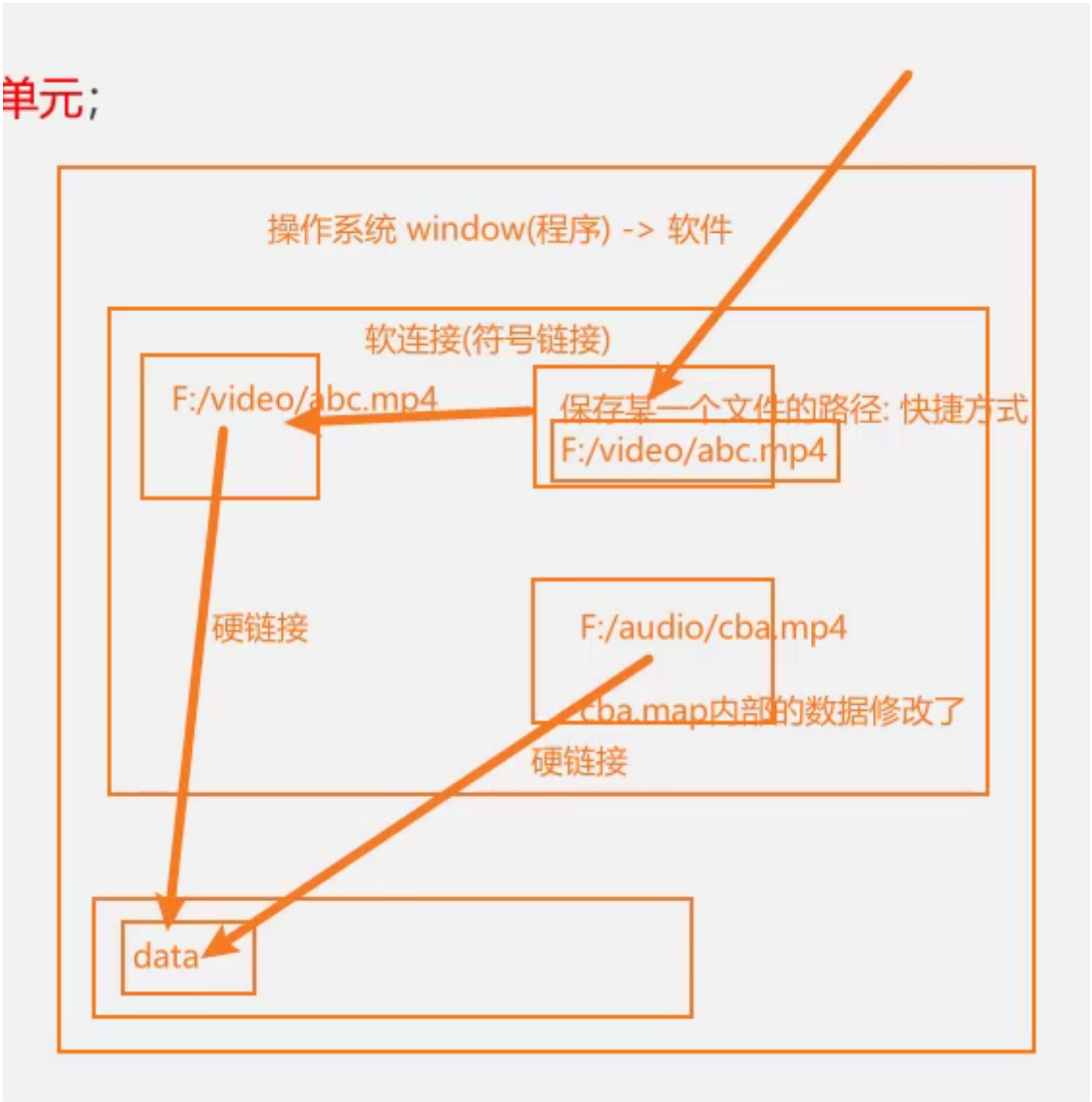
(理解)操作系统-硬链接和软链接的概念

> 操作系统 window(程序) -> 软件

硬链接和软连接的概念

- **硬链接（hard link）：**
  - **硬链接（英语：hard link）是电脑文件系统中的多个文件平等地共享同一个文件存储单元**
  - 删除一个文件名字后，还可以用其它名字继续访问该文件
- **符号链接（软链接soft link、Symbolic link）：**
  - **符号链接（软链接、Symbolic link）是一类特殊的文件**
  - 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**





