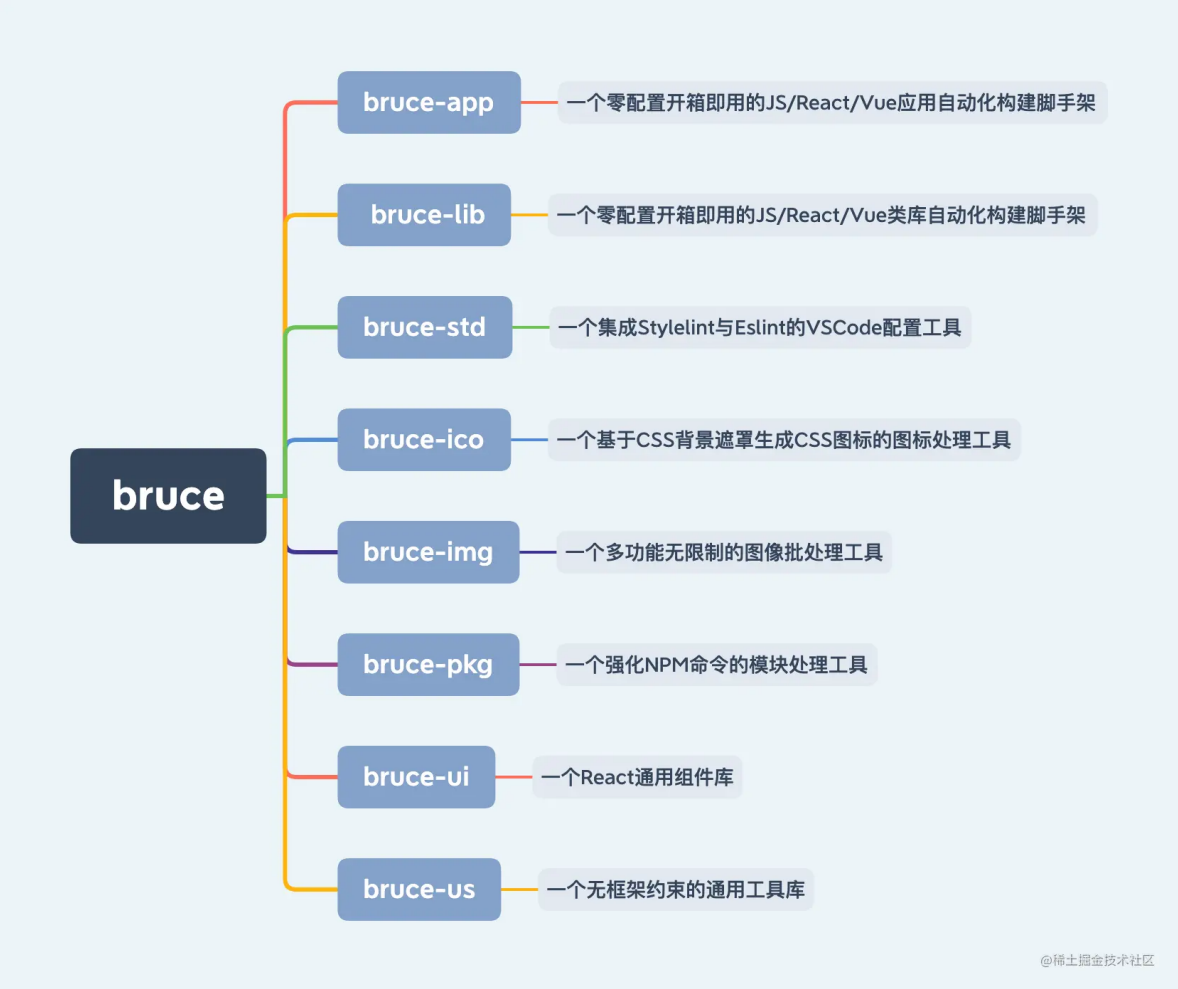
> 技术要点：Monorepo、Scope

前言

第15章开发了一个工具库，因为业务需求的不断发展，后续可能还会继续开发组件库、应用打包器、类库打包器等 前端工程化 项目，这样工程文件势必会增多。

以下是我最近一直在开发与迭代的项目bruce，初衷是开发一些功能强大的 前端工程化 工具集合，帮助开发者快速构建项目，使用工程化的手段解决开发问题。



根据以往方式，每个项目对应单独一个仓库，这种仓库管理风格称为 `Multirepo` 。其遵循着 模块化 与 组件化 的原则拆分代码，尽量不在一个项目中糅合太多东西，根据不同需求划分多个仓库，仓库间保持独立，每个项目都可独立开发独立部署，保证项目间不受其他项目影响。 `bruce` 整体结构如下，包括 `8` 个仓库。

```txt
bruce-app
├ src
└ package.json
bruce-lib
├ src
└ package.json
bruce-std
├ src
└ package.json
... # 另外5个仓库
```

