[发布 LTS 查看工具] Node 的生态利器 - NPM

> 本节目标：[开发一个查看 Node LTS 版本的命令行工具] 一沙一世界，模块成就雄伟工程，而模块的窝身之处就是包的海洋，也就是 NPM 所连接和管理的工具天堂。

Node 世界里，一切皆模块，而安装模块，皆是 npm i（也就是 npm install 的缩写）。npm install 想必是我们接触 Node 后，最先 Get 到的命令。它往往跟随 Node 大版本同时安装到本地，所以当你 which node 和 which npm 查看时：

```
1   which node
2 /Users/black/.nvm/versions/node/v20.14.0/bin/node
3   which npm
4 /Users/black/.nvm/versions/node/v20.14.0/bin/npm
```

这俩好兄弟如影如随从不分离，当然你也可以通过 npm install npm -g 来升级 npm 到某个特定版本或者最新版本。如果说 Node 是打开了前后端同一种语言的能力大门，那么 npm 就是让千军万马通过大门的高速公路，完全赋能了 Node，让它的背后长出了一个无比繁荣的军工城市群。成千上万的手艺人在那里无时无刻的制造趁手的工具，你能想到的几乎所有功能，只要想偷懒，就可以前往免费取回来。

大家可以输入 npm -h 或者 npm -l 来查看 npm 提供的命令集。如果这些命令相当一部分你都很熟悉，那么这一节就可以跳过了，我们会挑选几个常用的介绍一些。介绍之前，我们先来看下 npm 的模块安装策略。

包（模块）服务 - npm registry

npm 全称 Node Package Manager，只是它早就不是只为 Node 服务的包（模块）管理工具，海洋般的前端模块也一并被它纳入怀中。前后端的模块如此之多，甚至直接催生了商业机会 - npm.Inc 公司的诞生，也就是 Isaaz 辞职后（Node 第二任技术负责人）成立的专门维护 npm registry 的创业公司，主要为企业提供私有的 npm registry 服务和团队合作的 Saas 服务。

有了 npm，让包的下载变成一件特别 easy 的事情。比如命令行里丢进去 npm i lodash@4.17.21 运行后，就安装好了版本是 4.17.11 的 lodash 包（目录里需要先 npm init 创建 package.json 文件），这些包会被 npm 放到本地项目目录的 node_nodules 目录下。

这里简单区分下包和模块。从模块的角度，我们会看到一个有特定功能或者特定功能集合的文件夹，它里面有很多自身的模块文件也有很多依赖的其他文件，那么无论是从哪个颗粒度看，都可以称为模块，所以我们管 npm 叫模块管理工具也是成立的，但是颗粒度很难直观的表达。而包呢，就可以简单看做是一个特定功能的文件夹，无论它里面有多少模块组成，这个文件夹可以被发布到线上，那么可以把它看做是包，**我们知道包是模块的集合就行了**。

模块汇聚成了包，那包是从哪里下载的呢，答案就是 registry。

registry 直译就是注册表，再直白一点，就是所有可被下载的模块，需要有一个地方存储和记录他们，并且对外提供一个可查和下载的服务。对于 Node 模块来说，npm registry 就是这个服务，整个 npm 包括三个部分：

1. npm 官网，网址 www.npmjs.com，可以通过网站直接查询一个模块的相关信息
2. npm registry，https://registry.npmjs.org/ 则是模块查询下载的服务 API
3. npm cli，则是一个命令行下载工具，通过它来从 registry 下载模块

需要我们注意的是，这个 npm registry 不过是 Isaaz 公司提供给 Node 社区的服务，也是 npm 默认使用的服务，但它并不是唯一的服务。只要你有意愿，你都可以搭建自己的 registry，甚至是使用第三方的 registry，比如 淘宝NPM 镜像，它也有自己的 cli - cnpm cli，每三十分钟同步一次，大家在国内如果网络不通畅，建议使用淘宝镜像代替 npm 来作为模块下载的服务，使用的办法非常简单：

npm i lodash@4.17.21 --registry=https://registry.npmmirror.com

或者通过 cnpm 来代替：

```
npm install -g cnpm --registry=https://registry.npmmirror.com
cnpm install lodash@4.17.21
```

大家如果想给自己团队搭建私有 registry，除了 npm 官方的 registry 付费私有服务，给大家安利一个阿里云的 regsitry。首先注册一个阿里云账号，然后打开 Aliyun Registry，进去后，随便创建一个 scope 就可以使用了，待会我们会有一个代码案例，里面会演示如何发布到 Aliyun Registry。

包的地图 - npm init

npm init 可以直接创建一个 package.json，在它里面主要记录了：

- 当前模块/项目名称、版本、描述、作者
- 配置了当前项目依赖的模块及版本
- 配置了当前项目是在哪个 git repo 下

- 当前模块的主入口文件 main
- ...