这个硬链接是真实指向硬盘上的数据的(你可以以各种形式的名字去访问)，如果修改了一个指向F：/audio/cba.mp4的文件的话，其实就是修改了data的内容，那其他指向F：/audio/cba.mp4的文件内容也会同时被修改。这个跟拷贝一份是不同的概念

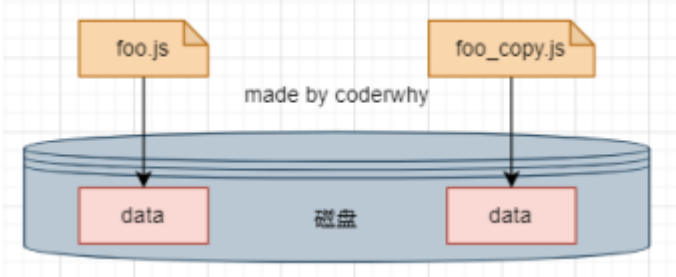
> 软链接就像是快捷方式，只是以绝对路径或者相对路径的形式指向其它文件或者目录

(理解)操作系统-硬链接和软连接的演练

硬链接和软连接的演练

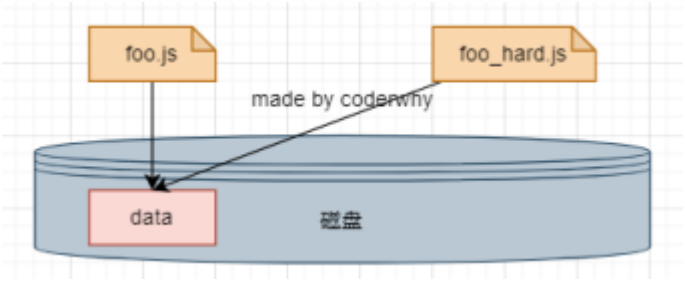
- **文件的拷贝**：文件的拷贝每个人都非常熟悉，会在硬盘中复制出来一份新的文件数据(也就是会造成磁盘的一部分空间)

```
window: copy foo.js foo_copy.js
macos : cp foo.js foo_copy.js
```



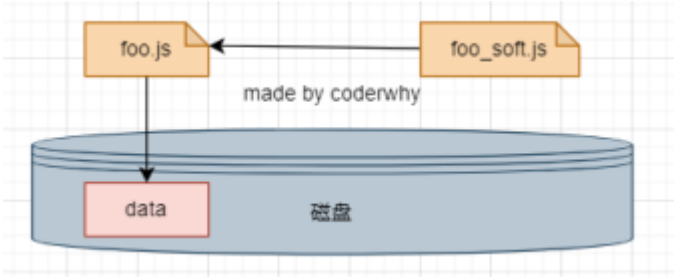
- **文件的硬链接**

```
window: mklink /H aaa_hard.js aaa.js//创建硬链接的命令，aaa_hard.js跟aaa.js建立起硬链接
macos : ln foo.js foo_hard.js
```



- **文件的软连接**:

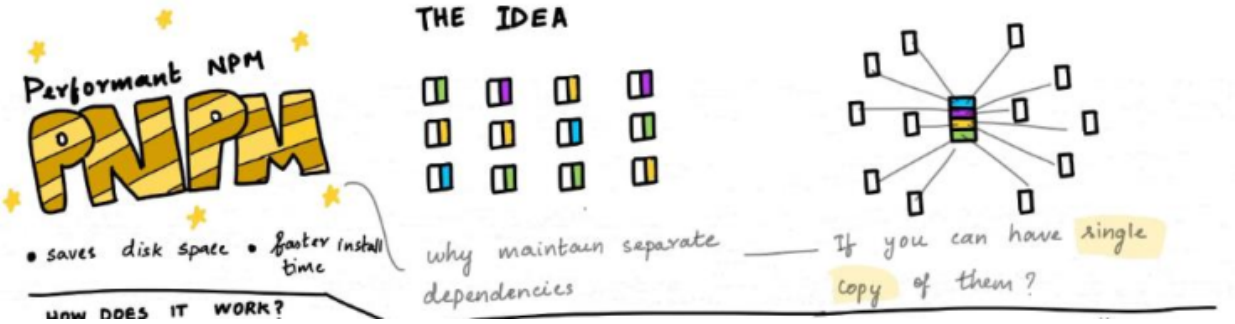
```
//需要权限，以管理员身份打开终端
window: mklink aaa_soft.js aaa.js//变成软链接后，类型变成.symlink
macos : ln -s foo.js foo_copy.js
//删除掉源文件，这个软链接就打不开了。相当于下面图片中foo.js被删掉了。foo_soft.js和data之间的联系就断开了
```



(理解)pnpm的原理-项目多个包建立硬链接

pnpm到底做了什么呢?

