yarn官方中文文档

书栈(BookStack.CN)

目　录

致谢

当前文档 《yarn官方中文文档》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-12-09。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/yarn-cn

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Yarn 对你的代码来说是一个包管理器， 你可以通过它使用全世界开发者的代码，或者分享自己的代码。 Yarn 做这些快捷、安全、可靠，所以你不用担心什么。

通过Yarn你可以使用其他开发者针对不同问题的解决方案，使自己的开发过程更简单。 使用过程中遇到问题，你可以将其上报或者贡献解决方案。一旦问题被修复，Yarn会更新保持同步。

代码通过包（**package**）（或者称为模块（**module**））的方式来共享。 一个包里包含所有需要共享的代码，以及描述包信息的文件，称为 `package.json` 。

> 原文: *https://yarnpkg.com/zh-Hans/docs/getting-started*

在你使用 Yarn 之前，需要先在系统中安装 Yarn 。有越来越多的各种的方法安装 Yarn:

macOS

Homebrew

你可以通过 Homebrew 包管理器安装 Yarn，如果没有安装 Node.js 它也可以安装。

```
1. brew install yarn
```

如果您使用 nvm 或类似的东西，您应该排除安装 Node.js 以便使用 nvm 的 Node.js 版本。

```
1. brew install yarn --without-node
```

MacPorts

您可以通过 MacPorts 安装 Yarn。 如果您未安装 Node.js，这也为您安装它。

```
1. sudo port install yarn
```

路径设置

如果您选择手动安装，用以下步骤添加 Yarn 到 path 变量，使其可以随处运行。

注意：您的配置文件可能是您的 `.profile` 、 `.bash_profile` 、 `.bashrc` 、 `.zshrc` 等。

- 将此项加入您的配置文件： `export PATH="$PATH:/opt/yarn-[version]/bin"` （路径可能根据您安装 Yarn 的位置而有差异）
- 在终端中，登录并登出以使更改生效
  为全局访问 Yarn 的可执行文件，您需要在您的终端中设置 `PATH` 环境变量。 若要执行此操作，请添加 `export PATH="$PATH:yarn global bin"` 到您的配置文件。

升级 Yarn

如果有新版本可用，Yarn 将会提醒。 若要升级 Yarn，您可以使用 Homebrew 来完成。

```
1. brew upgrade yarn
```

Windows

在 Windows 系统中安装 Yarn 有三种方法。

下载安装程序

这将给你一个 `.msi` 文件，当你运行它时带领你安装 Yarn 到 Windows 上。

如果你使用安装程序，你需要先安装 Node.js。

下载安装程序

用 Chocolatey 安装

Chocolatey 是 Windows 上的包管理器。可以用下面这些命令安装 Chocolatey 。

如果已经安装了 Chocolatey，则可以在控制台中运行下面的命令安装 yarn:

```
1. choco install yarn
```

这将确保你安装了 Node.js。

通过 Scoop 安装

Scoop 是一个 Windows 的命令行安装程序，你可以用下面这些指令安装 Scoop。

一旦安装了 Scoop，你可以在控制台里运行下面的代码安装 yarn:

```
1. scoop install yarn
```

如果 Node.js 没有安装，scoop 将给你一个建议来安装它。 例如:

```
1. scoop install nodejs
```

提示

请把你的项目目录和 Yarn 缓存目录 (%LocalAppData%\Yarn) 加到你的杀毒软件白名单里，否则安装包时会明显变慢，每个写入到硬盘时将被扫描。

备选

如果您使用其他操作系统，或者适用您系统的其他方式不能用，这里还有一些备选方案。 如果还没有的话，你需要安装 Node.js。

在常见的 Linux 系统上，比如说 Debian、Ubuntu 和 CentOS，推荐通过我们的包来安装 Yarn。

安装脚本

在 macOS 和通用 Unix 环境里安装 Yarn 的最容易方法之一是通过我们的 shell 脚本。你可以在你的终端里运行下列代码来安装 Yarn:

```
1. curl -o- -L https://yarnpkg.com/install.sh | bash
```

安装过程包括验证 GPG 签名。 在 GitHub 上查看代码

您也可以通过在您的终端中运行下面的代码指定版本：

```
curl -o- -L https://yarnpkg.com/install.sh | bash -s -- --version [version]
```

有哪些可能的版本请参阅 版本。

通过源码包手动安装

你可以下载源码包并解压到任意位置来安装 Yarn。

```
cd /opt
wget https://yarnpkg.com/latest.tar.gz
tar zvxf latest.tar.gz
# Yarn 现在在 /opt/yarn-[version]/ 里面
```

在提取 Yarn 前，推荐使用 GPG 验证 tar 包：

```
wget -qO- https://dl.yarnpkg.com/debian/pubkey.gpg | gpg --import
wget https://yarnpkg.com/latest.tar.gz.asc
gpg --verify latest.tar.gz.asc
# 在输出中看到 "Good signature from 'Yarn Packaging'"
```

通过 npm 安装

> 注意：一般来说，不推荐通过 *npm* 安装 *Yarn* 在用基于 *Node* 的包管理器安装 *Yarn* 时，该包未被签名， 并且只通过基本的 *SHA1* 散列进行唯一完整性检查。 这在安装系统级应用时有安全风险。 因为这些原因，高度推荐用你的操作系统最适合的方式来安装 *Yarn*。

如果有，你还可以通过 npm package manager 来安装 Yarn。 如果你已经装了Node.js，那你应该已经有 npm 了。

安装好 npm 后你可以用：

```
npm install --global yarn
```

路径设置

Unix/Linux/macOS

如果您选择手动安装，用以下步骤添加 Yarn 到 path 变量，使其可以随处运行。

注意：您的配置文件可能是您的 `.profile` 、 `.bash_profile` 、 `.bashrc` 、 `.zshrc` 等。

- 将此项加入您的配置文件： `export PATH="$PATH:/opt/yarn-[version]/bin"` （路径可能根据您安装 Yarn 的位置而有差异）
- 在终端中，登录并登出以使更改生效
  为全局访问 Yarn 的可执行文件，您需要在您的终端中设置 `PATH` 环境变量。 若要执行此操作，请添加 `export PATH="$PATH:yarn global bin"` 到您的配置文件。

Windows

你需要在终端里设置 `PATH` 环境变量来全局访问 Yarn 的二进制可执行程序。

添加 `set PATH=%PATH%;C:.yarn\bin` 到你的 shell 环境。

运行命令来测试 Yarn 是否安装：

```
1. yarn --version
```

> 有问题吗？ 如果你不能用这些安装程序安装 *Yarn*，请通过 *GitHub* 搜索一个 *issue* 或者开一个新的 *issue*。 搜索现有的 *issue* · 开一个新的 *issue*

> 原文：*https://yarnpkg.com/zh-Hans/docs/install*

现在Yarn已经 安装完毕，可以开始使用。以下是一些你需要的最常用的命令：

初始化新项目

```
1. yarn init
```

添加依赖包

```
1. yarn add [package]
2. yarn add [package]@[version]
3. yarn add [package]@[tag]
```

将依赖项添加到不同依赖项类别

分别添加到 `devDependencies` 、 `peerDependencies` 和 `optionalDependencies` ：

```
1. yarn add [package] --dev
2. yarn add [package] --peer
3. yarn add [package] --optional
```

升级依赖包

```
1. yarn upgrade [package]
2. yarn upgrade [package]@[version]
3. yarn upgrade [package]@[tag]
```

移除依赖包

```
1. yarn remove [package]
```

安装项目的全部依赖

```
1. yarn
```

或者

```
1. yarn install
```

> 原文: *https://yarnpkg.com/zh-Hans/docs/usage*

你的项目在引入了包管理器的同时，也引入了一套新的围绕着依赖项开发的工作流程。Yarn尽力不改变你的工作流程，并使流程中的每一步都简单明了。

关于基本工作流你应该知道几个简单的事：

- 创建一个新项目
- 增加 / 更新 / 删除依赖
- 安装 / 重装你的依赖
- 引入版本控制系统（例如 git）
- 持续集成

> 原文：*https://yarnpkg.com/zh-Hans/docs/yarn-workflow*

不论是已经有了现成的代码仓库（目录），还是正着手启动一个全新项目，你都可以使用同样的方法引入Yarn。

在命令行终端里，跳转到准备引入Yarn的目录（通常是一个项目的根目录），执行以下命令：

```
yarn init
```

这将打开一个用于创建Yarn项目的交互式表单，其中包含以下问题：

```
name (your-project):
version (1.0.0):
description:
entry point (index.js):
git repository:
author:
license (MIT):
```

