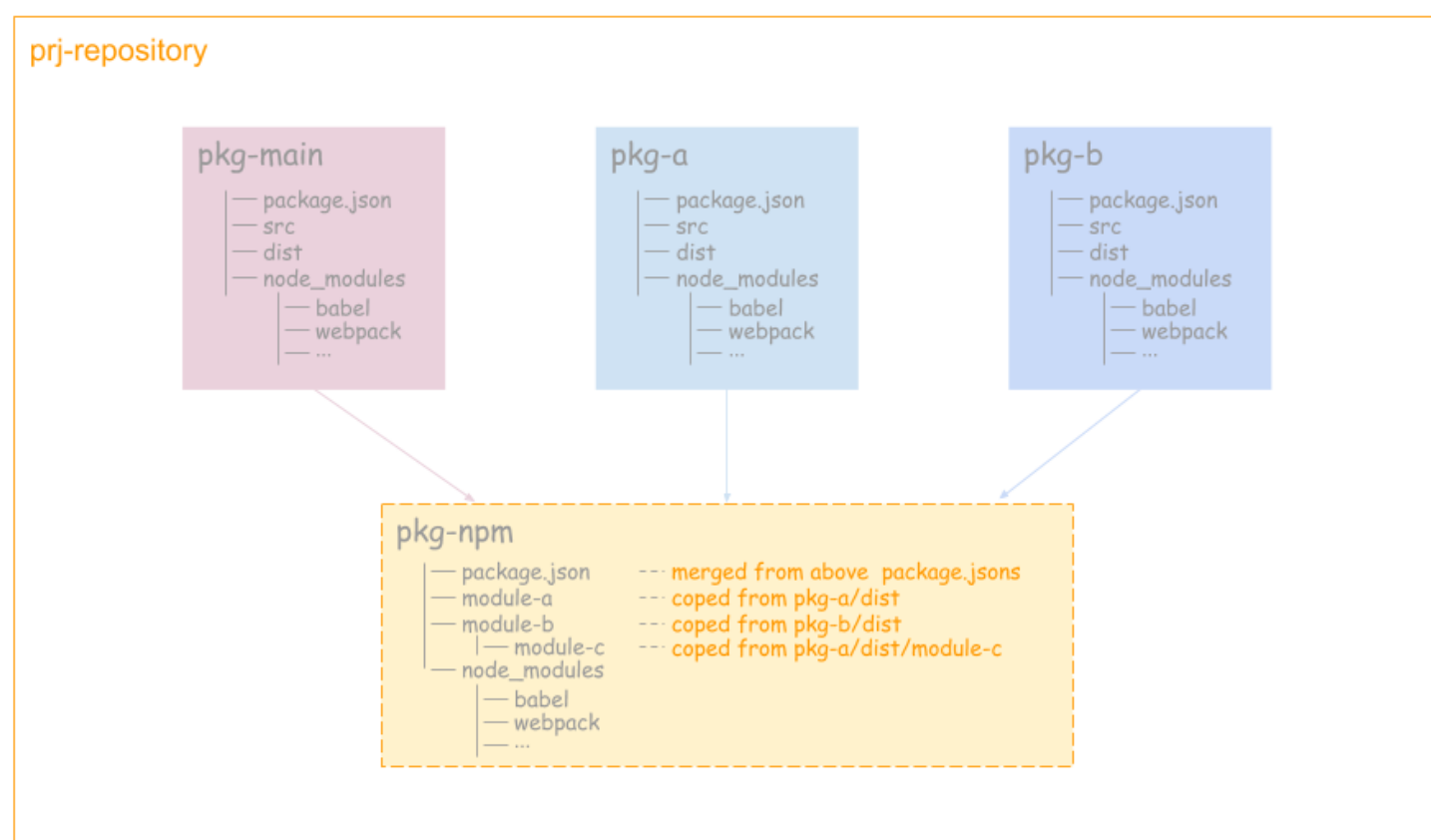


31 offer收割机之基于 Lerna 管理Monorepo

背景

最近在工作中接触到一个项目，这个项目是维护一套 CLI，发到 npm 上供开发者使用。先看一张图：



项目仓库中的根目录上就三个子模块的文件夹，分别对应三个 package，在熟悉了构建和发布流程后。工作流程如图中所示：

1. 使用webpack、babel和uglifyjs把 pkg-a 的 src 编译到 dist
2. 使用webpack、babel和uglifyjs把 pkg-b 的 src 编译到 dist
3. 使用webpack、babel和uglifyjs把 pkg-main 的 src 编译到 dist
4. 最后使用拷贝文件的方式，把pkg-main、pkg-a、pkg-b中编译后的文件组装到 pkg-npm 中，最终用于发布到 npm 上去。

痛点

1. **不好调试。**因为最终的包是通过文件拷贝的方式组装到一起的，并且都是压缩过的，无法组建一个自上到下的调试流程（实际工作中只能加log，然后重新把包编译组装一遍看效果）
- 1.
2. **包的依赖关系不清晰。**pkg-a、pkg-b索性没有版本管理，更像是源码级别的，但逻辑又比较独立。pkg-main中的package.json最终会拷贝到 pkg-npm 中，但又依赖pkg-a、pkg-b中的某些包，所以要把pkg-a、pkg-b中的依赖合并到pkg-main中。pkg-main和pkg-npm的package.json耦合在一起，导致一些本来是工程的开发依赖也会发布到 npm 上去，变成pkg-npm 的依赖包。
- 2.
3. **依赖的包冗余。**可以看到，pkg-a、pkg-b、pkg-main要分别编译，都依赖了babel、webpack等，要分别 `cd` 到各个目录安装依赖。
- 3.
4. **发布需要手动修改版本号。**因为最终只发布了一个包，但实际逻辑要求这个包即要全局安装又要本地安装，业务没有拆开，导致要安装两遍。耦合一起，即便使用 `npm link` 也会导致调试困难，
- 4.
5. **发版没有 `**CHANGELOG.md**`。**因为pkg-a、pkg-b都没有真正管理版本，所以也没有完善的CHANGELOG来记录自上个版本发布已来的变动。

整个项目像是一个没有被管理起来的 Monorepo。那什么又是 Monorepo 呢？

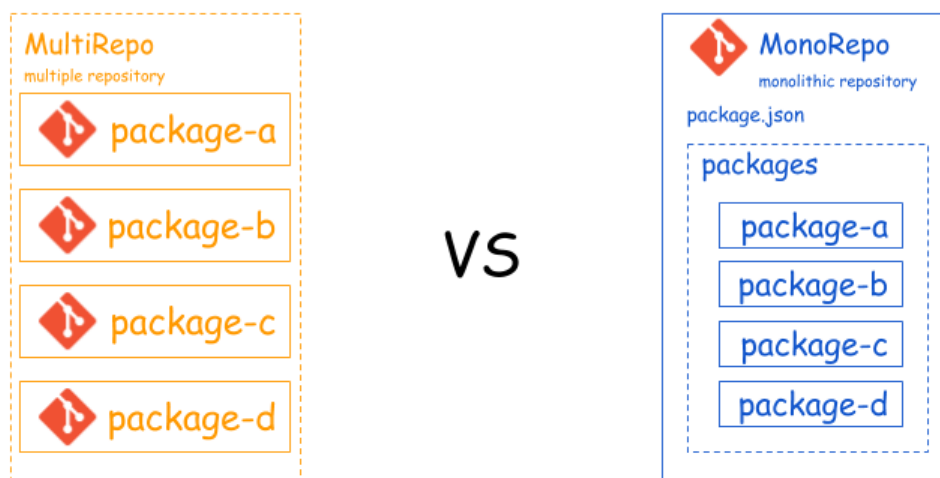
Monorepo vs Multirepo

Monorepo 的全称是 monolithic repository，即单体式仓库，与之对应的是 Multirepo(multiple repository)，这里的“单”和“多”是指每个仓库中所管理的模块数量。

Multirepo 是比较传统的做法，即每一个 package 都单独用一个仓库来进行管理。例如：Rollup, ...