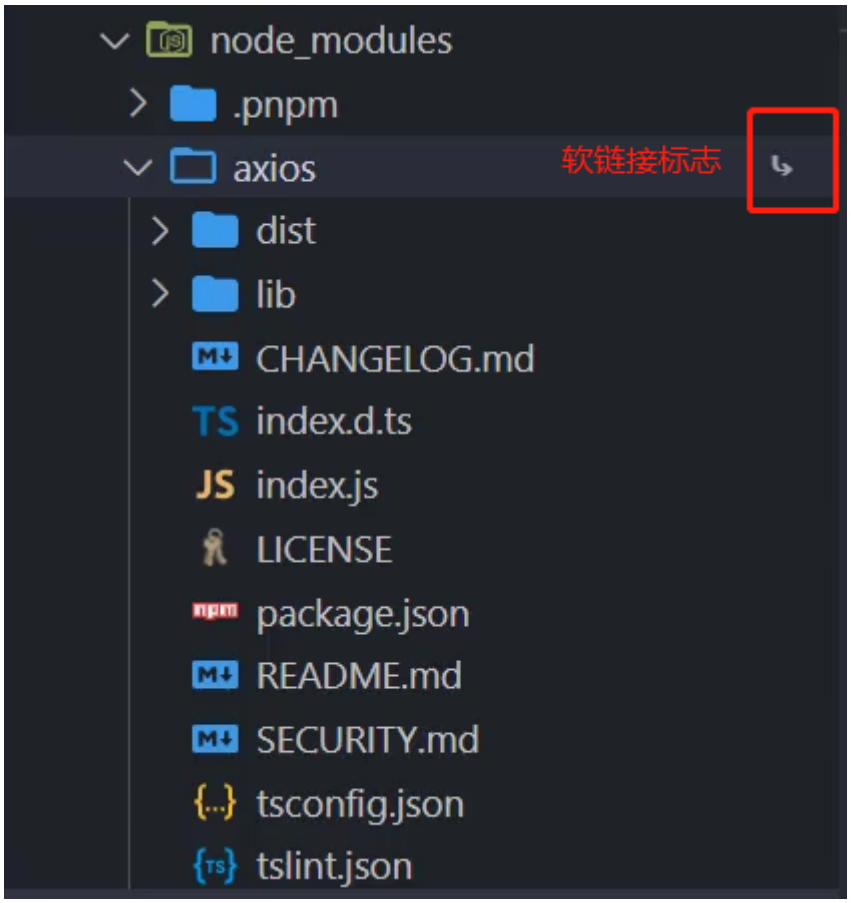
- **当使用 npm 或 Yarn 时，如果你有 100 个项目，并且所有项目都有一个相同的依赖包，那么， 你在硬盘上就需要保存 100 份该相同依赖包的副本**
- **如果是使用 pnpm，依赖包将被 存放在一个统一的位置，因此：**
  - 如果你对**同一依赖包使用相同的版本**，那么**磁盘上只有这个依赖包的一份文件**
  - 如果你对**同一依赖包需要使用不同的版本**，则仅有 **版本之间不同的文件会被存储起来**
  - 所有文件都**保存在硬盘上的统一的位置：**
    1. 当安装软件包时， 其包含的所有文件都会硬链接到此位置，而不会占用 额外的硬盘空间
    2. 这让你可以在项目之间方便地共享相同版本的 依赖包



