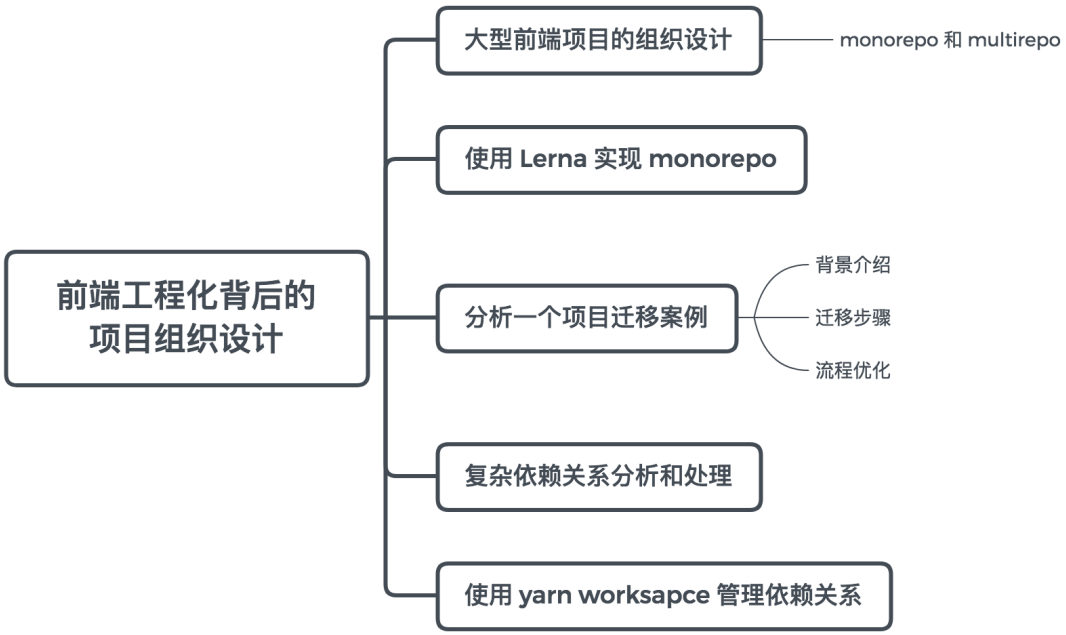
前端工程化背后的项目组织设计（上）

通过上一节的学习，我们看到了前端构建工具及其背后蕴含的技术设计。前端工程化包罗万象，本节课，我们将分析项目组织设计的相关话题，包括：



接下来，我们通过 2 节内容来学习这个主题。希望结束这个主题后大家可以从更高的视角看待项目管理和代码组织设计。

大型前端项目的组织设计

随着业务复杂度的直线上升，前端项目不管是从代码量上，还是从依赖关系上都爆炸式增长。同时，**团队中一般不止有一个业务项目，多个项目之间如何配合，如何维护相互关系？公司自己的公共库版本如何管理？这些话题随着业务扩展，纷纷浮出水面。**一名合格的高级前端工程师，在宏观上必需能妥善处理这些问题。

当然，不是每个开发者都有机会接触项目设计。如果读者没有面对过上述难题，也许并不容易理解这些问题究竟意味着什么。举个例子，团队主业务项目名为：App-project，这个仓库依赖了组件库：Component-lib，因此 App-project 项目的 package.json 会有类似的代码：

```
{
   "name": "App-project",
   "version": "1.0.0",
   "description": "This is our main app project",
   "main": "index.js",
   "scripts": {
       "test": "echo \\"Error: no test specified\\" &&
exit 1"
   },
   "dependencies": {
       "Component-lib": "^1.0.0"
   }
}
```

这时新的需求来了，产品经理需要更改 Component-lib 组件库中的 modal 组件样式及交互行为。作为开发者，我们需要切换到 Component-lib 项目，进行相关需求开发，开发完毕后进行测试。这里的测试包括 Component-lib 当中的单元测试，当然也包括在实际项目中进行效果验收。为方便调试，有经验的开发者也许会使用 npm link/yarn link 来开发和调试效果。当确认一切没问题后，我们还需要 npm 发包 Component-lib 项目，并提升版本为 1.0.1。在所有这些都顺利完成的基础上，才能在 App-project 项目中进行升级：

```
{
   //...
   "dependencies": {
       "Component-lib": "^1.0.1"
   }
}
```

