# 包管理器

**编写npm包的时候，可以办法自动生成changlog与自动更新tag【热度:455】**

**关键词：自动化 changelog**

在编写 npm 包时，可以使用自动化工具来生成 changelog 和自动更新 tag。以下是你可以使用的一些流行的工具以及它们的基本用法。

1. **semantic-release**: 这是一个全自动的版本管理和包发布工具。它能根据 commit 信息来自动决定版本号、生成变更日志（changelog）以及发布。

要使用 semantic-release，你需要按照以下步骤操作：

安装 semantic-release 工具：

npm install -D semantic-release

在项目中添加配置文件 (semantic-release.config.js) 或在 package.json 中配置。

在 CI 工具中（例如 GitHub Actions、Travis CI）配置发布脚本。

遵循规范化的 commit 消息风格（如 Angular 规范），因为 semantic-release 会根据 commit 消息来确定版本号和生成 changelog。

1. **standard-version**: 如果你更希望进行半自动化的版本管理，standard-version 是一个很好的替代选择。它可以自动地根据 commit 记录来生成 changelog。

使用 standard-version 的大致步骤如下：

安装 standard-version 工具：

npm install --save-dev standard-version

在 package.json 中配置脚本：

{

"scripts": {

"release": "standard-version"

}

}

当你准备发布新版本时，运行以下命令：

npm run release

standard-version 会自动根据 commit 消息创建一个新的 tag，并更新 changelog。然后，你可以手动推送这些改动到仓库。

在这两种情况下，都推荐使用遵循某种规范的 commit 消息，如 Conventional Commits 规范，这样可以让工具更准确地解析 commit 消息来进行版本管理。此外，确保你的 CI/CD 系统有足够的权限来推送 tags 到远程仓库。