我们也可以 npm init --yes 可以直接创建默认值的 package.json。

包的安装 - npm install

我们最常用的 npm install 通常会搭配 --save 和 --save-dev 来使用，分别把模块安装到 dependencies 和 devDependencies。一个是运行时依赖的模块，一个是本地开发时依赖的模块，当然也可以简写，比如 npm i xx -S 和 npm i xx -D。我个人会建议大家在本地安装模块，一定要指定 --save 或者 -S，来确保本地模块正确的添加到 package.json，那么具体 install 都支持哪些类型的模块呢，其实我们可以通过 npm help install 找到答案：

```
~ npm help install
# 项目中已有 package.json，可以直接 npm intall 安装所有依赖项
npm install (with no args, in package dir)
# scope 通常用于管理私有模块，以 @ 开头，没有 @ 则反之
# npm install some-pkg
# npm install @scott/some-pkg
npm install [<@scope>/]<name>
# 可以安装特定 tag 的模块，默认是 latest，如:
# npm install lodash@latest
npm install [<@scope>/]<name>@<tag>
# npm install lodash@4.17.11
# npm install @scott/some-pkg@1.0.0
npm install [<@scope>/]<name>@<version>
# 安装一个某个范围内的版本
# npm install lodash@">=2.0.0 <3.0.0"
# npm install @scott/som-pkg@">=2.0.0 <3.0.0"
npm install [<@scope>/]<name>@<version range>
# npm install git+ssh://git@github.com:tj/commander.js.git
npm install <git-host>:<git-user>/<repo-name>
# 以 git 仓库地址来安装
# npm install https://github.com/petkaantonov/bluebird.git
npm install <git repo url>
# 安装本地的 tar 包
# npm install /Users/black/Downloads/request-2.88.1.tar.gz
npm install <tarball file>
# 以 tar 包地址来安装
# npm install https://github.com/caolan/async/tarball/v2.3.0
# npm install https://github.com/koajs/koa/archive/2.5.3.tar.gz
npm install <tarball url>
# 从本地文件夹安装
# npm install ../scott/some-module
npm install <folder>
```

```
# 卸载也很简单
npm uninstall some-pkg -S
# 或者简写，加上 -S 是把卸载也同步到 package.json 中
npm un some-pkg -S
```

npm 有如此多样的安装方式，给了我们很多想象力，比如在强运维大量服务器存储保障的前提下，可以把所有的模块全部 tarball 形式从本地上传到服务器，可以保证所有模块代码的绝对一致性，甚至 npm registry 不稳定的时候也不影响，更不用提私有模块 scope 的组合使用。

抛开 npm 带来的便捷，反过来的一个问题就是模块的版本管理，怎样保证我本地安装的版本，跟服务器上运行的版本，两份代码是一模一样的，关于这一点我们先往下看，了解 npm versions 后再来讨论版本的管理问题。

包版本 - npm semver version

只要一个文件夹里面有 package.json，里面的基本信息完备，无论里面有多少个 js 模块，整个文件夹便可以看做是包。我们所谓的 npm - 包管理工具，其实本质上就是管理这个文件夹的版本。将几十上百个甚至上千个项目所依赖的包，全部下载到本地，全部整理到 node_modules 下面管理，每个包都是一个独立的文件夹，比如安装了bluebird 和 lodash 的项目，package.json 的依赖是这样的：

```
"dependencies": {
  "bluebird": "^3.5.2",
  "lodash": "^4.17.11"
}
```

而在 node_modules 里面是这样的：

```
├── node_modules
│   ├── bluebird
│   └── lodash
├── package-lock.json
└── package.json
```

每个文件夹都是一个 package（包），每个包都有自己依赖的其他包，每个包也都有自己的名称和版本。每个包作者对于版本的管理都不尽相同，有依赖大版本的，有依赖小版本的，有奇数偶数策略的等等。我们说下业界最常见的一种版本管理方式，就是 Semantic Versioning 2.0.0，大家可以前往 semver.org 查看详情。这里简单解释下，版本号比如 v4.5.1，v 是 version 的缩写，4.5.1 被分开成三端，这三端分别是：major、minor、patch，也就是主版本号.次版本号.修订号：

- major: breaking changes (做了不兼容的 API 修改)

- minor: feature add（向下兼容的功能性新增）
- patch: bug fix, docs（向下兼容的问题修正）

以 lodash@4.17.11 为例（不一定准确，仅示例），4 代表主版本，17 就是次版本，11 就是修订号，如果每一个变动都严格遵守，且每次都是 +1 的话，可以这样理解：lodash 经历了 3 次大的断代更新，即从 1 到 4，同时在 4 的大版本上，经历了 17 次的功能更新，并且向下兼容，至于 bug 修复之类也有 11 次。

每个包实际执行并不一定严格遵守这种语义化版本规范，所以也会带来一些管理困扰，但真正的困扰我们的反而不是版本号本身，而是包与包之间的依赖关系，以及包自身的版本稳定性（背后的代码稳定性），比如这是我多年前本地的一个项目，它的版本号是这样子的：

```
"dependencies": {
  "async": "~0.2.10",
  "bcrypt": "~0.7.8",
  "connect-mongo": "~0.3.3",
  "crypto": "~0.0.3",
  "express": "~3.4.8",
  "grunt": "~0.4.5",
  "grunt-concurrent": "~0.4.3",
  "grunt-contrib-jshint": "~0.10.0",
  "grunt-contrib-less": "~0.11.4",
  "grunt-contrib-uglify": "~0.5.1",
  "grunt-contrib-watch": "~0.6.1",
  "grunt-mocha-test": "~0.11.0",
  "grunt-nodemon": "~0.1.2",
  "jade": "~1.3.0",
  "moment": "~2.5.1",
  "mongoose": "~3.8.14",
  "underscore": "~1.6.0",
  "should": "^4.0.4"
}
```