你既可以回答这些问题，也可以直接敲回车键（enter/return）使用默认配置或者留空。

package.json

现在应该创建了一个和下面文件内容类似的 `package.json` ：

```
{
  "name": "my-new-project",
  "version": "1.0.0",
  "description": "My New Project description.",
  "main": "index.js",
  "repository": {
    "url": "https://example.com/your-username/my-new-project",
    "type": "git"
  },
  "author": "Your Name <you@example.com>",
  "license": "MIT"
}
```

执行 `yarn init` 之后，除了以上文件被创建之外，没有任何副作用。你可以随意编辑此文件。

`package.json` 文件里存储了项目的有关信息。 包括项目名称、维护者信息、代码托管地址，以及最重要的：项目依赖。

> 原文：*https://yarnpkg.com/zh-Hans/docs/creating-a-project*

你需要了解几若干个用于增加、更新、删除依赖项的命令。

每个命令都会更新 `package.json` 和 `yarn.lock` 文件。

添加依赖包

在使用一个包之前，你需要执行以下命令将其加入依赖项列表：

```
1. yarn add [package]
```

`[package]` 会被加入到 `package.json` 文件中的依赖列表，同时 `yarn.lock` 也会被更新。

```
1.   {
2.     "name": "my-package",
3.     "dependencies": {
4. +     "package-1": "^1.0.0"
5.     }
6.   }
```

你可以用以下参数添加其它类型的依赖：

- `yarn add –dev` 添加到 `devDependencies`
- `yarn add –peer` 添加到 `peerDependencies`
- `yarn add –optional` 添加到 `optionalDependencies`
  通过指定依赖版本和标签，你可以安装一个特定版本的包：

```
1. yarn add [package]@[version]
2. yarn add [package]@[tag]
```

`[version]` 或 `[tag]` 会被添加到 `package.json`，并在安装依赖时被解析。

例如：

```
1. yarn add package-1@1.2.3
2. yarn add package-2@^1.0.0
3. yarn add package-3@beta
```

```
1. {
2.   "dependencies": {
3.     "package-1": "1.2.3",
4.     "package-2": "^1.0.0",
5.     "package-3": "beta"
6.   }
7. }
```

更新依赖包

```
yarn upgrade [package]
yarn upgrade [package]@[version]
yarn upgrade [package]@[tag]
```

这会更新 `package.json` 和 `yarn.lock` 文件。

```
  {
    "name": "my-package",
    "dependencies": {
-     "package-1": "^1.0.0"
+     "package-1": "^2.0.0"
    }
  }
```

删除依赖包

```
yarn remove [package]
```

这会更新 `package.json` 和 `yarn.lock` 文件。

> 原文: *https://yarnpkg.com/zh-Hans/docs/managing-dependencies*

如果刚从版本控制系统里 checkout 一个包，则需要为其安装依赖。

> 如果是为现有的包增加依赖，那么这些新的依赖会自动安装。

安装依赖项

`yarn install` 是用于安装一个项目的所有依赖。 Yarn会从 `package.json` 中读取依赖，并将依赖信息存储到 `yarn.lock` 中。

如果你正在开发一个包，通常你会在以下情况之后进行依赖安装：

- 你刚检出需要这些依赖项的项目代码。
- 项目的另一个开发者添加了新的依赖，你需要用到。

安装选项

有很多参数可以控制依赖安装的过程，包括：

- 安装所有依赖：`yarn` 或 `yarn install`
- 安装一个包的单一版本：`yarn install --flat`
- 强制重新下载所有包：`yarn install --force`
- 只安装生产环境依赖：`yarn install --production`
  查看您可以传递给 `yarn install` 的 完整参数列表。

> 原文：*https://yarnpkg.com/zh-Hans/docs/installing-dependencies*

为了使其他人能够使用你的包，或者能够对其进行后续开发，你需要确保将所有必须的文件提交到你所使用的版本控制系统。

所需的文件

为了别人能使用你的包，以下文件必须被提交进版本控制系统：

- `package.json`：包含包的所有依赖信息；
- `yarn.lock`：记录每一个依赖项的确切版本信息；
- 包实现功能的实际项目代码。

> 请参阅*Yarn Example Package*项目，查看一个可用的*Yarn*包所需的最少文件配置。

> 原文：*https://yarnpkg.com/zh-Hans/docs/version-control*

Yarn 很容易在许多持续构建系统中使用。为了加速构建，Yarn 缓存目录可以跨构建保存起来。

AppVeyorCircleCICodeshipTravisSemaphoreSolano

从上面的选项中选择您正在使用持续集成系统

Yarn已预先安装在 AppVeyor 上，所以不需要在构建流程中做别的事情。

要让 build 更快，你可以把以下配置加到 `appveyor.yml` ，这会缓存 Yarn 的 缓存文件夹。

```
cache:
  - "%LOCALAPPDATA%\\Yarn"
```

CircleCI 为 Yarn 提供了文档。 你可以用他们的 Yarn 文档来开始运行。

Yarn 已在 Codeship Basic 上预装。

如果您正在使用 Codeship Pro (基于 Docker)，推荐您通过我们的 Debian/Ubuntu 包安装 Yarn。

Travis CI 根据项目根目录里面是否有 `yarn.lock` 文件检测是否使用 Yarn。 如果文件可用，Travis CI 会根据需要来安装 `yarn` ，并执行 `yarn` 作为默认的安装命令。

如果你的安装流程需要更多，要自己安装 Yarn，确保它在 build 镜像里已经预先安装好。

安装 Yarn 有两种方式：用 `sudo` ，或者不用。 如果你用的是基于容器的环境的话， 用第二种方式。

开启sudo的构建

```
sudo: required
before_install: # if "install" is overridden
  # Repo for Yarn
  - sudo apt-key adv --fetch-keys http://dl.yarnpkg.com/debian/pubkey.gpg
  - echo "deb http://dl.yarnpkg.com/debian/ stable main" | sudo tee /etc/apt/sources.list.d/yarn.list
  - sudo apt-get update -qq
  - sudo apt-get install -y -qq yarn
cache:
  yarn: true
```

建议您锁定使用特定 Yarn 版本，使每次构建都是使用同一版本的 Yarn。在切换之前，你也可以先测试新版本的 Yarn。 您可以在调用 `apt-get install` 命令时添加版本号：

```
sudo apt-get install -y -qq yarn=1.12.3-1
```

基于容器的构建

基于容器的构建没有 `sudo` 权限，必须通过其他方式安装。 比如：

```
sudo: false
```

```
before_install:
  - curl -o- -L https://yarnpkg.com/install.sh | bash -s -- --version 1.12.3
  - export PATH=$HOME/.yarn/bin:$PATH
cache:
  yarn: true
```

Semaphore 为所有受支持的 Node.js 版本预装了 Yarn，并且用户直接能使用已备好的 Yarn 缓存。

为确保本地 Yarn 版本与 Semaphore 上的一致，您可以在项目设置中加入以下几行到你的设置命令：

```
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -
echo "deb http://dl.yarnpkg.com/debian/ stable main" | sudo tee /etc/apt/sources.list.d/yarn.list
# install-package 是在 Semaphore 里缓存 APT 安装程序的工具
# 包版本号可以也可以不定义
install-package yarn=<version>
```

Yarn 已在 SolanoCI 上预装。 你可以跟着他们的 Yarn 文档快速启动和运行。 作为一个配置文件例子，取出一个他们的样例配置文件。

> 原文: *https://yarnpkg.com/zh-Hans/docs/install-ci*

Yarn提供了丰富的命令使你可以对Yarn包进行许多操作，包括安装、管理、发布等。

所有可用的命令都按照字母先后顺序列在此处，其中最常用的有：

- `yarn add`：为当前正在开发的包新增一个依赖包；
- `yarn init`：初始化包；
- `yarn install`：安装`package.json`文件里定义的所有依赖包；
- `yarn publish`：发布一个包到包管理器；
- `yarn remove`：从当前包里移除一个未使用的包。

