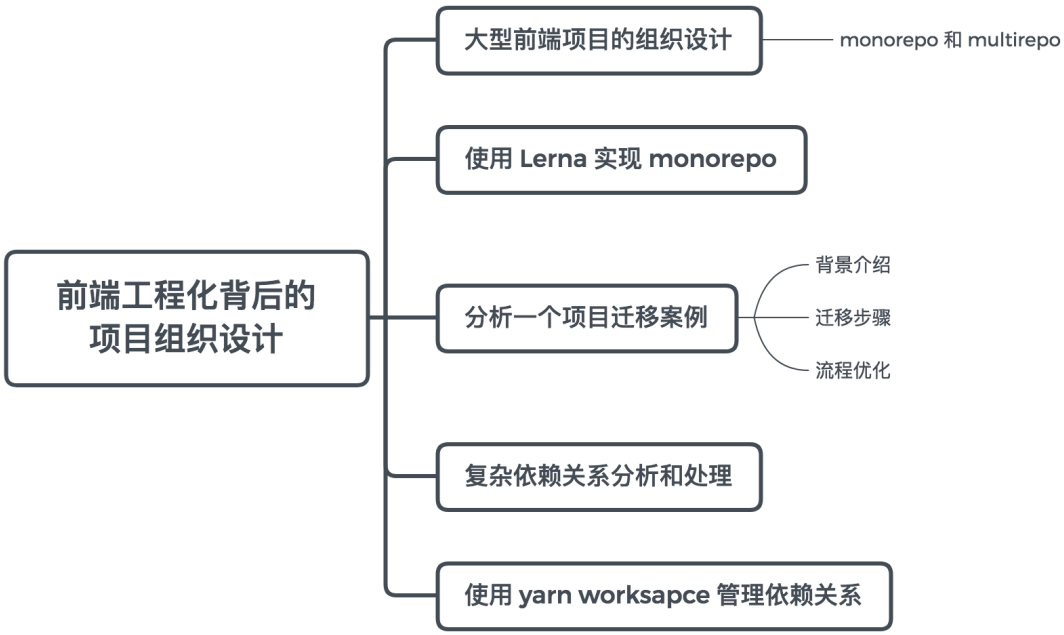
前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏
查看详情 >

前端工程化背后的项目组织设计（下）

承接上一节的内容，本节来继续学习前端工程化中依赖关系相关的内容。在此之前，先回顾一下「项目组织」主题的知识点：



说到项目中的依赖关系，我们往往会想到使用 yarn/npm 解决依赖问题。依赖关系大体上可以分为：

嵌套依赖

扁平依赖

项目中，我们引用了三个包：PackageA、PackageB、PackageC， 它们都依赖

知乎

PackageC 在各自的 node_modules 目录中分别含有 PackageD，那么我们将其理解为嵌套依赖：

```
PackageA
   node_modules/PackageD@v1.1
PackageB
   node_modules/PackageD@v1.2
PackageC
   node_modules/PackageD@v1.3
```

如果在安装时，先安装了 PackageA，那么 PackageA 依赖的 PackageD 版本成为主版本，它和 PackageA、PackageB、PackageC 一起平级出现，我们认为这是扁平依赖。此时 PackageB、PackageC 各自的 node_modules 目录中也含有各自的 PackageD 版本：

```
PackageA
PackageD@v1.1
PackageB
   node_modules/PackageD@v1.2
PackageC
   node_modules/PackageD@v1.3
```

npm 在安装依赖包时，会将依赖包下载到当前的 node_modules 目录中。对于嵌套依赖和扁平依赖的话题，npm 给出了不同的处理方案：npm3 以下版本在依赖安装时，非常直接，它会按照包依赖的树形结构下载到本地 node_modules 目录中，也就是说，每个包都会将该包的依赖放到当前包所在的 node_modules 目录中。

