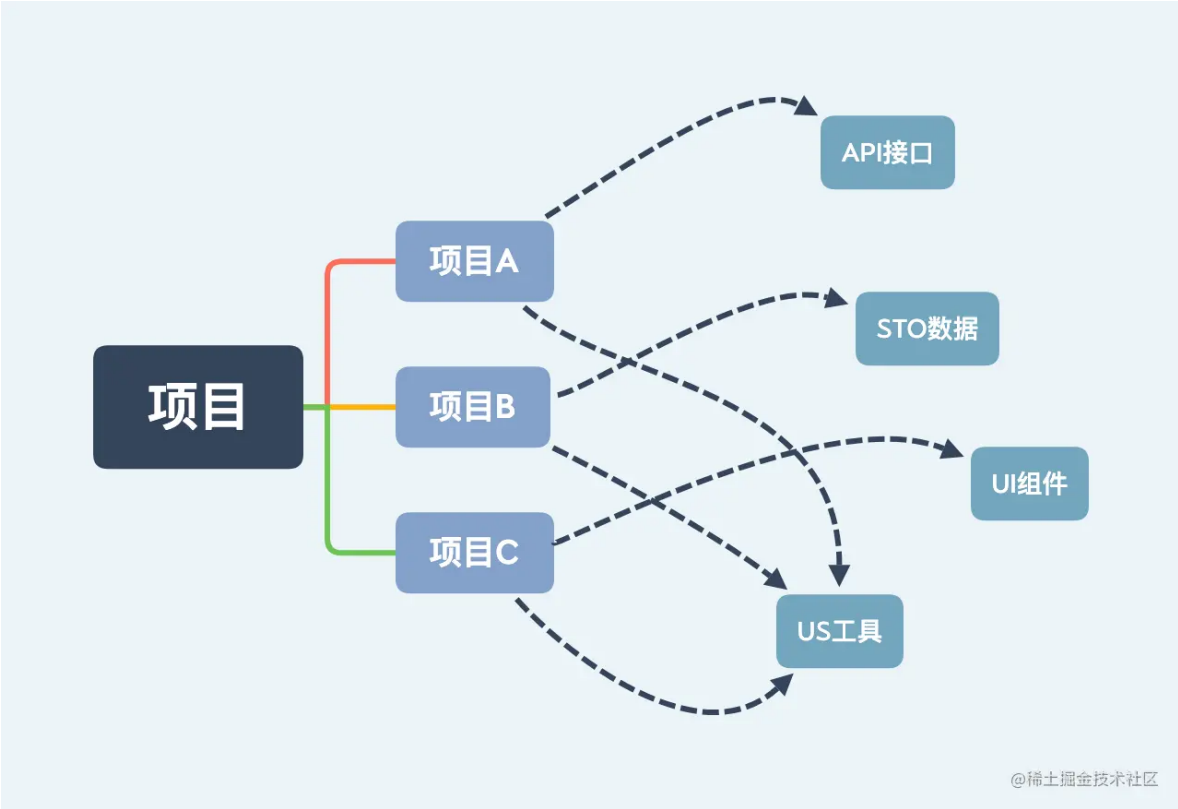
> 技术要点：Yarn、Learn

前言

当一个仓库规模逐渐升级并拆分为多个模块时，使用 `Monorepo` 的方式管理仓库再也适合不过了。这些模块通常在同一仓库中依赖其他不同模块, 同时不同模块间还会互相依赖, 那管理与组织这些依赖显然很重要。



为了解决这些依赖关系， `lerna` 就此诞生。 `lerna` 是一种 `Monorepo` 解决方案，用于管理包括多个子包的仓库，可优化使用 `Git` 与 `Npm` 管理 `多包仓库` 的工作流程。日常开发可在主仓库中管理多个模块，结合其他工具解决多个子包互相依赖的问题。本章将带领你**基于Yarn与Lerna搭建多包仓库基建模板**，将那些仓库内容出现关联的模块统一使用 `Monorepo` 的方式管理起来，降低日常开发的沟通难度，提升仓库管理的便利性。

背景：Monorepo仓库带来的收益

`lerna` 诞生于 `babel` 的仓库管理模式中，因为 `babel` 实在是太多模块了，甚至多到不可管理，就这样促使了 `lerna` 的诞生。像平时使用的 `angular` 、 `react` 、 `vue` 、

jest 等，其仓库都使用 lerna 管理。

打开它们的仓库，无一例外都是以下目录结构。

```txt
project
├─ lib1
│  ├─ src
│  └─ package.json
├─ lib2
│  ├─ src
│  └─ package.json
├─ lib3
│  ├─ src
│  └─ package.json
└─ package.json
```