默认命令

执行不带任何命令的`yarn`，等同于执行`yarn install`，并透传所有参数。

用户自定义脚本

执行`yarn <script> [<args>]`将会执行用户自定义`脚本`。参阅`yarn run`。

本地安装的 CLI{#locally-installed-clis.toc}

执行`yarn <command> [<args>]`将会执行当前包内安装过的（Local，而非Global）对应名称的命令，这样就可以不必为了一些简单的场景而专门去配置自定义脚本。

并发和 –mutex

当在同一个服务器上同时运行多个 yarn 实例时，你可以通过传递全局标志`–mutex`并跟一个`file`或`network`参数，确保任意给定时间只有一个实例运行（并且避免冲突）。

当使用`file`时 Yarn 默认会写/读当前工作目录里一个互斥锁文件`.yarn-single-instance`。你也可以指定一个备用或全局的文件名。

```
1. --mutex file
2. --mutex file:/tmp/.yarn-mutex
```

当使用`network`时，Yarn 默认会在`31997`端口创建一个服务器，你也可以指定一个备用端口。

```
1. --mutex network
2. --mutex network:30330
```

Verbose output with –verbose

运行`yarn <command> –verbose`会打印执行(创建目录、复制文件或 HTTP 请求等) 的详细信息。

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/*

安装一个包（以及任何它依赖的包）。

添加依赖包

一般而言，一个包是一个包含代码的文件夹和一个描述包内容的 `package.json` 文件。 如果你想使用其他包，首先要将其加入依赖列表。 也就是执行 `yarn add [package-name]` 命令，来为项目安装所需的包。

同时，这个命令会更新 `package.json` 和 `yarn.lock` 文件，以便使本项目的其他开发者可以使用 `yarn` 或者 `yarn install` 来安装相同的依赖。

大多数的包会从npm registry目录里以包名来安装。 例如， `yarn add react` 会从npm registry里安装 `react` 包。

你可以用以下方法指定版本号：

- `yarn add package-name` 会安装 latest 最新版本。
- `yarn add package-name@1.2.3` 会从 registry 里安装这个包的指定版本。
- `yarn add package-name@tag` 会安装某个 “tag” 标识的版本（比如 `beta`、`next` 或者 `latest`）。你也可以指定不同路径的包：
- `yarn add package-name` 从 npm registry 里安装包，除非你在 `package.json` 指定了其它 registry。
- `yarn add file:/path/to/local/folder` 从本地文件系统里安装一个包，可以用这种方式来测试还未发布的包。
- `yarn add file:/path/to/local/tarball.tgz` 安装一个 gzipped 压缩包，此格式可以用于在发布之前分享你的包。
- `yarn add <git remote url>` 从远程 git repo 里安装一个包。
- `yarn add <git remote url>#<branch/commit/tag>` 从一个远程 git 仓库指定的 git 分支、git 提交记录或 git 标签安装一个包。
- `yarn add https://my-project.org/package.tgz` 用一个远程 gzipped 压缩包来安装。

注意事项

如果你以前用过类似于 npm 的包管理器，你可能会想如何全局安装依赖。

对于绝大部分包来说，这是个坏习惯，因为它们是隐藏的。 最好本地安装你的依赖，这样它们都是明确的，每用你项目的人都能得到同样的依赖。

如果你想用有 `bin` 的命令行 CLI 工具，可以在 `./node_modules/.bin` 路径里访问。 你也可以用 `global` 命令：

```
1. yarn global add <package...>
```

命令

yarn add <package…>

这将安装您的 `dependencies` 中的一个或多个包。

`yarn add <package…> [–dev/-D]`

用 `–dev` 或 `-D` 会在 `devDependencies` 里安装一个或多个包。

`yarn add <package…> [–peer/-P]`

用 `–peer` 或者 `-P` 会在 `peerDependencies` 里安装一个或多个包。

`yarn add <package…> [–optional/-O]`

用 `–optional` 或者 `-O` 会在 `optionalDependencies` 里安装一个或多个包。

`yarn add <package…> [–exact/-E]`

用 `–exact` 或者 `-E` 会安装包的精确版本。 默认是安装包的主要版本里的最新版本。 比如说， `yarn add foo@1.2.3` 会接受 `1.9.1` 版，但是 `yarn add foo@1.2.3 –exact` 只会接受 `1.2.3` 版。

`yarn add <package…> [–tilde/-T]`

用 `–tilde` 或者 `-T` 来安装包的次要版本里的最新版。 默认是安装包的主要版本里的最新版本。 比如说， `yarn add foo@1.2.3 –tilde` 会接受 `1.2.9` ，但不接受 `1.3.0` 。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/add*

从包依赖里清除并移除不需要的文件。

yarn autoclean [-I/—init] [-F/—force]

autoclean 命令通过从依赖文件中移除不需要的文件和文件夹来释放空间 它减少了你的项目 `node_modules` 里的文件，在包直接签入到版本控制系统的环境里很有用。

注意：此命令只被用于高级用例。 除非你遇到了 `node_modules` 里安装的文件数问题，不推荐使用此命令。 这个命令会永久删除在 `node_modules` 里的那些会造成包停止工作的文件

autoclean 默认被禁用。 若要启动，手动创建一个 `.yarnclean` 文件，或者运行 `yarn autoclean —init` 来创建默认内容的文件。 `.yarnclean` 文件应该被加入到版本控制。

当一个包中存在 `.yarnclean` 文件，autoclean 功能就会启用。清除执行时机为：

- 在 `install` 后
- 在 `add` 后
- 如果运行了 `yarn autoclean —force`
  清除通过读取 `.yarnclean` 文件中的每一行执行，每行作为一个要删除文件的 glob 模式。

选项：

`-I/—init` ： 创建 `.yarnclean` 文件（如果尚不存在）并添加默认记录。 This file should then be reviewed and edited to customize which files will be cleaned. If the file already exists, it will not be overwritten.

`-F/—force` ： 如果 `.yarnclean` 文件存在，运行其清除流程。如果该文件不存在，不做任何事。

Defaults:

When the `yarn autoclean —init` command is used to create a `.yarnclean` file, it will be pre-populated with a set of default items for deletion. This default list is a guess at what is likely not needed. It is impossible to predict all directories and files that are actually unnecessary for all existing and future NPM packages, so this default list may cause a package to no longer work.

It is **highly recommended** that you manually review the default entries in `.yarnclean` and customize them to fit your needs.

如果你发现autoclean过程会删除影响正常工作的包，你应该移除 `.yarnclean` 文件里相关的入口。

示例：

You decide all YAML and Markdown files in all your dependencies installed in `node_modules` can be safely deleted. You make a `.yarnclean` file containing:

```
1. *.yaml
2. *.md
```

然后再运行 `yarn install` 或 `yarn autoclean —force` 。 The clean process will delete all `.yaml` *and* `.md` files within `node_modules/` recursively (including nested

transitive dependencies).

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/autoclean*

显示 yarn `bin` 目录的位置。

```
yarn bin
```

`yarn bin` 将打印 yarn 将把你的包里可执行文件安装到的目录。 一个可执行文件的例子也许是一个你定义在你的包里的，可以通过 `yarn run` 可执行脚本。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/bin*

`yarn cache list [–pattern]`

Yarn 将每个包存储在你的文件系统-用户目录-全局缓存中。 `yarn cache list` 将列出已缓存的每个包。

`yarn cache list –pattern <pattern>` 将列出匹配指定模式的已缓存的包。

示例:

```
1. yarn cache list --pattern gulp
2. yarn cache list --pattern "gulp|grunt"
3. yarn cache list --pattern "gulp-(match|newer)"
```

`yarn cache dir`

运行 `yarn cache dir` 会打印出当前的 yarn 全局缓存在哪里。

`yarn cache clean [<module_name…>]`

运行此命令将清除全局缓存。 将在下次运行 `yarn` 或 `yarn install` 时重新填充。 Additionally, you can specify one or more packages that you want to clean.