Monorep 是把所有相关的 package 都放在一个仓库里进行管理，**每个 package 独立发布**。例如：React, Angular, Babel, Jest, Umijs, Vue ...

一图胜千言：



优点：

- 各模块管理自由度较高，可自行选择构建工具，依赖管理，单元测试等配套设施
- 各模块仓库体积一般不会太大

缺点：

- 仓库分散不好找，当很多时，更加困难，分支管理混乱
- 版本更新繁琐，如果公共模块的版本发生了变化，需要对所有的模块进行依赖的更新
- CHANGELOG 梳理异常折腾，无法很好的自动关联各个模块的变动联系，基本靠口口相传

缺点：

- 统一构建工具，对构建工具提出了更高要求，要能构建各种相关模块
- 仓库体积会变大

优点：

- 一个仓库维护多个模块，不用到处找仓库
- 方便版本管理和依赖管理，模块之间的引用、调试都非常方便，配合相应工具，可以一个命令搞定
- 方便统一生成CHANGELOG，配合提并规范，可以在发布时自动生成CHANGELOG

当然到底哪一种管理方式更好，仁者见仁，智者见智。前者允许多元化发展（各项目可以有自己的构建工具、依赖管理策略、单元测试方法），后者希望集中管理，减少项目间的差异带来的沟通成本。

虽然拆分子仓库、拆分子 npm 包是进行项目隔离的天然方案，但当仓库内容出现关联时，没有任何一种调试方式比源码放在一起更高效。

结合我们项目的实际场景和业务需要，天然的 MonoRepo！因为工程化的最终目的是让业务开发可以 100% 聚焦在业务逻辑上，那么这不仅仅是脚手架、框架需要从自动化、设计上解决的问题，这涉及到仓库管理的设计。

一个理想的开发环境可以抽象成这样：

“**只关心业务代码，可以直接跨业务复用而不关心复用方式，调试时所有代码都在源码中。**”

在前端开发环境中，多 Git Repo，多 npm 则是这个理想的阻力，它们导致复用要关心版本号，调试需要 `npm link`。而这些是 MonoRepo 最大的优势。

上图中提到的利用相关工具就是今天的主角 Lerna！Lerna 是业界知名度最高的 Monorepo 管理工具，功能完整。

Lerna

Lerna 是什么

A tool for managing JavaScript projects with multiple packages.

Lerna is a tool that optimizes the workflow around managing multi-package repositories with git and npm.

Lerna 是一个管理多个 npm 模块的工具，是 Babel 自己用来维护自己的 Monorepo 并开源出的一个项目。优化维护多包的工作流，解决多个包互相依赖，且发布需要手动维护多个包的问题。

Lerna 现在已经被很多著名的项目组织使用，如：Babel, React, Vue, Angular, Ember, Meteor, Jest 。

一个基本的 Lerna 管理的仓库结构如下：

```
1  lerna-repo/
2      └─ packages/
3          └─ package-a/
4              └─ ...
5                  └─ package.json
6          └─ package-b/
7              └─ ...
8                  └─ package.json
9      └─ ...
10     └─ lerna.json
```

开始使用

安装

推荐全局安装，因为会经常用到 lerna 命令

```
1 npm i -g lerna
```

项目构建

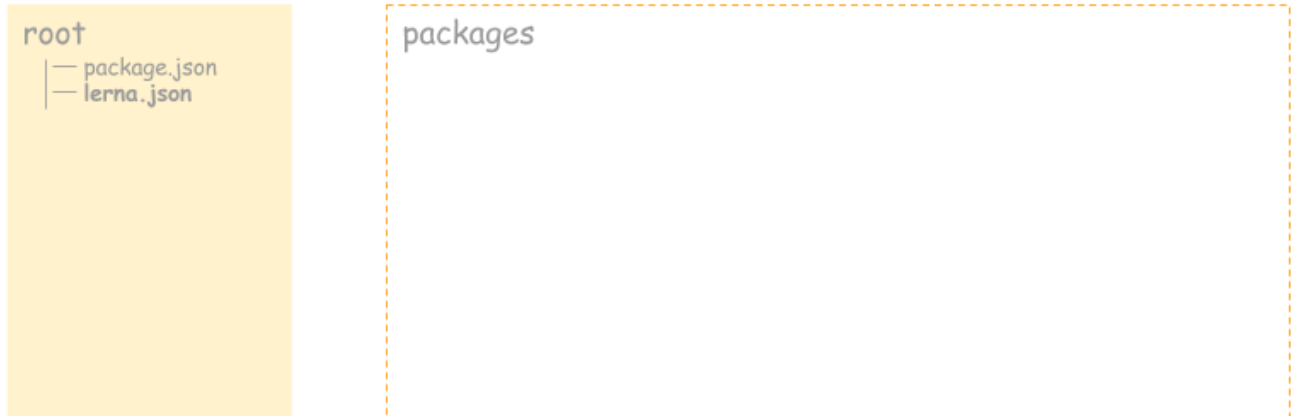
1. 初始化

```
1 lerna init
```

init 命令详情 请参考 [lerna init]

(<https://github.com/lerna/lerna/blob/master/commands/init/README.md>)

Repository



其中 package.json & lerna.json 如下:

```
1  // package.json
2  {
3    "name": "root",
4    "private": true, // 私有的，不会被发布，是管理整个项目，与要发布到npm的解耦
5    "devDependencies": {
6      "lerna": "^3.15.0"
7    }
8  }
9
10 // lerna.json
11 {
12   "packages": \[
13     "packages/\*"
14   \],
15   "version": "0.0.0"
16 }
```