那时候还是 grunt 全家桶，只要 grunt 的主版本发生大变化，那么它的插件，就有可能跑不起来，每次升级光弄插件版本的兼容性，就要折腾半死。这放到现在，对于 react-native 或者依赖 Babel 及它的各种插件的项目中，版本之间的兼容性管理都依然是容易出问题的地方。Node 社区也针对版本管理，有大量的讨论，包括有一些工具的产出，比如 facebook 开源的 Yarn 就是 npm 强有力的一个竞争对手。npm 也经历了几次大的升级，关于 npm 的包版本策略，我们放到锁包 - npm shrinkwrap 再来探讨。

包目录层级 - npm node_modules

在 npm 的升级历史中，有这样的一个重大的变化，那就是 node_module 是包依赖安装层级，在 npm2 时代和 npm3+ 时代，一个项目的 node_modules 目录是递归安装的，它是按照依赖关系进行文件夹的嵌套，比如：

```
├── connect-mongo
│ ├── node_modules
│ │ └── mongodb
│ │ ├── node_modules
├── mongoose
│ ├── node_modules
│ │ ├── mongodb
│ │ │ └── node_modules
│ │ └── sliced
├── async
├── grunt
│ ├── node_modules
│ │ ├── async
│ │ └── which
└── underscore
```

```
├── bluebird
└── request
├── node_modules
│ ├── har-validator
│ │ ├── node_modules
│ │ │ ├── ajv
│ │ │ │ ├── node_modules
│ │ │ │ │ ├── co
│ │ │ │ │ └── json-schema-traverse
│ ├── http-signature
│ │ ├── node_modules
│ │ │ └── sshpk
│ │ ├── node_modules
│ │ │ └── tweetnacl
│ └── uuid
└── request.js
```

从内心深处，我个人还是很喜欢这个时代的 npm 的，因为通常一个项目依赖三四十个包就算比较多了。在 node_modules 里面，也就三四十个目录，进去找一个包的源代码，或者去它的 node_modules 里继续向下找，会非常省事，尤其是当我去 review 源码去查找关键字的时候。但它的

缺点也有很多，比如嵌套可能会出现很深的情况，会遇到 windows 的文件路径长度限制，当然最敏感的是，会导致大量的代码冗余。比如我们上面的 connect-mongo 和mongoose 里面都用到 mongodb，grunt 里面也用到了 async 等等，这会导致整个项目体积特别的臃肿。

所幸是 npm3 时代里面策略改成了平铺结构，全部一股脑平铺到 node_modules 下面，比如 lodash 和 request 就变成了：

```
├── ajv
├── asn1
├── assert-plus
├── asynckit
├── aws-sign2
├── aws4
├── bcrypt-pbkdf
├── bluebird
├── caseless
├── co
├── combined-stream
├── delayed-stream
├── fast-deep-equal
├── fast-json-stable-stringify
├── forever-agent
├── form-data
├── uuid
└── ...省略
```

但是要注意，新的 npm 并不会无脑的平铺，而是会有一套算法来做同名且同版本的包去重，合理规划目录的嵌套层级。这样可以保证即便是有同名但是版本不同的模块，不会在 node_modules 里面冲突，同时只要不在同级冲突，npm 会尽可能把能复用的模块往高层级安装。这样可以达到最大程度的模块重用，代码冗余就大幅降低。我拿一个旧项目测试了下，npm2 安装后是 80MB 的体积，而用 npm3 安装后，node_modules 的体积降低到了 68MB，直降 15%，越大越复杂的项目，新的安装策略应该能带来更大的体积节省。

锁包 - npm shrinkwrap

除了安装策略外，npm 另外一个重大的升级，就是我们熟悉的 package-lock 文件，这是 npm5 以后带来的新特性。package-lock，顾名思义，就是把包的版本锁住，保证它的代码每一行每一个字节都恒久不变，为什么需要这样一种看上去奇葩的策略，我们还得结合上面的 Semantic Versioning 也就是包的语义化版本来说事。

在一个 package.json 里的 dependencies 里面，包的依赖版本可以这样写：

```
"lodash": "~3.9.0",
```

```
"lodash": "^3.9.0",
"lodash": ">3.9.0",
"lodash": ">=1.0.0-rc.2",
"lodash": "*"
// ... 更多写法不再列举
```

最常见的就是 ~ 和 ^ 这两种写法，它俩有什么区别呢?

~ 意思是，选择一个最近的小版本依赖包，比如 ~3.9.0 可以匹配到所有的 3.9.x 版本，但是不会匹配到 3.10.0，而 ^ 则是匹配最新的大版本，比如 ^3.9.0 可以匹配到所有的 3.x.x，但是不会匹配到 4.0.0。他们的好处很明显，就是当一个包有一些 bug， 作者修复之后，不需要我们开发者主动到 package.json 里，一个个的修改过去。事实上我们开发者也无从知晓作者什么时候升级了包，甚至我们都不知道里面有没有 bug，所以依靠 ~ 和 ^，它就能自动晋升到较新版本的包，里面包含了最新的代码。只不过 ^ 比 ~更加激进，可能会导致新包与项目的不兼容，而 ~ 会友好很多，但也不能保证 100% 的兼容，因为所有的包版本都是包作者自行管理的，作者的技术实力和版本意识也是有限的，它这次升级会不会导致你的项目出现问题，我们心里是没底的。

于是千古难题出现了，我们既想享受静默升级的好处，又要避免静默升级背后包代码的不兼容性，这两个实际上是冲突的。静默升级一定会带来代码变动，代码变动一定会带来兼容风险。而且，就算是我们把版本写死为 3.9.0 也无济于事，因为它自身向下依赖很多别的包，这些别的包又依赖了别的包，他们的包策略如果是语义化的，照样会带来包依赖树的不稳定（任何一个底层包代码有语义化升级）。