改变 yarn 缓存路径

设置 `cache-folder` 来配置缓存目录。

```
1. yarn config set cache-folder <path>
```

你也可以用 `–cache-folder` 标志指定缓存目录:

```
1. yarn <command> --cache-folder <path>
```

你还可以通过环境变量 `YARN_CACHE_FOLDER` 指定缓存目录：

```
1. YARN_CACHE_FOLDER=<path> yarn <command>
```

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/cache*

yarn check

验证当前项目 `package.json` 里的依赖版本和 yarn 的 lock 文件是否匹配。

yarn check –integrity

验证当前项目 `package.json` 里包内容的版本和 hash 值是否与 yarn 的 lock 文件一致。 这有助于验证包依赖没有更改。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/check*

管理 yarn 配置文件。

yarn config set <key> <value> [-g|—global]

设置配置项 `key` 为一个确切值 `value` 。

示例：

```
$ yarn config set init-license BSD-2-Clause
yarn config vx.x.x
success Set "init-license" to "BSD-2-Clause".
  Done in 0.05s.
```

yarn config get <key>

回显给定 `key` 的值到 `stdout` 。

示例：

```
$ yarn config get init-license
BSD-2-Clause
```

yarn config delete <key>

从配置里删除指定 `key` 。

示例：

```
$ yarn config delete test-key
yarn config vx.x.x
success Deleted "test-key".
  Done in 0.06s.
```

yarn config list

显示当前配置。

示例：

```
$ yarn config list
yarn config vx.x.x
info yarn config
{ 'version-tag-prefix': 'v',
  'version-git-tag': true,
  'version-git-sign': false,
  'version-git-message': 'v%s',
  'init-version': '1.0.0',
  'init-license': 'MIT',
  'save-prefix': '^',
  'ignore-scripts': false,
  'ignore-optional': true,
```

```
  registry: 'https://registry.yarnpkg.com',
  'user-agent': 'yarn/0.15.0 npm/? node/v6.2.1 darwin x64' }
info npm config
{ registry: 'https://registry.npmjs.org/',
  '//localhost:4873/:_authToken': 'some-auth-token' }
  Done in 0.05s.
```

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/config*

从任何 `create-*` 初学者工具包创建新项目。

yarn create <starter-kit-package> [<args>]

This command is a shorthand that helps you do two things at once:

- 全局安装 `create-<starter-kit-package>`，或更新包到最新版本（如果已存在）
- Run the executable located in the `bin` field of the starter kit's `package.json`, forwarding any `<args>` to it
  例如，`yarn create react-app my-app` 等同于：

```
$ yarn global add create-react-app
$ create-react-app my-app
```

For more information, check out the relevant blog entry.

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/create*

无需使用去重命令（dedupe），`yarn install` 命令会自动去重。

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/dedupe*

生成一个 lock 文件。

yarn generate-lock-entry

注：这个命令只给高级用例个工具。给它当前的 `package.json` 清单文件，它生成一个 *lock file* 入口。

```
$ yarn generate-lock-entry
# THIS IS AN AUTOGENERATED FILE. DO NOT EDIT THIS FILE DIRECTLY.
# yarn lockfile v1
yarnpkg@0.14.0:
  version "0.14.0"
  dependencies:
    babel-plugin-transform-inline-imports-commonjs "^1.0.0"
    babel-runtime "^6.0.0"
    bytes "^2.4.0"
    [...]
```

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/generate-lock-entry*

在你的操作系统上全局安装包。

`yarn global <add/bin/list/remove/upgrade> [–prefix]`

`yarn global` 是一个命令前缀，可用于 `add`、`bin`、`list` 和 `remove` 等命令。 它们的行为和他们的普通版本相同，只是它们用一个全局目录来存储包。 该 `global` 命令显示为您准备的可执行文件的位置。

注：不像 *npm* 里的 `–global` 标志，`global` 是一个必须跟在 `yarn` 后面的命令。 输入 `yarn add global package-name` 会把名为 `global` 和 `package-name` 的包添加到本地，而非全局添加 `package-name`。

这对于不是任何独立项目一部分、但用于本地命令的开发工具来说很有用。 一个这样的例子是 create-react-app，可以这样全局安装：

```
$ yarn global add create-react-app --prefix /usr/local
# the `create-react-app` command is now available globally:
$ which create-react-app
$ /usr/local/bin/create-react-app
$ create-react-app
```

定义安装位置

`yarn global bin` will output the location where Yarn will install symlinks to your installed executables. You can configure the base location with `yarn config set prefix <filepath>`. For example, `yarn config set prefix ~/.yarn` will ensure all global packages will have their executables installed to `~/.yarn/bin`.

`yarn global dir` will print the output of the global installation folder that houses the global `node_modules`. By default that will be: `~/.config/yarn/global`.

阅读更多可以和 `yarn global` 一起用的命令：

- `yarn add`：添加一个包用在你当前的项目里。
- `yarn bin`：显示 yarn bin 目录的位置。
- `yarn list`：列出已安装的包。
- `yarn remove`：从你当前包里移除一个不再使用的包。
- `yarn upgrade`: upgrade packages to their latest version based on the specified range.
- `yarn upgrade-interactive`: similar to `upgrade` command, but display the outdated packages before performing any upgrade, allowing the user to select which packages to upgrade.

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/global*

显示帮助信息。

```
yarn help
```

该命令会展示一系列可用的命令与选项，并对这些命令与选项的功能给出了简单的解释。

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/help*

依据原npm安装后的 `node_modules` 目录生成一份 `yarn.lock` 文件

yarn import

This command assists the migration of projects currently relying on `npm-shrinkwrap.json`, minimizing the differences between the lockfile and the existing dependency tree as best as it can.

动机

许多项目目前使用 `npm shrinkwrap` 或将 `node_modules` 签入它们的源代码控制，因为它们的依赖关系树很脆弱。 这些项目无法轻易迁移到 Yarn，因为 `yarn install` 可能产生有很大差异的逻辑依赖关系树。 不是所有树都可以用 Yarn 的 `yarn.lock` 表示， 并且部分有效的树会在安装后自动按重复剔除。 These nuances and others present a significant barrier to manual migration.

`yarn import` 旨在使用在 `node_modules` 内找到的版本根据普通的 `require.resolve()` 决议规则生成一个 `yarn.lock` 文件以缓解这一重大问题。 In cases where the Yarn resolution mechanism can't satisfy the existing dependency tree identically, alerts will be made so that you may manually review the changes. The existing `node_modules` tree will be checked for validity beforehand, and the resultant lockfile should be `yarn install` able without any surprises (failed compatibility, unresolvable dependencies, auto-dedupes, etc.)

```
$ yarn import
```

```
yarn import vx.x.x
success Folder in sync.
warning Using version "2.2.4" of "lru-cache" instead of "2.7.3" for "ngstorage > grunt > minimatch"
warning Using version "2.0.6" of "readable-stream" instead of "2.2.9" for "ngstorage > karma > chokidar > readdirp"
[...]
success Saved lockfile.
  Done in 11.96s.
```

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/import*

显示一个包的信息。

`yarn info <package> [<field>]`

这个命令会拉取包的信息并返回为树格式，包不必安装到本地。

```
yarn info react
```

```
yarn info vx.x.x
{ name: 'react',
  version: '15.4.0-rc.2',
  description: 'React is a JavaScript library for building user interfaces.',
  time: { modified: '2016-10-06T22:09:27.397Z', ... } ... }
```

这个命令默认的报告样式时单引号序列化的。 要输出有效的 JSON 行格式，使用标准的 `--json` 标志：

```
yarn info react --json
```

```
{"type":"inspect","data":{"name":"react","time":{...}}}
{"type":"finished","data":417}
```

指定版本的信息

追加 `@[version]` 到包名参数来提供那个特定版本的信息：

```
yarn info react@15.3.0
```

```
yarn info vx.x.x
{ name: 'react',
  version: '15.3.0',
  description: 'React is a JavaScript library for building user interfaces.',
  time: { modified: '2016-10-06T22:09:27.397Z', ... } ... }
```

选择特定字段

如果提供了可选字段参数，那么只有树的那部分被返回。

```
yarn info react description
```

```
yarn info vx.x.x
React is a JavaScript library for building user interfaces.
```

或检查可用的版本：

```
yarn info 反应版本
```

```
yarn info v1.1.0
[ '0.0.1',
  '0.0.2',
  '0.0.3',
(等)
```

如果指定的字段在一个内嵌的对象里，则返回子树：

```
yarn info react time
```

```
yarn info vx.x.x
{ modified: '2016-10-06T22:09:27.397Z',
  created: '2011-10-26T17:46:21.942Z', ... }

yarn info react time --json
{"type":"inspect","data":{"modified":"2016-10-06T22:09:27.397Z","created":...}}
...
```