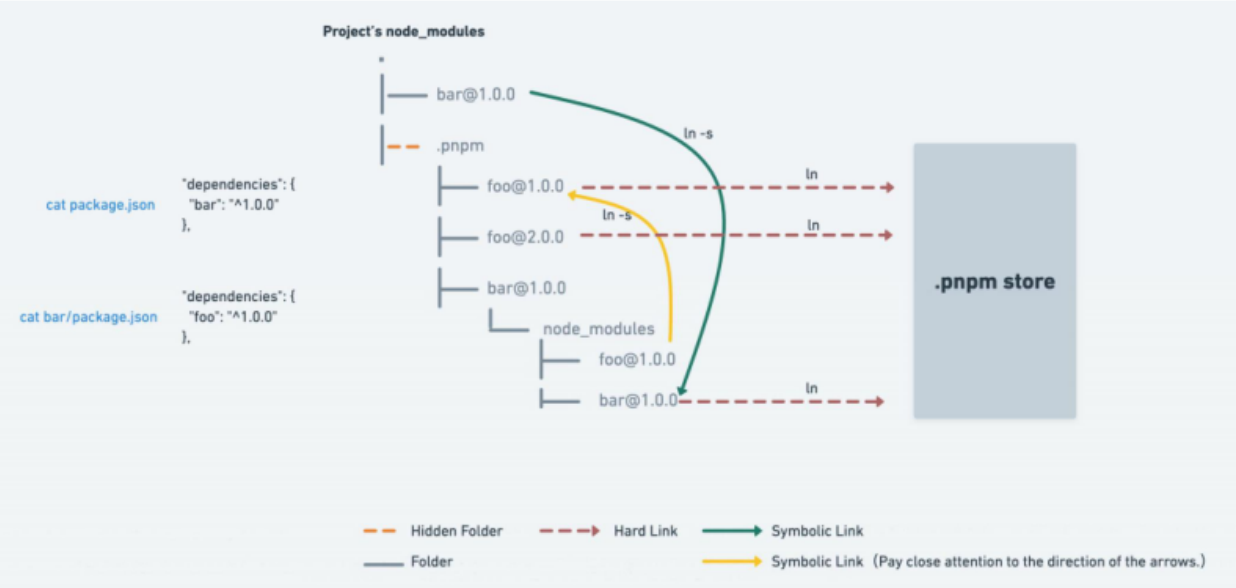
(理解)pnpm的创建非扁平化的node_module

pnpm创建非扁平的 node_modules 目录

- **当使用 npm 或 Yarn Classic 安装依赖包时，所有软件包都将被提升到 node_modules 的 根目录下**
  - 其结果是，源码可以访问 本不属于当前项目所设定的依赖包(就是你安装了webpack的时候，同时会下载下来一堆webpack所需要的包，这些不是你主动下载的，但是你也可以访问到，但是这个问题在pnpm中将不复存在，你下载了webpack就只能看到webpack这个文件夹，其他的有真实地址，硬链接指向了磁盘空间，但是他没有创建软链接，你没办法通过软链接找到硬链接再找到硬盘里的内容去调用)
  -



> 之所以是软链接，是因为真实的地址是由pnpm保管的，这里的只是指向真实地址的软链接，然后真实地址硬链接到硬盘中，这样重复使用的时候就创建软链接就行了，就不会因为多个项目使用重复的包而重复下载包造成的大量空间的浪费了



专业单词	意思
Hidden Folder	隐藏的文件夹

专业单词	意思
Folder	文件夹
Hard Link	硬链接
Symbolic Link	软链接
Symbolic Link(Pay close attention to the direction of the arrows)	软链接(密切注意箭头的方向)

(掌握)pnpm的常见命令和store存储

pnpm的安装和使用

- **那么我们应该如何安装pnpm呢?**
  - 官网提供了很多种方式来安装pnpm：https://www.pnpm.cn/installation
  - Node中有npm，所以我们通过npm安装即可

```
npm install -g pnpm
```

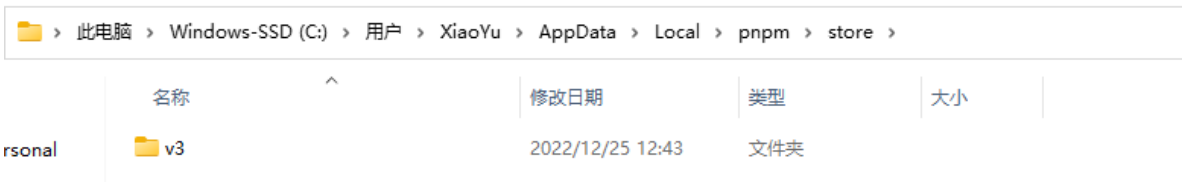
- **以下 是一个与 npm 等价命令的对照表，帮助你快速入门:**

npm命令(pkg = package)	pnpm等价命令
npm install	pnpm install
npm install	pnpm add
npm uninstall	pnpm remove
npm run	pnpm

pnpm的存储store

- **在pnpm7.0之前，统一的存储位置是 ~/.pnpm-score中的**

- **在pnpm7.0之后，统一的存储位置进行了更改：/store**
  - 在 Linux 上，默认是 ~/.local/share/pnpm/store
  - 在 Windows 上： %LOCALAPPDATA%/pnpm/store
  - 在 macOS 上： ~/Library/pnpm/store



- **我们可以通过一些终端命令获取这个目录**：获取当前活跃的store目录

```
pnpm store path//终端输入返回pnpm所在store地址
//store地址就是用来放我们下载的那些包的地方，属于硬链接的那部分，只下载一次供软链接去连接
```

- **另外一个非常重要的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

```
pnpm store prune
```