这么做的原因可以理解：它考虑到了包依赖的版本错综复杂的问题，同一个包因为被依赖的关系原因会出现多个版本，保证树形结构的安装能够简化和统一对于包的安装和删除行为。这样能够简单地解决多版本兼容问题，可是也带来了较大的冗余。

npm3 则采用了扁平结构，但是更加智能。在安装时，按照 package.json 里声明的顺序依次安装包，遇到新的包就把它放在第一级 node_modules 目录，后

知乎

判断包版本，如果版本一样则跳过安装，否则会按照 npm2 的方式安装在树形目录结构下。

npm3 这种安装方式只能够**部分解决**问题，比如：项目里依赖模块 PackageA、PackageB、PackageC、PackageD, 其中 PackageC、PackageB 依赖模块 PackageD v2.0，A 依赖模块 PackageD v1.0。那么可能在安装时，先安装了 PackageD v1.0，然后分别在 PackageC、PackageB 树形结构内部分别安装 PackageD v2.0。这也是一定程度的冗余。为了解决这个问题，因此也就有了 npm dedupe 命令。

npm 和 yarn 的内容足以单独开讲，我们这里不再展开。

另外，为了保证同一个项目中不同团队成员安装的版本依赖相同，我们往往使用 package-lock.json 或 yarn-lock.json 这类文件通过 git 上传以共享。在安装依赖时，依赖版本将会锁定。

这些内容与开发息息相关，但是往往被开发者所忽视。依赖问题说小很小，说复杂却也很复杂，我们再来看一个循环依赖的问题。

复杂依赖关系分析和处理

前端项目，安装依赖非常简单：

```
npm install / yarn add
```

安装一时爽，而带来的依赖关系慢慢地会让人头大。依赖关系的复杂性带来的主要副作用有就是**循环依赖**。

这里我们来重点说一下。简单来说，循环依赖就是模块 A 和模块 B 相互引用，在不同的模块化规范下，对于循环依赖的处理不尽相同。

Node.js 中，我们制造一个简单的循环引用场景。

模块 A：

```
exports.loaded = false
const b = require('./b')
module.exports = {
    bWasLoaded: b.loaded,
    loaded: true
}
```

模块 B：

```
exports.loaded = false
const a = require('./a')
module.exports = {
    aWasLoaded: a.loaded,
    loaded: true
}
```

在 index.js 中调用：

```
const a = require('./a');
const b = require('./b')
console.log(a)
console.log(b)
```

这种情况下，并未出现死循环崩溃的现象，而是输出：

```
{ bWasLoaded: true, loaded: true }
{ aWasLoaded: false, loaded: true }
```

原因是模块加载过程的缓存机制：Node.js 对模块加载进行了缓存。按照执行顺序，第一次加载 a 时，走到 const b = require('./b')，这样直接进入模块 B 当中，此时模块 B 中 const a = require('./a')，模块 A 已经被缓存，因此模块 B 返回的结果为：

```
{
```

```
    loaded: true
}
```

模块 B 加载完成，回到模块 A 中继续执行，模块 A 返回的结果为：

```
{
    aWasLoaded: true,
    loaded: true
}
```

据此分析，我们不难理解最终的打印结果。也可以总结为：

Node.js，或者 CommonJS 规范，得益于其缓存机制，在遇见循环引用时，程序并不会崩溃。但这样的机制，仍然会有问题：它只会输出已执行部分，对于未执行部分，export 内容为 undefined。

ES 模块化与 CommonJS 规范不同，ES 模块不存在缓存机制，而是动态引用依赖的模块。

《Exploring ES6》一文中的示例很好地阐明了这样的行为：

```
//------ a.js ------
import {bar} from 'b'; // (i)
export function foo() {
    bar(); // (ii)
}

//------ b.js ------
import {foo} from 'a'; // (iii)
export function bar() {
    if (Math.random()) {
        foo(); // (iv)
    }
}
```

```
//------ a.js ------
var b = require('b');
function foo() {
   b.bar();
}
exports.foo = foo;

//------ b.js ------
var a = require('a');
function bar() {
   if (Math.random()) {
       a.foo();
   }
}
exports.bar = bar;
```