**对比一下pnpm、npm、yarn特性【热度: 399】**



## pnpm 和 npm 的区别？

pnpm 和 npm 是两个不同的 JavaScript 包管理工具，它们有以下区别：

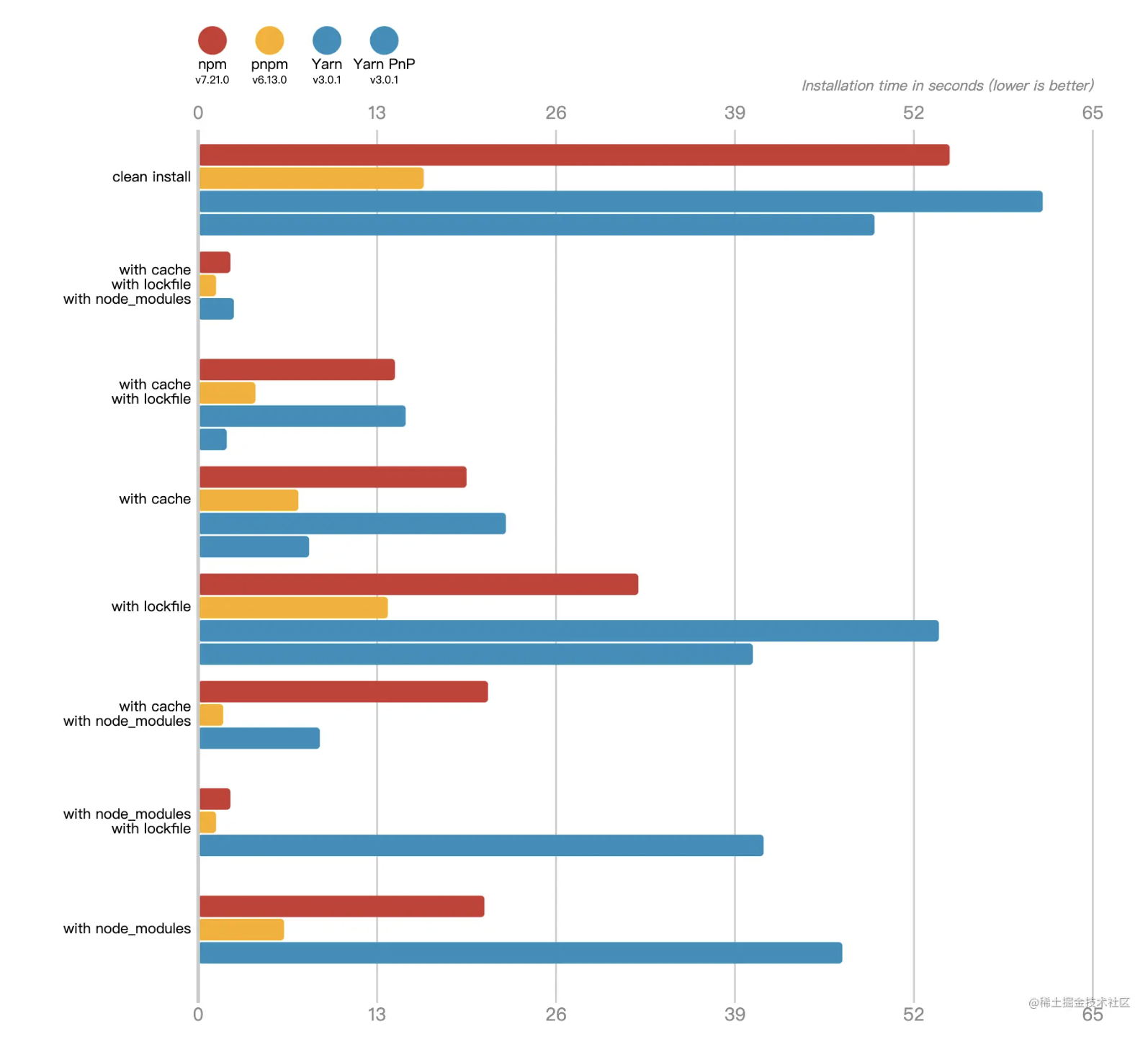
* 包的存储方式：npm 将每个包都下载到项目的 node\_modules 目录中，而 pnpm 会在全局安装一个存储库，并在项目中创建一个符号链接到该存储库中的每个包。
* 空间占用： 由于 pnpm 使用符号链接，它的空间占用通常比 npm 小，因为它避免了在多个项目中重复存储相同的依赖项。
* 安装速度： 由于 pnpm 在全局安装中共享依赖项，因此安装速度通常比 npm 更快。
* 命令行接口：pnpm 的命令行接口与 npm 不同，但它们都提供了一组相似的命令来管理包。
* 兼容性： 由于 pnpm 的存储方式不同于 npm，因此某些与 npm 兼容的工具可能无法与 pnpm 一起使用。

总的来说，pnpm 与 npm 相比具有更小的空间占用和更快的安装速度，但由于其不同的存储方式可能会导致与某些工具的不兼容。

## pnpm 了解多少？

pnpm，英文里面的意思叫做 performant npm ，意味“高性能的 npm”，官网地址可以参考 pnpm.io/。

pnpm 相比较于 yarn/npm 这两个常用的包管理工具在性能上也有了极大的提升，根据目前官方提供的 benchmark 数据可以看出在一些综合场景下比 npm/yarn 快了大概两倍：



[pnpm知多少?](https://github.com/pro-collection/interview-question/issues/140)

## npm script生命周期有哪些?【热度:519】

**关键词：npm 生命周期、script 生命周期**

### 安装和卸载

| 脚本名称 | 阶段 | 描述 | 执行时机 |
| --- | --- | --- | --- |
| preinstall | pre | 在 npm install 执行前运行，用于执行一些安装前的准备工作，例如检查依赖项或设置环境变量。 | 安装前 |
| install, postinstall | install | 在模块安装后执行，通常用于构建项目或者为其生成某些必须的文件，例如安装完毕后自动编译 TypeScript、ES6 等。 | 安装后 |
| preuninstall | pre | 在 npm uninstall 执行前运行，用于执行一些卸载前的准备工作。 | 卸载前 |
| uninstall | post | 在 npm uninstall 执行后运行，用于清理卸载后的一些操作。 | 卸载后 |
| postuninstall | post | 在 npm uninstall 执行后运行，用于执行一些卸载后的操作。 | 卸载后 |