2. 增加两个 packages

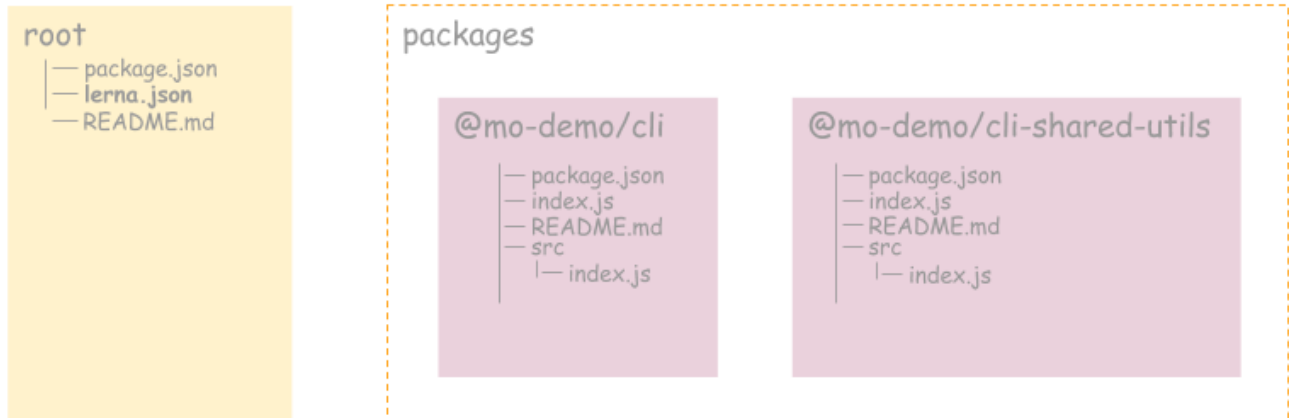
```
1  lerna create @mo-demo/cli
2  lerna create @mo-demo/cli-shared-utils
```

create 命令详情 请参考 [lerna create]

(<https://github.com/lerna/lerna/blob/master/commands/create/README.md>)

lerna create

Repository



3. 分别给相应的 package 增加依赖模块

- 1 `lerna add chalk` // 为所有 package 增加 chalk 模块
- 2 `lerna add semver --scope @mo-demo/cli-shared-utils` // 为 @mo-demo/cli-shared-utils 增加 semver 模块
- 3 `lerna add @mo-demo/cli-shared-utils --scope @mo-demo/cli` // 增加内部模块之间的依赖

add 命令详情 请参考 [lerna add]

(<https://github.com/lerna/lerna/blob/master/commands/add/README.md>)

`lerna add`

Repository

root

- package.json
- lerna.json
- README.md

packages

@mo-demo/cli

- package.json
- index.js
- README.md
- src
 - index.js
- node_modules
 - @xx-utils
 - chalk
 - ...

@mo-demo/cli-shared-utils

- package.json
- index.js
- README.md
- src
 - index.js
- node_modules
 - chalk
 - semver
 - ...

4. 发布

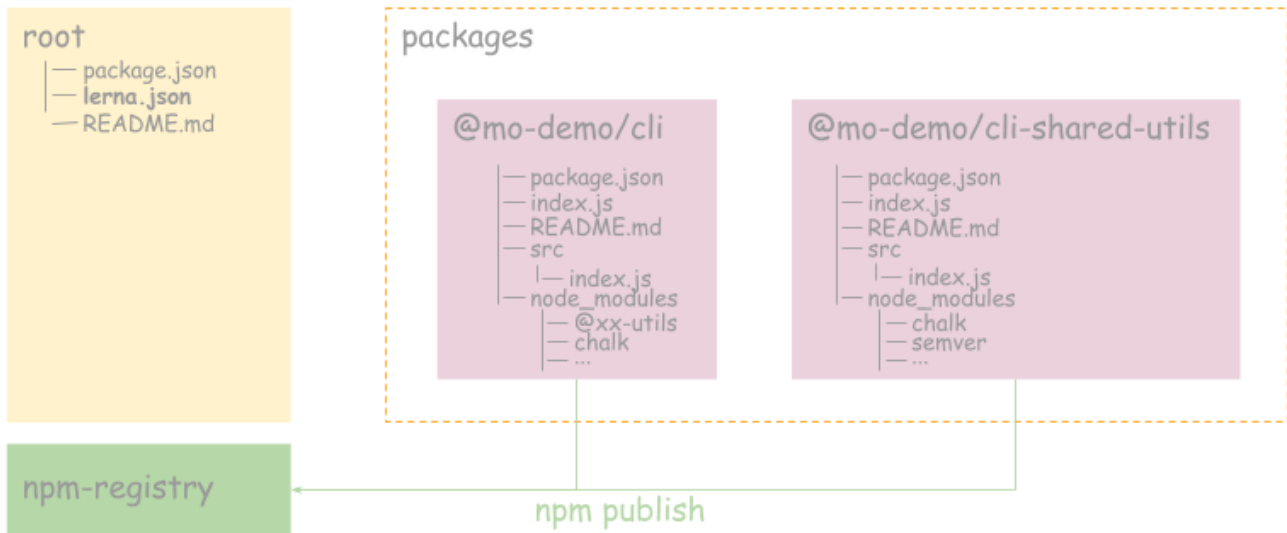
1 `lerna publish`

publish 命令详情 请参考 [lerna publish]

(<https://github.com/lerna/lerna/blob/master/commands/publish/README.md>)

lerna publish

Repository



如下是发布的情况，lerna会让你选择要发布的版本号，我发了@0.0.1-alpha.0 的版本。

发布 npm 包需要登陆 npm 账号

```
lerna notice cli v3.15.0
lerna info current version 0.0.0
lerna info Assuming all packages changed
? Select a new version (currently 0.0.0) Prepatch (0.0.1-alpha.0)

Changes:
- @mo-demo/cli-shared-utils: 0.0.0 => 0.0.1-alpha.0
- @mo-demo/cli: 0.0.0 => 0.0.1-alpha.0

? Are you sure you want to publish these packages? Yes
lerna info execute Skipping releases
lerna info git Pushing tags...
lerna info publish Publishing packages to npm...
lerna info Verifying npm credentials
lerna http fetch GET 200 https://registry.npmjs.org/-/npm/v1/user 1323ms
lerna http fetch GET 200 https://registry.npmjs.org/-/org/morrain/package?format=cli 909ms
lerna http fetch PUT 401 https://registry.npmjs.org/@mo-demo%2fcli-shared-utils 564ms
? This operation requires a one-time password: 255185
lerna success published @mo-demo/cli-shared-utils 0.0.1-alpha.0
lerna notice
lerna notice 📦 @mo-demo/cli-shared-utils@0.0.1-alpha.0
lerna notice === Tarball Contents ===
lerna notice 1.1kB LICENSE
lerna notice 25B index.js
lerna notice 72B src/index.js
lerna notice 799B package.json
lerna notice 146B README.md
lerna notice === Tarball Details ===
lerna notice name: @mo-demo/cli-shared-utils
lerna notice version: 0.0.1-alpha.0
lerna notice filename: mo-demo-cli-shared-utils-0.0.1-alpha.0.tgz
lerna notice package size: 1.4 kB
lerna notice unpacked size: 2.1 kB
lerna notice shasum: 9a925aa6982dcfee78926b803e9ec79e9183a15a
lerna notice integrity: sha512-UCuCBm0DOY8Fy[...]X2V4p16M0s6EQ==
lerna notice total files: 5
lerna notice
lerna http fetch PUT 200 https://registry.npmjs.org/@mo-demo%2fcli-shared-utils 10735ms
lerna http fetch PUT 401 https://registry.npmjs.org/@mo-demo%2fcli 899ms
lerna success published @mo-demo/cli 0.0.1-alpha.0
lerna notice
lerna notice 📦 @mo-demo/cli@0.0.1-alpha.0
lerna notice === Tarball Contents ===
lerna notice 1.1kB LICENSE
lerna notice 25B index.js
lerna notice 46B src/index.js
lerna notice 800B package.json
lerna notice 120B README.md
lerna notice === Tarball Details ===
lerna notice name: @mo-demo/cli
lerna notice version: 0.0.1-alpha.0
lerna notice filename: mo-demo-cli-0.0.1-alpha.0.tgz
lerna notice package size: 1.4 kB
lerna notice unpacked size: 2.1 kB
lerna notice shasum: 39a281a6437809b3930d057d5d70d3f20cb861e1
lerna notice integrity: sha512-9poLv/HadKYIw[...]R2H5UB1EIophw==
lerna notice total files: 5
lerna notice
lerna http fetch PUT 200 https://registry.npmjs.org/@mo-demo%2fcli 7690ms
Successfully published:
- @mo-demo/cli-shared-utils@0.0.1-alpha.0
- @mo-demo/cli@0.0.1-alpha.0
lerna success published 2 packages
```