获取 readme 字段

请注意，默认情况下，`yarn info` 不返回 `readme` 字段（因为它通常很长）。要显式请求那个字段，使用第二个参数：

```
yarn info react readme
```

```
yarn info vx.x.x
## react

An npm package to get you immediate access to
[React](https://facebook.github.io/react/).
...
```

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/info*

交互式创建或更新 package.json 文件。

yarn init

这个命令通过交互式会话带你创建一个 `package.json` 文件。 一些默认值比如 license 和初始版本可以在 yarn 的 `init-*` 配置里找到。

这是一个在名为 `testdir` 的目录里运行命令的例子：

```
$ yarn init
```

```
question name (testdir): my-awesome-package
question version (1.0.0):
question description: The best package you will ever find.
question entry point (index.js):
question git repository: https://github.com/yarnpkg/example-yarn-package
question author: Yarn Contributor
question license (MIT):
question private:
success Saved package.json
  Done in 87.70s.
```

这导致下面的 `package.json`：

```
{
  "name": "my-awesome-package",
  "version": "1.0.0",
  "description": "The best package you will ever find.",
  "main": "index.js",
  "repository": {
    "url": "https://github.com/yarnpkg/example-yarn-package",
    "type": "git"
  },
  "author": "Yarn Contributor",
  "license": "MIT"
}
```

> *By default, if answer given to `question private` is passed in as empty, the `private` key will not be added to `package.json`*

如果你已经有一个现成的 `package.json`，它会用这个文件的条目作为默认值。

现有下面的 `package.json`：

```
{
  "name": "my-existing-package",
  "version": "0.1",
  "description": "I exist therefore I am.",
  "repository": {
    "url": "https://github.com/yarnpkg/example-yarn-package",
    "type": "git"
```

```
  },
  "license": "BSD-2-Clause"
}
```

下面交互式会话期间默认值的结果：

```
$ yarn init
```

```
question name (my-existing-package):
question version (0.1):
question description (I exist therefore I am.):
question entry point (index.js):
question git repository (https://github.com/yarnpkg/example-yarn-package):
question author: Yarn Contributor
question license (BSD-2-Clause):
question private:
success Saved package.json
  Done in 121.53s.
```

为 yarn init 设置默认值

下面的 `config` 变量可被用于自定义 `yarn init` 的默认值：

- `init-author-name`
- `init-author-email`
- `init-author-url`
- `init-version`
- `init-license`

yarn init –yes/-y

这个命令跳过上面提到的交互式会话，并生成一个基于你的默认值的 `package.json` 。 一些默认值可以被上面提到的 `init-*` 配置改变。 例如，给定一个全新安装的 Yarn 并在一个 `yarn-example` 目录里：

```
$ yarn init --yes
```

```
warning The yes flag has been set. This will automatically answer yes to all questions which may have security implications.
success Saved package.json
  Done in 0.09s.
```

这会生成下面的 `package.json` ：

```
{
  "name": "yarn-example",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT"
}
```

yarn init –private/-p

> 自动添加 *private: true* 到 *package.json*

```
$ yarn init --private
```

If the `private` flag is set, the `private` key will be automatically set to `true` and you still complete the rest of the init process.

```
question name (testdir): my-awesome-package
question version (1.0.0):
question description: The best package you will ever find.
question entry point (index.js):
question git repository: https://github.com/yarnpkg/example-yarn-package
question author: Yarn Contributor
question license (MIT):
success Saved package.json
  Done in 87.70s.
```

```
{
  "name": "my-awesome-package",
  "version": "1.0.0",
  "description": "The best package you will ever find.",
  "main": "index.js",
  "repository": {
    "url": "https://github.com/yarnpkg/example-yarn-package",
    "type": "git"
  },
  "author": "Yarn Contributor",
  "license": "MIT",
  "private": true
}
```

可以同时使用 `yes` 和 `private` 标志。

像是：

```
$ yarn init -yp
```

```
warning The yes flag has been set. This will automatically answer yes to all questions which may have security implications.
success Saved package.json
  Done in 0.05s.
```

这会生成下面的 `package.json` ：

```
{
  "name": "yarn-example",
```

```
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "private": true
}
```

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/init*

`yarn install` 用于安装一个项目的所有依赖。 这个命令最常见的使用场景是在你刚Check out一份项目代码之后，或者在你需要使用其他开发者新增加的项目依赖的时候。

如果习惯使用 npm， 你可能希望使用 `–save` 或 `–save-dev`， 这些已经被 `yarn add` 和 `yarn add –dev` 所取代。 更多信息，请参阅 `yarn add` 文档。

执行不带任何命令的 `yarn`，等同于执行 `yarn install`，并透传所有参数。

如果需要可重现的依赖环境（比如在持续集成系统中），应该传入 `–frozen-lockfile` 标志。

`yarn install`

在本地 `node_modules` 目录安装 `package.json` 里列出的所有依赖。

`yarn install –check-files`

验证 `node_modules` 中已安装的文件没有被移除。

`yarn install –flat`

安装所有依赖，但每个依赖只允许有一个版本存在。 第一次运行这个命令时，会提示你在每个依赖包的多个版本范围中选择一个版本。 这会被添加到你的 `package.json` 文件的 `resolutions` 字段。

```
1. "resolutions": {
2.   "package-a": "2.0.0",
3.   "package-b": "5.0.0",
4.   "package-c": "1.5.2"
5. }
```

`yarn install –force`

这回重新拉取所有包，即使之前已经安装的。

`yarn install –har`

从安装期间的所有网络请求输出一个 HTTP archive。 HAR 文件通常用于排查网络性能，并能用 Google’s HAR Analyzer 或 HAR Viewer 这样的工具分析。

`yarn install –ignore-scripts`

不执行项目 package.json 及其依赖定义的任何脚本。

`yarn install –modules-folder <path>`

为 `node_modules` 目录指定另一位置，代替默认的 `./node_modules`。

`yarn install –no-lockfile`

不读取或生成 `yarn.lock` 锁文件。

`yarn install –production[=true|false]`

如果 `NODE_ENV` 环境变量设为 `production`，Yarn 将不安装任何列于 `devDependencies` 的包。 使用此标志指示 Yarn 忽略 `NODE_ENV` 并用它取代“生产”与否的状态。

> 注意：`–production` 等同 `–production=true`。`–prod` 是 `–production` 的别名。

`yarn install –pure-lockfile`

不生成 `yarn.lock` 锁文件。

`yarn install –frozen-lockfile`

不生成 `yarn.lock` 锁文件，并且，如果需要更新则会报错。

`yarn install –silent`

执行 yarn install 而不显示安装日志

`yarn install –ignore-engines`

忽略引擎检查。

`yarn install –ignore-optional`

Don’t install optional dependencies.

`yarn install –offline`

Run yarn install in offline mode.

`yarn install –non-interactive`

Disable interactive prompts, like when there’s an invalid version of a dependency.

`yarn install –update-checksums`

Update checksums in the `yarn.lock` lockfile if there’s a mismatch between them and their package’s checksum.

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/install*

列出已安装包的许可证。

yarn licenses list

运行这个命令将按字母顺序列出所有被 `yarn` 或 `yarn install` 安装的包，并且给你每个包关联的许可证（和源代码的 URL）。

```
yarn licenses list
```

```
yarn licenses v0.14.0
├─ abab@1.0.3
│  ├─ License: ISC
│  └─ URL: git+https://github.com/jsdom/abab.git
├─ abbrev@1.0.9
│  ├─ License: ISC
│  └─ URL: http://github.com/isaacs/abbrev-js
├─ acorn-globals@1.0.9
│  ├─ License: MIT
│  └─ URL: https://github.com/ForbesLindesay/acorn-globals.git
├─ acorn@2.7.0
│  ├─ License: MIT
│  └─ URL: https://github.com/ternjs/acorn.git
├─ align-text@0.1.4
│  ├─ License: MIT
│  └─ URL: git://github.com/jonschlinkert/align-text.git
├─ amdefine@1.0.0
│  ├─ License: BSD-3-Clause AND MIT
│  └─ URL: https://github.com/jrburke/amdefine.git
├─ ansi-escapes@1.4.0
│  ├─ License: MIT
│  └─ URL: https://github.com/sindresorhus/ansi-escapes.git
├─ ansi-regex@2.0.0
│  ├─ License: MIT
│  └─ URL: https://github.com/sindresorhus/ansi-regex.git
...
```

yarn licenses generate-disclaimer

运行这个命令将返回一个从所有你安装的包得到的排序后的许可证列表，打印在 `stdout` 。

```
yarn licenses generate-disclaimer
```

```
The following software may be included in this product: package-1. This software contains the following
license and notice below:

[[LICENSE TEXT]]

-----

The following software may be included in this product: package-2. 本软件包含如下证书和注意事项：