本章将带领你**基于Monorepo模式拆分仓库**，了解 Monorepo 的收益与落地，着手改造多个仓库的项目结构，基于 Npm Scope 发布一个仓库中的多个 Npm模块 。

背景：Multirepo引发的痛点

这样的前端业务通常涉及多个仓库，时间一长，多个仓库共同管理的弊端就会日益显露。

代码复用

在维护多个项目时肯定会遇到一些公共逻辑被复用的情况，很多同学可能都会把这些公共逻辑复制多份，应用到不同项目中，优点是简单快捷无脑，缺点是万一出现问题维护成本可高了。

稍微具备一点 前端工程化 知识的同学可能会把这些公共逻辑封装为一个 Npm模块 并发布到 Npm公有仓库 供其他项目安装。这样貌似解决了公共逻辑被复用的问题，若公共逻辑出现 Bug ，那得修复后再次发布到 Npm公有仓库 并再次安装。

可能只改一行代码也需走这么多流程，但开发阶段很难保证不出任何 Bug ，细想下有必要一直走这些重复步骤吗？上述问题可能是 Multirepo 的致命伤害，因为不同项目的工作区的割裂，导致复用代码的成本很高，开发调试的流程很繁琐，甚至在基础库频繁改动的情况下让人感到很抓狂，体验很差。

基建复用

上述公共逻辑被复用的情况，在项目基建中也一样存在。每个项目都会拥有自己的环境配置、构建、打包、 CI/CD 等，这些代码块肯定也会存在很多公共逻辑被复用

的情况。

例如基于 `rollup` 编写一个打包脚本用于打包工具库，而打包组件库也可用到该脚本，只在脚本中加入一些处理样式的代码块，但不能认为两种情况完全不同而单独区分。有时发布一个需求可能同时需发布多个项目，项目间存在构建、测试、打包、部署和发布的规范不统一也很严峻，这样维护起来就更麻烦了。

版本管理

管理多个项目时，每次定义与更新版本都是一件很头疼的事情。刚开始每个项目的版本都是 `1.0.0` ，经过不同需求的迭代，后续版本就会变得很不协调。例如依赖的一个工具库，其版本从 `v1` 升级到 `v2` ，与原版本的 `API` 发生很大出入，那肯定会导致引用该工具库的项目在未升级工具库版本的情况下产生一些莫名奇妙的错误。

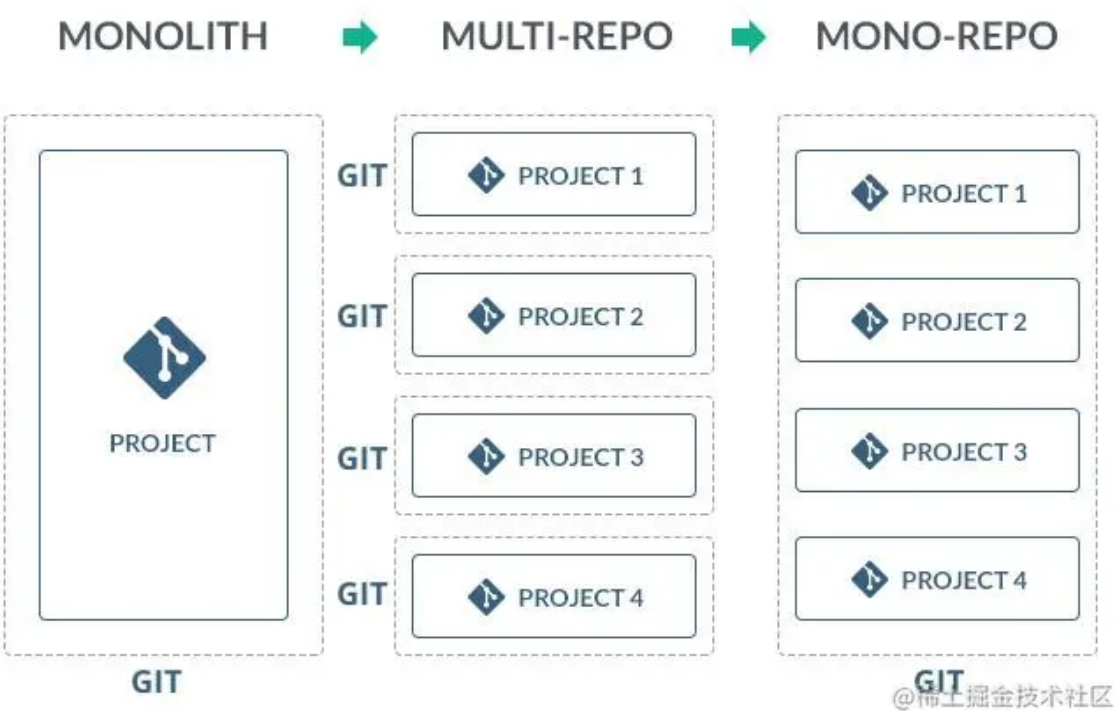
项目数量多起来后，出现这种依赖更新不及时的情况很常见，导致查漏补缺的时间成本大大增加。