mo-demo

Packages

Members

Teams

Billing

2 packages

+ Add Package

@mo-demo/cli-shared-utils

A demo for lerna

morrain published 0.0.1-alpha.0 • 2 minutes ago

@mo-demo/cli

A demo for lerna

morrain published 0.0.1-alpha.0 • 2 minutes ago

5. 安装依赖包 & 清理依赖包

上述1-4步已经包含了 Lerna 整个生命周期的过程了，但当我们维护这个项目时，新拉下来仓库的代码后，需要为各个 package 安装依赖包。

我们在第4步 `lerna add` 时也发现了，为某个 package 安装的包被放到了这个 package 目录下的 `node_modules` 目录下。这样对于多个 package 都依赖的包，会被多个 package 安装多次，并且每个 package 下都维护 `node_modules`，也不清爽。于是我们使用 `--hoist` 来把每个 package 下的依赖包都提升到工程根目录，来降低安装以及管理的成本。

```
1 lerna bootstrap --hoist
```

bootstrap 命令详情 请参考 [lerna bootstrap]
(<https://github.com/lerna/lerna/blob/master/commands/bootstrap/README.md>)

```
lerna bootstrap -- hoist
```

Repository

root

- package.json
- lerna.json
- README.md
- node_modules
 - semver
 - chalk
 - ...

packages

@mo-demo/cli

- package.json
- index.js
- README.md
- src
 - index.js
- node_modules
 - @xx-utils

@mo-demo/cli-shared-utils

- package.json
- index.js
- README.md
- src
 - index.js

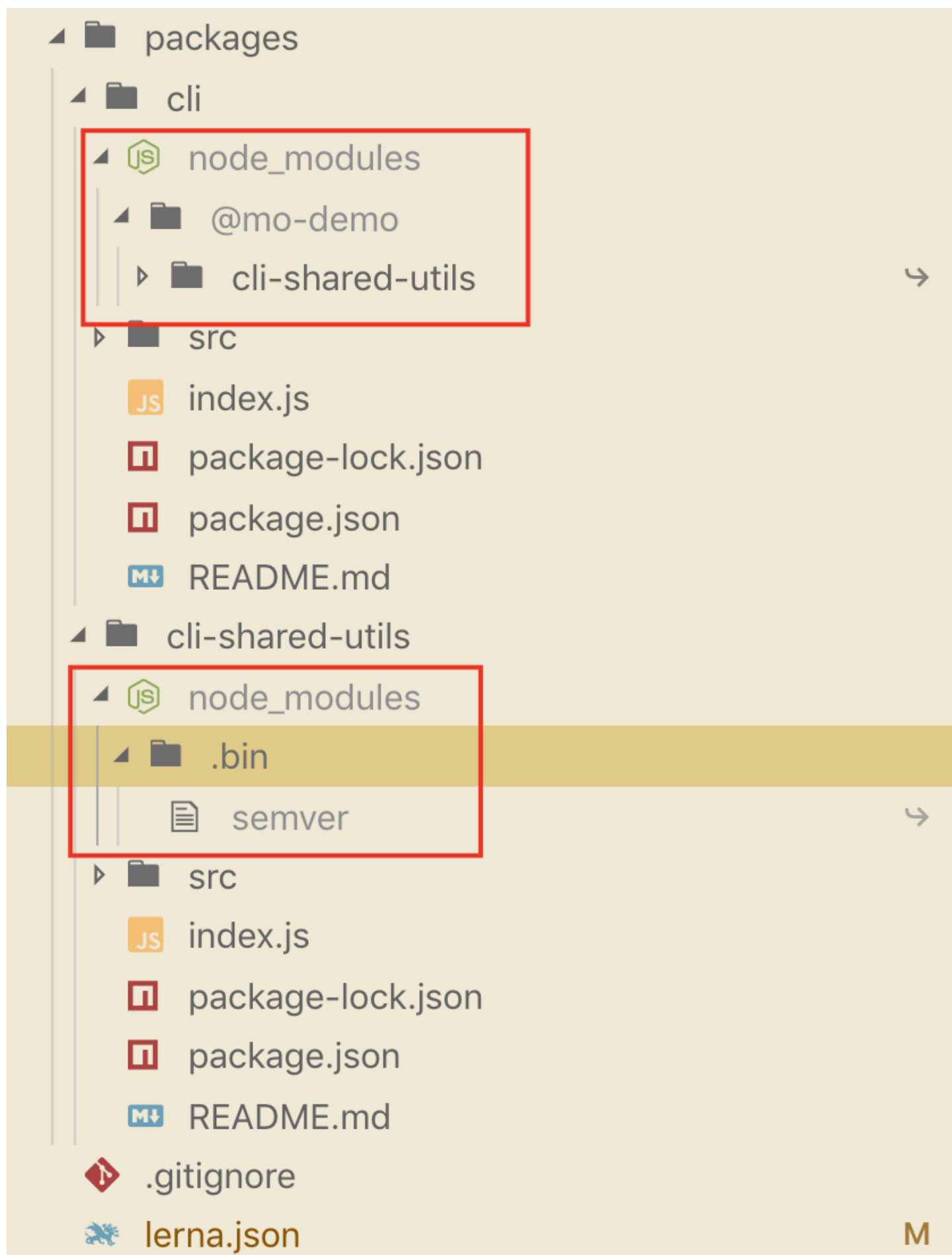
为了省去每次都输入 `--hoist` 参数的麻烦，可以在 `lerna.json` 配置：

```
1  {
2    "packages": \[
3      "packages/\*"
4    \],
5    "command": {
6      "bootstrap": {
7        "hoist": true
8      }
9    },
10   "version": "0.0.1-alpha.0"
11 }
```


配置好后，对于之前依赖包已经被安装到各个 `package` 下的情况，我们只需要清理一下安装的依赖即可：

```
1  lerna clean
```


然后执行 `lerna bootstrap` 即可看到 package 的依赖都被安装到根目录下的 `node_modules` 中了。



 LICENSE

 package-lock.json

U

 package.json

 README.md

M

Lerna的最佳实践

lerna不负责构建，测试等任务，它提出了一种集中管理package的目录模式，提供了一套自动化管理程序，让开发者不必再深耕到具体的组件里维护内容，在项目根目录就可以全局掌控，基于 npm scripts，使用者可以很好地完成组件构建，代码格式化等操作。接下来我们就来看看，如果基于 Lerna，并结合其它工具来搭建 Monorepo 项目的最佳实践。