```

```
9. [[LICENSE TEXT]]
```

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/licenses*

开发过程中符号链接一个包的目录。

为了开发，一个包可以链接到另一个项目。通常用于测试出新功能，或者尝试调试包在其他项目中表现的问题时。

有两个命令来促成此流程：

`yarn link (在你想连接的包里)`

这个命令在你想链接的包里运行。 例如，如果你正工作在 `react` 并且向用你的本地版本来调试一个 `react-relay` 里的问题，在 `react` 项目里简单运行 `yarn link` 。

`yarn link [package…]`

使用 `yarn link [package]` 来链接另一个你想在当前项目里测试的包。 按照上面的例子，在 `react-relay` 项目里，你可以运行 `yarn link react` 来使用你之前链接的 `react` 本地版本。

`Complete example, assuming two project folders react and react-relay next to each other:`

```
$ cd react
$ yarn link
yarn link vx.x.x
success Registered "react".
info You can now run `yarn link "react"` in the projects where you want to use this module and it will be used instead.
```

```
$ cd ../react-relay
$ yarn link react
yarn link vx.x.x
success Registered "react".
```

这会创建一个符号链接 `react-relay/node_modules/react` 连接到你本地的 `react` 项目副本。

要逆转这个过程，只需使用 `yarn unlink` 或 `yarn unlink [package]` 。另请参见：

- `yarn unlink` ：取消已链接的包。

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/link*

列出已安装的包。

yarn list

```
1. yarn list
```

`yarn list` 命令模仿 Unix 列目录命令的预期行为。 In Yarn, the `list` command lists all dependencies for the current working directory by referencing all package manager meta data files, which includes a project’s dependencies.

```
1. yarn list vx.x.x
2. ├─ package-1@1.3.3
3. ├─ package-2@5.0.9
4. │  └─ package-3@^2.1.0
5. └─ package-3@2.7.0
```

yarn list [–depth] [–pattern]

默认情况下，所有包和它们的依赖会被显示。 要限制依赖的深度，你可以给 `list` 命令添加一个标志 `–depth` 所需的深度。

```
1. yarn list --depth=0
```

记住，深度层级时从零索引的。

`yarn list –pattern <pattern>` 会根据模式标志会筛选出依赖列表。

示例：

```
1. yarn list --pattern gulp
2. yarn list --pattern "gulp|grunt"
3. yarn list --pattern "gulp|grunt" --depth=1
```

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/list*

无需使用 lockfile 命令， `yarn install` 命令会自动创建 lockfile 文件。

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/lockfile*

存储 registry 用户名和 email。

yarn login

运行此命令会提示你输入你 npm registry 的用户名和 email。 它不会要求你提供密码。 之后当你运行像 `yarn publish` 这样的命令请求验证时，你必须输入密码才能做。

```
yarn login
```

```
yarn login vx.x.x
question npm username: my-username
question npm email: my-username@example.com
  Done in 6.03s.
```

使用 `yarn logout` 你可以删除你的用户名和 email。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/login*

清除 registry 用户名和 email。

```
yarn logout
```

这将移除你用 `yarn login` 保存给 npm registry 的用户名和 email。 你需要运行这个来解除认证，registry actions 单独认证。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/logout*

检查过时的包依赖。

`yarn outdated`

列出包的所有依赖项的版本信息，包括当前已安装的版本、最符合语义版本定义（semver）的版本和最新的可用版本。

例如，`package.json` 中列出了以下依赖项：

```
{
  "dependencies": {
    "underscore": "~1.6.0"
  },
  "devDependencies": {
    "lodash": "4.15.0"
  }
}
```

命令的输出结果类似以下：

```
yarn outdated
```

```
Package    Current Wanted Latest Package Type    URL
lodash     4.15.0  4.15.0 4.16.4 devDependencies https://github.com/lodash/lodash#readme
underscore 1.6.0   1.6.0  1.8.3  dependencies    https://github.com/jashkenas/underscore#readme
  Done in 0.72s.
```

`yarn outdated [package…]`

列出一个或多个依赖项的版本信息。

例如，对于上述 `package.json` 文件，当检查一个依赖包的版本信息时，会有以下输出：

```
yarn outdated lodash
```

```
Package Current Wanted Latest Package Type    URL
lodash  4.15.0  4.15.0 4.16.4 devDependencies https://github.com/lodash/lodash#readme
  Done in 1.04s.
```

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/outdated*

管理包的所有者。

什么是包的所有者？

包管理平台里一个包的“所有者”是一个能修改包的用户。如果你想的话，一个包可以有多个所有者。

所有者有权执行以下任务：

- 发布包的新版本
- 增加或移除包的所有者
- 修改包的元数据

注意事项

现在没有任何其他访问权限级别。所有用户都可以修改或不能修改包，也许会有更多角色类型，但现在还没有。

命令

`yarn owner list <package>`

列出 `<package>` 的全部所有者。

`yarn owner add <user> <package>`

增加 `<user>` 为 `<package>` 的所有者。要运行这个命令，你必须已经是 `<package>` 的一个所有者。

`yarn owner remove <user> <package>`

从 `<package>` 里移除所有者 `<user>`。要运行这个命令，你必须已经是 `<package>` 的所有者。

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/owner*

`yarn pack`

创建一个压缩的包依赖 `gzip` 档案。

`yarn pack –filename <filename>`

创建一个压缩的包依赖 `gzip` 档案并命名为 *`filename`*。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/pack*

The prune command isn’t necessary. `yarn install` will prune extraneous packages.

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/prune*

发布一个包到 npm 库。

一旦一个包被发布了，你不能修改那个特定版本，所以发布前要小心。

yarn publish

发布当前目录里 `package.json` 定义的包。

yarn publish [tarball]

发布一个 `.tgz` gzip 压缩文件定义的包。

yarn publish [folder]

发布包含在指定目录里的包。 `<folder>/package.json` 应该指定包的细节。

yarn publish –new-version <version>

使用 `version` 的值，跳过对新版本的询问。

yarn publish –tag <tag>

为 `yarn publish` 提供一个标签（tag）可以让你发布的包带有一个指定的标签。 例如，如果你运行了 `yarn publish –tag beta`，并且你的包名叫 `blorp`，那么其他人可以用 `yarn add blorp@beta` 安装那个包。

yarn publish –access <public|restricted>

The `–access` flag controls whether the npm registry publishes this package as a public package, or restricted.

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/publish*

`yarn remove <package…>`

运行 `yarn remove foo` 会从你的直接依赖里移除名为 `foo` 的包，在此期间会更新你的 `package.json` 和 `yarn.lock` 文件。

此项目上工作的其他开发者可以运行 `yarn install` 来同步他们的 `node_modules` 目录为新的依赖集。

当你移除一个包时，它被从所有类型的依赖里移除： `dependencies` 、 `devDependencies` 等等。

> 注意： *`yarn remove`* 总是会更新你的 *`package.json`* 和 *`yarn.lock`* 。 这可以确保不同的开发人员在同一个项目上得到相同的依赖集。 不可能禁止这个行为。 注意： *`yarn remove <package> –<flag>`* 使用与 *`yarn install`* 命令相同的 *`flag`* 。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/remove*

运行一个定义好的包脚本。

你可以在你的 `package.json` 文件中定义 `scripts`。

```
{
  "name": "my-package",
  "scripts": {
    "build": "babel src -d lib",
    "test": "jest"
  }
}
```

`yarn run [script] [<args>]`

如果你已经在你的包里定义了 `scripts`，这个命令会运行指定的 `[script]`。例如：

```
yarn run test
```

运行这个命令会执行你的 `package.json` 里名为 `"test"` 的脚本。

您可以在脚本名称后放置要传递给您的脚本的额外参数。

```
yarn run test -o --watch
```

运行这个命令会执行 `jest -o –watch`。

`[script]` 也可以是任何 `node_modules/.bin/` 里本地安装的可执行程序。

也可以在该命令中忽略 `run`，每个脚本都可以用其名字执行：

```
yarn test -o --watch
```

执行该命令会和 `yarn run test` 有一样的效果。 注意内置的 `cli` 命令将优先于你的脚本，因此不应在其他脚本中一直依赖该快捷方式。

`yarn run env`

执行该命令将会会列出脚本运行时可用的环境变量

如果想覆盖此命令，可以在 `package.json` 中定义自己的 `"env"` 脚本。

`yarn run`

如果你不指定一个脚本给 `yarn run` 命令，`run` 命令会列出包里所有可运行的脚本。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/run*