如果模块 a.js 先被执行，a.js 依赖 b.js，在 b.js 中，因为 a.js 此刻还并没有暴漏出任何内容，因此如果在 b.js 中，对于顶层 a.foo() 的调用，会得到报错。但是如果 a.js 模块执行完毕后，再调用 b.bar()，b.bar() 当中的 a.foo() 可以正常运行。

但是这样的方式的局限性：

如果 a.js 采用 module.exports = function () { ··· } 的方式，那么 b.js 当中的 a 变量在赋值之后不会二次更新。

ESM 不会存在这样的局限性。ESM 加载的变量，都是动态引用其所在的模块。只要引用是存在的，代码就能执行。回到：

```
//------ a.js ------
import {bar} from 'b'; // (i)
export function foo() {
   bar(); // (ii)
}
```

```
import {foo} from 'a'; // (iii)
export function bar() {
    if (Math.random()) {
        foo(); // (iv)
    }
}
```

代码，第 ii 行和第 iv 行，bar 和 foo 都指向原始模块数据的引用。ESM 的设计目的之一就是支持循环引用。

ES 的设计思想是：尽量静态化，这样在编译时就能确定模块之间的依赖关系。这也是 import 命令一定要出现在模块开头部分的原因。在模块中，import 实际上不会直接执行模块，而是只生成一个引用。在模块内真正引用依赖逻辑时，再到模块里取值。这样的设计非常有利于 tree shaking 技术的实现，我们在《深入浅出模块化相关话题（含 tree shaking）》课程中继续展开。

在工程实践中，循环引用的出现往往是由设计不合理造成的。如果使用 webpack 进行项目构建，可以使用 webpack 插件 circular-dependency-plugin 来帮助检测项目中存在的所有循环依赖。循环依赖这个问题说大不大，说小不小，我们应该尽可能在设计源头规避。

另外复杂的依赖关系还会带来以下等问题：

- 依赖版本不一致
- 依赖丢失

针对此，需要开发者根据真实情况进行处理，同时，合理使用 npm/yarn 工具，也能起到非常关键的作用。

笔者团队中通过：

```
"scripts": {
    // ...
```

```
    // ...
}
```

即

```
yarn run analyzeDeps
```

来对依赖进行分析。具体流程是 analyzeDeps 脚本会对依赖版本冲突和依赖丢失的情况进行处理，这个过程依赖 missingDepsAnalyze 和 versionConflictsAnalyze 两个任务：

其中 missingDepsAnalyze 依赖 depcheck，depcheck 可以找出哪些依赖是没有用到的，或者对比 package.json 声明中缺少的依赖项。

同时 missingDepsAnalyze 会读取 lerna.json 配置，获得项目中所有 package，接着对所有 package 中的 package.json 进行遍历，检查是否存在相关依赖，如果不存在则自动执行 yarn add XXXX 进行安装。

versionConflictsAnalyze 任务类似，只不过在获得每个 package 的 package.json 中定义的依赖之后，检查同一个依赖是否有重复声明且存在版本不一致的情况。对于版本冲突，采用交互式命令行，让开发者选择正确的版本。

相关代码并不难实现，感兴趣的读者可以在评论区交流或者向我提问，出于隐私原因，这里不再贴出。

使用 yarn workspace 管理依赖关系

monorepo 项目中依赖管理问题值得重视。现在我们来看一下非常流行的 yarn workspace 如何处理这种问题。

workspace 的定位为：

翻译过来，workspace 能帮助你更好地管理有多个子 package 的 monorepo。开发者既可以在每个子 package 下使用独立的 package.json 管理依赖，又可以享受一条 yarn 命令安装或者升级所有依赖的便利。