所以，路被堵死了，意味着除非我们把整个 node_modules 保存到本地，上传到 git 仓库，全量上传到服务器，我们根本无法保证代码的不变性，据淘宝的工程师讲，他们某段时间也确实是这么干的，全包上传，全包回滚，粗暴但实用。

那么到底应该怎么办呢，大家可能猜到了，答案就是 package-lock.json，也就是 npm 的锁包。

大家可以在本地的一个空目录下，执行 npm init --yes && npm i lodash async -S，然后我们来看下 package-lock.json 里面的内容：

```
{
  "name": "npm",
  "version": "1.0.0",
  "lockfileVersion": 1,
  "requires": true,
  "dependencies": {
    "async": {
      "version": "2.6.1",
      "resolved": "http://registry.npm.taobao.org/async/download/async-2.
      6.1.tgz",
```

```
      "integrity": "sha1-skWiPKcZMAROxT+kaqAKPofGphA=",
      "requires": {
        "lodash": "^4.17.10"
      }
    },
    "lodash": {
      "version": "4.17.11",
      "resolved": "http://registry.npm.taobao.org/lodash/download/lodash-
    4.17.11.tgz",
      "integrity": "sha1-s56mIp72B+zYniyN8SU2iRysm40="
    }
  }
}
```

version 就是包的准确版本号（无语义化的跃迁），resolved 则是一个明确 tar 包地址，它是唯一不变的，并且还有 integrity 这个内容 hash 的值，他们三个就决定了这个包准确身份信息，这样第一个问题就解决了，那就是特定版本的包代码不变性。然后第二个问题，这些包向下依赖的包如何不变？

这个是通过每个包的 requires 字段实现。它实际上跟每个包的内部 package.json 的 dependencies 里的包是一一对应的，所以包的依赖关系也有了，无论嵌套多少层级，在 lock 文件里面，它都有 version、resolved、integrity 来保证单包不变性，那么整包就保证了代码不变。

可以把 package-lock.json 理解为一个详细描述代码版本的快照文件，它储存了 node_modules 当前的包代码状态，无论被哪个团队成员拿走项目，无论是本地还是服务器上 npm install，都能依据 package-lock.json 里面的包状态，原封不动的复原node_modules 里面的代码版本。这个就是锁包功能，其实在 npm5 之前就提供了，也就是 npm shrinkwrap，它需要手动执行，而现在则是自动生成。

如果你完全不依赖锁包功能，则可以将它关闭：npm config set package-lock false

包脚本 - npm scripts

npm 最强大的能力，除了 install 安装能力，就是脚本能力。在 package.json 里的 scripts 里配置的各种任务，都可以这样直接调用：

```
npm start
npm run dev
npm run build:prod
```

结合 npm 社区海量的包资源，跨平台执行也完全没有问题，比如 rm -rf 在 windows 下不支持，或者考虑支持 windows/linux 都可以设置环境变量，都可以换一个模块来执行，比如：

```
"scripts": {
  "build": "npm run build:prod",
  "clean:dist": "rimraf ./dist",
  "build:prod": "cross-env NODE_ENV=production webpack"
}
# 如下命令行均可执行
  npm run clean:dist
  npm run build:prod
  npm run build
```

npm scripts 如此之强大，甚至直接替换历史产物 grunt/gulp，尤其是处理一些构建预准备工作或构建后任务，比如先检查代码规范，再跑单元测试，最后跑构建，构建成功了就发一个钉钉通知到团队等等，这些任务可能是级联关系也可能是并行关系，在 npm scripts 里面也轻松搞定，比如：

```
"scripts": {
  // 通过 && 分隔，如果 clean:dist 任务失败，则不会执行后面的构建任务
  "build:task1": "npm run clean:dist && npm run build:prod"
  // 通过 ; 分隔，无论 clean:dist 是否成功，运行后都继续执行后面的构建任务
  "build:task2": "npm run clean:dist;npm run build:prod"
  // 通过 || 分隔，只有当 clean:dist 失败，才会继续执行后面的构建任务
  "build:task3": "npm run clean:dist||npm run build:prod"
  "clean:dist": "rimraf ./dist",
  "build:prod": "cross-env NODE_ENV=production webpack",
  // 对一个命令传配置参数，可以通过 -- --prod
  // 比如 npm run compile:prod 相当于执行 node ./r.js --prod
  "compile:prod": "npm run compile -- --prod",
  "compile": "node ./r.js",
}
```

通过上面的案例，我们可以发现，npm scripts 可以构建非常复杂的任务。不过 npm scripts 也会带来一些问题，比如非常复杂的 scripts 会带来非常复杂的依赖队列，不好维护。针对这一点，建议把每个独立的任务都分拆开进行组合，可以把复杂的任务独立写入到一个本地的脚本中，比如task.js。如果需要底层系统命令支撑，又实在找不到跨平台的包，也可以在它里面使用 shelljs 来调用系统命令，甚至不仅仅局限于 Node 的包，在 script 里面调用 python 脚本和 bash 脚本也一样溜。相信我，npm scripts 会给你打开一片新天地，大家有时间也可以研究下 npmasbuildtool 的 scripts 清单。