方案：基于Monorepo模式拆分仓库

Monorepo

相对 `Multirepo` ，另一种仓库管理风格就是 `Monorepo` 。 `Monorepo` 其实不是一个新概念，它在软件工程领域已有十多年历史。它把每个项目放到不同仓库中，每个项目对应一个单独仓库分散管理。

对于 `Monorepo` 而言，只是把多个项目根据预设场景组织到一起，它的粒度还是保持原有的划分。对于团队某个成员而言，他的关注点还是在其中一个项目中。可能听上去这种方式相对于 `Multirepo` 而言有些多此一举，已把仓库拆分，为何还要再组织回去?



现代 前端工程化 越来越离不开 Monorepo ， 一些明星项目像 react 、 vue 、 babel 、 rollup 等都是基于 Monorepo 管理。 Monorepo 的项目结构一般根据以下目录划分。

```txt
bruce-us
├─ app
│  ├─ src
│  └─ package.json
├─ lib
│  ├─ src
│  └─ package.json
├─ std
│  ├─ src
│  └─ package.json
└─ ... # 另外5个仓库
```

收益

> 简化组织

在普通情况下，一开始开发项目时并不能遇见项目以后的规模大小。随着业务需求的不断迭代，工程文件会越来越多，代码逻辑越来越复杂，于是在后期就要对项目拆分。使用 Monorepo模式 管理这些项目，可简化项目结构的组织。很多拆分都不是正确的拆分，可能只是因为代码量多，或其他非必要原因对仓库拆分，那就需重新将它们组织起来。

> 减少依赖

对于前端项目， `npm i` 已深深刻在开发者的基因中，但 `npm i` 安装了庞大的 `node_modules` 后，其实很多项目安装的 Npm模块 都是重复的。使用 Monorepo模式 管理这些项目，可把这些依赖提取出来，而引用这些依赖的子项目只需通过软链接的方式引用依赖项，就能消除重复安装依赖的影响了。

> 跨域开发

若同时为几个 Npm模块 迭代功能，在多个仓库中调试起来可能就很不方便了，还需手动维护 `npm link` 。使用 Monorepo模式 管理这些项目，可直接在本地跨项目联调，提升开发效率。

> 方便管理

对于一些大型项目或开源项目，多个仓库意味着要在多个地方处理 `Issue` 或 `PR` ，当然更倾向于统一管理这些问题啦，一个仓库就能处理的事情干嘛要分开多个仓库处理。

落地

若还未接触过 Monorepo模式 管理项目，到这可能会产生疑惑: 是直接把多个项目的项目结构合并为上述项目结构就完事了吗?

当然不是，在实际场景落地 Monorepo ，需一套完整的工程体系支撑，因为基于 Monorepo模式 管理项目，绝不是仅仅修改项目结构，把代码放到一起就完事，还需考虑项目间的依赖分析、依赖安装、依赖卸载、构建流程、测试流程、打包流程、部署流程、发布流程等诸多工程环节，同时还要考虑项目规模到达一定程度后的性能问题，例如某个流程的执行时间，在实现全面工程化能力的同时也需兼顾性能问题。

想从零开始定制一套完善的 Monorepo 工程化工具，是一件难度极高的事情。不过社区已提供了一些较成熟的方案，可直接拿来使用。

其中底层方案的 lerna ，封装了 Monorepo 中的依赖安装、依赖卸载、脚本批量执行等基本功能，但它无法提供一套完整的构建、测试、打包、部署和发布功能的工具链，整体 Monorepo 功能较弱，但要将其用到前端业务中，往往需基于它封装顶层方案，提供更全面的工程支撑能力。

Scope

将 bruce 每个项目发布到 Npm公有仓库 ，还是以 bruce-xyz 的包名发布吗?