### 发布和更新版本

| 脚本名称 | 阶段 | 描述 | 执行时机 |
| --- | --- | --- | --- |
| prepublish | pre | 在 publish（npm发布）执行前，运行 npm pack。 | 发布前 |
| prepare | pre | 在包被发布前或安装前执行，可以用来设置编译或验证文件的操作。 | 发布前、安装前 |
| prepublishOnly | pre | 在 npm publish 执行前运行，用于确保在 publish 命令执行时不会意外发布不必要的文件。 | 发布前 |
| prepack | pre | 在 npm pack（打包命令）执行前运行，用于执行一些打包前的准备工作。 | 打包前 |
| postpack | post | 在 npm pack 执行后运行，用于清理和重置打包相关的操作。 | 打包后 |
| publish | post | 在包被成功发布后执行。 | 发布后 |
| postpublish | post | 在包被成功发布后执行，用于执行一些发布后的操作。 | 发布后 |
| preversion | pre | 在项目版本号更新（npm version）之前执行。 | 更新版本号前 |
| version | post | 在 npm version 执行后执行，用于执行一些版本更新后的操作。 | 更新版本号后 |
| postversion | post | 在项目版本号更新（npm version）之后执行。 | 更新版本号后 |

### 测试和运行

| 脚本名称 | 阶段 | 描述 | 执行时机 |
| --- | --- | --- | --- |
| pretest | pre | 在 npm test 执行前执行，用于执行某些测试前的准备工作。 | 测试前 |
| test | test | 执行 npm run test 命令时执行。通常用于执行单元测试，并返回任何错误状态。 | 默认测试阶段 |
| posttest | post | 在 npm test 执行后执行，用于执行某些测试后的操作。 | 测试后 |
| prestart | pre | 在 npm start 执行前运行，用于执行某些启动进程前的准备工作。 | 启动前 |
| start | start | 执行 npm start 命令时执行，通常用于启动 Web 服务器、Node 服务器、实时编译器等。 | 默认启动阶段 |
| poststart | post | 在 npm start 执行后执行，用于执行某些启动进程后的操作。 | 启动后 |
| prerestart | pre | 在 npm restart 执行前执行，用于执行一些重新启动进程前的准备工作。 | 重新启动前 |
| restart | stop/start | 执行 npm restart 命令时执行，通常用于停止正在运行的 Node 服务器、Web 服务器等，然后以更新的源码重新启动服务。 | 默认重新启动阶段，但是该命令会触发停止和启动两个标准阶段 |
| postrestart | post | 在 npm restart 执行后执行，用于执行一些重新启动进程后的操作。 | 重新启动后 |

### 其他生命周期

| 脚本名称 | 阶段 | 描述 | 执行时机 |
| --- | --- | --- | --- |
| prestop | pre | 在 npm stop 执行前运行，用于执行某些停止进程前的准备工作。 | 停止前 |
| stop | stop | 执行 npm stop 命令时执行，通常用于停止正在运行的 Web 服务器、Node 服务器、实时编译器等。 | 默认停止阶段 |
| poststop | post | 在 npm stop 执行后执行，用于执行某些停止进程后的操作。 | 停止后 |

