

1. 包管理器npm介绍

1.1 什么是npm

NPM的全称是Node Package Manager，是一个NodeJS包管理和分发工具，已经成为了非官方的发布Node模块（包）的标准。

2020年3月17日，[Github](#)宣布收购npm，GitHub现在已经保证npm将永远免费。

小贴士：

简单的讲，npm就是现代工程化的JavaScript项目中的依赖管理工具，工程化项目中的JavaScript依赖包全部通过npm工具进行安装和管理，开发者也可以通过npm工具发布个人开发的依赖包项目提供给世界范围内的程序员使用。

1.2 如何使用npm

本地安装了NodeJS环境后，在系统的命令行工具中除node命令外，还包含npm命令。npm命令可以帮助开发者快速的安装和管理项目以来。

查看npm版本的方式

```
npm -v
```

控制台上会展示当前npm以来的版本信息

```
zhangyunpeng@zhangyunpengdeMacBook-Pro ~ % npm -v
7.21.0
```

查看npm可用功能

开发者成功安装了npm依赖管理工具后，可以通过命令行的方式查看npm所包含的所有功能，利用 `npm -h` 指令，查看下面的注释了解npm主要功能介绍。

```
zhangyunpeng@zhangyunpengdeMacBook-Pro ~ % npm -h
npm <command>

Usage:
#npm install 会自动安装你的项目package.json文件中所包含的所有依赖到本地
npm install    install all the dependencies in your project
#npm install <包名> 将指定依赖包下载并安装到你的项目中
npm install <foo> add the <foo> dependency to your project
#npm test 运行当前项目的测试用例
npm test       run this project's tests
#npm run <命令名称> 会自动运行当前项目scripts中所包含的同名指令
npm run <foo>    run the script named <foo>
#npm <命令> -h 快速查看当前命令的帮助文档
npm <command> -h quick help on <command>
#npm -l 列出所有命令的使用说明
npm -l         display usage info for all commands
#以下命令不常用
npm help <term> search for help on <term>
npm help npm   more involved overview
```

所有可用命令列表

All commands:

```
access, adduser, audit, bin, bugs, cache, ci, completion,
config, dedupe, deprecate, diff, dist-tag, docs, doctor,
edit, exec, explain, explore, find-dupes, fund, get, help,
hook, init, install, install-ci-test, install-test, link,
ll, login, logout, ls, org, outdated, owner, pack, ping,
pkg, prefix, profile, prune, publish, rebuild, repo,
restart, root, run-script, search, set, set-script,
shrinkwrap, star, stars, start, stop, team, test, token,
uninstall, unpublish, unstar, update, version, view, whoami
```

Specify configs in the ini-formatted file:

```
/Users/zhangyunpeng/.npmrc
```

or on the command line via: `npm <command> --key=value`

More configuration info: `npm help config`

Configuration fields: `npm help 7 config`

```
npm@7.21.0 /usr/local/lib/node_modules/npm
zhangyunpeng@zhangyunpengdeMacBook-Pro ~ %
```

1.3 镜像地址管理

关于npm依赖中心

自从有了npm依赖管理工具后，所有互联网中存在的公共依赖都存在于<https://www.npmjs.com/> 网站中。

所以通过使用npm包管理器安装的依赖都可以在该网站中查询到依赖包的安装方式和使用文档，如图所示。

The screenshot shows the npm website interface. At the top, there's a navigation bar with links for Products, Pricing, and Documentation. Below the navigation bar is a search bar with the text "Search packages" and a "Search" button. A message banner indicates that two-factor authentication (2FA) is not enabled on the user's npm account. The main content area is divided into three sections: "Popular libraries" listing packages like lodash, react, chalk, tslib, axios, commander, express, react-dom, moment, and vue; "Discover packages" with filters for Front-end, Back-end, CLI, Documentation, CSS, Testing, IoT, Coverage, Mobile, Frameworks, and Robotics; and "By the numbers" showing statistics: 1,939,292 Packages, 36,464,275,213 Downloads - Last Week, and 180,415,337,947 Downloads - Last Month.

接下来可以通过浏览NPM网站进行简单的依赖包学习。

npm镜像管理配置

由于日常的大量的npm依赖都通过世界的npm依赖中心提供，所以使用npm工具安装JavaScript依赖的时候需要开发者必须连接互联网，此时就涉及到npm的镜像地址配置工作了。

由于不同国家的开发者所存在的网络环境不同，而npm的依赖对于国内开发者来说都是保存在国外，所以使用npm下载依赖包时连接默认地址会出现访问慢的问题，所以在刚刚安装npm依赖管理工具时大多数人都会先使用配置工具将镜像地址进行修改。

查看当前的npm镜像地址：

```
npm config get registry
```

默认的情况下npm返回的的镜像地址为<https://registry.npmjs.org/>

设置国内的npm镜像地址：

为了保证npm依赖的访问速度提升，各国都提供了很多的镜像地址，设置npm镜像地址的方式为

```
npm config set registry "镜像地址"
```

常用的npm镜像地址

```
npm ----- https://registry.npmjs.org/  
yarn ----- https://registry.yarnpkg.com/  
tencent ---- https://mirrors.cloud.tencent.com/npm/  
cnpm ----- https://r.cnpmjs.org/  
taobao ----- https://registry.npmirror.com/  
npmMirror ---- https://skimdb.npmjs.com/registry/
```

镜像管理工具

频繁使用 npm config 命令来切换镜像是非常麻烦的事情，所以重复的事情当然要交给工具去做，这时我们可以使用nrm镜像管理工具来实现快速的npm镜像地址切换。

镜像工具安装指令

```
npm i nrm -g #-g和--global代表全局安装的意思
```

安装完成后命令行工具中便可以使用nrm指令来进行镜像的管理和切换。

列出所有可用地址

```
nrm ls
```

添加新的地址

```
nrm add <key> <address>
```

删除已有的地址

```
nrm del <key>
```

切换现有的镜像地址

```
nrm use <key>
```

1.4 npm config的介绍

npm的命令行工具不仅仅可以切换镜像地址，还可以对npm所有的属性进行设置，在上面的章节中展示了设置或获取镜像地址的完整指令为

```
npm config get/set registry [<address>]
```

实际上 `npm config` 部分就代表操作了npm的配置文件的某个属性，所以set和get对应的就是获取或设置指定属性的结果。

查询详细的config数据

npm在电脑上的默认配置已经足以满足日常开发需求，但是如果涉及到npm的一些特殊参数查看时，还需要通过指令进行操作。

列出npm默认配置信息

```
npm config list #获取精简的npm配置信息
```

查看下面的案例

```
zhangyunpeng@zhangyunpengdeMacBook-Pro ~ % npm config list
#这行代表当前的npm配置信息保存在/Users/zhangyunpeng/.npmrc文件中
; "user" config from /Users/zhangyunpeng/.npmrc

//localhost:4873/:_authToken = "vY61v52JVw2iMlRQm8yS7g=="
//registry.npmjs.org/:_authToken = (protected)
home = "https://npm.taobao.org"
http://ssss = ""
ignore-scripts = false
registry = "https://registry.npmmirror.com/"
ssss = ""
# node命令所存放的目录
; node bin location = /usr/local/bin/node
#运行命令所在目录
; cwd = /Users/zhangyunpeng
; HOME = /Users/zhangyunpeng
; Run `npm config ls -l` to show all defaults.
```

根据日志中给出的地址找到文件中的.npmrc文件



.npmrc

```
ssss=
http://ssss=
ignore-scripts=false
//registry.npmjs.org/:_authToken=npm_H8cKmilBTv5yQhcx6GcmKlCJIJl0JW0o7KIW
registry=https://registry.npmmirror.com/
home=https://npm.taobao.org
//localhost:4873/:_authToken="vY61v52JVw2iMlRQm8yS7g=="
|
```

接下来执行 `npm config list -l`，这里列出了本地的所有npm配置属性。

```
zhangyunpeng@zhangyunpengdeMacBook-Pro ~ % npm config list -l
; "default" config from default values

_auth = (protected)
access = null
all = false
```

```
allow-same-version = false
also = null
audit = true
audit-level = null
auth-type = "legacy"
before = null
bin-links = true
browser = null
ca = null
cache = "/Users/zhangyunpeng/.npm"
cache-max = null
cache-min = 0
cafile = null
call = ""
cert = null
ci-name = null
cidr = null
color = true
commit-hooks = true
depth = null
description = true
dev = false
diff = []
diff-dst-prefix = "b/"
diff-ignore-all-space = false
diff-name-only = false
diff-no-prefix = false
diff-src-prefix = "a/"
diff-text = false
diff-unified = 3
dry-run = false
editor = "vi"
engine-strict = false
fetch-retries = 2
fetch-retry-factor = 10
fetch-retry-maxtimeout = 60000
fetch-retry-mintimeout = 10000
fetch-timeout = 300000
force = false
foreground-scripts = false
format-package-lock = true
fund = true
git = "git"
git-tag-version = true
global = false
global-style = false
globalconfig = "/usr/local/etc/npmrc"
heading = "npm"
https-proxy = null
if-present = false
; ignore-scripts = false ; overridden by user
include = []
include-staged = false
init-author-email = ""
init-author-name = ""
init-author-url = ""
init-license = "ISC"
init-module = "/Users/zhangyunpeng/.npm-init.js"
```

```
init-version = "1.0.0"
init.author.email = ""
init.author.name = ""
init.author.url = ""
init.license = "ISC"
init.module = "/Users/zhangyunpeng/.npm-init.js"
init.version = "1.0.0"
json = false
key = null
legacy-bundling = false
legacy-peer-deps = false
link = false
local-address = null
location = "user"
loglevel = "notice"
logs-max = 10
; long = false ; overridden by cli
maxsockets = 15
message = "%s"
metrics-registry = "https://registry.npmirror.com/"
node-options = null
node-version = "v16.5.0"
noproxy = [""]
npm-version = "7.21.0"
offline = false
omit = []
only = null
optional = null
otp = null
pack-destination = "."
package = []
package-lock = true
package-lock-only = false
parseable = false
prefer-offline = false
prefer-online = false
prefix = "/usr/local"
preid = ""
production = null
progress = true
proxy = null
read-only = false
rebuild-bundle = true
; registry = "https://registry.npmjs.org/" ; overridden by user
save = true
save-bundle = false
save-dev = false
save-exact = false
save-optional = false
save-peer = false
save-prefix = "^"
save-prod = false
scope = ""
script-shell = null
searchexclude = ""
searchlimit = 20
searchopts = ""
searchstaleness = 900
```

```
shell = "/bin/zsh"
shrinkwrap = true
sign-git-commit = false
sign-git-tag = false
sso-poll-frequency = 500
sso-type = "oauth"
strict-peer-deps = false
strict-ssl = true
tag = "latest"
tag-version-prefix = "v"
timing = false
tmp = "/var/folders/36/2z_w159s5yx56w2rd37z2bcr0000gn/T"
umask = 0
unicode = true
update-notifier = true
usage = false
user-agent = "npm/7.21.0 node/v16.5.0 darwin x64 workspaces/false"
userconfig = "/Users/zhangyunpeng/.npmrc"
version = false
versions = false
viewer = "man"
which = null
workspace = []
workspaces = false
yes = null

; "user" config from /Users/zhangyunpeng/.npmrc

//localhost:4873/:_authToken = "vY61v52Jvw2iM1RQm8ys7g=="
//registry.npmjs.org/:_authToken = (protected)
home = "https://npm.taobao.org"
http://ssss = ""
ignore-scripts = false
registry = "https://registry.npmmirror.com/"
ssss = ""

; "cli" config from command line options

long = true
zhangyunpeng@zhangyunpengdeMacBook-Pro ~ %
```