使用基于 Monorepo 的 lerna 管理仓库，能为仓库带来更好的收益。

> 节省存储空间

若一个仓库中多个子包都依赖 react 与 react-dom ，在为每个子包安装依赖时会在各自的 node_modules 文件夹中产生大量冗余的 Npm模块 。

若使用 lerna ，会将相同版本的 Npm模块 提升到仓库根目录中的 node_modules 文件夹，以降低模块沉积。

> 解决依赖升级

若一个仓库中多个子包都依赖某一个或多个其他子包 X ，当 X 升级版本时需将它们发布到 Npm公有仓库 ，再更新依赖 X 的子包的依赖，才能将改动的代码应用起来。

若使用 lerna ，则会直接跳过将 X 发布到 Npm公有仓库 的流程，子包可在本地环境直接 link 到 X 。

> 共用同一仓库

若一个仓库中多个子包都因为需求更新而发版，那将引起一堆重复操作，导致每个子包都要走一次提交流程。

若使用 `lerna` ，则解决了这些子包的工作流程，就可放心将它们存放到同一仓库中管理了。

其实 `lerna` 的最终目标还是围绕着 `统一工作流程` 与 `分割通用代码` ，毕竟当仓库内容出现关联时，无任何一种调试方式比源码放在一起更高效，而 `lerna` 就是解决源码放在一起带来的副作用，使用 `Monorepo` 的方式解决这些问题，使 `多包仓库` 管理起来更高效。

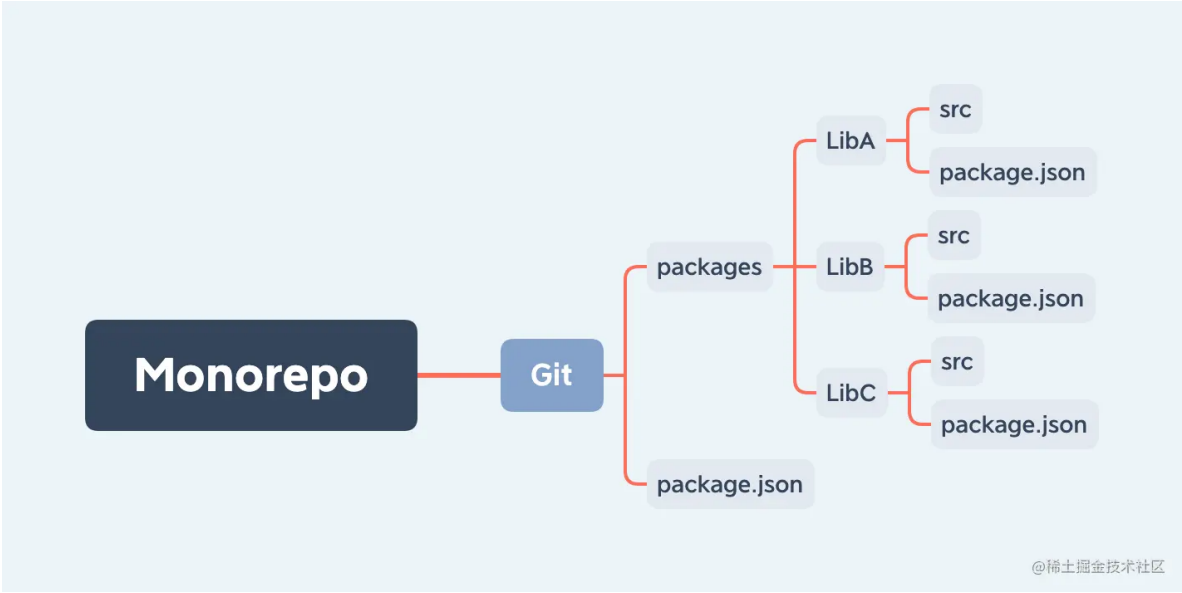
虽然 `lerna` 管理仓库会导致仓库的存储空间变得更大，但其带来的优势也是常见方案不能企及的。

```txt
仓库管理的工作流程包括但不限于构建、测试、打包、发布、部署等流程。
```