更新 Yarn 到最新版。

```
yarn self-update
```

注：`self-update` 当前不可用，意味着更新必须手动完成。 详情请看 *issue #1139*。

这个命令用于更新 Yarn 到最新可用版本。

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/self-update*

增加、删除或列出一个包的标签。

什么是标签？

发布标签（或 dist-tags）是一种把已发布版本标记为一个标签的方法。你的包的用户可以用这个标签代替版本号来安装。

例如，如果你有一个 **stable** 发版通道和一个 **canary** 发版通道，你可以把标签作为一种方式允许用户这样输入：

```
1. yarn add your-package-name@stable
2. yarn add your-package-name@canary
```

不同标签有不同的含义：

- `latest`：包的当前版本
- `stable`：包的最新稳定版本，通常和和 latest 版本相同，除非你有长期支持版本（LTS）
- `beta`：成为 latest 或 stable 前的版本，用在即将到来的变更完成前分享。
- `canary`：晚间构建或 beta 预发布版本，如果你的项目频繁更新并被很多人依赖，你可以用这个来尽早分享代码。
- `dev`：有时你想尽可能通过注册表测试一个你还在处理的修订版，这很有用。
  有些项目会创造他们看着合适或者替代更标准标签的自有标签。比如用 `next` 等同 `beta`。

虽然这些是公认的“标准”标签，唯一有实际意义的是 `latest`，当没有指定版本时用来确定安装哪个版本。

注意事项

你不能用和潜在版本号匹配的标签，因为它们共享一个命名空间：

```
1. yarn add your-package-name@<version>
2. yarn add your-package-name@<tag>
```

任何可以作为有效语义版本范围的标签将会被拒绝。比如，你不能有一个名叫 `v2.3` 的标签，因为在语义版本里它意味着 `>=2.3.0 <2.4.0`。

一般情况下，避免使用看起来像版本号的标签，它们通常只会把人弄糊涂。

命令

`yarn tag add <package>@<version> <tag>`

为一个 `<package>` 的指定 `<version>` 添加一个名为 `<tag>` 的标签。

`yarn tag remove <package> <tag>`

从 `<package>` 里删除一个不再使用的名叫 `<tag>` 的标签。

> 注意：你在包里移动一个标签到另一个版本前不需要删除它，不删更好。

`yarn tag list [<package>]`

列出 `<package>` 的所有标签。如果没有指定， `<package>` 默认为你当前所在目录。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/tag*

维护团队成员

`yarn team`

管理组织里的团队，修改团队成员。

命令

`yarn team create <scope:team>`

创建一个新团队。

`yarn team destroy <scope:team>`

解散一个现有团队。

`yarn team add <scope:team> <user>`

增加一个用户到现有团队。

`yarn team remove <scope:team> <user>`

从用户所属团队中移除用户。

`yarn team list <scope>|<scope:team>`

如果用一个组织名称做参数，会返回那个组织下的现有团队列表。 如果对一个团队执行它，它会返回那个特定团队所属的所有用户列表。

细节

`yarn team` 总是操作当前包管理器注册表，可以从命令行上用 `–registry=<registry url>` 配置。

为了创建团队和管理团队成员，你在给定的组织里必须是一个团队管理员。组织的任何成员可以列出团队和团队成员。

组织的建立、团队管理员和组织成员的管理可以通过 `npm` 网站而不是 `CLI` 来完成。

要使用团队来管理属于你组织的包的权限，使用 `yarn access` 命令来授予或取消合适的权限。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/team*

运行包定义的测试脚本。

yarn test

如果你的包里有一个定义好的 `scripts` 对象，这个命令会运行指定的 `test` 脚本。

例如，如果你的包里有一个 bash 脚本 `scripts/test`：

```
#!/bin/bash

echo "Hello, world!"
```

并且下面是你 `package.json` 里的：

```
{
  "name": "my-tribute-package",
  "version": "1.0.0",
  "description":
    "This is not the best package in the world, this is just a tribute.",
  "main": "index.js",
  "author": "Yarn Contributor",
  "license": "MIT",
  "scripts": {
    "test": "scripts/test"
  }
}
```

那么运行 `yarn test` 将得到：

```
$ yarn test
yarn test v0.15.1
$ "./scripts/test"
Hello, world!
 Done in 0.17s.
```

yarn run test

`yarn test` 也是 `yarn run test` 的快捷命令。

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/test*

取消一个以前创建的包符号链接。

要移除一个使用 `yarn link` 创建的符号链接的包，可以使用 `yarn unlink` 。

yarn unlink

在之前用来创建链接的目录里运行 `yarn unlink` 。

yarn unlink [package]

要取消开发期间你项目里用符号链接的包，简单运行 `yarn unlink [package]` 即可。 你需要运行 `yarn` 或 `yarn install` 重新安装已经用符号链接的包。

Continued example from the `yarn link` documentation: assume two folders `react` and `react-relay` that are located next to each other with `react` linked into the `react-relay` project:

```
$ cd react
$ yarn unlink
yarn link vx.x.x
success Unregistered "react".
```

```
$ cd ../react-relay
$ yarn unlink react
yarn link vx.x.x
success Unregistered "react".
```

另请参见：

- `yarn link`： 在本地开发环境里符号链接一个包。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/unlink*

升级包到它们基于规范范围的最新版本。

yarn upgrade [package | package@tag | package@version | @scope/]… [–ignore-engines] [–pattern]

该命令会根据在 `package.json` 文件中所指定的版本范围将依赖更新到其最新版本。也会重新生成 `yarn.lock` 文件。

可以选择指定一个或多个包名称。指定包名称时，将只升级这些包。未指定包名称时，将升级所有依赖项。

`[package]` : When a specified package is only a name then the latest patching version of this package will be upgraded to.

`[package@tag]` ：指定包包含标签时，将升级到该标签的版本。 标签 名称由项目维护者选择，通常你用这个命令来安装一个活跃开发中的包的实验版本或长期支持版本。 你选择的标签会成为出现在你的 `package.json` 文件里的版本。

`[package@version]` ： 当指定包包含版本时，将升级到该版本。 `package.json` 中指明的依赖也将同时更改为指定的版本。 你可以使用任何语义版本的版本号或版本范围。

`–ignore-engines` ： 此标志可用于跳过引擎检查。

示例：

```
1. yarn upgrade
2. yarn upgrade left-pad
3. yarn upgrade left-pad@^1.0.0
4. yarn upgrade left-pad grunt
5. yarn upgrade @angular
```

`yarn upgrade –pattern <pattern>` 将升级所有匹配此模式的包。

示例：

```
1. yarn upgrade --pattern gulp
2. yarn upgrade left-pad --pattern "gulp|grunt"
3. yarn upgrade --latest --pattern "gulp-(match|newer)"
```

yarn upgrade [package]… –latest|-L [–caret | –tilde | –exact] [–pattern]

The `upgrade –latest` command upgrades packages the same as the `upgrade` command, but ignores the version range specified in `package.json`. Instead, the version specified by the `latest` tag will be used (potentially upgrading the packages across major versions).

The `package.json` file will be updated to reflect the latest version range. By default, the existing range specifier in `package.json` will be reused if it is one of: ^, ~, <=, >, or an exact version. 否则，它将被更改为插入符号 (^)。 One of the flags `–caret`, `–tilde` or `–exact` can be used to explicitely specify a range.

示例：

```
1. yarn upgrade --latest
2. yarn upgrade left-pad --latest
3. yarn upgrade left-pad grunt --latest --tilde
```

yarn upgrade (–scope|-S) @scope [–latest] [–pattern]

`–作用域 @scope/` : 指定作用域时，只会升级以作用域开头的包。作用域必须以“@”开头。

`–latest` : 忽略在 `package.json` 中指定的版本范围。 Instead, the version specified by the `latest` tag will be used (potentially upgrading the packages across major versions).