如果想要更加方便的查看全局属性可以使用下列指令

```
npm config list --json
```

如果需要编辑文件可以通过下列指令

```
npm config edit
```

2. 企业级npm包管理器实用攻略

对于npm的应用其实大多数开发者涉及到的场景都是以下集中情况：

- 安装项目的所有依赖包 `npm install`

- 安装执行依赖包到项目 `npm install <packageName>`
- 删除置顶依赖包 `npm uninstall <packageName>`
- 执行项目的功能脚本

```
npm run build #构建项目
npm run dev/serve/start #运行项目
npm run eslint/... #其他项目功能
```

若仅掌握以上类型的npm指令正常在开发过程中能解决日常的大部分问题，但是对于想要晋升的开发者来说，如果只对npm了解的很片面代表所从事的项目级别和研发级别不是很高，所以接下来围绕Node项目进一步的了解npm。

2.1 初始化工程化项目

对于工程化项目来说，仅仅创建一个文件夹是不够的，我们需要在创建文件夹的基础上为该项目创建一个描述文件，也即是在前端项目中特别常见的package.json文件。那么这个文件究竟包含多少属性，每个属性有什么样的规则呢？

npm init 介绍

可以通过 `npm init -h` 查看该指令究竟有多少种使用方式：

```
zhangyunpeng@zhangyunpengdeMacBook-Pro test1 % npm init -h
npm init

Create a package.json file

Usage:
#npm init -f | -y 代表初始化当前项目的package.json文件
npm init [--force|-f|--yes|-y|--scope]
#npm init 名称 相当于以某个线上的模版初始化项目，比如npm init react-app 项目名称
npm init <@scope> (same as `npm <@scope>/create`)
#同上
npm init [<@scope>/]<name> (same as `npm [<@scope>/]create-<name>`)

Options:
[-y|--yes] [-f|--force]
[-w|--workspace <workspace-name> [-w|--workspace <workspace-name> ...]]
[-ws|--workspaces]

aliases: create, innit

Run "npm help init" for more info
```

该指令最常用的部分就是 `npm init -y`

初始化一个项目

1. 在编辑器中创建一个文件夹test
2. 在命令行工具中打开test文件夹并且输入指令

```
npm init -y
```

3. 命令行工具中会弹出如下日志，并且文件夹中会自动生成package.json配置文件


```
zhangyunpeng@zhangyunpengdeMacBook-Pro test % npm init -y
wrote to /Users/zhangyunpeng/Desktop/JavaScript/test/package.json:

{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

4. 打开package.json文件查看默认属性

```
{
  "name": "test", //项目名称
  "version": "1.0.0", //项目当前的版本号
  "description": "", //项目的描述内容
  "main": "index.js", //项目作为依赖包被别人引用时所执行的文件
  "scripts": { //项目的调试命令
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [], //项目作为依赖发布后的搜索关键字
  "author": "", //项目作者
  "license": "ISC" //软件许可证
}
```

2.2 依赖管理介绍

项目开发阶段，使用npm管理项目依赖是开发者做的最多的工作，所以我们接下来学习一下如何通过npm管理项目依赖。

安装依赖

安装项目依赖通过 `npm install` 指令完成，不过npm安装依赖的方式多种多样，并且存在很多的指令组合，可以使用 `npm install -h` 查看可用的npm安装方式

```
zhangyunpeng@zhangyunpengdeMacBook-Pro test % npm install -h
npm install

Install a package

Usage:
npm install [<@scope>/]<pkg>
npm install [<@scope>/]<pkg>@<tag>
npm install [<@scope>/]<pkg>@<version>
npm install [<@scope>/]<pkg>@<version range>
npm install <alias>@npm:<name>
npm install <folder>
npm install <tarball file>
npm install <tarball url>
```

```
npm install <git:// url>
npm install <github username>/<github project>

Options:
#这里为install后面的可选指令，不同指令代表不同的安装模式
[-S|--save|--no-save|--save-prod|--save-dev|--save-optional|--save-peer]
[-E|--save-exact] [-g|--global] [--global-style] [--legacy-bundling]
[--strict-peer-deps] [--no-package-lock]
[--omit <dev|optional|peer>] [--omit <dev|optional|peer> ...]] [--ignore-scripts]
[--no-audit] [--no-bin-links] [--no-fund] [--dry-run]
[-w|--workspace <workspace-name> [-w|--workspace <workspace-name> ...]]
[-ws|--workspaces]
#这里代表别名所以npm i npm in npm install 等指令均代表npm install
aliases: i, in, ins, inst, insta, instal, isnt, isnta, isntal, add

Run "npm help install" for more info
zhangyunpeng@zhangyunpengdeMacBook-Pro test %
```

查看安装指令后很多人才会发现，原来npm安装一个依赖包都有这么多的可选项，那么这里实际上开发者使用更多的是如下几个指令：

```
npm install xxx -S|--save|-s #代表本地安装一个依赖包在项目中使用
```

```
npm install xxx -D|--save-dev #代表安装一个依赖包到项目，该依赖包为运行项目所需的依赖
```

```
npm install xxx -g|--global #代表全局安装一个依赖包到npm本地电脑的全局依赖文件夹中
```

全局安装的依赖可以通过如下指令查看：

```
npm ls -g
```

运行效果如下：

```
zhangyunpeng@zhangyunpengdeMacBook-Pro test % npm ls -g
/usr/local/lib
├─ @nestjs/cli@8.2.0
├─ @vue/cli@5.0.1
├─ cnpm@6.1.1
├─ npm@7.21.0
├─ nrm@1.2.5
├─ nvm@0.0.4
├─ p-nrm@1.0.7
├─ typescript@4.5.5
├─ verdaccio@5.8.0
└─ yarn@1.22.11
```

全局安装的依赖包会自导保存到日志输出的目录下，访问该目录可以查看所有的全局依赖会统一存放在/usr/local/lib目录下的node_module文件夹中

node_modules

> @nestjs

> @vue

> cnpm

> npm

> nrm

> nvm

> p-nrm

> typescript

> verdaccio

> yarn

除以上几个指令外，实际上npm安装依赖时还可以选择如下指令：

```
npm install xxx --save-optional #代表可选依赖
```

```
npm install xxx --save-peer #代表当前项目作为依赖包提供给其他项目使用时，项目需要自行安装当前依赖
```

依赖安装实战

1. 在项目中执行如下命令安装webpack

```
npm i webpack -D
```

2. 在项目中执行如下命令安装vue

```
npm i vue -S
```

3. 在项目中执行如下命令安装react

```
npm i react --save-peer
```

4. 在项目中执行如下命令安装webpack-cli

```
npm i webpack-cli --save-optional
```

全部执行完毕之后该项目的package.json会变成如下内容：

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
```

```
"test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [],
"author": "",
"license": "ISC",
//该属性下的依赖代表当前项目在实际运行时所需要依赖的包，并不会与当前项目核心代码组合到一起，项目打包构建成为静态资源或发布成依赖包供给其他开发者使用时不会进入构建后的代码中
"devDependencies": {
  "webpack": "^5.72.0"
},
//该属性下的依赖代表当前项目的核心代码依赖，项目打包构建成为静态资源或发布成依赖包供给其他开发者使用时所必要的依赖包
"dependencies": {
  "vue": "^3.2.33"
},
//该属性代表同级依赖，指的是当前的项目在构建后的运行时所需要的依赖，并构建后的项目中不包含此依赖，当其他开发者安装安装本依赖包时需要额外下载当前依赖才能保证本依赖包正常工作
"peerDependencies": {
  "react": "^18.0.0"
},
//此选项代表可选依赖
"optionalDependencies": {
  "webpack-cli": "^4.9.2"
}
}
```

关于npm install 和 npm ci

简而言之，使用npm install和npm ci之间的主要区别是：

- 该项目必须具有现有的package-lock.json或npm-shrinkwrap.json。
- 如果程序包锁中的依赖项与package.json中的依赖项不匹配，则npm ci将退出并显示错误，而不是更新程序包锁。
- npm ci一次只能安装整个项目：此命令不能添加单个依赖项。
- 如果已经存在node_modules，它将在npm ci开始安装之前自动删除。
- 它永远不会写入package.json或任何包锁：安装实际上是冻结的。

本质上，`npm install` 读取 `package.json` 会创建依赖项列表，并用于 `package-lock.json` 告知要安装这些依赖项的版本。**如果不存在依赖项 package-lock.json，则将添加 npm install。**

`npm ci`（以C intinuous I ntegration 命名）直接从中安装依赖项，`package-lock.json` 并且 `package.json` 仅用于验证没有不匹配的版本。**如果缺少任何依赖项或版本不兼容，它将抛出错误。**

使用 `npm install` 到一个项目中添加新的依赖，并更新相关的更新。通常，在拉动更新依赖项列表的更改之后，您将在开发期间使用它，但是 `npm ci` 在这种情况下使用它可能是一个好主意。