优雅的提交

1. commitizen && cz-lerna-changelog

[commitizen](<http://commitizen.github.io/cz-cli/>) 是用来格式化 git commit message 的工具，它提供了一种问询式的方式去获取所需的提交信息。

[cz-lerna-changelog](<https://github.com/atlassian/cz-lerna-changelog>) 是专门为 Lerna 项目量身定制的提交规范，在问询的过程，会有类似影响哪些 package 的选择。如下：

```
> git-cz

cz-cli@3.1.1, cz-lerna-changelog@2.0.2

Line 1 will be cropped at 100 characters. All other lines will be wrapped after 100 characters.

? Select the type of change that you're committing: (Use arrow keys or type to search)
> feat:      ✨ A new feature (note: this will indicate a release)
  fix:       🐛 A bug fix (note: this will indicate a release)
  docs:      Documentation only changes
  style:     Changes that do not affect the meaning of the code
             (white-space, formatting, missing semi-colons, etc)
  refactor:  A code change that neither fixes a bug nor adds a feature
  perf:      A code change that improves performance
(Move up and down to reveal more choices)
```

```
> git-cz

cz-cli@3.1.1, cz-lerna-changelog@2.0.2

Line 1 will be cropped at 100 characters. All other lines will be wrapped after 100 characters.

? Select the type of change that you're committing: feat:      ✨ A new feature (note: this will indicate a release)
? Denote the scope of this change:
? Write a short, imperative tense description of the change:
  fix a bug about cli
? Provide a longer description of the change (optional). Use "|" to break new line:

? List any BREAKING CHANGES (if none, leave blank):

? List any ISSUES CLOSED by this change (optional). E.g.: #31, #34:

? The packages that this commit has affected (0 detected)
(Press <space> to select, <a> to toggle all, <i> to invert selection)
> @mo-demo/cli-shared-utils
  @mo-demo/cli
```

我们使用 commitizen 和 cz-lerna-changelog 来规范提交，为后面自动生成日志作好准备。

因为这是整个工程的开发依赖，所以在根目录安装：

- 1 npm i -D commitizen
- 2 npm i -D cz-lerna-changelog

安装完成后，在 package.json 中增加 config 字段，把 cz-lerna-changelog 配置给 commitizen。同时因为 commitizen 不是全局安全的，所以需要添加 scripts 脚本来执行 `git-cz`

```

1  {
2    "name": "root",
3    "private": true,
4    "scripts": {
5      "c": "git-cz"
6    },
7    "config": {
8      "commitizen": {
9        "path": "./node_modules/cz-lerna-changelog"
10     }
11   },
12   "devDependencies": {
13     "commitizen": "^3.1.1",
14     "cz-lerna-changelog": "^2.0.2",
15     "lerna": "^3.15.0"
16   }
17 }

```

之后在常规的开发中就可以使用 `npm run c` 来根据提示一步一步输入，来完成代码的提交。

```

j@mac ~/Documents/lerna-learning master • npm run c
> root@ c /Users  jg/Documents/lerna-learning
> git-cz

cz-cli@3.1.1, cz-lerna-changelog@2.0.2

Line 1 will be cropped at 100 characters. All other lines will be wrapped after 100 characters.

? Select the type of change that you're committing: feat:  ✨ A new feature (note: this will indicate a release)
? Denote the scope of this change: main
? Write a short, imperative tense description of the change:
  增加 commitizen,规范提交,为后面自动生成CHANGELOG 准备
? Provide a longer description of the change (optional). Use "|" to break new line:

? List any BREAKING CHANGES (if none, leave blank):

? List any ISSUES CLOSED by this change (optional). E.g.: #31, #34:

? The packages that this commit has affected (0 detected)

This commit does not indicate any release

Commit message:

feat(main): 增加 commitizen,规范提交,为后面自动生成CHANGELOG 准备

[master 73f33e1] feat(main): 增加 commitizen,规范提交,为后面自动生成CHANGELOG 准备
2 files changed, 679 insertions(+), 4 deletions(-)

```


2. commitlint & husky

上面我们使用了 commitizen 来规范提交，但这个要靠开发自觉使用 `npm run c`。万一忘记了，或者直接使用 `git commit` 提交怎么办？答案就是在提交时对提交信息进行校验，如果不符合要求就不让提交，并提示。校验的工作由 [commitlint](<https://commitlint.js.org/#/>) 来完成，校验的时机则由 [husky](<https://github.com/typicode/husky>) 来指定。husky 继承了 Git 下所有的钩子，在触发钩子的时候，husky 可以阻止不合法的 commit, push 等等

```
1 // 安装 commitlint 以及要遵守的规范
2 npm i -D @commitlint/cli @commitlint/config-conventional
3
4
5 // 在工程根目录为 commitlint 增加配置文件 commitlint.config.js 为commitlint 指定相应的规范
6 module.exports = { extends: ['@commitlint/config-conventional'] }
7
8
9 // 安装 husky
10 npm i -D husky
11
12 // 在 package.json 中增加如下配置
13 "husky": {
14   "hooks": {
15     "commit-msg": "commitlint -E HUSKY\_GIT\_PARAMS"
16   }
17 }
```

"commit-msg"是git提交时校验提交信息的钩子，当触发时便会使用 commitlit 来校验。安装配置完成后，想通过 `git commit` 或者其它第三方工具提交时，只要提交信息不符合规范就无法提交。****从而约束开发者使用 `**npm run c**` 来提交。**

3. standardjs & lint-staged

除了规范提交信息，代码本身肯定也少了靠规范来统一风格。

[standardjs](<https://standardjs.com/readme-zhcn.html>)就是完整的一套 JavaScript 代码规范，自带 linter & 代码自动修正。它无需配置，自动格式化代码并修正，提前发现风格以及程序问题。

[lint-staged](<https://github.com/okonet/lint-staged>) staged 是 Git 里的概念，表示暂存区，lint-staged 表示只检查并矫正暂存区中的文件。一来提高校验效率，二来可以为老的项目带去巨大的方便。

```
1  // 安装
2  npm i -D standard lint-staged
3
4
5  // package.json
6  {
7    "name": "root",
8    "private": true,
9    "scripts": {
10      "c": "git-cz"
11    },
12    "config": {
13      "commitizen": {
14        "path": "./node_modules/cz-lerna-changelog"
15      }
16    },
17    "husky": {
18      "hooks": {
19        "pre-commit": "lint-staged",
20        "commit-msg": "commitlint -E HUSKY\\_GIT\\_PARAMS"
21      }
22    },
23    "lint-staged": {
24      "\\*.js": [
25        "standard --fix",
26        "git add"
27      ]
28    },
29    "devDependencies": {
30      "@commitlint/ cli ": "^8.1.0",
31      "@commitlint/config-conventional": "^8.1.0",
32      "commitizen": "^3.1.1",
33      "cz-lerna-changelog": "^2.0.2",
34      "husky": "^3.0.0",
35      "lerna": "^3.15.0",
36      "lint-staged": "^9.2.0",
37      "standard": "^13.0.2"
38    }
39  }
```