方案：基于Yarn与Lerna搭建多包仓库基建模板

在使用 `lerna` 前，一个 `Multirepo仓库` 的组织形式可能是这样的。

在使用 `lerna` 后， `Multirepo仓库` 就会变成 `Menorepo仓库` ，其组织形式就会变成这样。



通过上图基本可判断出使用 `lerna` 前后的差异。 `lerna` 显式地改变项目结构，把多个子包合并为一个仓库，然后使用 `packages` 文件夹存放每个子包，以使用一个仓库管理多个子包。另外还可用 `lerna` 提供的命令轻松管理 `Menorepo仓库` ，掌握以下命令就能操作 `lerna` 。

命令	功能	描述
`lerna init`	初始项目	
`lerna boostrap`	安装依赖	自动解决子包间的依赖关系 子包内部互相依赖会使用软链接处理
`lerna clean`	卸载依赖	只能卸载每个子包的 `node_modules` 不能卸载根目录的 `node_modules`
`lerna create <name>`	新增子包	在 `packages` 文件夹中创建由 `lerna` 管理的子包
`lerna add <pkg>`	安装模块	为所有子包安装模块 可通过 `--scope=pkg` 安装指定子包模块
`lerna run <script>`	执行命令	为所有子包执行命令 可通过 `--scope=pkg` 执行指定子包命令
`lerna version`	标记版本	标记存在修改行为的子包
`lerna publish [bump]`	发布子包	发布全部 `private` 不为 `true` 的子包

版本管理

lerna 有两种版本管理模式，分别是 固定模式(Fixed) 与 独立模式(Independent) 。

> 固定模式

固定模式 通过在 lerna.json 中指定 version 统一管理版本。该模式自动将所有子包版本捆绑在一起，迭代任何一个或多个子包都会导致所有子包版本升级。

angular 早期版本迭代使用该模式处理，导致很多无任何迭代的子包因为其他迭代的子包升级而升级。 固定模式 的优势是统一管理版本，让开发者专注于开发而无需理会版本的迭代过程，反正每次发生迭代就统一升级所有子包版本。其劣势也很明显，那些无任何迭代的子包无缘无故被升级，不符合 变更就发版 的开发逻辑，导致追溯源码版本时很易产生多个版本源码一样的情况。

执行 lerna init 初始项目，就可生成 固定模式 的项目结构。其中 package.json 与 lerna.json 的内容分别如下。

json

```
// package.json
{
        "name": "root",
        "private": true,
        "devDependencies": {
                "lerna": "4.0.0"
        }
}
```

json

```
// lerna.json
{
        "version": "0.0.0",
        "packages": [
                "packages/*"
        ]
}
```

> 独立模式

独立模式 通过在 lerna.json 中指定 version 为 independent ，允许具体维护每个子包版本。子包版本由每个子包的 package.json 的 version 维护，每次发布时都会收

到一个提示，以说明每个迭代子包是主版本(major)、次版本(minor)、修订版本(patch)还是自定义更改版本(custom change)。

每次发布版本时， lerna.json 的 version 不会发生变化，始终保持为 independent 。得益于 独立模式 能更好地区分每个子包版本，所以目前很多明星项目都使用 独立模式 处理子包版本。

执行 lerna init --independent 初始项目，就可生成 独立模式 的项目结构。其中 package.json 与 lerna.json 的内容分别如下。

```json
// package.json
{
        "name": "root",
        "private": true,
        "devDependencies": {
                "lerna": "4.0.0"
        }
}
```

```json
// lerna.json
{
        "version": "independent",
        "packages": [
                "packages/*"
        ]
}
```

为了更好地管理每个子包版本，选择 独立模式 会更适合仓库的发展。因为子包间可能存在依赖关系，例如 子包B 依赖 子包A ，因此需将 子包A 链接到 子包B 的 node_module 中，一旦子包间的依赖关系很多，手动管理这些 link 操作是很麻烦的。能不能 自动化 执行这些 link操作 ，根据拓扑排序将各个依赖 link 起来?

lerna 作为一个 Menorepo 解决方案并不提供这些依赖关系复杂化的处理，但 yarn 的 Workspaces 却具备这样的功能。

Workspaces 顾名思义是 工作空间 ，其提供一种机制使项目作用域在一个片区产生隔离效果，使内部模块通过软链接(symlink)的方式产生依赖但又不影响全局依赖。其优点也不言而喻。

- 开发多个互相依赖的模块时， `Workspaces` 会自动对模块的引用设置软链接，比 `yarn link` 更方便且链接仅局限在当前 `Workspaces` 中，不会对整个系统造成影响
- 所有模块的依赖会安装在根目录的 `node_modules` 文件夹中，节省磁盘空间且给 `yarn` 更大的依赖优化空间
- 所有模块使用同一个 `yarn.lock` ，减少依赖冲突且易于审查