使用 `npm ci` 如果你需要一个确定的，可重复的构建。例如，在持续集成，自动化作业等过程中，以及在首次安装依赖项时，而不是 `npm install`。

npm install

- 安装软件包及其所有依赖项。
- 依赖关系由 `npm-shrinkwrap.json` 和 `package-lock.json`（按此顺序）驱动。
- **不带参数**：安装本地模块的依赖项。
- 可以安装全局软件包。

- 将在中安装所有缺少的依赖项 `node_modules`。
- 它可能会写入

```
package.json
```

或

```
package-lock.json
```

。

- 与参数（`npm i packagename`）结合使用时，它可能会写入 `package.json` 以添加或更新依赖项。
- 当不带参数使用时，（`npm i`）可能会写入以 `package-lock.json` 锁定某些依赖项的版本（如果它们尚未在此文件中）。

npm ci

- 至少需要 `npm v5.7.1`。
- 需要 `package-lock.json` 或 `npm-shrinkwrap.json` 存在。
- 如果这两个文件的依赖项不匹配，则会引发错误 `package.json`。
- 一次删除 `node_modules` 并安装所有依赖项。
- 它从不写入 `package.json` 或 `package-lock.json`。

npm ci的算法

而 `npm ci` 从生成整个依赖树 `package-lock.json` 或 `npm-shrinkwrap.json`，`npm install` 更新的内容 `node_modules` 使用以下算法（源）：

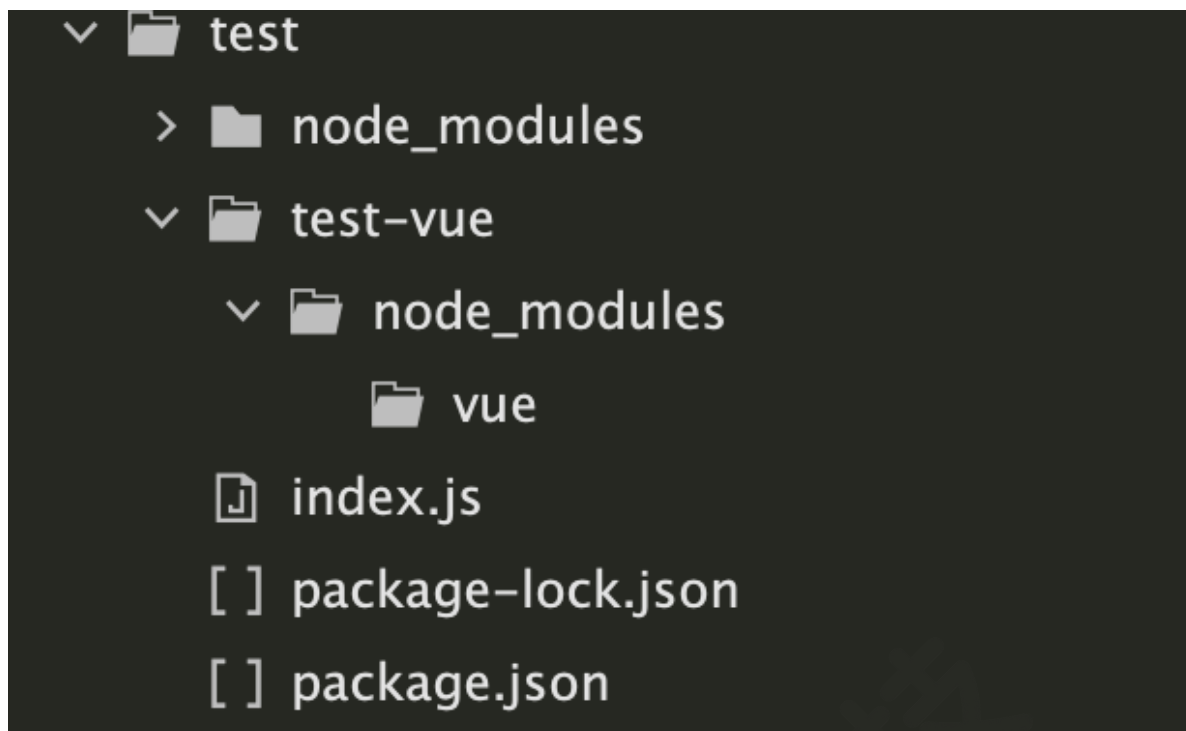
```
load the existing node_modules tree from disk
clone the tree
fetch the package.json and assorted metadata and add it to the clone
walk the clone and add any missing dependencies
dependencies will be added as close to the top as is possible
without breaking any other modules
compare the original tree with the cloned tree and make a list of
actions to take to convert one to the other
execute all of the actions, deepest first
kinds of actions are install, update, remove and move
```

2.3 npm依赖加载规则

依赖包安装位置

使用 `npm` 所安装的依赖，都会被保存到当前项目目录下的 `node_modules` 文件夹中，项目中的依赖加载规则是自底向上寻找 `node_modules` 文件夹中的依赖包。

为验证 `npm` 的依赖包加载规则，在 `test` 项目中创建名为 `test-vue` 的文件夹，在文件夹内部创建 `node_modules` 文件夹，在其内部创建 `vue` 文件夹，如图所示



此文件结构代表N个嵌套的工程化项目，用此项目识别npm加载依赖是按照什么规则加载的，所以接下来使用命令行工具打开test-vue/node_modules/vue文件夹，在其中执行 `npm init -y` 将该项目初始化为一个npm项目。

```
Wrote to /Users/zhangyunpeng/Desktop/JavaScript/test/test-  
vue/node_modules/vue/package.json:
```

```
{  
  "name": "vue",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC"  
}
```

```
zhangyunpeng@zhangyunpengdeMacBook-Pro vue %
```

初始化项目后，在vue文件夹中创建src文件，并在其中创建index.js文件，在文件内部编写代码

```
const { version, name } = require('../package.json')  
module.exports = { version, name }
```

此时项目的文件结构为

```
test-vue
├── node_modules
│   └── vue
│       ├── package.json
│       └── src
│           └── index.js
```

最后在test-vue/node_modules/vue中创建的package.json中找到main属性并做如下改造

```
{
  "name": "vue",
  "version": "1.0.0",
  "description": "",
  //main代表当前开发者使用import xx from 'vue'或require('vue')时加载的是src下的
  index.js文件
  "main": "./src/index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

接下来在test-vue根目录下创建index.js文件并初始化如下代码

```
const vue = require('vue')
const react = require('react')
console.log(vue)
console.log(react)
```

接下来在test-vue中打开命令行工具并执行运行命令

```
zhangyunpeng@zhangyunpengdeMacBook-Pro test-vue % node index
{ version: '1.0.0', name: 'vue' }
{
  Children: {
    map: [Function: mapChildren],
    forEach: [Function: forEachChildren],
    count: [Function: countChildren],
    toArray: [Function: toArray],
    only: [Function: onlyChild]
  },
  Component: [Function: Component],
  Fragment: Symbol(react.fragment),
  Profiler: Symbol(react.profiler),
  PureComponent: [Function: PureComponent],
  StrictMode: Symbol(react.strict_mode),
  Suspense: Symbol(react.suspense),
  __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED: {
    ReactCurrentDispatcher: { current: null },
    ReactCurrentBatchConfig: { transition: null },
    ReactCurrentOwner: { current: null },
    ReactDebugCurrentFrame: {
```

```

    setExtraStackFrame: [Function (anonymous)],
    getCurrentStack: null,
    getStackAddendum: [Function (anonymous)]
  },
  ReactCurrentActQueue: {
    current: null,
    isBatchingLegacy: false,
    didScheduleLegacyUpdate: false
  }
},
cloneElement: [Function: cloneElementWithValidation],
createContext: [Function: createContext],
createElement: [Function: createElementWithValidation],
createFactory: [Function: createFactoryWithValidation],
createRef: [Function: createRef],
forwardRef: [Function: forwardRef],
isValidElement: [Function: isValidElement],
lazy: [Function: lazy],
memo: [Function: memo],
startTransition: [Function: startTransition],
unstable_act: [Function: act],
useCallback: [Function: useCallback],
useContext: [Function: useContext],
useDebugValue: [Function: useDebugValue],
useDeferredValue: [Function: useDeferredValue],
useEffect: [Function: useEffect],
useId: [Function: useId],
useImperativeHandle: [Function: useImperativeHandle],
useInsertionEffect: [Function: useInsertionEffect],
useLayoutEffect: [Function: useLayoutEffect],
useMemo: [Function: useMemo],
useReducer: [Function: useReducer],
useRef: [Function: useRef],
useState: [Function: useState],
useSyncExternalStore: [Function: useSyncExternalStore],
useTransition: [Function: useTransition],
version: '18.0.0-fc46dba67-20220329'
}
zhangyunpeng@zhangyunpengdeMacBook-Pro test-vue %

```

执行命令后会发现在test-vue中的node_modules的vue会作为本项目的依赖优先加载，而项目外部的react对象仍然可以作为依赖被加载到本项目中，这个就是node_modules的依赖加载顺序。

所以为了进一步验证我们的猜想，接下来在test项目的根目录中的index.js文件中编写代码并运行

```

const vue = require('vue')
console.log(vue)

```

```

zhangyunpeng@zhangyunpengdeMacBook-Pro test % node index
{
  EffectScope: [class EffectScope],
  ReactiveEffect: [class ReactiveEffect],
  customRef: [Function: customRef],
  effect: [Function: effect],
  effectScope: [Function: effectScope],

```