包执行工具 - npx

npx 是 npm 自带的非常酷炫的功能，直接执行依赖包里的二进制文件，比如：

```
# 先安装一个 cowsay
 npm install cowsay -D
# 直接通过 npx 来调用 cowsay 里的二进制文件
 npx cowthink Node 好玩么
 ______________
( Node 好玩么 )
 --------------
        o   ^__^
         o  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


 npx cowsay 爽爆了
 ________
< 爽爆了 >
 --------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

甚至我们 npm i webpack -D 以后，可以直接 npx http-server 把静态服务开起来。

包发布 - npm publish

好的，看过了 npm 的主要命令，我们来看下如何发布一个包。首先你要有一个 npm 的账号和 Github 账号，可以分别到 npmjs.com 和 github.com 注册（Github 还需要配置 ssh key）。这些都搞定后，就可以准备开发和发布 NPM 包了，整个流程很简单，总共都不超过 10 步：

1. 本地（或者从 Github 上）创建创建一个空项目，拉到本地
2. 增加 .gitignore 忽略文件和 README
3. npm init 生成 package.json
4. 编写功能代码，增加到目录 /lib
5. npm install 本地包进行测试
6. npm publish 发布包
7. npm install 线上包进行验证

8. 修改代码发布一个新版本

那到底如何实操呢，我们且往下看。

编程练习 - 实现一个 Node LTS 查看工具

我们知道 Node 版本就像做火箭一样，一直飙升，有的是 LTS 版本，有的不是，我们想时不时回头看 Node 都发布过哪些版本，总是不太方便。那我们就来开发一个这样的工具，给它起名字叫 nlts 吧，然后可以到 github 上新建一个 repo 名字就叫做 nlts，大家可以换一个其他名字。因为在 npm 上面，一个包名是唯一的，不能重复，然后就按照上面的几个步骤，我们逐个来实现。

1. 项目初始化

本地新建一个文件夹，叫做 node-lts，命令行到这个目录下，如我的电脑上就是：

```
  cd nlts
# 通过 touch 新建一个 markdown 的文件，用来描述包功能
  touch README.md
# 通过 touch 新建一个 git 忽略文件
  touch .gitignore
```

打开 .gitignore，输入如下内容，把一些无关文件排除出去。

```
.DS_Store
npm-debug.log
node_modules
yarn-error.log
.vscode
.eslintrc.json
```

如果 Github 上创建了项目，可以再把本地的项目和线上 repo 做关联；或者把空仓库拉下来，我们在空仓库里，增加上述文件，再进行后面的操作。

2. npm init 生成 package.json

```
  npm init
Press ^C at any time to quit.
# 回车确认或者输入另外一个名字作为包名
package name: (nlts)
# 版本就从 1.0.0 开始
version: (1.0.0)
```

```
# 简单的描述
description: CommandLine Tool for Node LTS
# 包的入口文件地址，通过 index.js 暴露内部函数
entry point: (index.js) index.js
# 测试脚本，可以先留空，大家根据实际情况取舍
test command:
# 包的 github 仓库地址
git repository:
# 一些功能关键词描述
keywords: Node LTS
# 作者自己
author:
# 开源的协议，默认是 ISC，我个人喜欢 MIT
license: (ISC) MIT

# 检查信息无误，输入 yes 回车即可
About to write to /Users/xiaojusurvey/nlts/package.json:
{
  "name": "nlts",
  "version": "1.0.0",
  "description": "CommandLine Tool for Node LTS",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Node", "LTS"
  ],
  "author": "",
  "license": "MIT"
}
Is this OK? (yes) yes


# 输入 ls 查看当前包内文件
 ls
package.json README.md
```

上面的 lib/index.js 要检查下，如果是放到仓库目录下的话，就不需要加 lib，直接 index.js 就行。

如果想更省事一些，可以用脚手架来做，先安装：

```
npm i yo generator-nm -g
```

然后运行 yo nm，会有一堆类似上面的问询，它会帮你把 .gitignore licence 这些模块中必备的文件都生成好，非常方便。

3. 增加功能代码目录 /libs

现在进入编码环节了，首先，根目录下增加一个 index.js，通过它来暴露 libs 下的模块：

```
const query = require('./libs/query')
const update = require('./libs/update')
module.exports = {
  query,
  update
}
```

/libs/update.js 可以用来放数据源的获取和更新，而 /libs/query.js 里面可以放对数据的二次加工格式化之类，首先是 /libs/update.js 获取 Node LTS 数据：

```
const axios = require('axios')
const { compareVersions } = require('compare-versions')
module.exports = async (v) => {
  // 拿到所有的 Node 版本
  const { data } = await axios
    .get('https://nodejs.org/dist/index.json')
  // 把目标版本的 LTS 都挑选出来
  return data.filter(node => {
    const cp = v
    ? (compareVersions(node.version, 'v' + v + '.0.0') >= 0)
    : true
    return node.lts && cp
  }).map(it => {
    // 踢出去 file 这个字段，其他的全部返回
    const { files, ...rest } = it
    return { ...rest }
  })
}
```

然后是 /libs/query.js：

```
const Table = require('cli-table')
function query(dists) {
  const keys = Object.keys(dists[0])
  // 建立表头
```

```
  const table = new Table({
    head: keys
  })
  // 拼接出表格的每一行
  return dists
    .reduce((res, item) => {
      table.push(
      Object.values(item)
      )
      return res
    }, table).toString()
}
module.exports = query
```

最后，再增加一个 bin 文件夹，在它里面增加一个 nlts 脚本文件，在里面写入：

