node_modules简介

`node_modules` 是 Node.js 项目中的一个目录，用于存放项目的依赖包（模块）。

当您在 Node.js 项目中使用第三方库或模块时，通常会通过 npm（Node.js 包管理器）或其他包管理工具来安装这些依赖项。安装完成后，相关的依赖包将会被下载并存放在项目的 `node_modules` 目录下。

`node_modules` 目录是一个自动创建的目录，它位于项目根目录或子目录中（根据项目的结构而定），用于组织和存储项目所需的依赖项。

node_modules检索规则

在 Node.js 中，当需要加载某个包时，Node.js 会按照一定的规则来检索并加载该包。检索包的过程包括以下步骤：

1. 检查内置模块：首先，Node.js 会检查所需的模块是否是 Node.js 的内置模块。内置模块是 Node.js 提供的一些核心模块，如 `fs`、`http` 等。如果所需模块是内置模块，Node.js 会直接加载它。

2. 检查核心模块：如果所需的模块不是内置模块，Node.js 会检查是否是核心模块。核心模块是指 Node.js 默认提供的模块，如 `path`、`util` 等。如果所需模块是核心模块，Node.js 会直接加载它。

3. 检查局部模块：如果所需模块既不是内置模块也不是核心模块，Node.js 会按照以下顺序检查模块是否存在于局部安装的依赖项中：

   a. 当前目录的 `node_modules` 目录：Node.js 首先检查当前目录下的 `node_modules` 目录，查找是否存在所需模块的安装。

   b. 父级目录的 `node_modules` 目录：如果当前目录的 `node_modules` 目录中不存在所需模块，Node.js 会逐级向上查找父级目录，直到找到包含所需模块的 `node_modules` 目录，或者到达文件系统的根目录。

4. 检查全局模块：如果所需模块既不是内置模块、核心模块，也不存在于局部安装的依赖项中，Node.js 会检查是否是全局安装的模块。全局安装的模块是通过 `npm install -g` 命令安装的。Node.js 会检查全局安装目录中的 `node_modules` 目录，查找是否存在所需模块。

5. 报错：如果以上步骤都未找到所需模块，Node.js 会抛出模块未找到的错误。

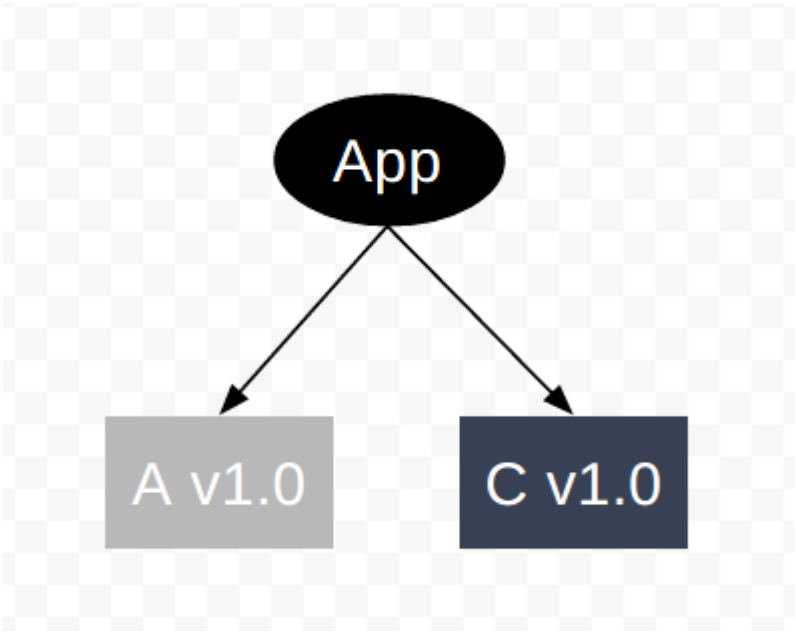
需要注意的是，当有多个版本的模块存在时，Node.js 会根据模块的依赖关系和加载顺序，优先加载符合依赖要求的模块版本。