## npm包管理了解多少?【热度: 1,321】

关键词：npm 依赖管理、npm 包管理、npm 缓存

### npm 是如何进行依赖管理的？

npm 是通过 package.json 文件来进行依赖管理的。当在项目中使用第三方库时，我们可以在 package.json 中添加对应的依赖项及版本号，npm 会根据 package.json 中的依赖关系，自动安装相应的依赖包及其依赖项。当我们执行 npm install 命令时，npm 会自动根据 package.json 中的依赖信息进行依赖包的安装。

npm 的依赖管理还涉及到依赖的版本控制，可以在 package.json 中指定对应的版本号，常见的版本号控制符号有：

* ^（caret）：匹配到次要版本号（第二个数字）的最新版本。例如，^1.2.3 表示安装 1.2.x 的最新版本（除了 1.3.0）。
* ~（tilde）：匹配到修订版本号（第三个数字）的最新版本。例如，~1.2.3 表示安装 1.2.3 到 1.2.x 的最新版本（除了 1.3.0）。
* \*：匹配到最新的版本。
* >=：匹配到大于或等于指定版本的最新版本。
* <、<=、>：匹配到小于、小于等于或大于指定版本的最新版本。

在 npm 的依赖管理中，还有两种类型的依赖关系：生产依赖和开发依赖。生产依赖是指在应用程序运行时必须要加载的依赖，开发依赖是指在应用程序开发过程中使用的依赖。在 package.json 中，生产依赖使用 dependencies 字段，开发依赖使用 devDependencies 字段。这样可以让项目更加清晰地管理其依赖关系。

### npm 有缓存包的能力吗？

npm有缓存包的能力。当你第一次使用npm安装一个包时，npm会自动将该包缓存在本地。这样，当你下次需要安装相同版本的该包时，npm就不必重新从网络上下载该包，而是直接使用缓存中的包。这样可以提高包的下载速度，节省网络带宽。

npm的缓存位于本地文件系统中的一个隐藏目录。默认情况下，缓存位于当前用户的主目录下的.npm目录中。你可以使用以下命令查看npm缓存的路径：

npm config get cache

你也可以通过npm cache命令来管理npm缓存，例如清空缓存：

npm cache clean

或者查看缓存的统计信息：

npm cache ls

### npm 是如何使用缓存中的包的？

使用缓存中的包可以通过以下两种方式实现：

* 使用 npm ci 命令

npm ci 命令会首先检查 package-lock.json 或 npm-shrinkwrap.json 文件，以确保安装的依赖与锁定的版本一致。然后，它会在 node\_modules 目录下安装依赖，如果缓存中存在符合要求的包，npm ci 会直接从缓存中复制到 node\_modules 目录下，而不需要重新下载和编译。

* 手动指定缓存路径

如果需要手动使用缓存中的包，可以在 npm install 命令中指定缓存路径，例如：

npm install --cache /path/to/npm-cache

然后，执行 npm install 命令时，npm 会尝试从指定的缓存路径中获取包，如果找到匹配的包，就会直接复制到 node\_modules 目录下。

需要注意的是，手动指定缓存路径的方式可能会导致不同的项目之间共用缓存，因此需要确保缓存路径的唯一性。

## npm workspaces 是什么概念， 主要是解决什么问题【热度: 100】

关键词：npm workspaces 概念

npm workspaces是一个在 npm 中用于管理多包项目的功能。它主要解决以下几个问题：

### 一、项目结构管理

#### 多包项目组织：

在大型项目中，可能包含多个独立的软件包或模块。npm workspaces允许将这些包组织在一个统一的项目结构中，方便管理和开发。

例如，一个企业级应用可能由一个前端库、一个后端服务和一个共享的工具包组成。使用npm workspaces，可以将这些包放在同一个项目目录下，而不是作为独立的项目进行管理。