安装完成后，在 package.json 增加 lint-staged 配置，如上所示表示对暂存区中的 js 文件执行 `standard --fix` 校验并自动修复。那什么时候去校验呢，就又用到了上面安装的 husky，**husky的配置中增加'pre-commit'的钩子用来执行 lint-staged 的校验操作**，如上所示。

此时提交 js 文件时，便会自动修正并校验错误。即保证了代码风格统一，又能提高代码质量。

自动生成日志

有了之前的规范提交，自动生成日志便水到渠成了。再详细看下 `lerna publish` 时做了哪些事情：

1. 调用 `lerna version`

- a. 找出从上一个版本发布以来有过变更的 package
- b. 提示开发者确定要发布的版本号

```
j@mac ~/Documents/lerna-learning master lerna version
info cli using local version of lerna
lerna notice cli v3.15.0
lerna info current version 0.0.1-alpha.0
lerna info Looking for changed packages since v0.0.1-alpha.0
? Select a new version (currently 0.0.1-alpha.0) Custom Prerelease
? Enter a prerelease identifier (default: "alpha", yielding 0.0.1-alpha.1) 0.0.1-alpha.1

Changes:
- @mo-demo/cli: 0.0.1-alpha.0 => 0.0.1-alpha.1

? Are you sure you want to create these versions? (ynH) [ ]
```

- c. 将所有更新过的的 package 中的package.json的version字段更新
- d. 将依赖更新过的 package 的 包中的依赖版本号更新
- e. 更新 lerna.json 中的 version 字段
- f. 提交上述修改，并打一个 tag
- g. 推送到 git 仓库

lerna version

Repository 📁 @tag

root

- package.json
- **lerna.json**
- README.md
- node_modules
 - semver
 - chalk
 - ...

packages

@mo-demo/cli

- package.json
- index.js
- README.md
- src
 - index.js
- node_modules
 - @xx-utils

@mo-demo/cli-shared-utils

- **package.json**
- index.js
- README.md
- src
 - index.js

2. 使用 `npm publish` 将新版本推送到 npm

CHANGELOG 很明显是和 version 一一对应的，所以需要在 `lerna version` 中想办法，查看 `[lerna version]`(<https://github.com/lerna/lerna/blob/master/commands/version/README.md>) 命令的详细说明后，会看到一个配置参数 `--conventional-commits`。没错，只要我们按规范提交后，在 `lerna version` 的过程中会自动生成当前这个版本的 CHANGELOG。为了方便，不用每次输入参数，可以配置在 `lerna.json` 中，如下：

```
1  {
2    "packages": \[
3      "packages/\ "
4    \],
5    "command": {
6      "bootstrap": {
7        "hoist": true
8      },
9      "version": {
10       "conventionalCommits": true
11     }
12   },
13   "ignoreChanges": \[
14     *    "\*\*/\*.md"
15   \],
16   "version": "0.0.1-alpha.1"
17 }
```

`lerna version` 会检测从上一个版本发布以来的变动，但有一些文件的提交，我们不希望触发版本的变动，譬如 `.md` 文件的修改，并没有实际引起 `package` 逻辑的变化，不应该触发版本的变更。可以通过 `ignoreChanges` 配置排除。如上。

`lerna version`

Repository @tag

root

```
— package.json
— lerna.json
— README.md
— CHANGELOG.md
— node_modules
  — semver
  — chalk
  — ...
```

packages

@mo-demo/cli

```
— package.json
— index.js
— README.md
— CHANGELOG.md
— src
  |— index.js
— node_modules
  |— @xx-utils
```

@mo-demo/cli-shared-utils

```
— package.json
— index.js
— README.md
— CHANGELOG.md
— src
  |— index.js
```

实际 `lerna version` 很少直接使用，因为它包含在 `lerna publish` 中了，直接使用 `lerna publish` 就好了。

Lerna 在管理 `package` 的版本号上，提供了两种模式供选择 `Fixed` or `Independent`。默认是 `Fixed`，更多细节，以及 Lerna 的更多玩法，请参考官网文档：

[<https://github.com/lerna/lerna/blob/master/README.md>]

(<https://github.com/lerna/lerna/blob/master/README.md>)

编译、压缩、调试

采用 **Monorepo** 结构的项目，各个 **package** 的结构最好保持统一 根据目前的项目状况，设计如下：

1. 各 `package` 入口统一为 `index.js`

2. 各 package 源码入口统一为 src/index.js
3. 各 package 编译入口统一为 dist/index.js
4. 各 package 统一使用 ES6 语法、使用 Babel 编译、压缩并输出到 dist
5. 各 package 发布时只发布 dist 目录，不发布 src 目录
6. 各 package 注入 LOCAL_DEBUG 环境变量，在 index.js 中区分是调试还是发布环境，调试环境
`require('./src/index.js)` 保证所有源码可调试。发布环境
`require('./dist/index.js)` 保证所有源码不被发布。

因为 dist 是 Babel 编译后的目录，我们在搜索时不希望搜索它的内容，所以在工程的设置中把 dist 目录排除在搜索的范围之外。

接下来，我们按上面的规范，搭建 package 的结构。

首先安装依赖

```
1  npm i -D @babel/cli @babel/core @babel/preset-env // 使用 Babel 必备 详见官网用法
2  npm i -D @babel/node // 用于调试 因为用了
   import&export 等 ES6 的语法
3  npm i -D babel-preset-minify // 用于压缩代码
```

由于各 package 的结构统一，所以类似 Babel 这样的工具，只在根目录安装就好了，不需要在各 package 中安装，简直是清爽的要死了。

增加 Babel 配置

```
1  // 根目录新建 babel.config.js
2  module.exports \= function (api) {
3    api.cache(true)
4
5    const presets \= \[
6      \[
7        '@babel/env',
8        {
```

```

9      targets: {
10        node: '8.9'
11      }
12    }
13  \]
14 \]
15
16  // 非本地调试模式才压缩代码，不然调试看不到实际变量名
17  if (!process.env\['LOCAL\_DEBUG'\]) {
18    presets.push(\[
19      'minify'
20    \])
21  }
22
23  const plugins \= \[\]
24
25  return {
26    presets,
27    plugins,
28    ignore: \['node\_modules'\]
29  }
30 }

```

修改各 package 的代码

```

1  // @mo\demo/cli/index.js
2  if (process.env.LOCAL\_DEBUG) {
3    require('./src/index') // 如果是调试模式，加载src中的源
4  } else {                  码
5    require('./dist/index') // dist会发到npm
6  }
7
8  // @mo\demo/cli/src/index.js
9  import { log } from '@mo-demo/cli-shared-utils' // 从 utils 模块引入依赖并使用
10 log('cli/index.js as cli entry exec!') log 函数
11
12 // @mo\demo/cli/package.json
13 {
14   "main": "index.js",
15   "files": \[

```

```

16     "dist"                                // 发布 dist
17     \]
18   }
19
20   // @mo\demo/cli-shared-utils/index.js
21   if (process.env.LOCAL_DEBUG) {
22     module.exports \= require('./src/index')    // 如果是调试模式，加载src中的
    源码
23   } else {
24     module.exports \= require('./dist/index')  // dist会发到npm
25   }
26
27   // @mo\demo/cli-shared-utils/src/index.js
28   const log \= function (str) {
29     console.log(str)
30   }
31   export {                                //导出 log 接口
32     log
33   }
34
35   // @mo\demo/cli-shared-utils/package.json
36   {
37     "main": "index.js",
38     "files": \[
39       "dist"
40     \]
41   }