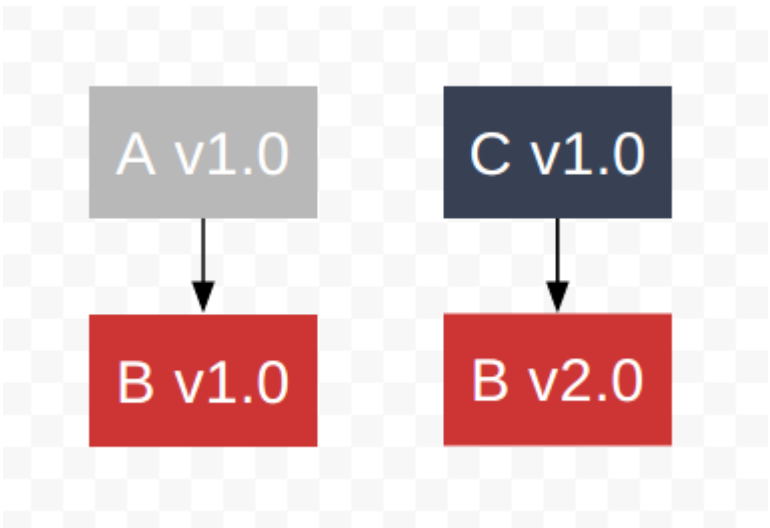
总结起来，Node.js 在加载模块时会按照内置模块、核心模块、局部模块和全局模块的顺序进行检索。这样的检索过程确保了模块的正确加载，并满足依赖关系。

node_modules的弊端

Dependency Hell 依赖地狱问题

现在项目里有两个依赖A和C，A和C分别依赖B的不同版本，如何处理





这里存在两个问题：

1. 首先是B本身支持多版本共存，只要B本身没有副作用，这是很自然的，但是对于很多库如core-js会污染全局环境，本身就不支持多版本共存，因此我们需要尽早的进行报错提示（conflict的warning和运行时的conflict的check）

2. 如果B本身支持多版本共存，那么需要保证A正确的加载到B v1.0和C正确的加载到B v2.0

   我们重点考虑第二个问题：node的解决方式是**依赖的node加载模块的路径查找算法和node_modules的目录结构来配合**

如何从node_modules加载package?

核心是递归向上查找node_modules里的package，如果在 `'/home/ry/projects/foo.js'` 文件里调用了 `require('bar.js')`，则 Node.js 会按以下顺序查找：

- `/home/ry/projects/node_modules/bar.js`
- `/home/ry/node_modules/bar.js`
- `/home/node_modules/bar.js`
- `/node_modules/bar.js`