这个过程已经比较复杂了。如果中间环节出现任何纰漏，我们都要重复上述所有步骤。另外，这只是单一依赖关系，现实中 App-project 不可能只依赖

Component-lib。这种项目管理的方式无疑是低效且痛苦的。那么在项目设计哲学上，有更好的方式吗?

monorepo 和 multirepo

答案是肯定的，管理组织代码的方式主要分为两种：

multirepo

monorepo

顾名思义，multirepo 就是将应用按照模块分别在不同的仓库中进行管理，即上述 App-project 和 Component-lib 项目的管理模式；而 monorepo 就是将应用中所有的模块一股脑全部放在同一个项目中，这样自然就完全规避了前文描述的困扰，不需要单独发包、测试，且所有代码都在一个项目中管理，一同部署上线，在开发阶段能够更早地复现 bug，暴露问题。

这就是项目代码在组织上的不同哲学：**一种倡导分而治之，一种倡导集中管理。**究竟是把鸡蛋放在同一个篮子里，还是倡导多元化，这就要根据团队的风格以及面临的实际场景进行选型。

我试着从 multirepo 和 monorepo 两种处理方式的**弊端说起**，希望给读者更多的参考和建议。

multirepo 存在以下问题：

开发调试以及版本更新效率低下

团队技术选型分散，不同的库实现风格可能存在较大差异（比如有的库依赖 Vue，有的依赖 React）

changelog 梳理困难，issues 管理混乱（对于开源库来说）

而 monorepo 缺点也非常明显：

库体积超大，目录结构复杂度上升

需要使用维护 monorepo 的工具，这就意味着学习成本比较高

清楚了不同项目组织管理的缺点，我们再来看一下社区上的经典选型案例。

Babel 和 React 都是典型的 monorepo，其 issues 和 pull requests 都集中到唯一的项目中，changelog 可以简单地从一份 commits 列表梳理出来。我们参看 React 项目仓库，从目录结构即可看出其强烈的 monorepo 风格：

```
react-16.2.0/
 packages/
   react/
   react-art/
   react-.../
```

因此，react 和 react-dom 在 npm 上是两个不同的库，它们只不过在 React 项目中通过 monorepo 的方式进行管理。至于为什么 react 和 react-dom 是两个包，我把这个问题留给读者。

而著名的 Rollup 目前是 multirepo 方式。对于 monorepo 和 multirepo，选择了 monorepo 的 Babel 贡献了文章：Why is Babel a monorepo?， 其中提到：

monorepo 的优势：

所有项目拥有一致的 lint，以及构建、测试、发布流程

不同项目之间容易调试、协作

方便处理 issues

容易初始化开发环境

易于发现 bug

monorepo 的劣势：

源代码不易理解

项目体积过大

这些分析与我们前文提到的类似。但是，从业内技术发展来看，monorepo 目前越来越受欢迎。了解了 monorepo 的利弊，我们应该如何实现 monorepo 呢?

使用 Lerna 实现 monorepo

Lerna 是 Babel 管理自身项目并开源的工具，官网对 Lerna 的定位非常简单直接：

A tool for managing JavaScript projects with multiple packages.

我们来建立一个简单的 demo，首先安装依赖，并创建项目：

```
mkdir new-monorepo && cd new-monorepo
npm init -y
npm i -g lerna（有需要的话要 sudo）
git init new-monorepo
lerna init
```

成功后，Lerna 会在 new-monorepo 项目下自动添加以下三个文件目录：

packages

lerna.json

package.json

我们添加第一个项目 module-1：

```
cd packages
mkdir module-1
```

```
cd module-1
npm init -y
```

这样，我们在 ./packages 目录下新建了第一个项目：module-1，并在 module-1 中添加了一些依赖，模拟更加真实的场景。同样的方式，建立 module-2 以及 module-3。

此时，读者可以自行观察 new-monorepo 项目下的目录结构为：

```
packages/
 module-1/
   package.json
 module-2/
   package.json
 module-3/
   package.json
```

接下来，我们退到主目录下，安装依赖：

```
cd ..
lerna bootstrap
```

关于该命令的作用，官网直述为：

Bootstrap the packages in the current Lerna repo. Installs all of their dependencies and links any cross-dependencies.