```

修改发布的脚本

`npm run b` 用来对各 package 执行 babel 的编译，从 src 目录输出到 dist 目录，使用根目录的配置文件 babel.config.js。

`npm run p` 用来取代 `lerna publish`，在 publish 前先执行 `npm run b` 来编译。

其它常用的 lerna 命令也添加到 scripts 中来，方便使用。

```

1   // 工程根目录 package.json
2   "scripts": {

```



```
3     "c": "git-cz",
4     "i": "lerna bootstrap",
5     "u": "lerna clean",
6     "p": "npm run b && lerna publish",
7     "b": "lerna exec -- babel src -d dist --config-file ../../babel.config.js"
8 }
```

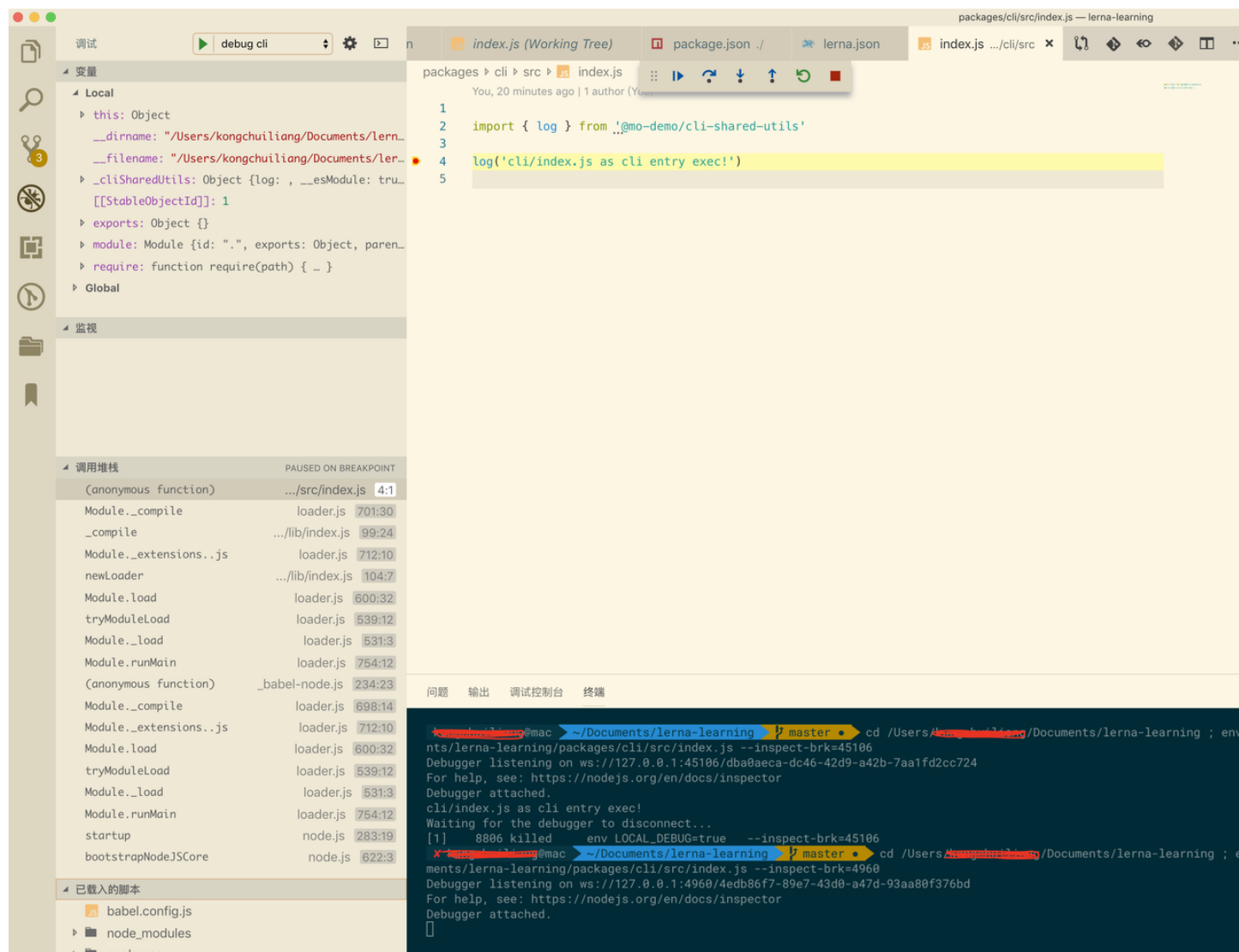
调试

我们使用vscode自带的调试功能调试，也可以使用 Node + Chrome 调试，看开发者习惯。我们就vscode 为例，请参考 [<https://code.visualstudio.com/docs/editor/debugging>。]
(<https://code.visualstudio.com/docs/editor/debugging%E3%80%82>)