  该算法有两个核心：（1）优先读取最近的node_modules的依赖（2）递归向上查找node_modules的依赖

  该算法即简化了 Dependency hell 的解决方式，也带来了非常多的问题

node_modules的目录结构

nest mode嵌套结构

利用require先在最近的node_module里查找依赖的特性，我们能想到一个很简单的方式，直接在node_module维护原模块的拓扑图即可

这样根据mod-a就近的使用mod-b的1.0版本，而mod-c就近的使用了mod-b的2.0版本

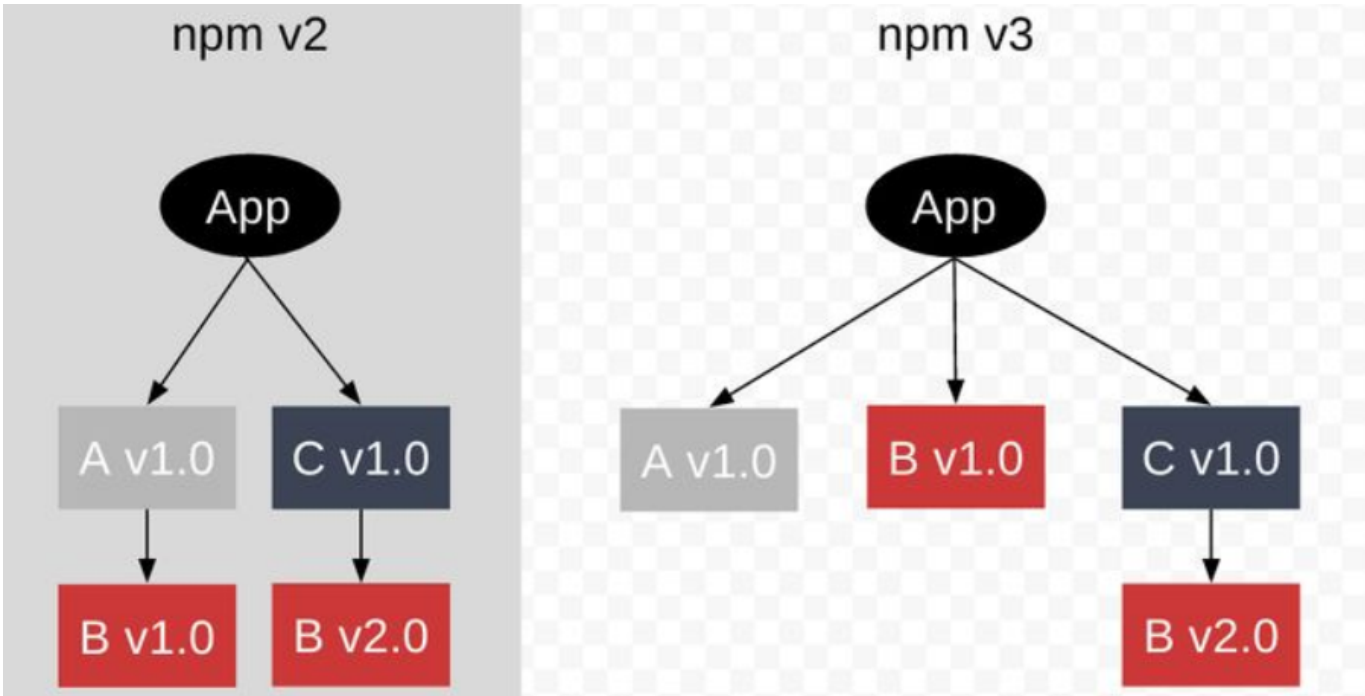
但是这样带来了另一个问题，如果我们此时再依赖一个mod-d，该mod-d也同时依赖的mod-b的2.0版本，这时候node_modules就变成下面这样

```
node_modules
    mod-a
        node_modules
            mod-b@1.0
    mod-c
        node_modules
            mod-b@2.0
    mod-d
        node_modules
            mod-b@2.0
```

我们发现这里存在个问题，虽然 mod-a 和 mod-d 依赖了同一个mod-b的版本，但是mod-b却安装了两遍，如果你的应用了很多的第三方库，同时第三方库共同依赖了一些很基础的第三方库如lodash，你会发现你的node_modules里充满了各种重复版本的lodash，造成了极大的空间浪费，也导致npm install很慢，这既是 node_modules hell

flat mode 平坦模式

我们还可以利用向上递归查找依赖的特性，将一些公共依赖放在公共的node_module里



根据require的查找算法：

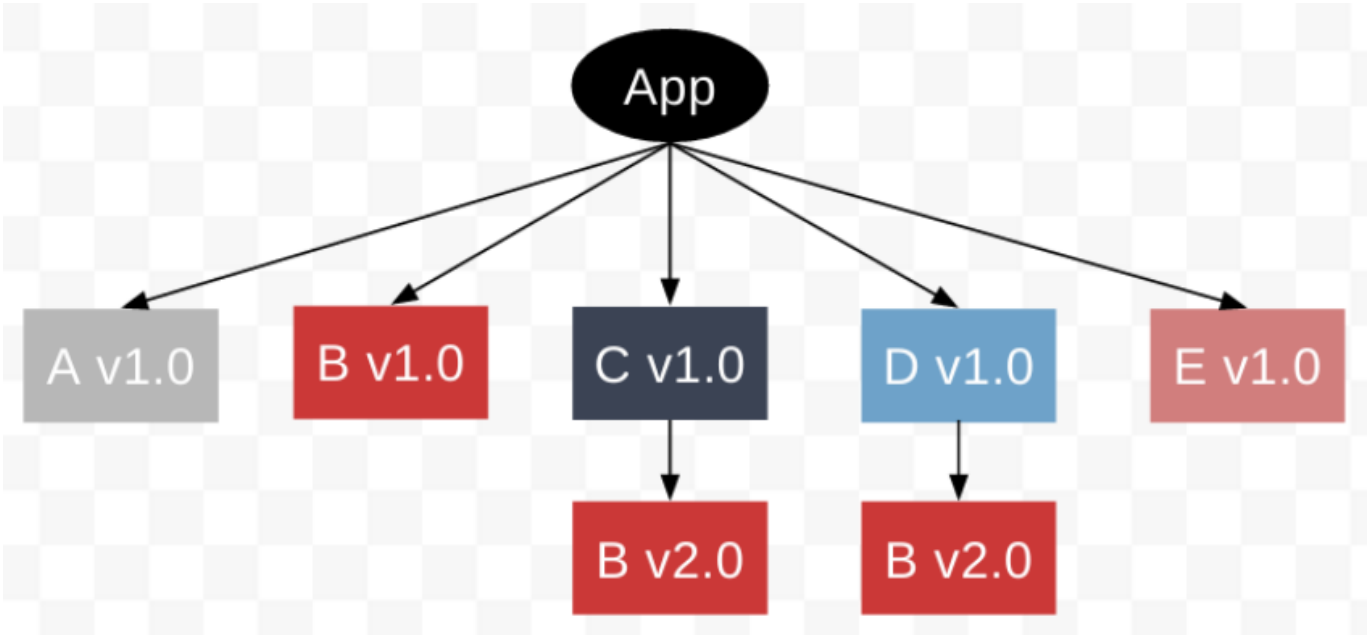
1. A和D首先会去自己的node_module里去查找B，发现不存在B，然后递归的向上查找，此时查找到了B的 v1.0版本，符合预期