```

getCurrentScope: [Function: getCurrentScope],
isProxy: [Function: isProxy],
isReactive: [Function: isReactive],
isReadonly: [Function: isReadonly],
isRef: [Function: isRef],
isShallow: [Function: isShallow],
markRaw: [Function: markRaw],
onScopeDispose: [Function: onScopeDispose],
proxyRefs: [Function: proxyRefs],
reactive: [Function: reactive],
readonly: [Function: readonly],
ref: [Function: ref],
shallowReactive: [Function: shallowReactive],
shallowReadonly: [Function: shallowReadonly],
shallowRef: [Function: shallowRef],
stop: [Function: stop],
toRaw: [Function: toRaw],
toRef: [Function: toRef],
toRefs: [Function: toRefs],
triggerRef: [Function: triggerRef],
unref: [Function: unref],
camelize: [Function (anonymous)],
capitalize: [Function (anonymous)],
normalizeClass: [Function: normalizeClass],
normalizeProps: [Function: normalizeProps],
normalizeStyle: [Function: normalizeStyle],
toDisplayString: [Function: toDisplayString],
toHandlerKey: [Function (anonymous)],
BaseTransition: {
  name: 'BaseTransition',
  props: {
    mode: [Function: String],
    appear: [Function: Boolean],
    persisted: [Function: Boolean],
    onBeforeEnter: [Array],
    onEnter: [Array],
    onAfterEnter: [Array],
    onEnterCancelled: [Array],
    onBeforeLeave: [Array],
    onLeave: [Array],
    onAfterLeave: [Array],
    onLeaveCancelled: [Array],
    onBeforeAppear: [Array],
    onAppear: [Array],
    onAfterAppear: [Array],
    onAppearCancelled: [Array]
  },
  setup: [Function: setup]
},
Comment: Symbol(Comment),
Fragment: Symbol(Fragment),
KeepAlive: {
  name: 'KeepAlive',
  __isKeepAlive: true,
  props: { include: [Array], exclude: [Array], max: [Array] },
  setup: [Function: setup]
},
Static: Symbol(Static),

```

```
Suspense: {
  name: 'Suspense',
  __isSuspense: true,
  process: [Function: process],
  hydrate: [Function: hydrateSuspense],
  create: [Function: createSuspenseBoundary],
  normalize: [Function: normalizeSuspenseChildren]
},
Teleport: {
  __isTeleport: true,
  process: [Function: process],
  remove: [Function: remove],
  move: [Function: moveTeleport],
  hydrate: [Function: hydrateTeleport]
},
Text: Symbol(Text),
callWithAsyncErrorHandling: [Function: callWithAsyncErrorHandling],
callWithErrorHandling: [Function: callWithErrorHandling],
cloneVNode: [Function: cloneVNode],
compatUtils: null,
computed: [Function: computed],
createBlock: [Function: createBlock],
createCommentVNode: [Function: createCommentVNode],
createElementBlock: [Function: createElementBlock],
createElementVNode: [Function: createBaseVNode],
createHydrationRenderer: [Function: createHydrationRenderer],
createPropsRestProxy: [Function: createPropsRestProxy],
createRenderer: [Function: createRenderer],
createSlots: [Function: createSlots],
createStaticVNode: [Function: createStaticVNode],
createTextVNode: [Function: createTextVNode],
createVNode: [Function: createVNodeWithArgsTransform],
defineAsyncComponent: [Function: defineAsyncComponent],
defineComponent: [Function: defineComponent],
defineEmits: [Function: defineEmits],
defineExpose: [Function: defineExpose],
defineProps: [Function: defineProps],
getCurrentInstance: [Function: getCurrentInstance],
getTransitionRawChildren: [Function: getTransitionRawChildren],
guardReactiveProps: [Function: guardReactiveProps],
h: [Function: h],
handleError: [Function: handleError],
initCustomFormatter: [Function: initCustomFormatter],
inject: [Function: inject],
isMemoSame: [Function: isMemoSame],
isRuntimeOnly: [Function: isRuntimeOnly],
isVNode: [Function: isVNode],
mergeDefaults: [Function: mergeDefaults],
mergeProps: [Function: mergeProps],
nextTick: [Function: nextTick],
onActivated: [Function: onActivated],
onBeforeMount: [Function (anonymous)],
onBeforeUnmount: [Function (anonymous)],
onBeforeUpdate: [Function (anonymous)],
onDeactivated: [Function: onDeactivated],
onErrorCaptured: [Function: onErrorCaptured],
onMounted: [Function (anonymous)],
onRenderTracked: [Function (anonymous)],
```

```
onRenderTriggered: [Function (anonymous)],
onServerPrefetch: [Function (anonymous)],
onUnmounted: [Function (anonymous)],
onUpdated: [Function (anonymous)],
openBlock: [Function: openBlock],
popScopeId: [Function: popScopeId],
provide: [Function: provide],
pushScopeId: [Function: pushScopeId],
queuePostFlushCb: [Function: queuePostFlushCb],
registerRuntimeCompiler: [Function: registerRuntimeCompiler],
renderList: [Function: renderList],
rendersSlot: [Function: rendersSlot],
resolveComponent: [Function: resolveComponent],
resolveDirective: [Function: resolveDirective],
resolveDynamicComponent: [Function: resolveDynamicComponent],
resolveFilter: null,
resolveTransitionHooks: [Function: resolveTransitionHooks],
setBlockTracking: [Function: setBlockTracking],
setDevtoolsHook: [Function: setDevtoolsHook],
setTransitionHooks: [Function: setTransitionHooks],
ssrContextKey: Symbol(ssrContext),
ssrUtils: {
  createComponentInstance: [Function: createComponentInstance],
  setupComponent: [Function: setupComponent],
  renderComponentRoot: [Function: renderComponentRoot],
  setCurrentRenderingInstance: [Function: setCurrentRenderingInstance],
  isVNode: [Function: isVNode],
  normalizeVNode: [Function: normalizeVNode]
},
toHandlers: [Function: toHandlers],
transformVNodeArgs: [Function: transformVNodeArgs],
useAttrs: [Function: useAttrs],
useSSRContext: [Function: useSSRContext],
useSlots: [Function: useSlots],
useTransitionState: [Function: useTransitionState],
version: '3.2.33',
warn: [Function: warn],
watch: [Function: watch],
watchEffect: [Function: watchEffect],
watchPostEffect: [Function: watchPostEffect],
watchSyncEffect: [Function: watchSyncEffect],
withAsyncContext: [Function: withAsyncContext],
withCtx: [Function: withCtx],
withDefaults: [Function: withDefaults],
withDirectives: [Function: withDirectives],
withMemo: [Function: withMemo],
withScopeId: [Function: withScopeId],
Transition: [Function: Transition] {
  displayName: 'Transition',
  props: {
    mode: [Function: String],
    appear: [Function: Boolean],
    persisted: [Function: Boolean],
    onBeforeEnter: [Array],
    onEnter: [Array],
    onAfterEnter: [Array],
    onEnterCancelled: [Array],
    onBeforeLeave: [Array],
```

```

    onLeave: [Array],
    onAfterLeave: [Array],
    onLeaveCancelled: [Array],
    onBeforeAppear: [Array],
    onAppear: [Array],
    onAfterAppear: [Array],
    onAppearCancelled: [Array],
    name: [Function: String],
    type: [Function: String],
    css: [Object],
    duration: [Array],
    enterFromClass: [Function: String],
    enterActiveClass: [Function: String],
    enterToClass: [Function: String],
    appearFromClass: [Function: String],
    appearActiveClass: [Function: String],
    appearToClass: [Function: String],
    leaveFromClass: [Function: String],
    leaveActiveClass: [Function: String],
    leaveToClass: [Function: String]
  }
},
TransitionGroup: {
  name: 'TransitionGroup',
  props: {
    mode: [Function: String],
    appear: [Function: Boolean],
    persisted: [Function: Boolean],
    onBeforeEnter: [Array],
    onEnter: [Array],
    onAfterEnter: [Array],
    onEnterCancelled: [Array],
    onBeforeLeave: [Array],
    onLeave: [Array],
    onAfterLeave: [Array],
    onLeaveCancelled: [Array],
    onBeforeAppear: [Array],
    onAppear: [Array],
    onAfterAppear: [Array],
    onAppearCancelled: [Array],
    name: [Function: String],
    type: [Function: String],
    css: [Object],
    duration: [Array],
    enterFromClass: [Function: String],
    enterActiveClass: [Function: String],
    enterToClass: [Function: String],
    appearFromClass: [Function: String],
    appearActiveClass: [Function: String],
    appearToClass: [Function: String],
    leaveFromClass: [Function: String],
    leaveActiveClass: [Function: String],
    leaveToClass: [Function: String],
    tag: [Function: String],
    moveClass: [Function: String]
  },
  setup: [Function: setup]
},

```

```

VueElement: [class VueElement],
createApp: [Function: createApp],
createSSRApp: [Function: createSSRApp],
defineCustomElement: [Function: defineCustomElement],
defineSSRCustomElement: [Function: defineSSRCustomElement],
hydrate: [Function: hydrate],
initDirectivesForSSR: [Function: initDirectivesForSSR],
render: [Function: render],
useCssModule: [Function: useCssModule],
useCssVars: [Function: useCssVars],
vModelCheckbox: {
  deep: true,
  created: [Function: created],
  mounted: [Function: setChecked],
  beforeUpdate: [Function: beforeUpdate]
},
vModelDynamic: {
  created: [Function: created],
  mounted: [Function: mounted],
  beforeUpdate: [Function: beforeUpdate],
  updated: [Function: updated]
},
vModelRadio: {
  created: [Function: created],
  beforeUpdate: [Function: beforeUpdate]
},
vModelSelect: {
  deep: true,
  created: [Function: created],
  mounted: [Function: mounted],
  beforeUpdate: [Function: beforeUpdate],
  updated: [Function: updated]
},
vModelText: {
  created: [Function: created],
  mounted: [Function: mounted],
  beforeUpdate: [Function: beforeUpdate]
},
vShow: {
  beforeMount: [Function: beforeMount],
  mounted: [Function: mounted],
  updated: [Function: updated],
  beforeUnmount: [Function: beforeUnmount]
},
withKeys: [Function: withKeys],
withModifiers: [Function: withModifiers],
compile: [Function: compileToFunction]
}

```

通过本案例的开发，我们完全理解了npm依赖包的加载机制以及import 或require时项目到底如何执行的。

2.4 bin属性的介绍

我们发现在使用npm安装依赖时，并不是所有依赖都是需要import到我们的项目中才能运行的依赖，比如nrm包就是当项目安装后，会在全局命令行工具中默认可以使用nrm指令来操作镜像地址。这个能力取决于package.json的bin属性。

接下来继续通过实践的方式学习如何创建带有命令行工具的项目。

1. 在编辑器中创建名为demo的项目
2. 在项目的根目录中创建bin文件夹并在其中创建index.js文件
3. 在index.js文件中编写如下命令