```
#!/usr/bin/env node
const pkg = require('../package')
const query = require('..').query
const update = require('..').update
function printResult(v) {
  update(v).then(dists => {
    const results = query(dists, v)
    console.log(results)
    process.exit()
  })
}
function printVersion() {
  console.log('nlts ' + pkg.version)
  process.exit()
}
function printHelp(code) {
  const lines = [
    '',
    '  Usage:',
    '    nlts [8]',
    '',
    '  Options:',
    '    -v, --version          print the version of vc',
    '    -h, --help             display this message',
    '',
    '  Examples:',
    '    $ nlts 8',
    ''
  ]
```

```
  console.log(lines.join('\n'))
  process.exit(code || 0)
}
// 包的入口函数，里面对参数做剪裁处理，拿到入参并给予
// 不同入参的处理逻辑
function main(argv) {
  // 命令行的入参
  if (!argv) {
    printHelp(1)
  }
  // 兼容 nlts --lts=10
  const getArg = function() {
    let args = argv.shift()
    args = args.split('=')
    if (args.length > 1) {
      argv.unshift(args.slice(1).join('='))
    }
    return args[0]
  }
  let arg
  while (argv.length) {
    arg = getArg()
    switch(arg) {
      case '-v':
      case '-V':
      case '--version':
        printVersion()
      break
      case '-h':
      case '-H':
      case '--help':
        printHelp()
      break
        default:
        printResult(arg)
      break
    }
  }
}
// 启动程序就开始执行主函数
main(process.argv.slice(2))
module.exports = main
```

#!/usr/bin/env node 加上 #! 这里是定义当前脚本的执行环境是用 Node 执行。安装包以后我们就可以直接在命令行来调用执行，那么可以到 package.json 来配置下执行路径，在 package.json 里面增加一个配置属性（少数同学可能遇到 bin/nlts 失败，可以尝试 bin/nlts.js 试试看）：

```
"bin": {
  "nlts": "bin/nlts.js"
},
```

然后对于用到的模块，我们在包目录下，执行：

npm i axios cli-color cli-table compare-versions -S

这样安装后，package-lock.json 也自动创建了， 整个的目录结果如下：

```
~ tree -L 2
.
├── README.md
├── bin
│   └── nlts
├── index.js
├── libs
│   ├── query.js
│   └── update.js
├── node_modules
├── package-lock.json
└── package.json
```

再把 README.md 文档内容完善一下，我们的代码就准备好了。

4. npm install 本地包进行测试

等到代码写完，就可以本地测试了，本地测试最简单的办法，就是通过 npm link 安装下，如果失败，可以试下 sudo npm link。

```
  sudo npm link
npm WARN nlts@1.0.0 No description
npm WARN nlts@1.0.0 No repository field.
up to date in 0.468s
4 packages are looking for funding
  run
npm fund
 for details
/usr/local/bin/nlts -> /usr/local/lib/node_modules/nlts/bin/nlts.js
/usr/local/lib/node_modules/nlts -> /Users/xiaojusurvey/nlts
```

然后边调试代码边测试，测试完毕后，可以直接在本地指定目录来全局安装，首先卸载掉之前可能测试安装过的全局包：

```
npm uninstall nlts -g
```

然后可以在命令行窗口用绝对路径，或者直接进入到包目录下，执行全局安装动作：

```
npm i ./ -g
```

安装后，测试下 nlts 16，会拿到这样一个截图：

version	date	npm	v8	uv	zlib	openssl	modules	lts	security
v20.16.0	2024-07-24	10.8.1	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false
v20.15.1	2024-07-08	10.7.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	true
v20.15.0	2024-06-20	10.7.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false
v20.14.0	2024-05-28	10.7.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false
v20.13.1	2024-05-09	10.5.2	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false
v20.13.0	2024-05-07	10.5.2	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false
v20.12.2	2024-04-10	10.5.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	true
v20.12.1	2024-04-03	10.5.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	true
v20.12.0	2024-03-26	10.5.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false
v20.11.1	2024-02-13	10.2.4	11.3.244.8	1.46.0	1.2.13.1-motley	3.0.13+quic	115	Iron	true
v20.11.0	2024-01-09	10.2.4	11.3.244.8	1.46.0	1.2.13.1-motley	3.0.12+quic	115	Iron	false
v20.10.0	2023-11-22	10.2.3	11.3.244.8	1.46.0	1.2.13.1-motley	3.0.12+quic	115	Iron	false
v20.9.0	2023-10-24	10.1.0	11.3.244.8	1.46.0	1.2.13.1-motley	3.0.10+quic	115	Iron	false
v18.20.4	2024-07-08	10.7.0	10.2.154.26	1.44.2	1.3.0.1-motley	3.0.13+quic	108	Hydrogen	true
v18.20.3	2024-05-20	10.7.0	10.2.154.26	1.44.2	1.3.0.1-motley	3.0.13+quic	108	Hydrogen	false
v18.20.2	2024-04-10	10.5.0	10.2.154.26	1.44.2	1.3.0.1-motley	3.0.13+quic	108	Hydrogen	true
v18.20.1	2024-04-02	10.5.0	10.2.154.26	1.44.2	1.3.0.1-motley	3.0.13+quic	108	Hydrogen	true
v18.20.0	2024-03-26	10.5.0	10.2.154.26	1.44.2	1.3.0.1-motley	3.0.13+quic	108	Hydrogen	false
v18.19.1	2024-02-13	10.2.4	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.13+quic	108	Hydrogen	true
v18.19.0	2023-11-29	10.2.3	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.12+quic	108	Hydrogen	false
v18.18.2	2023-10-13	9.8.1	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.10+quic	108	Hydrogen	true
v18.18.1	2023-10-10	9.8.1	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.10+quic	108	Hydrogen	false
v18.18.0	2023-09-18	9.8.1	10.2.154.26	1.46.0	1.2.13.1-motley	3.0.10+quic	108	Hydrogen	false
v18.17.1	2023-08-08	9.6.7	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.10+quic	108	Hydrogen	false
v18.17.0	2023-07-18	9.6.7	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.9+quic	108	Hydrogen	false
v18.16.1	2023-06-20	9.5.1	10.2.154.26	1.44.2	1.2.13	3.0.9+quic	108	Hydrogen	true
v18.16.0	2023-04-12	9.5.1	10.2.154.26	1.44.2	1.2.13	3.0.8+quic	108	Hydrogen	false
v18.15.0	2023-03-05	9.5.0	10.2.154.26	1.44.2	1.2.13	3.0.8+quic	108	Hydrogen	false
v18.14.2	2023-02-21	9.5.0	10.2.154.26	1.44.2	1.2.13	3.0.8+quic	108	Hydrogen	false
v18.14.1	2023-02-16	9.3.1	10.2.154.23	1.44.2	1.2.13	3.0.8+quic	108	Hydrogen	true
v18.14.0	2023-02-01	9.3.1	10.2.154.23	1.44.2	1.2.13	3.0.7+quic	108	Hydrogen	false
v18.13.0	2023-01-05	8.19.3	10.2.154.23	1.44.2	1.2.13	3.0.7+quic	108	Hydrogen	false
v18.12.1	2022-11-04	8.19.2	10.2.154.15	1.43.0	1.2.11	3.0.7+quic	108	Hydrogen	true
v18.12.0	2022-10-25	8.19.2	10.2.154.15	1.43.0	1.2.11	3.0.5+quic	108	Hydrogen	false

5. npm publish 发布包

代码写完测试完，就可以发布到 npm 上了，再发布之前，别忘记先把代码 push 到 github 上来记录这一版本的变化，至于发布动作则很简单，先确保到 npmjs.com 注册好一个账号且邮箱验证完毕（有的国内邮箱会验证失败，比如 yeah.net 网易邮箱），然后在本地命令行窗口登录：