也就是说，假设我们在 module-1 项目中添加了依赖 module-2，那么执行 lerna bootstrap 命令后，会在 module-1 项目的 node_modules 下创建软链接直接指向 module-2 目录。也就是说 lerna bootstrap 命令会建立整个项目内子 repo 之间的依赖关系，这种建立方式不是通过「硬安装」，而是通过软链接指向相关依赖。

Linux 中关于硬链接和软链接的区别，可以参考文章：linux 硬链接与软链接。

在正确连接了 Git 远程仓库后，我们可以发布：

```
lerna publish
```

这条命令将各个 package 一步步发布到 npm 当中。Lerna 还可以支持自动生成 changelog 等功能。这里我们不再统一介绍。

到这里，你可能觉得 Lerna 还挺简单。但其实里面还是有更多学问，比如 Lerna 支持下面两种模式。

Fixed/Locked 模式

Babel 便采用了这样的模式。这个模式的特点是，开发者执行 lerna publish 后，Lerna 会在 lerna.json 中找到指定 version 版本号。如果这一次发布包含某个项目的更新，那么会自动更新 `version` 版本号。对于各个项目相关联的场景，这样的模式非常有利，任何一个项目大版本升级，其他项目的大版本号也会更新。

Independent 模式

不同于 Fixed/Locked 模式，Independent 模式下，各个项目相互独立。开发者需要独立管理多个包的版本更新。也就是说，我们可以具体到更新每个包的版本。每次发布，Lerna 会配合 Git，检查相关包文件的变动，只发布有改动的 package。

开发者可以根据团队需求进行模式选择。

我们也可以使用 Lerna 安装依赖，该命令可以在项目下的任何文件夹中执行：

```
lerna add dependencyName
```

Lerna 默认支持 hoist 选项，即默认在 lerna.json 中：

```
{ bootstrap: { hoist: true } }
```

这样项目中所有的 package 下 package.json 都会出现 dependencyName 包：

```
 packages/
module-1/
  package.json(+ dependencyName)
  node_modules
module-2/
  package.json(+ dependencyName)
  node_modules
module-3/
  package.json(+ dependencyName)
  node_modules
node_modules
     dependencyName
```

这种方式，会在父文件夹的 node_modules 中高效安装 dependencyName（Node.js 会向上在祖先文件夹中查找依赖）。对于未开启 hoist 的情况，执行 lerna add 后，需要执行：

```
lerna bootstrap --hoist
```

如果我们想有选择地升级某个依赖，比如只想为 module-1 升级 dependencyName 版本，可以使用 scope 参数：

```
lerna add dependencyName --scope=module-1
```

这时候 module-1 文件夹下会有一个 node_modules，其中包含了 dependencyName 的最新版本。

分析一个项目迁移案例

接下来，我选取一个正在线上运行的 multirepo 项目，并演示使用 Lerna 将其迁移到 monorepo 的过程。此案例来自 mitter.io，该团队以往一直以 multirepo 的形式维护以下几个项目：

@mitter-io/core，mitter.io SDK 核心基础库

@mitter-io/models，TypeScript models 库

@mitter-io/web，Web 端 SDK 应用

@mitter-io/react-native，React Native 端 SDK 应用

@mitter-io/node，Node.js 端 SDK 应用

@mitter-io/react-scl，React.js 组件库

背景介绍

项目使用 TypeScript 和 Rollup 工具，以及 TypeDoc 生成规范化文档。在使用 Lerna 做 monorepo 化之前，这样的技术方案带来的困扰显而易见，我们来分析一下当前技术栈的弊端，以及 monorepo 化能为这些项目带来哪些收益。

如果 @mitter-io/core 中出现任何一处改动，其他所有的包都需要升级到 @mitter-io/core 最新版本，不管这些改动是 feature 还是 bug fix，成本都比较大

如果所有这些包能共同分享版本，那么带来的收益也是非常巨大的

这些不同的仓库之间，由于技术栈近似，一些构建脚本大体相同，部署流程也都一致，如果能够将这些脚本统一抽象，也将带来便利

迁移步骤

我们运用 Lerna 构建 monorepo 项目，第一步：

```
mkdir my-new-monorepo && cd my-new-monorepo
git init .
lerna init
```

不同于之前的示例，这是从现有项目中导入，因此我们可以使用命令：

```
lerna import ~/projects/my-single-repo-package-1 --flatten
```

这行命令不仅可以导入项目，同时也会将已有项目中的 git commit 一并搬迁过来。我们可以放心地在新 monorepo 仓库中使用 git blame 来进行回溯。

如此一来，得到了这样的项目结构：

```
packages/
 core/
 models/
 node/
 react-native/
 web/
lerna.json
package.json
```