```
//#所声明的部分代表通过node命令行工具执行该文件
#!/usr/bin/env node
console.log('hello demo')
```

4. 执行 `npm init -y` 命令初始化package.json
5. 在demo根目录下创建一个空的index.js文件（此步骤为支持打包构建）
6. 检查package.json文件会发现内部多出一个属性，将该属性的值改为bin/index.js

```
{
  "name": "demo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "bin": { //bin代表可执行的命令行指令所调用的js文件映射规则，安装此项目后便可以直接通过
    命令行工具中使用demo指令
    "demo": "bin/index.js"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

7. 接下来要思考的就是如何让命令行工具中可以直接运行demo指令，这里不需要通过node命令在本地测试index.js因为这种测试毫无意义
8. 所以下一步需要做的就是将我们当前的项目变成一个真正的依赖包，这里便需要一个指令 `npm pack`
9. 在命令行工具中执行该指令

```
zhangyunpeng@zhangyunpengdeMacBook-Pro demo % npm pack
npm notice
npm notice 📦 demo@1.0.0
npm notice === Tarball Contents ===
npm notice 45B bin/index.js
npm notice 257B package.json
npm notice === Tarball Details ===
npm notice name: demo
npm notice version: 1.0.0
npm notice filename: demo-1.0.0.tgz
npm notice package size: 317 B
npm notice unpacked size: 302 B
npm notice shasum: 841d4a8ae108dfcab191d55170fef69699e41de7
npm notice integrity: sha512-EP4jbQIUgR8EP[...]rPNE5PnJwpHAg==
npm notice total files: 2
```

```
npm notice
demo-1.0.0.tgz
```

10. 会发现本项目中出现名为demo-1.0.0.tgz的文件，这个文件便是我们平时使用npm install所安装的依赖包的本体
11. 接下来需要将当前的demo安装包安装到我们的本地电脑上作为全局依赖，所以需要在命令行工具中打开demo目录并执行如下指令

```
zhangyunpeng@zhangyunpengdeMacBook-Pro demo % npm i ./demo-1.0.0.tgz -g

added 1 package in 668ms
```

12. 安装完毕后执行 `npm ls -g` 会发现我们的demo项目会包含在全局的npm依赖列表中

```
zhangyunpeng@zhangyunpengdeMacBook-Pro demo % npm ls -g
/usr/local/lib
├─ @nestjs/cli@8.2.0
├─ @vue/cli@5.0.1
├─ cnpm@6.1.1
├─ demo@1.0.0
├─ npm@7.21.0
├─ nrm@1.2.5
├─ nvm@0.0.4
├─ p-nrm@1.0.7
├─ typescript@4.5.5
├─ verdaccio@5.8.0
└─ yarn@1.22.11
```

13. 接下来在命令行工具中执行如下命令

```
zhangyunpeng@zhangyunpengdeMacBook-Pro demo % demo
hello demo
```

总结：

刚才的步骤完成后我们会发现，实际上使用npm安装的命令行依赖也是一个node项目，package.json的bin属性有着强大的功能，可以帮助我们创建一个命令行工具的映射，通过命令行工具便可以触发命令所对应的JS文件中的代码执行，代码中第一行带#的部分就是声明该文件是一个命令行运行文件，这样我们便可以省略很多shell编程环节，通过更简单的JavaScript语法来构建很多服务器工具应用。

2.5 scripts的作用

作为预设项目启动命令

从开始到现在我们都是在使用node命令执行项目中的.js文件，这种方式在实际调试时非常的麻烦，因为有些工具执行的调试命令非常长，属性很多，所以此时我们频繁使用node命令运行项目是非常不可取的，所以我们通常会使用package.json中的scripts来配置项目中各种运行的场景。

接下来回到最开始的test项目中，在其中的package.json中改造scripts属性

```
{
  "scripts": {
    "test": "node index"
  },
}
```

然后在命令行工具中执行

```
zhangyunpeng@zhangyunpengdeMacBook-Pro test % npm run test
#npm run 指令名称会在控制台上开启node执行（xing）行（hang）来运行test所对应的命令行指令，所以
在scripts中的key为npm run执行的命令名称，scripts中的key对应的value为实际npm执行的命令行
代码
> test@1.0.0 test
> node index

{
  EffectScope: [class EffectScope],
  ReactiveEffect: [class ReactiveEffect],
  customRef: [Function: customRef],
  effect: [Function: effect],
  effectScope: [Function: effectScope],
  getCurrentScope: [Function: getCurrentScope],
  isProxy: [Function: isProxy],
  isReactive: [Function: isReactive],
  isReadonly: [Function: isReadonly],
  isRef: [Function: isRef],
  isShallow: [Function: isShallow],
  markRaw: [Function: markRaw],
  onScopeDispose: [Function: onScopeDispose],
  proxyRefs: [Function: proxyRefs],
  reactive: [Function: reactive],
  readonly: [Function: readonly],
  ref: [Function: ref],
  shallowReactive: [Function: shallowReactive],
  shallowReadonly: [Function: shallowReadonly],
  shallowRef: [Function: shallowRef],
  stop: [Function: stop],
  toRaw: [Function: toRaw],
  ...
}
```

会发现我们使用npm run test 所执行的命令会触发node index执行。有些同学可能会觉得这种方式简直是多此一举，但实际上命令行工具在运行时会有大量的指令所对应的属性，比如npm就具备大量的指令和指令的自有属性，所以解析来，将test命令改为

```
{
  "scripts": {
    "test": "node index --port=8080 --host=127.0.0.1 --indexPath=index.html"
  },
}
```

然后将index.js中的代码改造为如下效果


```
const argv = process.argv
argv.filter(item => item.indexOf('=') !== -1).forEach(item => {
  let [ key , value ] = item.split('=')
  console.log(`参数${key}的值为:${value}`)
})
```

接下来在命令行工具中执行 `npm run test`

```
zhangyunpeng@zhangyunpengdeMacBook-Pro test % npm run test

> test@1.0.0 test
> node index --port=8080 --host=127.0.0.1 --indexPath=index.html

参数--port的值为:8080
参数--host的值为:127.0.0.1
参数--indexPath的值为:index.html
```

所以此时此刻便证明了scripts属性的重要性。

作为命令行工具的执行容器

scripts属性除上述作用外，还可以作为其他命令行工具的运行容器。简单的讲，我们在项目开发时会发现很多项目的scripts中运行的并不是node开头的命令而是如下代码

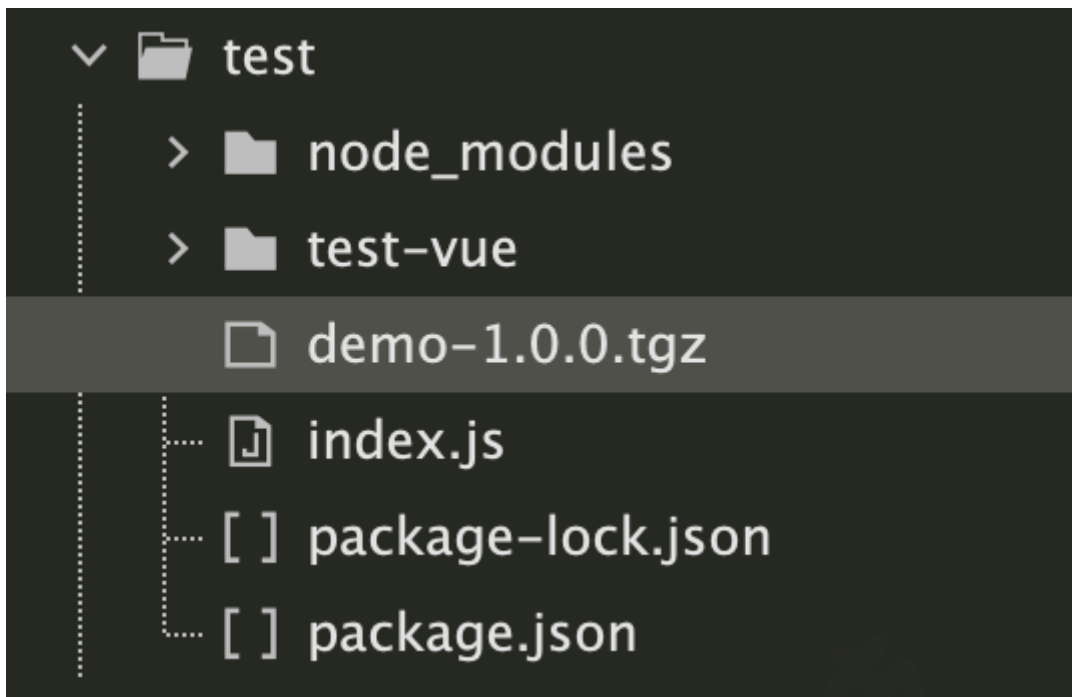
```
{
  "scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  }
}
```

我们在使用npm运行该指令时项目没有任何问题，而直接在命令行工具中执行vue-cli-service时会提示

```
zhangyunpeng@zhangyunpengdeMacBook-Pro test % vue-cli-service
zsh: command not found: vue-cli-service
```

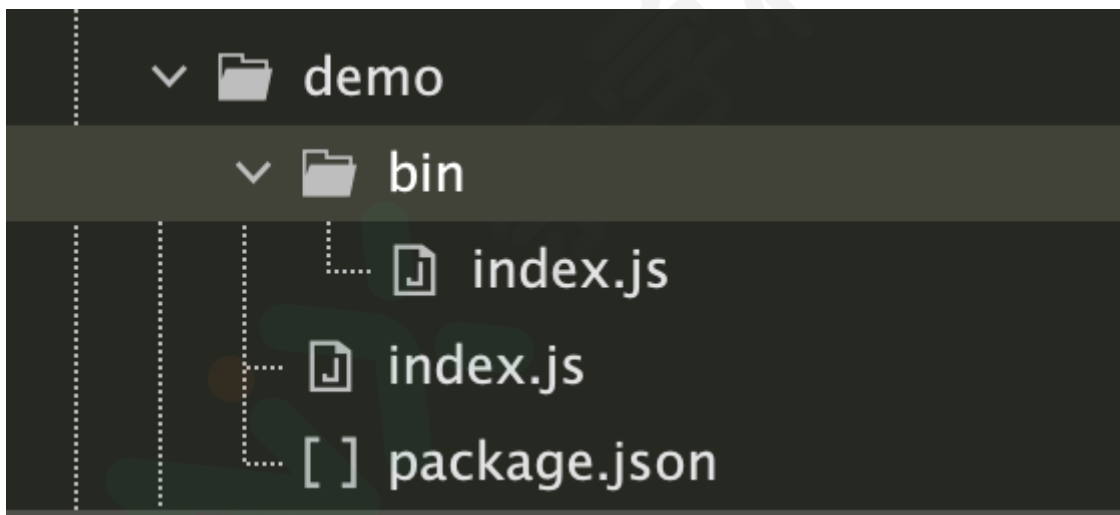
其原因是当前命令并没有被global安装而是指安装到了项目内部，所以我们并没有能力通过命令行工具全局使用该命令。之所以这样设计是因为不同的前端项目包含大量的运行指令，若全部指令都需要全局安装的话，会导致我们的电脑中存在大量的全局指令包，还会造成全局命令行指令的重名，所以npm的scripts在执行时可以获取到当前项目中的node_modules中的带有bin的项目，并且在npm run的运行阶段可以执行该指令，这样便可以实现减少全局命令的创建。