2. C会先查找到自己的node_module里查找到了B v2.0，符合预期

这时我们发现了即解决了depdency hell也避免了npm2 的nest模式导致的重复依赖问题

新问题： doppelgangers 二重身问题

但是问题并没有结束，如果此时引入的D依赖的是B v2.0，而引入的E依赖的是B v1.0，我们发现无论是把Bv2.0还是Bv1.0放在top level，都会导致任何另一个版本会存在重复的问题，如这里的B的v2.0的重复问题



大多数情况都是这样，但有可能造成很多问题

1.全局冲突

2.破坏单例模式

3.Phantom dependency 幻影依赖

我们发现flat mode相比nest mode节省了很多的空间，然而也带来了一个问题即phantom depdency，考察下如下的项目

```
// monorepo/mylib/package.json
{
  "name": "my-library",
  "version": "1.0.0",
  "main": "lib/index.js",
  "dependencies": {
    "minimatch": "^3.0.4"
  },
  "devDependencies": {
    "rimraf": "^2.6.2"
  }
}

```

```
var minimatch = require("minimatch")
var expand = require("brace-expansion");  // ???
var glob = require("glob")  // ???

// (more code here that uses those libraries)

```

这里的glob和brace-expansion都不在我们的depdencies里，但是我们开发和运行时都可以正常工作（因为这个是rimraf的依赖），一旦将该库发布，因为用户安装我们的库的时候并不会安装库的devDepdency，这导致在用户的地方会运行报错。

我们把一个库使用了不属于其depdencies里的package称之为phantom depdencies，phantom depdencies不仅会存在库里，当我们使用monorepo管理项目的情况下，问题更加严重，一个package不但可能引入DevDependency引入的phantom依赖，更很有可能引入其他package的依赖，当我们部署项目或者运行项目的时候就可能出问题。

在基于yarn或者npm的node_modules的结构下，doppelganger和phantom dependency似乎并没有太好的解决方式。其本质是因为npm和yarn通过 node resolve算法配合node_modules的树形结构对原本depdency graph的模拟

后续一系列解决方案

Semver、lock、pnpm

workspace 的工作原理

workspaces就是多空间的概念，在npm中可以理解为多包。它的初衷是为了用来进行多包管理的，它可以让多个npm包在同一个项目中进行开发和管理变得非常方便：

- 它会将子包中所有的依赖包都提升到根目录中进行安装，提升包安装的速度；
- 它初始化后会自动将子包之间的依赖进行关联（软链接）；
- 因为同一个项目的关系，从而可以让各个子包共享一些流程，比如：eslint、stylelint、git hooks、publish flow等；

这个设计模式最初来自于Lerna，但Lerna对于多包管理，有着更强的能力，而且最新版的Lerna可以完全兼容npm或yarn的workspaces模式。

workspace基本使用

定义工作区

工作区通常通过 `package.json` 文件的 `workspaces` 属性定义，例如：

```
{
  "name": "my-workspaces-powered-project",
  "workspaces": [
    "packages/workspace-a"
  ]
}
```

鉴于上述 `package.json` 示例位于当前工作目录 `.` 中，该目录包含一个名为 `packages/workspace-a` 的文件夹，该文件夹本身包含一个 `package.json`，定义了一个 Node.js 包，例如：

```
.
+-- package.json
`-- packages
    `-- workspace-a
        `-- package.json
```