接下来，运行熟悉的：

```
lerna boostrap
lerna publish
```

进行依赖维护和发布。注意并不是每次都需要执行 lerna bootstrap，只需要在第一次切换到项目，安装所有依赖时运行。

对于每一个 package 来说，其 pacakge.json 文件中都有以下雷同的 npm script 声明。

```
"scripts": {
   ...
   "prepare": "yarn run build",
   "prepublishOnly": "./../../ci-scripts/publish-
tsdocs.sh",
   ...
   "build": "tsc --module commonjs && rollup -c
rollup.config.ts && typedoc --out docs --target es6 --
```

```
theme minimal --mode file src"
}
```

受益于 monorepo，所有项目得以集中管理在一个仓库中，这样我们将所有 package 公共的 npm 脚本移到 ./scripts 文件中。在单一的 monorepo 项目里，我们就可以在不同 package 之间共享构建脚本了 。

运行公共脚本时，有时候有必要知道当前运行的项目信息。npm 是能够读取到每个 package.json 信息的。因此，对每个 package，在其 package.json 中添加以下信息：

```
{
   "name": "@mitter-io/core",
   "version": "0.6.28",
   "repository": {
       "type": "git"
   }
}
```

之后，如下变量都可以被 npm script 使用：

```
npm_package_name = @mitter-io/core
npm_package_version = 0.6.28
npm_package_repository_type = git
```

流程优化

团队中正常的开发流程是每个程序员新建一个 git branch，通过代码审核之后进行合并。整套流程在 monorepo 架构下变得非常清晰，我们来梳理一下。

step1：当开发完成后，我们计划进行版本升级，只需要运行：lerna version

step2：Lerna 会提供交互式 prompt，对下一版本进行序号升级

```
 lerna version --force-publish
lerna notice cli v3.8.1
```

```
   lerna info current version 0.6.2
   lerna info Looking for changed packages since v0.6.2
   ? Select a new version (currently 0.6.2) (Use arrow
keys)
   ❯ Patch (0.6.3)
   Minor (0.7.0)
   Major (1.0.0)
   Prepatch (0.6.3-alpha.0)
   Preminor (0.7.0-alpha.0)
   Premajor (1.0.0-alpha.0)
   Custom Prerelease
   Custom Version
```

新版本被选定之后，Lerna 会自动改变每个 package 的版本号，在远程仓库中创建一个新的 tag，并将所有的改动推送到 GitLab 实例当中。

接下来，CI 构建实际上只需要两步：

Build 构建

Publish 发布

构建实际就是运行：

```
lerna bootstrap
lerna run build
```

而发布也不复杂，需要执行：

```
git checkout master
lerna bootstrap
git reset --hard
lerna publish from-package --yes
```

注意，这里我们使用了 lerna publish from-package，而不是简单的 lerna publish。因为开发者在本地已经运行了 lerna version，这时候再运行 lerna

publish 会收到「当前版本已经发布」的提示。而 from-package 参数会告诉 Lerna 发布所有非当前 npm package 版本的项目。

通过这个案例，我们了解了 Lerna 构建 monorepo 的经典套路，Lerna 还封装了更多的 API 来支持更加灵活的 monorepo 的创建，感兴趣的读者可以自行研究，欢迎在评论区留言讨论，或者直接向我提问。个人认为，未来 monorepo 和 multirepo 将会持续并存，每个开发者都应该根据项目特点来进行选择。

到此，我们分析了 multirepo 和 monorepo 方案的各自特点，通过实例和项目迁移了解了如何构建 monorepo 项目。但是，项目组织不光这些内容，下一节我们将讨论依赖关系这一话题。

总结

monorepo 目前来看是一个流行趋势，笔者为项目团队引入了 monorepo 的架构方案之后收益非常明显，我们也是国内最早采用 monorepo 架构的团队之一。

但是这篇课程难以做到面面俱到，并且任何一个项目都有自己的独立性和特殊性，究竟该如何组织调配、生产部署，需要每一个开发者开动脑筋。

比如：monorepo 方式会导致整个项目体积变大，在上线部署时，用时更长，甚至难以忍受。在工程中如何解决这类问题？针对于此，我设计了增量部署构建方案，通过分析项目依赖以及拓扑排序，优化项目编译构建，这里不再多做介绍。

如果对工程化话题格外感兴趣的读者较多，我会专门进行讲解。希望大家一起讨论。

点击查看下一节

前端工程化背后的项目组织设计（下）