所以接下来，我们把之前的demo.1.0.0.tgz粘贴到test项目中并使用npm命令安装到本项目中



```
npm install ./demo-1.0.0.tgz -s
```

安装后检查当前项目的node_modules中是否出现demo文件夹并检查其文件结构，如图所示



接下来要卸载掉全局的demo命令

```
npm uninstall demo -g
```

确保命令行中无法运行demo命令

```
zhangyunpeng@zhangyunpengdeMacBook-Pro test % demo  
zsh: command not found: demo
```

然后在test的package.json中做如下改造

```
{
  "scripts": {
    "test-demo": "demo",
    "test": "node index --port=8080 --host=127.0.0.1 --indexPage=index.html"
  },
}
```

然后在命令行工具中执行

```
zhangyunpeng@zhangyunpengdeMacBook-Pro test % npm run test-demo

> test@1.0.0 test-demo
> demo

hello demo
```

此时便可以发现scripts标签可以将命令降级在项目中使用，防止对全局产生过度负担。

2.6 发布配置

通过上面的学习，我们已经了解了package.json中的常用属性的作用以及在实际场景中，项目作为依赖包和作为项目主体的区别，当我们创建的仍然是依赖包项目并且要发布给其他人使用时，仍然需要很多的其他配置。

版本迭代配置

每个 npm 包都有一个 package.json 文件，文件中的 version 字段即为当前包的版本号。version 字段一般由三位数构成，格式如下：x.x.x，分别对应着 version 里面的：major, minor, patch.; 若带预发号的话，格式为：x.x.x-x，最后一位表示预发号。

npm version 命令用于更改版本号的信息，并执行 commit 操作；该命令执行后，package.json 里的 version 会自动更新。

一般来说，当版本有较大改动时，变更第一位，执行命令：npm version major -m "description"，例如 1.0.0 -> 2.0.0；

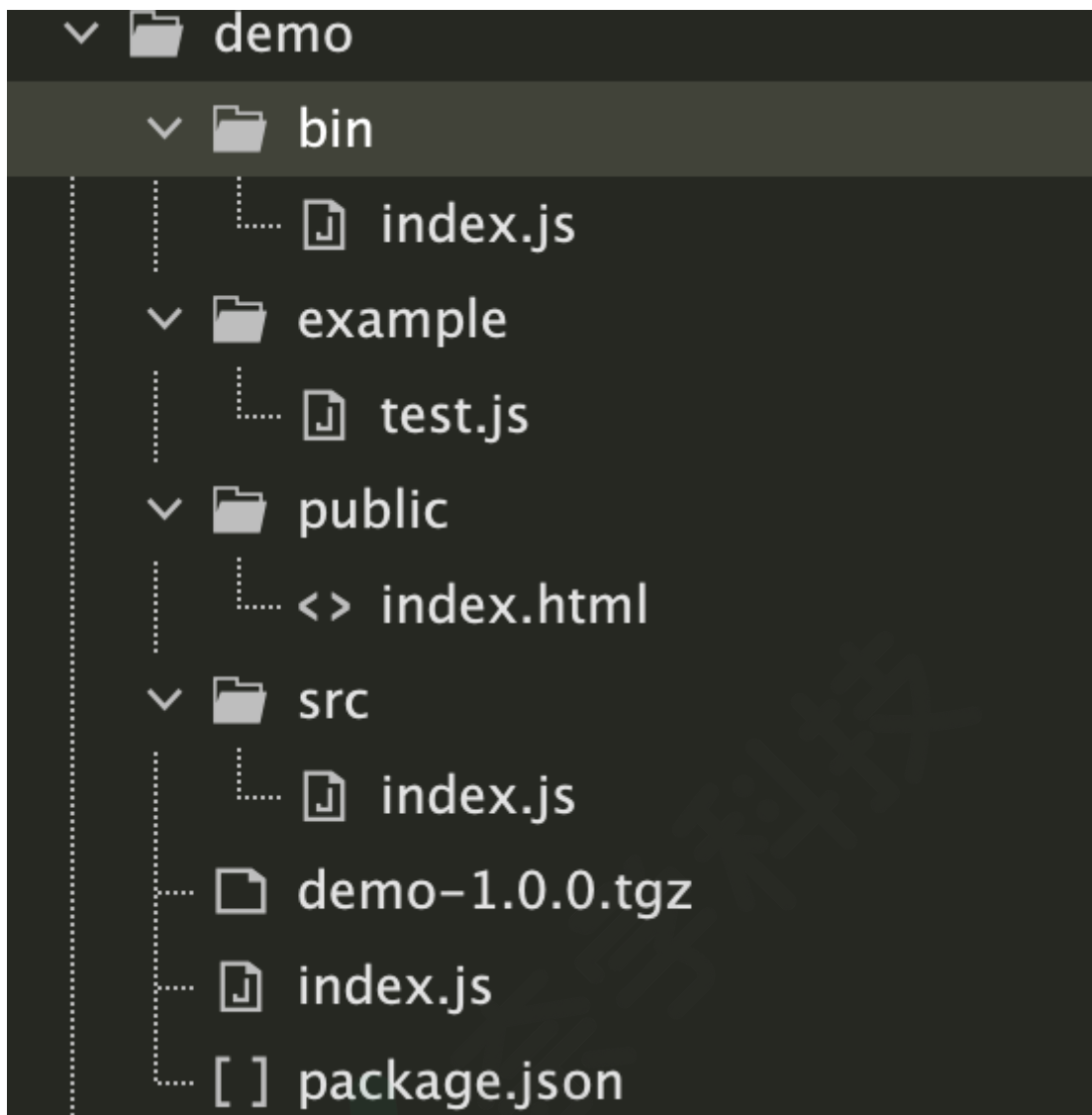
当前包变动较小时，可变更第二位，执行命令：npm version minor -m "description"，例如：1.0.0 -> 1.1.0；

当前包只是修复了些问题时，可变更第三位，执行命令：npm version patch -m "description"，例如：1.0.0 -> 1.0.1；

公共文件配置

我们在实际项目开发时，会发现实际上的node_modules中已经安装的依赖大多数都不包含其打包构建工具的配置以及项目的源代码目录，这并代表作者通过一个空项目只发布构建后的代码而是通过package.json中的files属性控制了项目中的目录结构如何发布。

接下来在demo项目中创建多个目录和文件如图所示。



接下来我们在当前项目的package.json文件中创建files实行并输入以下结果

```
{  
  "files":["bin","example"],  
}
```

然后在项目的目录下的命令行工具中输入

```
zhangyunpeng@zhangyunpengdeMacBook-Pro demo % npm version patch -m "版本更新"  
v1.0.1
```