在当前工作目录 `.` 中运行 `npm install` 后，预期的结果是文件夹 `workspace-a` 将符号链接(npm link)到当前工作目录的 `node_modules` 文件夹。

下面是一个帖子 `npm install` 示例，假设文件和文件夹的先前示例结构相同：

```
.
+-- node_modules
|   `-- workspace-a -> ../workspace-a
+-- package-lock.json
+-- package.json
    `-- packages
        `-- workspace-a
            `-- package.json
```

您可以使用 npm init 自动执行定义新工作区所需的步骤。例如，在已经定义了 `package.json` 的项目中，您可以运行：

```
npm init -w ./packages/a
```

此命令将创建丢失的文件夹和新的 `package.json` 文件（如果需要），同时确保正确配置根项目 `package.json` 的 `"workspaces"` 属性。

将依赖项添加到工作区

可以使用 `workspace` 直接添加/删除/更新工作区的依赖项。

例如，假设以下结构：

```
.
+-- package.json
`-- packages
   +-- dgov
   |   `-- package.json
   `-- dgov-mobile
       `-- package.json
```

如果您想从注册表中添加一个名为 `abbrev` 的依赖项作为工作区 adgov的依赖项，您可以使用工作区配置告诉 npm 安装程序应将包添加为所提供工作区的依赖项：

```
npm install abbrev -w dgov
```

注意：其他安装命令如 `uninstall`、`ci` 等也将尊重提供的 `workspace` 配置。

使用工作区

给定 Node.js 如何处理模块解析的细节，可以通过声明为 `package.json` `name` 来使用任何定义的工作区。继续上面定义的示例，让我们还创建一个需要 `workspace-a` 示例模块的 Node.js 脚本，例如：

```
// ./workspace-a/index.js
module.exports = 'a'

// ./lib/index.js
const moduleA = require('workspace-a')
console.log(moduleA) // -> a
```

运行时：

```
node lib/index.js
```

这展示了 `node_modules` 分辨率的性质如何允许工作空间启用可移植的工作流程，以要求每个工作空间也易于 发布 这些嵌套的工作空间在其他地方使用。

在工作区上下文中运行命令

您可以使用 `workspace` 配置选项在已配置工作区的上下文中运行命令。

下面是一个关于如何在嵌套工作区的上下文中使用 `npm run` 命令的快速示例。对于包含多个工作区的项目，例如：

```
.
+-- package.json
`-- packages
    +-- a
    |   `-- package.json
    `-- b
        `-- package.json
```

通过使用 `workspace` 选项运行命令，可以在该特定工作区的上下文中运行给定的命令。例如：

```
npm run test --workspace=a
```

这将运行在 `./packages/a/package.json` 文件中定义的 `test` 脚本。

请注意，您还可以在命令行中多次指定此参数以针对多个工作区，例如：

```
npm run test --workspace=a --workspace=b
```

也可以使用 `workspaces`（复数）配置选项来启用相同的行为，但在所有配置的工作区的上下文中运行该命令。例如：

```
npm run test --workspaces
```

将在 `./packages/a` 和 `./packages/b` 中运行 `test` 脚本。

命令将按照它们在 `package.json` 中出现的顺序在每个工作区中运行

```
{
  "workspaces": [ "packages/a", "packages/b" ]
}
```

运行顺序与以下不同：

```
{
  "workspaces": [ "packages/b", "packages/a" ]
}
```

请注意，`npm workspace` 只适用于 npm 7 或更高版本。

例子

1. 创建项目

```
mkdir demo-workspaces-multi-project
```

2. 初始化项目

```
npm init -y
```

·. └── package.json`

3. 声明本项目是workspaces模式

`package.json` 新增配置：

```
"private":"true",
"workspaces": [
 "projects/*"
],
```

4. 初始化子项目 `zoo`

创建子项目 `zoo`：

```
npm init -w package/zoo -y
```

```
. ├── package.json └── packages     └── zoo        └── package.json
```

创建模板文件 `index.html`，主内容为：

```
<!-- package/zoo/index.html -->
<body>
  <h1>Welcome to Zoo!</h1>
  <div id="app"></div>
</body>
```

创建项目入口js文件 `index.js`，内容为：

```
console.log('Zoo')
```

安装项目构建依赖包：

```
npm i -S webpack webpack-cli webpack-dev-server html-webpack-plugin webpack-merge --
workspace=zoo

# projects/zoo/package.json
"private":"true",
"dependencies": {
  "html-webpack-plugin": "^5.5.0",
  "webpack": "^5.65.0",
  "webpack-cli": "^4.9.1",
  "webpack-dev-server": "^4.7.2"
}
```