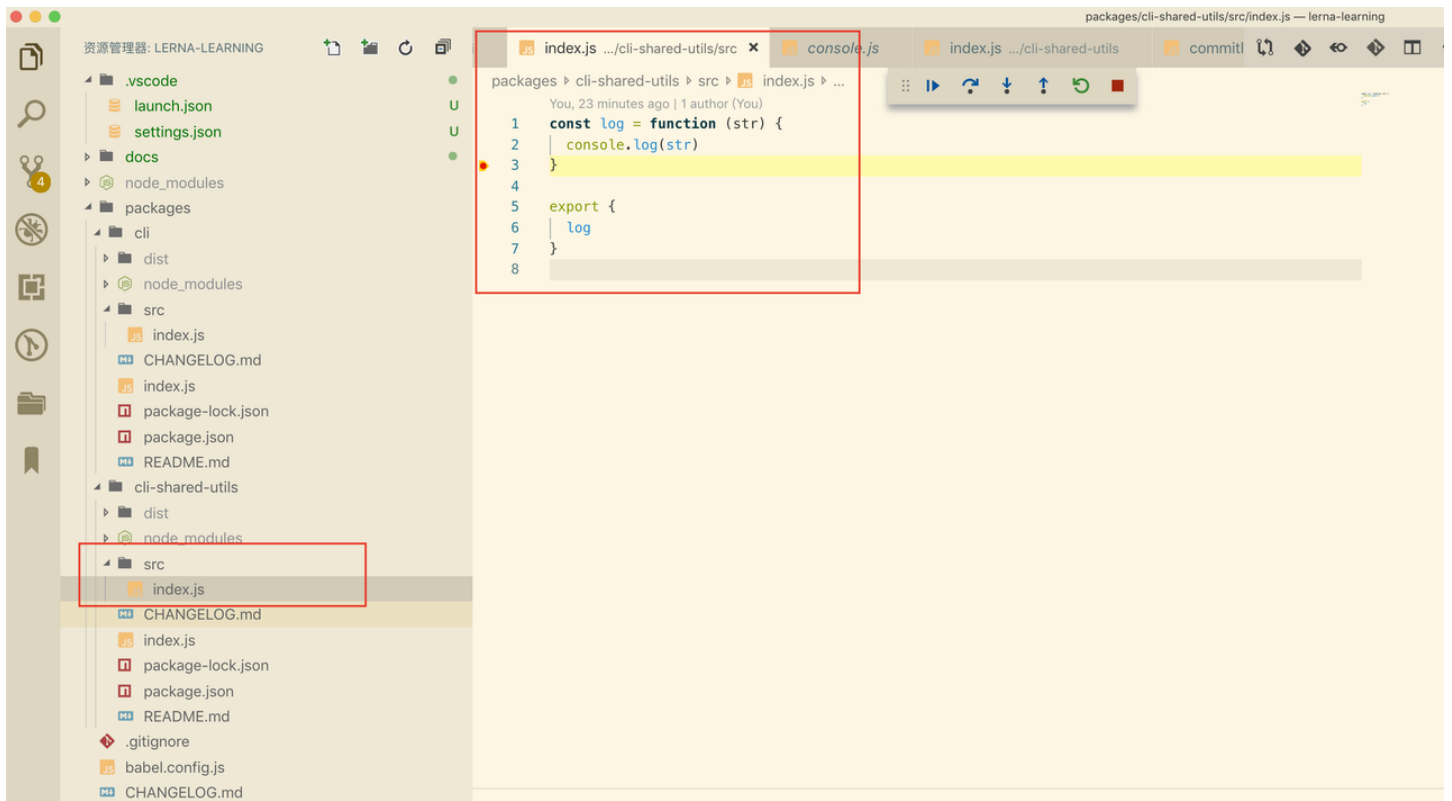
增加如下调试配置文件：

```
1  // .vscode/launch.json
2  {
3      // 使用 IntelliSense 了解相关属性。
4      // 悬停以查看现有属性的描述。
5      // 欲了解更多信息，请访问：https://go.microsoft.com/fwlink/?linkid=830387
6      "version": "0.2.0",
7      "configurations": \[
8          {
9              "type": "node",
10             "request": "launch",
11             "name": "debug cli",
12             "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/babel-
node",
13             "runtimeArgs": \[
14                 "${workspaceRoot}/packages/cli/src/index.js"
15             \],
16             "env": {
17                 "LOCAL_DEBUG": "true"
18             },
19             "console": "integratedTerminal"
20         }
21     \]
22 }
```

因为 src 的代码是 ES6 的，所以要使用 `babel-node` 去跑调试，`@babel/node` 已经在前面安装过了。

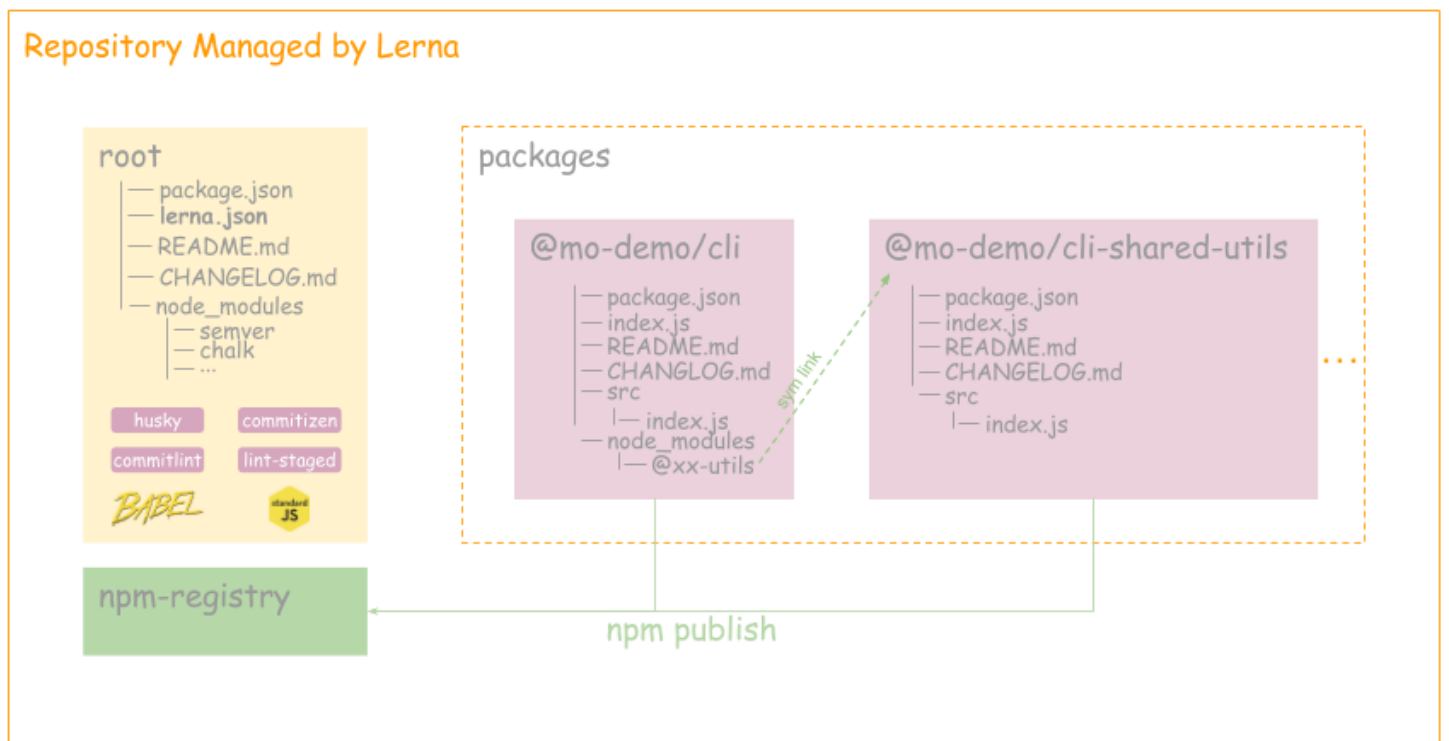


最棒的是，可以直接使用单步调试，调到依赖的模块中去，如上图，我们要执行 `@mo-demo/cli-shared-utils` 模块中的 `log` 方法，单步进入，会直接跳到 `@mo-demo/cli-shared-utils` src 源码中去执行。如下图



结语

到这里，基本上已经构建了基于 Lerna 管理 packages 的 Monorepo 项目的最佳实践了，该有的功能都有：完善的工作流、流畅的调试体验、风格统一的编码、一键式的发布机制、完美的更新日志……



当然，Lerna 还有更多的功能等待着你去发掘，还有很多可以结合 Lerna 一起使用的工具。构建一套完善的仓库管理机制，可能它的收益不是一些量化的指标可以衡量出来的，也没有直接的价值输出，但它能在日常的工作中极大的提高工作效率，解放生产力，节省大量的人力成本。

参考文献

[手把手教你玩转 Lerna](<http://www.uedlinker.com/2018/08/17/lerna-training/>)

[精读《Monorepo 的优势》](<https://mp.weixin.qq.com/s/f2ehHTNK9rx8jNBUyhSwAA>)

[使用lerna优雅地管理多个package](<https://zhuanlan.zhihu.com/p/35237759>)

[用 husky 和 lint-staged 构建超溜的代码检查 workflow]
(<https://segmentfault.com/a/1190000009546913>)