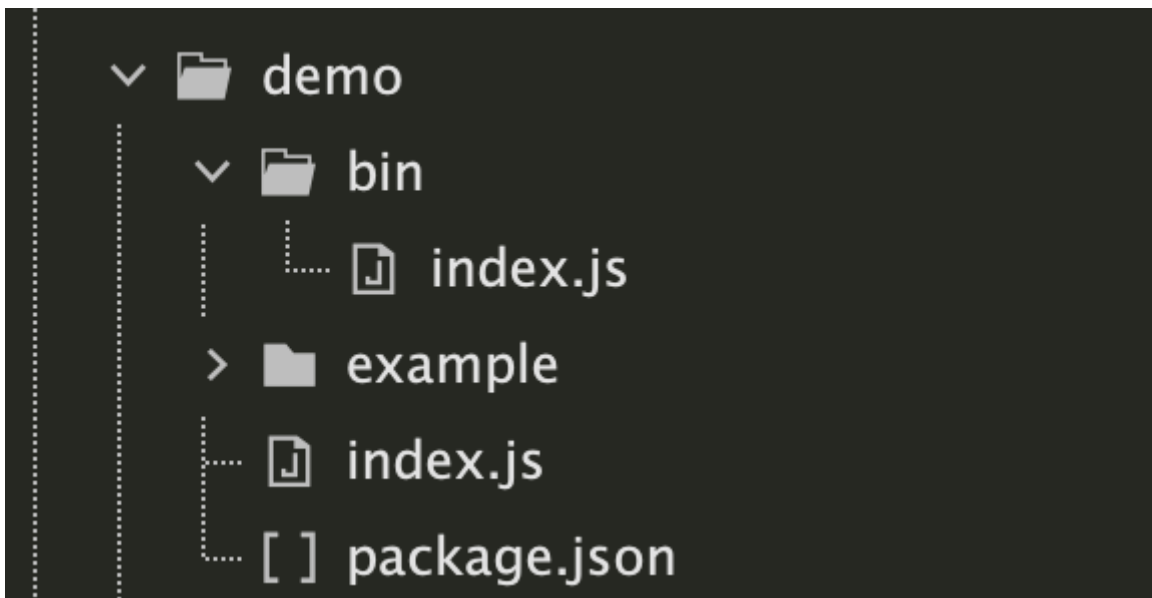
继续使用构建命令将项目打成压缩包

```
npm pack
```

将生成的demo-1.0.1.tgz转移到test项目中，并安装到test项目中

```
npm i ./demo-1.0.1.tgz -s
```

安装后找到依赖包中的文件结构会发现如图



只有files属性中设置的文件夹被发布出去了，这样便可以实现选择性发布。

3. 本地npm私服搭建

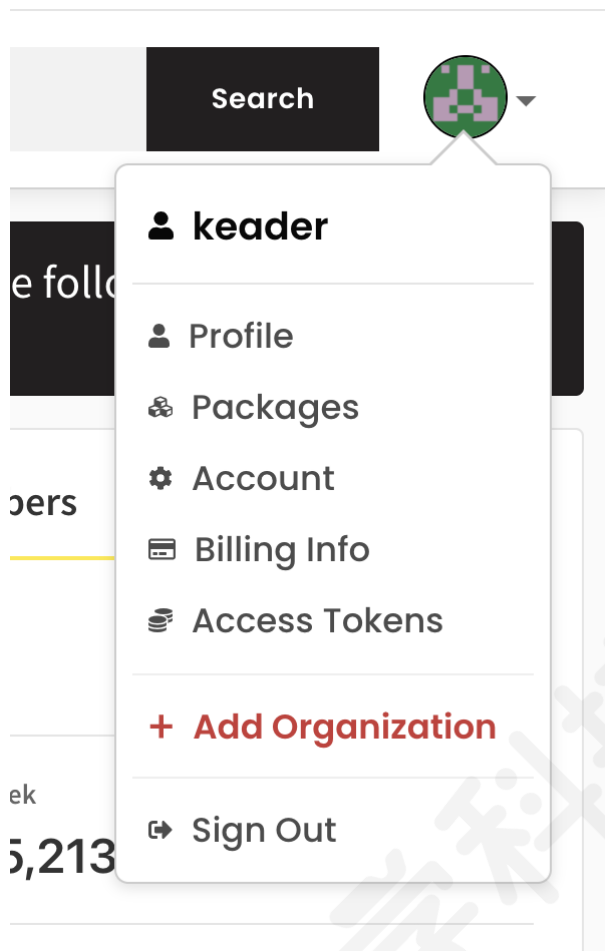
3.1 npm账号的注册与登陆

npm官方网站中集合了全世界的JavaScript依赖包，所以想要将做好的依赖提供给其他开发者使用最好的方式就是发不到npm平台，不过发布到npm平台首先要注册一个npm的账号。

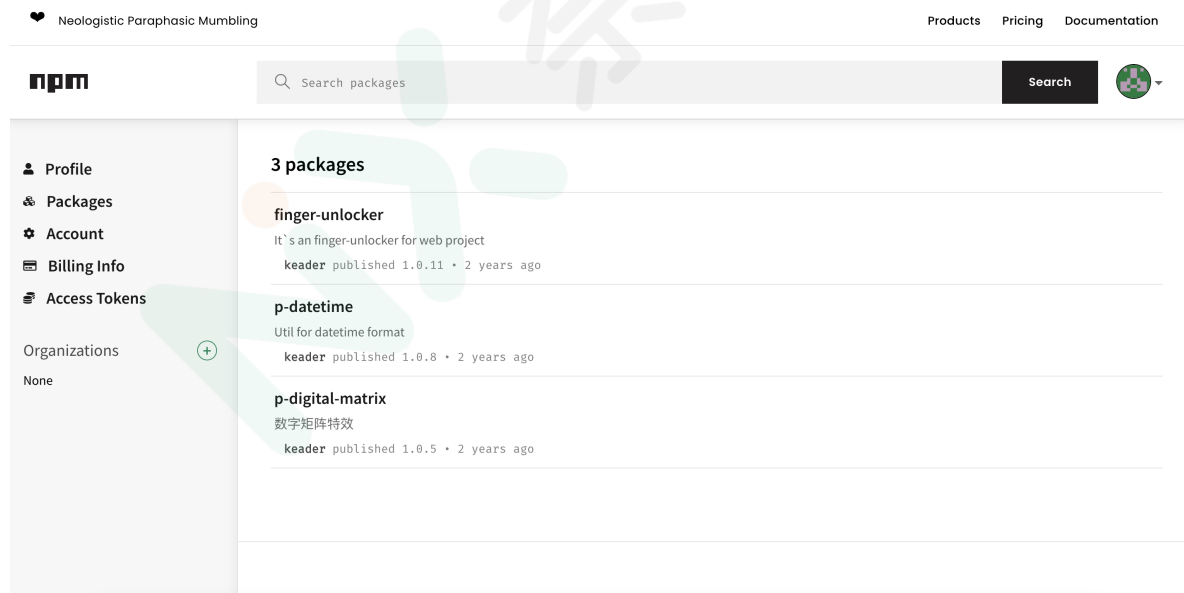
注册账号的地址为：<https://www.npmjs.com/signup>

注册过程就不演示了，纯GUI界面操作，所以不需要多介绍。

注册成功的账号登录后可以进入账号管理页面



在packages中可以看见自己发布的包以及包的情况



在此平台上所发布的包默认可以提供给全世界所有的开发者下载和使用，所以有些公司中技术团队开发的依赖包并不想开源或开放访问时，该平台就不适用于这些公司了，此时便需要使用npm私服。

3.2 npm私服的选择与搭建

市面上现存的npm私服种类并不是很多，但是也多种选择，不过早期的npm私服安装繁琐，搭建步骤复杂，非常不易于管理，直到verdaccio问世。

为什么选择verdaccio

1. 付费选择：

- MyGet (<https://www.myget.org>) 9美元/月，且只能有两个账号和1GB的存储空间。
- NPM Org (<https://www.npmjs.com>) 每个账号每月7美元

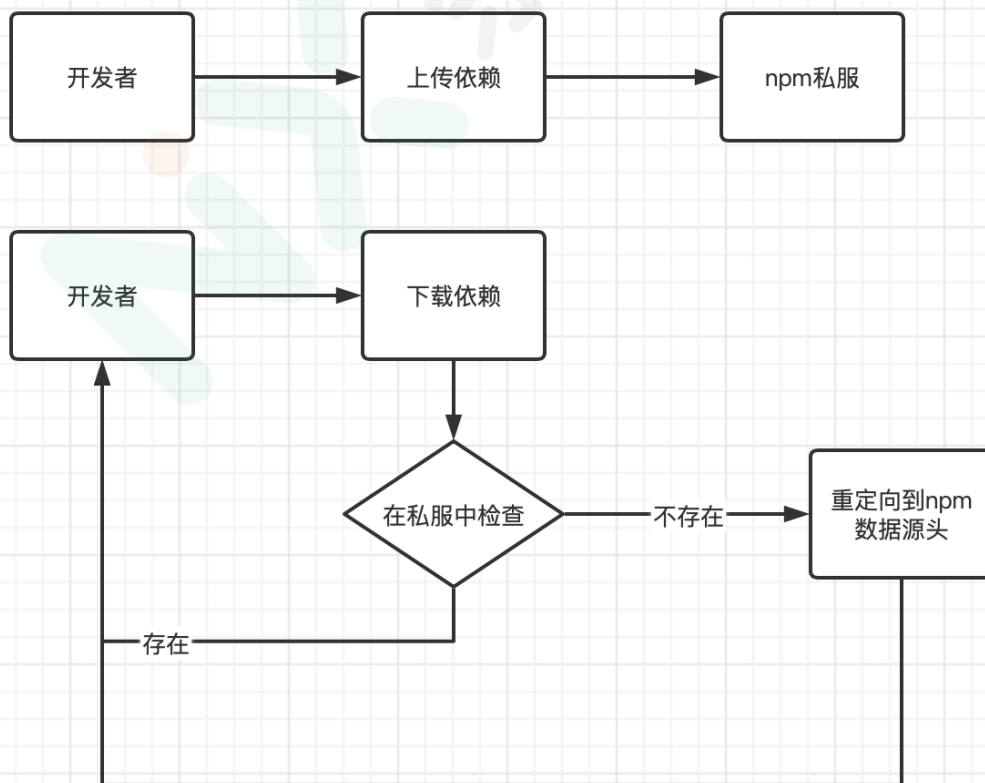
2. 免费选择:

- DIY NPM (<https://docs.npmjs.com/misc/registry>)
- Git, 这也是一种选择，在package.json中指定git仓库的URL即可，但是这种做法有些别扭，第一，使得package.json不够优雅，第二，当git仓库为private时，你需要HTTPS或SSH凭据，而且通常我们并没有每个团队的权限。
- Sinopia (<https://www.npmjs.com/package/sinopia>)
- Cnpmjs.org (<https://github.com/cnpm/cnpmjs.org>)

付费的我们就不考虑了，没这个必要，而且付费的也不是就更好。sinopia搭建十分简单友好，不过这玩意儿已经停止维护了，最近的更新在好多年前，但有一群人出了sinopia的一个分支，起了个名字叫 **verdaccio**，这个就是这次主要推荐的方案，这个库一直在积极维护中，github start 7000+，看来还是比较靠谱的，而且国内外各种资料参考下来，这个方案也是受到极力推荐的。verdaccio搭建私服很简单，相比于cnpm搭建，还需要安装配置mysql，这个绝对会少走一些坑。

什么是私服

首先我们要了解什么是私服，npm私服并不代表我们在本地搭建一个和npm官方完全一样的数据库并且将npm现有的依赖包完全克隆一份安装到我们的电脑上，这个思路就是不现实的思路，因为能装得下全世界所有npm依赖以及各版本的所有依赖对于个人来说是非常的困难的，所以私服可以理解为与淘宝镜像等镜像地址类型，首先他运行在本地，然后连接这个私服也可以下载npm上存在而私服中不存在的依赖包，当我们创建好私服之后，开发者可以在本地私服中上传需要私有化发布的依赖包，局域网内的用户都可以通过连接私服下载该依赖，连接私服的开发者仍然可以通过私服跳转到npm服务器下载依赖，这个就是私服使用场景。



verdaccio的安装和使用

Verdaccio 是一个简单的、零配置本地私有 npm 软件包代理注册表。Verdaccio 开箱即用，拥有自己的小型数据库，能够代理其它注册表（例如 npmjs.org），缓存下载模块。此外 Verdaccio 还易于扩展存储功能，它支持各种社区制作的插件，以连接到亚马逊的 s3、谷歌云存储等服务或创建自己的插件。