创建webpack配置(此处忽略，可去github上看详细配置)

zoo下的`package.json`新增命令：

```
"scripts": {
  "dev": "webpack-dev-server --config webpack/dev.config.js --open",
  "prod": "webpack --config webpack/prod.config.js"
},
```

接下来就可以运行了，只需要在项目根目录使用：

```
npm run dev --workspace=zoo
```

即可进行本地开发。

效果：

5. 初始化子项目`shop`

创建子项目`shop`：

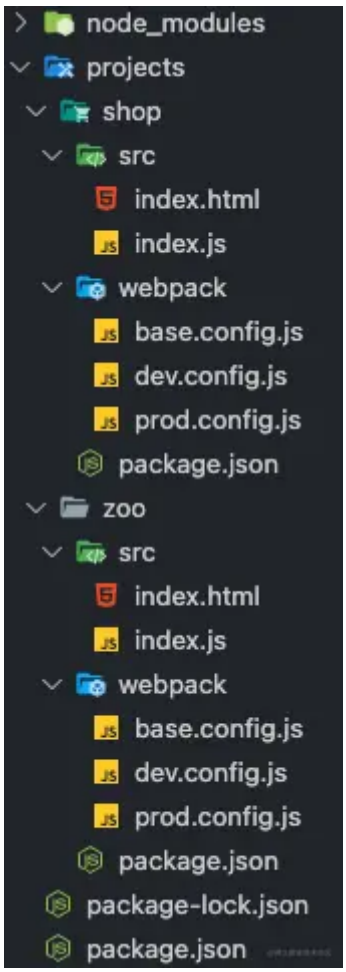
shell

复制代码

```
npm init -w package/shop -y
```

其余步骤同初始化子项目`zoo`几乎一模一样，所以不再赘述。

最后的目录结构：



共享

对于Monorepo，共享是最重要的一个优势。所以，我们来做一些共享的事情。

1. 在packages下加个 `share` 空间，作为共享资源空间，并创建共享文件 `Fish.js`：

```
npm init -w projects/share -y
mkdir projects/share/js
touch projects/share/js/Fish.js`
```

```
// projects/share/js/Fish.js
class Fish {
 constructor(name, age) {
 this.name = name
 this.age = age
 }
 swim() {
 console.log('swim~')
 }
 print() {
 return '🐠 '
 }
}
module.exports = Fish
```

子项目`zoo`的入口文件改为：

```
// projects/zoo/src/index.js
const Fish = require('share/js/Fish')
const fish = new Fish()
document.getElementById('app').textContent = fish.print()
```

运行`zoo`的`dev`看效果：🐟

修改子项目`shop`的入口文件后，会出现同样的效果。

也就是说，share文件夹下的东西，`zoo`和`shop`可以公用了，需要做的仅仅是新增一个webpack的`alias`而已！🎉

优点：便捷的项目管理，共享依赖项，统一的配置和脚本，子项目的独立性，简化的构建和部署，交叉引用和调试

风险：

坑一：npm install 默认模式的坑

npm v7开始，install会默认安装依赖包中的`peerDependencies`声明的包。新项目可能影响不大，但是，如果你是改造现有的项目。因为用了统一管理的方式，所以一般都会把子项目中的lock文件删掉，在根目录用统一的lock管理。然后，当你这么做了以后，可能坑爹的事情就出现了。 场景：我的子项目中用的是`webpack4`，然后，我们的构建相关的工具（webpack、babel、postcss、eslint、stylint等）都会封装到基础包中。这些包的依赖包中有一个包，在`package.json`声明中使用这样写：

json

复制代码

```
"peerDependencies": { "webpack": "^5.1.0" },
```

然后，在根目录中`npm install`，然后再跑子项目发现项目跑不起来了。原因就是，项目居然安装的是`webpack5`的版本！

解决方案

- 方案1：在子项目的`package.json`中显示声明用的`webpack`版本；
- 方案2：去github和作者商量修复依赖包，如果他的包即兼容`webpack4`也兼容`webpack5`，应该写成，把声明改为：`"webpack": "^4.0.0 || ^5.0.0"`
- 方案3：`npm install --legacy-peer-deps`