#### 依赖共享：

多个包之间可能存在共享的依赖项。npm workspaces可以自动管理这些共享依赖，避免重复安装和版本冲突。

当一个包安装了某个依赖项时，其他包可以直接使用这个依赖，而不需要再次安装。这可以减少项目的体积和安装时间，提高开发效率。

### 二、开发效率提升

#### 单一命令行界面：

使用npm workspaces，可以在项目的根目录下使用单一的npm命令来管理所有的包。这避免了在每个包的目录下分别运行npm命令的繁琐过程。

例如，可以在根目录下运行npm install来安装所有包的依赖项，或者运行npm run test来执行所有包的测试。

#### 同时开发多个包：

开发人员可以在同一个项目中同时开发多个包，而不需要在不同的项目目录之间切换。这可以提高开发效率，特别是在需要频繁修改多个包的情况下。

例如，可以在一个编辑器中打开多个包的代码，进行同时编辑和调试。

### 三、版本管理和发布

#### 统一版本控制：

在多包项目中，需要确保各个包的版本保持一致。npm workspaces可以帮助管理包的版本，确保在发布时所有的包都使用相同的版本号。

可以在项目的根目录下使用npm version命令来统一更新所有包的版本号，并生成相应的版本标签。

#### 简化发布流程：

发布多个包时，通常需要分别在每个包的目录下执行发布命令。使用npm workspaces，可以在项目的根目录下使用单一的发布命令来发布所有的包。

例如，可以使用npm publish --workspaces来发布项目中的所有包，而不需要分别进入每个包的目录进行发布。

## pnpm install 和 npm install 有何区别

pnpm install 和 npm install 都是用于安装 JavaScript 项目依赖的命令，但它们背后的包管理器（分别是 pnpm 和 npm）在处理依赖安装、存储和优化方面有一些关键区别。

### 1. 存储方式的区别

npm：在每个项目的 node\_modules 文件夹中分别存储其依赖。这意味着如果你有多个项目，它们共享相同的依赖库，这些依赖库的多个副本将在你的文件系统中的每个项目内分别存储。这样做会占用更多的磁盘空间。

pnpm：采用一种称为内容寻址文件系统的方式来存储依赖。所有项目的依赖被存储在一个共享的位置，各个项目中的 node\_modules 目录通过硬链接（hard links）或符号链接（symlinks）指向这个共享位置。该方法有效地减少了磁盘空间的占用，并加快了依赖的安装速度。

### 2. 性能与速度

pnpm：由于对依赖进行了有效的复用，并且使用硬链接来减少磁盘上的副本数量，通常可以提供比 npm 更快的安装速度。

npm：近几个大版本中也进行了许多性能改进，但在多个项目中共享相同依赖时，它可能仍然比 pnpm 更慢，尤其是在首次安装依赖时。

### 3. 依赖平面结构 vs. 嵌套结构

npm：自版本 3 以后，默认创建扁平的 node\_modules 结构（尽可能），这样做是为了避免 Windows 系统中路径过长的问题。但在必要时， npm 仍然会创建嵌套的 node\_modules 目录结构，以解决依赖冲突。

pnpm：通过使用符号链接，pnpm 维护了一个严格的嵌套依赖结构，更接近每个包的 package.json 文件所声明的依赖树形态。这提供了更高的一致性和在某些情况下更好的包隔离性。

### 4. 依赖隔离与安全性

pnpm：更好地隔离了依赖，每个包只能访问其在 package.json 中声明的依赖。这一特性增强了项目的安全性，因为它阻止了未声明的依赖被意外引入的情况。

npm：虽然 npm 也遵循 package.json 中的声明，但其扁平化的 node\_modules 结构有时可能会容易地让包访问到未明确声明的依赖。

### 5. 命令行界面（CLI）和配置

这两个工具的命令行界面（CLI）和配置都非常直观且类似，但它们的某些命令和选项可能会有细微差别。pnpm 为了优化性能和安全性引入了一些特有的命令和配置选项。

总的来说，pnpm 在多项目管理、磁盘空间和安装速度方面提供了优于 npm 的性能和利益。对于新项目或在寻求性能优化的既存项目，考虑试用 pnpm 可能是个不错的选择。

npx了解多少?【热度:290】

npm lock文件了解多少?【热度:258】

npm script了解多少?【热度: 364】