只需要通过 npm 命令就可以将 Verdaccio 安装到本地，步骤如下：

1. 在命令行工具中执行安装命令

```
npm i verdaccio -g
```

2. 运行 verdaccio 服务器

```
verdaccio
```

3. 运行后输出结果为

```
zhangyunpeng@zhangyunpengdeMacBook-Pro p-ui % verdaccio
warn --- config file - /Users/zhangyunpeng/.config/verdaccio/config.yaml
warn --- Plugin successfully loaded: verdaccio-htpasswd
warn --- Plugin successfully loaded: verdaccio-audit
warn --- http address - http://localhost:4873/ - verdaccio/5.8.0
```

4. 访问首页<http://localhost:4873/>

5. 在 i 图标的菜单内部可以设置语言

verdaccio 中的账号注册和项目发布

1. 使用 nrm 将本地的 verdaccio 数据源添加到列表中

```
nrm add local http://localhost:4873/
```

2. 通过 use 命令将本地的 npm 镜像切换为私服服务器地址

```
zhangyunpeng@zhangyunpengdeMacBook-Pro test % nrm use local

Registry has been set to: http://localhost:4873/
```

3. 通过命令行工具注册账号

```
zhangyunpeng@zhangyunpengdeMacBook-Pro demo % npm adduser
npm notice Log in on http://localhost:4873/
Username: test1
Password:
Email: (this IS public) 273274517@qq.com
Logged in as test1 on http://localhost:4873/.
```

4. 注册成功的用户可以直接在视图网站登陆



5. 接下来在demo项目中打开命令行工具并执行

```
zhangyunpeng@zhangyunpengdeMacBook-Pro demo % npm publish
npm notice
npm notice 📦 demo@1.0.1
npm notice === Tarball Contents ===
npm notice 45B bin/index.js
npm notice 0B example/test.js
npm notice 0B index.js
npm notice 304B package.json
npm notice === Tarball Details ===
npm notice name: demo
npm notice version: 1.0.1
npm notice filename: demo-1.0.1.tgz
npm notice package size: 381 B
npm notice unpacked size: 349 B
npm notice shasum: 7497c372c311ad956f162528cb0b0431a95123e7
npm notice integrity: sha512-qajewSNZopKID[...]US/FionrqUJfw==
npm notice total files: 4
npm notice
+ demo@1.0.1
```

6. 执行发布后查看<http://localhost:4873/>



7. 接下来在test项目中执行

```
npm uninstall demo -s
```

8. 在执行

```
npm i demo -s
```

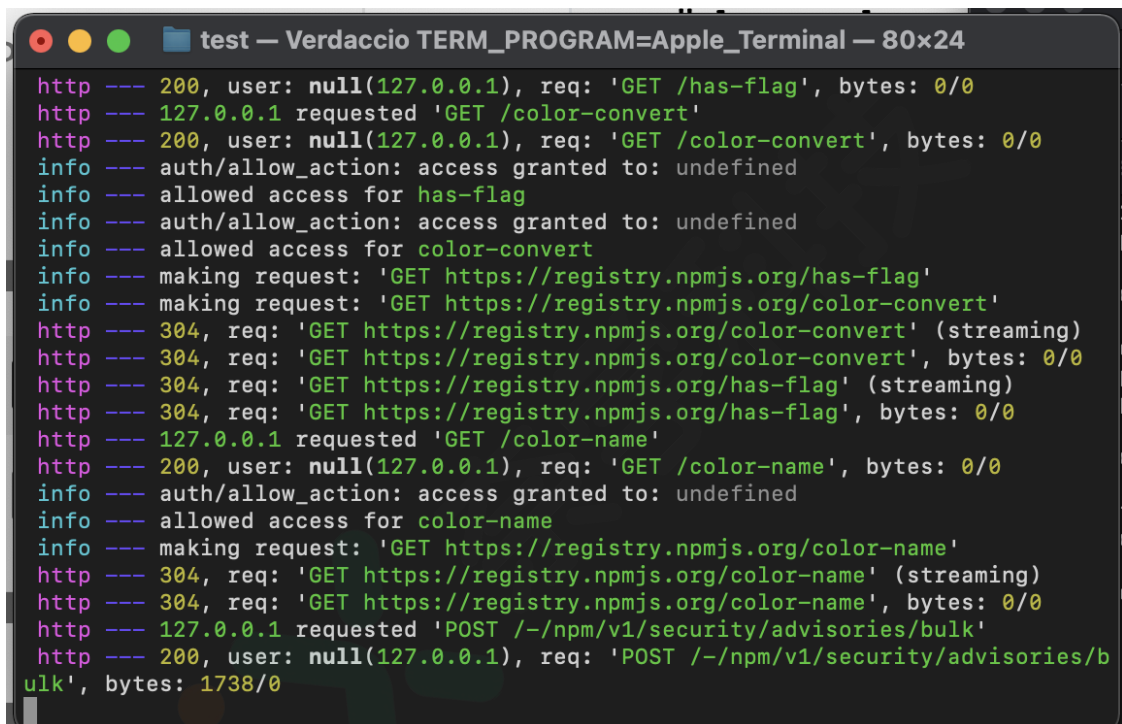
9. 会发现package.json中出现线上下载的版本号

```
{
  "dependencies": {
    "demo": "^1.0.1",
    "vue": "^3.2.33"
  }
}
```

10. 此时代表私服管理依赖的功能实现了
11. 下一步测试一下安装私服中不存在的依赖效果

```
npm i @babel/core -D
```

12. 会发现verdaccio识别到本地不存在该依赖包时会自动联网下载线上的依赖包如图



```
test — Verdaccio TERM_PROGRAM=Apple_Terminal — 80x24
http --- 200, user: null(127.0.0.1), req: 'GET /has-flag', bytes: 0/0
http --- 127.0.0.1 requested 'GET /color-convert'
http --- 200, user: null(127.0.0.1), req: 'GET /color-convert', bytes: 0/0
info --- auth/allow_action: access granted to: undefined
info --- allowed access for has-flag
info --- auth/allow_action: access granted to: undefined
info --- allowed access for color-convert
info --- making request: 'GET https://registry.npmjs.org/has-flag'
info --- making request: 'GET https://registry.npmjs.org/color-convert'
http --- 304, req: 'GET https://registry.npmjs.org/color-convert' (streaming)
http --- 304, req: 'GET https://registry.npmjs.org/color-convert', bytes: 0/0
http --- 304, req: 'GET https://registry.npmjs.org/has-flag' (streaming)
http --- 304, req: 'GET https://registry.npmjs.org/has-flag', bytes: 0/0
http --- 127.0.0.1 requested 'GET /color-name'
http --- 200, user: null(127.0.0.1), req: 'GET /color-name', bytes: 0/0
info --- auth/allow_action: access granted to: undefined
info --- allowed access for color-name
info --- making request: 'GET https://registry.npmjs.org/color-name'
http --- 304, req: 'GET https://registry.npmjs.org/color-name' (streaming)
http --- 304, req: 'GET https://registry.npmjs.org/color-name', bytes: 0/0
http --- 127.0.0.1 requested 'POST /-/npm/v1/security/advisories/bulk'
http --- 200, user: null(127.0.0.1), req: 'POST /-/npm/v1/security/advisories/bulk', bytes: 1738/0
```

13. 关于更多Verdaccio的配置可以访问<https://verdaccio.org/zh-cn/docs/installation/>查看

4. yarn/pnpm包管理器介绍

npm

<https://www.npmjs.com/>

npm 是 Node Package Manager 的缩写，是一个 NodeJS 包管理和分发工具，我们可以使用它发布、安装和卸载 NodeJS 包。npm 是 JavaScript 运行时环境 Node.js 的默认包管理器。

yarn

<https://yarnpkg.com/>

yarn 是 facebook 等公司在 npm v3 时推出的一个新的开源的 Node Package Manager，它的出现是为了弥补 npm 当时安装速度慢、依赖包版本不一致等问题。

yarn 有以下优点：

- 安装速度快

- 并行安装：npm 是按照队列依次安装每个 package，当前一个 package 安装完成之后，才能继续后面的安装。而 Yarn 是同步执行所有任务。
- 缓存：如果一个 package 之前已经安装过，yarn 会直接从缓存中获取，而不是重新下载。
- 版本统一

yarn 创新性的新增了 yarn.lock 文件，它是 yarn 在安装依赖包时，自动生成的一个文件，作用是记录 yarn 安装的每个 package 的版本，保证之后 install 时的版本一致。不过随着后来 npm 也新增了作用相同的 package-lock.json，这个优势已经不太明显。

pnpm

<https://pnpm.io/>

2017 年 pnpm 推出。全称 Performance NPM，即高性能的 npm。相比较于 yarn，pnpm 在性能上又有了极大的提升。

pnpm 的出现解决了 npm、yarn 重复文件过多、复用率低等问题。我们知道，不管是 npm 还是 yarn，它们的安装方法都是将项目依赖包的原封不动的从服务器上下载到本地，写入到 node_modules 文件夹，而每个 package 又都有自己的 node_modules，所以当 package 在不同的依赖项中需要时，它会被多次复制粘贴并生成多份文件，形成一个很深的依赖树。

另外，如果同一个 package 在我们本地的多个项目中使用，每次安装的时候它都会被重新下载一次。比如我们本地有 100 个项目，都依赖 lodash，那么使用 npm 或 yarn 进行安装，lodash 很可能就被下载、安装了 100 次，也就是说我们的磁盘中有 100 个地方写入了 lodash 的代码，这种方式是极其低效的。

pnpm 内部使用基于内容寻址的文件系统来存储磁盘上所有的文件，这个文件系统出色的地方在于：

同一个包 pnpm 只会安装一次，磁盘中只有一个地方写入，后面再次使用都会直接使 hardlink。即使一个包的不同版本，pnpm 也会极大程度地复用之前版本的代码。举个例子，比如 lodash 有 100 个文件，更新版本之后多了一个文件，那么磁盘当中并不会重新写入 101 个文件，而是保留原来的 100 个文件的 hardlink，仅仅写入那一个新增的文件。

npm yarn 和 pnpm 之间命令的区别

-	npm	yarn	pnpm
Install all	npm install	yarn	pnpm install
Install	npm install [package]	yarn add [package]	pnpm add [package]
	npm install [package] -D	yarn add [package] -D	pnpm add -D [package]
	npm install [package] -g	yarn global add [package]	pnpm add -g [package]
Uninstall	npm uninstall [package]	yarn remove [package]	pnpm remove [package]
Update	npm update [package]	yarn upgrade [package]	pnpm update [package]