若使用 `lern` 的 `独立模式` 管理仓库，为了解决互相依赖的问题必须引用 `yarn` 的 `Workspaces` 。执行 `npm i -g yarn` 安装 `yarn` ，输出版本表示安装成功。

在 `package.json` 中指定 `workspaces` 并修改 `name` 。

```json
{
        "name": "@yangzw/bruce", // 自行修改
        "private": true,
        "devDependencies": {
                "lerna": "4.0.0"
        },
        "workspaces": [
                "packages/*"
        ]
}
```

在 `lerna.json` 中指定以下字段并删除 `packages` 。

- **npmClient**：模块管理工具，可选 `npm/yarn`
- **useWorkspaces**：是否使用 `yarn` 的 `Workspaces`
- **command**：命令配置
  - **publish.ignoreChanges**：指定文件夹或文件在改动情况下不会被发布
  - **publish.message**：发布时提交消息的格式
  - **publish.registry**：发布到指定 `Npm镜像`

```json
{
        "version": "independent",
        "npmClient": "yarn", // 主要
        "useWorkspaces": true, // 主要
        "command": {
                "publish": {
                        "ignoreChanges": [
                                ".DS_Store",
                                "node_modules",
                                "package-lock.json",
```

```
                "yarn.lock"
            ],
            "message": "chore: publish release %v",
            "registry": "https://registry.npmjs.org/"
        }
    } // 次要
}
```

子包管理

执行 `lerna create <name>` 创建三个子包。 `app` 是一个打包应用的工具， `ui` 是一个通用组件库， `us` 是一个通用工具库。

```
lerna create app && lerna create ui && lerna create us
```

推荐手动创建每个子包的内容，这样更易掌握 多包仓库 的项目结构。在每个子包的 `package.json` 中指定 `name` 为 范围模块 。

```json
{
    "name": "@yangzw/bruce-app" // 其他子包同样处理
}
```

依赖管理

三个子包都缺乏内容，演示起来有点空荡，把 `packages` 文件夹中的内容删除，复制该目录中的 `app` 、 `ui` 和 `us` 到 `packages` 文件夹中。

`app` 与 `ui` 都依赖了 `us` ，与上述情况保持一样。接着增加**整包安装依赖**与**整包卸载依赖**的操作。在根目录的 `package.json` 中指定 `srcipts` 。

```json
{
    "scripts": {
        "clean": "lerna clean && rimraf node_modules package-lock.json yarn.lock
        "init": "lerna bootstrap"
    }
}
```

当首次克隆仓库或重装依赖时，执行 `yarn run init` 。该命令会执行 `lerna bootstrap` 自动处理好三个子包间的依赖关系并将部分依赖提升到根目录的 `node_modules` 文件夹中。

当卸载依赖时，执行 `yarn run clean` 。该命令会执行 `lerna clean` 卸载每个子包的 `node_modules` 文件夹，但会留下根目录的 `node_modules` 文件夹。还记得第11章使用 `rimraf` 卸载根目录的 `node_modules` 文件夹吗？将这两个命令组合起来即可。

当依赖乱掉或工程混乱时，根据顺序执行 `yarn run clean` 与 `yarn run init` ，让仓库的依赖关系保持最佳状态。

在开发时肯定会安装某些模块。使用 `lerna` 安装依赖有些特殊，需分三种情况考虑。

若全部子包都安装 `semver` ，那执行以下命令，不带任何参数。安装好的 `semver` 会存放到根目录的 `node_modules` 文件夹中。

```
lerna add semver
```

若只有 `app` 安装 `semver` ，执行以下命令需带上 `--scope` 指定子包，子包名称以 `package.json` 的 `name` 为准。

```
lerna add semver --scope @yangzw/bruce-app
```

若 `app` 依赖了 `us` ，为 `app` 安装 `us` 也通过上述方式完成。

```
lerna add @yangzw/bruce-us --scope @yangzw/bruce-app
```

命令管理

有些子包可能需输出编译后的 `bundle文件` ，通常会在 `package.json` 中指定 `scripts` ，使用 `build` 字段映射相关打包命令。

上述所有子包都存在 `build` 命令，执行以下命令就会顺序为每个子包执行 `yarn run build` 。

```
lerna run build
```

为了方便控制，在根目录的 `package.json` 中指定 `srcipts` 。

```json
{
        "scripts": {
                "build": "lerna run build"
        }
}
```