Npm 也有一种类似 Monorepo模式 用于管理 Npm模块 ，它就是 Scope ，被 Scope 管理的模块称为**范围模块**。 范围模块 可被发布到任意支持 Scope 的 Npm仓库 ，包括 Npm公有仓库 与 Npm私有仓库 ， Npm公有仓库 从 2015年4月19日 就开始支持 范围模块 的公共发布。

命名

范围模块 的命名规则与 普通模块 差不多，同样不能是 URL 非法字符或符号开头。 范围模块 以 @ 开头，后跟一个 / ，再跟一个包名，在 package.json 中显示如下。

```json
{
        "name": "@scope/package"
}
```

Scope 是一种把相关模块组织到一起的模块管理风格，也会在某些地方影响 Npm 对模块的处理。 Npm公有仓库 支持 范围模块 ，同时 Npm 对无 Scope 的模块也是向后兼容的，所以可同时使用两者。

每个 Npm用户/组织 都有自己的 Scope ，只有当前账号才能在自己的 Scope 中增加 Npm模块 。这意味着不必担心有人抢走你的包名，因此这也是向组织发出正式模块的好方式。

安装

范围模块 安装在 node_modules 文件夹中同一个子目录。 普通模块 安装在 node_modules/package 目录中，那 范围模块 安装在 node_modules/@scope/package 目录中， @scope 文件夹中可包括多个 Npm模块 ，就像 Monorepo 的项目结构那样。

安装一个 范围模块 也很简单，例如安装 bruce 中的几个 Npm模块 。因为 bruce 已被注册，所以只能使用我自己名字的拼音命名了。

```
npm i @yangzw/bruce-ui @yangzw/bruce-us
```

安装完毕在 package.json 中显示如下。

```json
{
        "dependencies": {
                "@yangzw/bruce-ui": "^1.0.0",
                "@yangzw/bruce-us": "^1.0.0"
        }
}
```

若 @ 省略，那 Npm 会尝试从 Github 中安装相关模块，可查看npm-install。

发布

若要发布一个公共的 范围模块 ，必须在执行 npm publish 时指定 --access public 。这样该 Npm模块 被标记为可公开使用，在 Npm官网 中可搜索出来，也能像上述 bruce 的 Npm模块 那样被广大开发者安装并使用。

若不想在执行 npm publish 时指定 --access public ，可在 package.json 中指定 publishConfig 。在发布模块时会自动发布到指定 Scope 的 Npm公有仓库 中。

```json
{
        "publishConfig": {
                "access": "public"
        }
}
```

有些 范围模块 可能包括隐私信息，不想发布到 Npm公有仓库 ，也可自行搭建 Npm私有仓库 ，使用该仓库托管 范围模块 ，后续会花费一章带领你从零到一搭建一个 Npm私有仓库 。

总结

从 Multirepo 的角度来看，每个子团队拥有自己的仓库，可用自己擅长的工具与工作流程。多元化能促使各个团队尽可能地提升自己的效率，当然 Multirepo 的代价在于增加很多沟通成本。若在其中一个项目发现 Bug ，就必须修复后再次发布到 Npm公有仓库 并再次安装，然后再回到原来的项目中继续工作。在不同仓库间，不仅需处理不同代码与工具，甚至是不同工作流程，甚至在无权限的情况下只能低声下气求人。

从 Monorepo 的角度来看，让不同团队走自己的路并不见得能提高生产力。虽然有些团队可能会找到自己最佳的工作方式，但他们的收益也会被其他团队不好的工作方式抵消。相反，严格统一的管理更能提升开发效率，团队每个成员都可修改任何东西。虽然把所有鸡蛋都放到一个篮子中，但也更方便照顾鸡蛋。

若团队最终选择 Monorepo ，那主要的挑战自然是随着项目发展，其规模会变得很庞大，因此需使用很多工具应对这些挑战。本章更多是一些理论与概念的知识，下章将基于 yarn 与 lerna 开发一个 多包仓库 。

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（7）

ticktock76323 JY.4 3月前

menorepo 请问这里是不是笔误呀
正确拼写monorepo?

1 1

JowayYoung （作者） 3月前

是的，笔误了，已更新

点赞 回复

SteveCGC JY.3 大哥你搞前端，前端有... 4月前

前面不是用的pnpm吗，怎么突然转到yarn了

点赞 2

JowayYoung （作者） 4月前

课程没有提过pnpm吧

点赞 回复

SteveCGC 回复 JowayYoung 4月前

额，可能看多个小册，有点岔劈了

“课程没有提过pnpm吧”

点赞 回复

忘带伞的阿离 LV.2 JY.2 高级前端开发工程师 4月前

建议可以添加关于 pnpm+monorepo的实践

1 1

JowayYoung （作者） 4月前

好的，后续安排

点赞 回复