```
1 ~ npm login
```

```
Username: xxxxx
Password:
Email: (this IS public) xxxx@gmail.com
Logged in as 4liangge on https://registry.npmjs.com/.

~ npm publish
```

6. 修改代码发布一个新版本

有时候我们会修一个 bug，或者增加一个新特性，甚至有断代更新，这时候版本管理就参考前面的语义化版本来管理就行。比如我们想要增加一个小特性，可以支持在表格里多呈现一个信息，就是每一个 LTS 版本它们有一个 API，比如 v10.14.1 和 v10.13.0 是两个独立的 API 文档地址，文档内容也是有差异的，那么我们首先到代码中，找到 /libs/update 里面的 map 函数，在里面增加一句代码：

```
const terminalLink = require('terminal-link')
const color = require('cli-color')
//...
.map(it => {
  const { files, ...rest } = it
  const doc = color.yellow(terminalLink('API',
https://nodejs.org/dist/   ${it.version}/docs/api/documentation.html
))
  return { ...rest, doc }
})
```

这里用到了 terminal-link、cli-color，我们安装一下 npm i terminal-link cli-color -S。安装后，同样走上面的测试步骤，测试通过后，把 packge.json 的版本号改一下，这个功能是增加一个链接，在展示上有变化，但是向下是兼容的，所以可以把版本号从 1.0.0 改为 1.1.0。改完后，我们继续推送到 github 上后，再 npm publish 就可以了。

version	date	npm	v8	uv	zlib	openssl	modules	lts	security	doc
v20.16.0	2024-07-24	10.8.1	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false	API (https://nodejs.org/dist/v20.16.0/docs/api/documentation.html)
v20.15.1	2024-07-08	10.7.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	true	API (https://nodejs.org/dist/v20.15.1/docs/api/documentation.html)
v20.15.0	2024-06-20	10.7.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false	API (https://nodejs.org/dist/v20.15.0/docs/api/documentation.html)
v20.14.0	2024-05-28	10.7.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false	API (https://nodejs.org/dist/v20.14.0/docs/api/documentation.html)
v20.13.1	2024-05-09	10.5.2	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false	API (https://nodejs.org/dist/v20.13.1/docs/api/documentation.html)
v20.13.0	2024-05-07	10.5.2	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false	API (https://nodejs.org/dist/v20.13.0/docs/api/documentation.html)
v20.12.2	2024-04-10	10.5.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	true	API (https://nodejs.org/dist/v20.12.2/docs/api/documentation.html)
v20.12.1	2024-04-03	10.5.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	true	API (https://nodejs.org/dist/v20.12.1/docs/api/documentation.html)
v20.12.0	2024-03-26	10.5.0	11.3.244.8	1.46.0	1.3.0.1-motley	3.0.13+quic	115	Iron	false	API (https://nodejs.org/dist/v20.12.0/docs/api/documentation.html)
v20.11.1	2024-02-13	10.2.4	11.3.244.8	1.46.0	1.2.13.1-motley	3.0.13+quic	115	Iron	true	API (https://nodejs.org/dist/v20.11.1/docs/api/documentation.html)
v20.11.0	2024-01-09	10.2.4	11.3.244.8	1.46.0	1.2.13.1-motley	3.0.12+quic	115	Iron	false	API (https://nodejs.org/dist/v20.11.0/docs/api/documentation.html)
v20.10.0	2023-11-22	10.2.3	11.3.244.8	1.46.0	1.2.13.1-motley	3.0.12+quic	115	Iron	false	API (https://nodejs.org/dist/v20.10.0/docs/api/documentation.html)
v20.9.0	2023-10-24	10.1.0	11.3.244.8	1.46.0	1.2.13.1-motley	3.0.10+quic	115	Iron	false	API (https://nodejs.org/dist/v20.9.0/docs/api/documentation.html)
v18.20.4	2024-07-08	10.7.0	10.2.154.26	1.44.2	1.3.0.1-motley	3.0.13+quic	108	Hydrogen	true	API (https://nodejs.org/dist/v18.20.4/docs/api/documentation.html)
v18.20.3	2024-05-20	10.7.0	10.2.154.26	1.44.2	1.3.0.1-motley	3.0.13+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.20.3/docs/api/documentation.html)
v18.20.2	2024-04-10	10.5.0	10.2.154.26	1.44.2	1.3.0.1-motley	3.0.13+quic	108	Hydrogen	true	API (https://nodejs.org/dist/v18.20.2/docs/api/documentation.html)