引入 workspace 之后，在根目录执行：

```
yarn install / yarn updrade XX
```

所有的依赖都会被安装或者更新。

当然，如果只想更新某一个包内的版本，可以通过以下代码完成：

```
yarn workspace upgrade XX
```

在使用 yarn 的项目中，如果想使用 yarn workspace，我们不需要安装其他的包，只要简单更改 package.json 便可以工作：

```
// package.json
{
 "private": true,
 "workspaces": ["workspace-1", "workspace-2"]
}
```

需要注意的是，**如果需要启用 workspace，那么这里的 private 字段必须设置成 true。** 同时 workspaces 这个字段值对应一个数组，数组每一项是个字符串，表示一个 workspace（可以理解为一个 repo）。

接着，我们可以在 workspace-1 和 workspace-2 项目中分别添加 package.json 内容：

```
{
 "name": "workspace-1",
 "version": "1.0.0",
```

```
 }
}
```

以及：

```
{
 "name": "workspace-2",
 "version": "1.0.0",

 "dependencies": {
   "react": "16.2.3",
   "workspace-1": "1.0.0"
 }
}
```

执行 yarn install 之后，发现项目根目录下的 node_modules 内已经包含所有声明的依赖，且各个子 package 的 node_modules 里面不会重复存在依赖，只会有针对根目录下 node_modules 中的 React 引用。

我们发现，yarn workspace 跟 Lerna 有很多共同之处，解决的问题也部分重叠。**下面我们对比一下 workspace 和 Lerna。**

> yarn workspace 寄存于 yarn，不需要开发者额外安装工具，同时它的使用也非常简单，只需要在 package.json 中进行相关的配置，不像 Learn 那样提供了大量 API
>
> yarn workspace 只能在根目录中引入，不需要在各个子项目中引入

事实上，Lerna 可以与 workspace 共存，搭配使用能够发挥更大作用。在我们团队中：Lerna 负责版本管理与发布，依靠其强大的 API 和设置，做到灵活细致；workspace 负责依赖管理，整个流程非常清晰。

在 Lerna 中使用 workspace，首先需要修改 lerna.json 中的设置：

```
  "npmClient": "yarn",
  "useWorkspaces": true,
  ...
}
```

然后将根目录下的 package.json 中的 workspaces 字段设置为 Lerna 标准 packages 目录：

```
{
 ...
 "private": true,
 "workspaces": [
   "packages/*"
 ],
 ...
}
```

注意：如果我们开启了 workspace 功能，lerna.json 中的 packages 值便不再生效。原因是 Lerna 会将 package.json 中 workspaces 中所设置的 workspaces 数组作为 lerna packages 的路径，也就是各个子 repo 的路径。换句话说，Lerna 会优先使用 package.json 中的 workspaces 字段，在不存在该字段的情况下，再使用 lerna.json 中的 packages 字段。如果未开启 workspace 功能，lerna.json 配置为：

```
{
 "npmClient": "yarn",
 "useWorkspaces": false,
 "packages": [
   "packages/11/*",
   "packages/12/*"
 ]
}
```

根目录下的 package.json 配置为：

```
{
   "private": true,
   "workspaces": [
     "packages/21/*",
     "packages/22/*",
   ],
     ...
}
```

那么这就意味着使用 yarn 管理的是 package.json 中 workspaces 所对应的项目路径下的依赖：`packages/21/*` 以及 `packages/22/*`。而 Leran 管理的是 lerna.json 中 packages 所对应的 `packages/11/*` 以及 `packages/12/*` 的项目。

总结

本节主要抛出了大型前端项目的组织选型问题，着重分析了 monorepo 方案，内容注重实战。对于大型代码库的组织，本节梳理出一条完善的工作流程。找到适合自己团队的风格，是一名合格的开发者所需要具备的技能。

但是关于 npm 和 yarn 以及所牵扯出的依赖问题、monorepo 设计问题仍然将是挑战，其中的话题仍然值得深挖和系统展开。具体工程化项目的代码组织选型和设计，开发者一定要通过动手来理解。在此学习过程中，有任何疑问和想法，都欢迎与我交流，也希望能有更多机会和大家交流。

点击查看下一节

代码规范工具及背后技术设计（上）