若只有 `app` 执行 `build` ， 执行以下命令需带上 `--scope` 指定子包。

```
lerna run build --scope @yangzw/bruce-app
```

各个子包的依赖还可能存在顺序关系，例如 `app` 依赖 `us` ， 因此必须先执行 `yarn run build --scope @yangzw/bruce-us` ， 再执行 `yarn run build --scope @yangzw/bruce-app` 。这实际上要求命令以一种拓扑排序的规则进行。

很不幸 `yarn` 的 `Workspaces` 暂时并未支持根据拓扑排序规则执行命令，幸运的是 `lerna` 支持根据拓扑排序规则执行命令， `--sort` 参数可控制以拓扑排序规则执行命令。

```
lerna run --stream --sort build
```

发布管理

在发布子包前需执行 `lerna version` 迭代版本，执行该命令时做了以下工作。

- 标识自上个标记版本依赖有更新的子包
- 提示输入新版本
- 修改子包的元数据以反映新版本，在根目录与每个子包中运行适当的生命周期脚本
- 提交与标记修改记录
- 推送到 `Git` 远端

> 版本更新原则

- [x] 存在 BREAKING CHANGE提交 ： 需更新 主版本
- [x] 存在 FEAT提交 ： 需更新 次版本
- [x] 存在 FIX提交 ： 需更新 修订版本

发布子包需执行 `lerna publish` ，该命令既可包括 `lerna version` 的工作，也可只做发布操作。若子包的 `package.json` 的 `private` 设置为 `true` ，则不会被发布出去。

发布子包可分为以下场景。

> lerna publish

该命令实际封装了 `lerna version && lerna publish from-git` ，用于发布自上次发布以来有更新的子包， 上次发布 也是基于上次执行 `lerna publish` 而言。

> lerna publish from-git

发布当前提交中已标记的子包。 `from-git` 根据 `git commit` 中的 `annotaed tag` 发包。

> lerna publish from-package

发布 Npm镜像 中不存在的最新版本的子包。 `from-package` 根据 `package.json` 的 `version` 变动发包。

总结

通过 `lerna` 就能很方便地管理 Menorepo方式 的仓库了，因此当仓库达到一定规模需拆分子包时就要考虑使用 Menorepo方式 管理仓库中所有子包了。以下情况，直接建议基于 `yarn` 与 `lerna` 完成一个 多包仓库 。

- 个人或团队开源的 框架类项目
- 公司内部的 组件库项目 或 工具库项目

本章内容到此为止，希望能对你有所启发，欢迎你把自己的学习心得打到评论区！

- [x] 示例项目：fe-engineering
- [x] 正式项目：bruce

留言

输入评论（Enter换行，Ctrl + Enter发送）

发表评论

全部评论（17）

litefe JY.3 3天前

请问老师公司多个业务项目是否可用这种方式管理

点赞 回复

AaronLei LV.3 JY.3 2月前

安装依赖是 lerna bootstrap， 不是lerna boostrap

点赞 回复

aComputerGeek JY.4 逗比一枚~ 3月前

在调研 monorepo的时候，用yarn workspace, lerna 这些在我们当时的项目上都出现一些问题。而lerna 所解决的问题， 感觉pnpm是有覆盖到空间节省这个部分的。

点赞 2

JowayYoung （作者） 3月前

每一个解决方案都有优势与劣势，根据自身需求探讨与使用就好

点赞 回复

qingkooo 1月前

补充，npm也有workspace

点赞 回复

amipei_y JY.4 4月前

有没有多仓库代码共享的方法😭，之前项目是一个仓库的，后来需求拆了几个仓库出来，单独部署，怎么管理这些共享的代码啊。

点赞 4

JowayYoung （作者） 3月前

拆包吧，做成一个menorepo仓库吧

点赞 回复

Mahousho... 2月前

我也有这个疑问，请问你是怎么做的呢

点赞 回复

查看更多回复

SteveCGC JY.3 大哥你搞前端，前端有... 4月前

lerna团队不是都不维护了吗？vue3团队都用的pnpm, pnpm会不会是更好的方案？能来一篇pnpm的文章吗

点赞 6

JowayYoung （作者） 4月前

不维护并不代表不能用，相反个人觉得lerna更稳定，后续会研究一下pnpm，再增加这部分内容

1 回复

Escape 回复 JowayYoung 4月前

同求能更新一篇pnpm如何管理多包仓库的文章吗

“不维护并不代表不能用，相反个人觉得lerna更稳定，后续会研究一下... ”

1 回复

查看更多回复