v18.20.1	2024-04-02	10.5.0	10.2.154.26	1.44.2	1.3.0.1-motley	3.0.13+quic	108	Hydrogen	true	API (https://nodejs.org/dist/v18.20.1/docs/api/documentation.html)
v18.20.0	2024-03-26	10.5.0	10.2.154.26	1.44.2	1.3.0.1-motley	3.0.13+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.20.0/docs/api/documentation.html)
v18.19.1	2024-02-13	10.2.4	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.13+quic	108	Hydrogen	true	API (https://nodejs.org/dist/v18.19.1/docs/api/documentation.html)
v18.19.0	2023-11-29	10.2.3	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.12+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.19.0/docs/api/documentation.html)
v18.18.2	2023-10-13	9.8.1	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.10+quic	108	Hydrogen	true	API (https://nodejs.org/dist/v18.18.2/docs/api/documentation.html)
v18.18.1	2023-10-10	9.8.1	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.10+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.18.1/docs/api/documentation.html)
v18.18.0	2023-09-18	9.8.1	10.2.154.26	1.46.0	1.2.13.1-motley	3.0.10+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.18.0/docs/api/documentation.html)
v18.17.1	2023-08-08	9.6.7	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.10+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.17.1/docs/api/documentation.html)
v18.17.0	2023-07-18	9.6.7	10.2.154.26	1.44.2	1.2.13.1-motley	3.0.9+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.17.0/docs/api/documentation.html)
v18.16.1	2023-06-20	9.5.1	10.2.154.26	1.44.2	1.2.13	3.0.9+quic	108	Hydrogen	true	API (https://nodejs.org/dist/v18.16.1/docs/api/documentation.html)
v18.16.0	2023-04-12	9.5.1	10.2.154.26	1.44.2	1.2.13	3.0.8+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.16.0/docs/api/documentation.html)
v18.15.0	2023-03-05	9.5.0	10.2.154.26	1.44.2	1.2.13	3.0.8+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.15.0/docs/api/documentation.html)
v18.14.2	2023-02-21	9.5.0	10.2.154.26	1.44.2	1.2.13	3.0.8+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.14.2/docs/api/documentation.html)
v18.14.1	2023-02-16	9.3.1	10.2.154.23	1.44.2	1.2.13	3.0.8+quic	108	Hydrogen	true	API (https://nodejs.org/dist/v18.14.1/docs/api/documentation.html)
v18.14.0	2023-02-01	9.3.1	10.2.154.23	1.44.2	1.2.13	3.0.7+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.14.0/docs/api/documentation.html)
v18.13.0	2023-01-05	8.19.3	10.2.154.23	1.44.2	1.2.13	3.0.7+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.13.0/docs/api/documentation.html)
v18.12.1	2022-11-04	8.19.2	10.2.154.15	1.43.0	1.2.11	3.0.7+quic	108	Hydrogen	true	API (https://nodejs.org/dist/v18.12.1/docs/api/documentation.html)
v18.12.0	2022-10-25	8.19.2	10.2.154.15	1.43.0	1.2.11	3.0.5+quic	108	Hydrogen	false	API (https://nodejs.org/dist/v18.12.0/docs/api/documentation.html)

可以看到打印的表格里，多了 API 这一栏，同时 nlts 的版本也升级到了 1.1.0 了。到这里为止，发包流程全部搞定了。但是这些包是公开的，当我们希望自己的包不是发布到 npm 公共空间时候，我们就可以选择发布到私有源，npm 也提供了收费的服务，让我们无论个人还是组织都可以管理维护自己的私有源。那么有免费的私有源，比如前面我们提到的 Aliyun Registry，我自己差不多试用了1 年，感觉还是很好用的。使用也很简单，首先大家到阿里云注册一个账号，拿到自己的用户 ID，然后打开这个地址，进去后，新建一个 registry，可以拿到自己的账号和密码，有了这个就好办了。

我们把现在本地的包代码，改成 @scope/ntsl，然后本地 npm registry 切换到这个源，重新 npm login/npm publish 就可以了，流程一样就不再演示了。

npm 是我们学习 Node 过程中必须掌握的一个技能，npm 用的溜，不仅可以给我们的生活学习带来很多好用三方或者自研的工具，也可以帮我们打开视野，看到Node 社区生机勃勃充满想象力的一面。