示例:

```
1. yarn upgrade --scope @angular
2. yarn upgrade -S @angular
```

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/upgrade*

This is similar to npm-check interactive update mode. It provides an easy way to update outdated packages.

yarn upgrade-interactive [–latest]

`upgrade-interactive` 与 `upgrade` 命令采用相同的参数和功能。 在执行升级操作之前，此命令将显示已过期的包列表，并允许用户选择相应的想要升级的包。 Yarn will respect the version ranges in `package.json` when determining the version to upgrade to.

You can think of `yarn upgrade-interactive` as a combination of the `yarn outdated` and `yarn upgrade [package…]` commands. Where `yarn outdated` displays the list of outdated packages and `yarn upgrade [package…]` can then be used to upgrade desired packages, `yarn upgrade-interactive` displays the same outdated package list and lets you immediately chose which to upgrade.

`–latest` ：此标志告知 yarn 忽略 `package.json` 中指定的版本范围，改用资源库中标为 `latest` 的版本。

```
1.  [1/? 选择更新哪些包。 (按空格键选择, a 键切换所有, i 键反选选择)
2.   devDependencies
3.  ❯○ autoprefixer      6.7.7  ❯  7.0.0          https://github.com/postcss/autoprefixer#readme
4.   ○ webpack           2.4.1  ❯  2.5.1          https://github.com/webpack/webpack
5.
6.   dependencies
7.   ○ bull              2.2.6  ❯  3.0.0-alpha.3  https://github.com/OptimalBits/bull#readme
8.   ○ fs-extra          3.0.0  ❯  3.0.1          https://github.com/jprichardson/node-fs-extra
9.   ○ socket.io         1.7.3  ❯  1.7.4          https://github.com/socketio/socket.io#readme
10.  ○ socket.io-client  1.7.3  ❯  1.7.4          https://github.com/Automattic/socket.io-client#readme
```

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/upgrade-interactive*

更新包版本。

更新版本

用 `yarn version` 命令你可以通过命令行更新你的包版本。

例如，从这个 `package.json` 开始：

```
{
  "name": "example-yarn-package",
  "version": "1.0.1",
  "description": "An example package to demonstrate Yarn"
}
```

但我们运行 `yarn version` 命令：

```
yarn version
```

```
info Current version: 1.0.1
question New version: 1.0.2
info New version: 1.0.2
  Done in 9.42s.
```

我们会得到这个更新的 `package.json`：

```
{
  "name": "example-yarn-package",
  "version": "1.0.2",
  "description": "An example package to demonstrate Yarn"
}
```

> 注意： 你输入的新版本必须是有效的语义版本版本号。

Git 标签

如果你在一个 Git 仓库内运行 `yarn version`，一个 Git 标签 默认会被以 `v0.0.0` 格式创建。

你可以自定义被创建的 git 标签或通过 `yarn config set` 禁用这个行为。

要修改 git 标签的前缀，你可以使用 `version-tag-prefix`：

```
yarn config set version-tag-prefix "v"
```

或者你也可以用 `version-git-message` 修改 git 版本消息（ `%s` 时版本号字符串）。

```
yarn config set version-git-message "v%s"
```

您也可以使用 `version-sign-git-tag` 开关 git 标签签名：

```
1. yarn config set version-sign-git-tag false
```

你甚至可以用 `version-git-tag` 完全启用或禁用 git 打标签行为。

```
1. yarn config set version-git-tag true
```

命令

yarn version

用交互式会话提示你输入一个新版本号来创建新版本。

yarn version —new-version <version>

创建一个由 `< 版本 >` 指定的新版本。

yarn version —no-git-tag-version

创建一个新版本，不创建 git 标签。

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/version*

Displays version information of the currently installed Yarn, Node.js, and its dependencies.

yarn versions

```
yarn versions
```

```
yarn versions v0.24.5
{ http_parser: '2.7.0',
  node: '7.10.0',
  v8: '5.5.372.43',
  uv: '1.11.0',
  zlib: '1.2.11',
  ares: '1.10.1-DEV',
  modules: '51',
  openssl: '1.0.2k',
  icu: '58.2',
  unicode: '9.0',
  cldr: '30.0.3',
  tz: '2016j' }
  Done in 0.04s.
```

> 原文: *https://yarnpkg.com/zh-Hans/docs/cli/versions*

显示有关一个包为何被安装的信息。

`yarn why <query>`

这个命令将确定为什么安装了一个包，详述其它哪些包依赖它，例如，它是否在 `package.json` 清单里被显式标记为一个依赖。

```
1. yarn why jest
```

```
1. yarn 为什么 vx.x.x
2. [1/4]
3. [2/4]
4. [3/4]
5. [4/4]
6. info Has been hoisted to "jest"
7. info This module exists because it's specified in "devDependencies".
8. info Disk size without dependencies: "1.29kB"
9. info Disk size with unique dependencies: "101.31kB"
10. info Disk size with transitive dependencies: "20.35MB"
11. info Amount of shared dependencies: 125
```

查询参数

`yarn why` 强制性的查询参数可以是以下之一：

- 一个包名（作为上面例子里的）
- 一个包目录；例如：`yarn why node_modules/once`
- 一个包目录里的文件；例如：`yarn why node_modules/once/once.js`
  文件也可以是绝对路径。

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/why*

查看这些链接以了解更多有关工作区信息：

- Yarn 的工作区
- 工作区

`yarn workspace <workspace_name> <command>`

这会在所选工作区中运行所选的 Yarn 命令。

示例：

```
1. yarn workspace awesome-package add react react-dom --dev
```

这会将 `react` 和 `react-dom` 作为 `devDependencies` 加到 `packages/awesome-package/package.json` 中。

如果想移除一个包：

```
1. yarn workspace web-project remove some-package --save
```

上面的例子将会从 `packages/web-project/package.json` 中移除 `some-package`

> 原文：*https://yarnpkg.com/zh-Hans/docs/cli/workspace*

对多数用户来说，从npm迁移的过程应该非常简单。Yarn和npm使用相同的 `package.json` 格式，而且Yarn可以从npm安装依赖包。

如果你打算在现有项目中尝试Yarn，只需执行：

```
1. yarn
```

Yarn将通过自己的解析算法来重新组织 `node_modules` 目录，这个算法和node.js 模块解析算法是兼容的。

如果出错，请查阅issue列表，或者向Yarn issue tracker报告。

执行 `yarn` 命令或者 `yarn add <package>` 命令后，Yarn都会在项目根目录下生成 `yarn.lock` 文件。 你无需理解此文件的具体内容，但请记得将其提交到代码管理系统。 当其他开发者也从 `npm` 迁移到Yarn时， `yarn.lock` 文件的存在会确保他们得到的依赖包与你的完全相同。

多数情况下，第一次执行 `yarn` 或者 `yarn add` 都会成功。 有些情况下， `package.json` 文件里的信息不足以找出冗余依赖，Yarn安装依赖时采用的确定性算法就会导致依赖冲突。 这种情况常常出现在那些由于 `npm install` 执行出现问题， `node_modules` 文件夹被多次删除，并重新安装的大型项目里。 如果发生这种情况，请在迁移到Yarn前尝试使用 `npm` 命令来让依赖的版本更明确。

项目的其他开发者可以继续使用 `npm` ，所以无需让每个人同时迁移。 使用 `yarn` 会让开发者得到完全相同的配置，而使用 `npm` 却未必，但这是 `npm` 的预期行为。

如果你之后发现Yarn并不适合自己，你无需任何特别修改就能迁移回 `npm` 。 如果项目里所有人都不再使用Yarn，就可以删除 `yarn.lock` 文件（但不是必须）。

如果项目目前使用了 `npm-shrinkwrap.json` 文件，请小心你可能会得到一组不同的依赖。 Yarn不支持npm shrinkwrap文件，因为文件里没有足够的信息来支撑Yarn的确定性算法。 所以如果项目正在使用 shrinkwrap 文件，那么团队成员同时迁移到Yarn可能会更容易一点。 只需删除现有的 `npm-shrinkwrap.json` 文件，并提交新创建的 `yarn.lock` 文件。

CLI 命令比较

npm (v5)	Yarn
`npm install`	`yarn install`
(不适用)	`yarn install --flat`
(不适用)	`yarn install --har`
`npm install --no-package-lock`	`yarn install --no-lockfile`
(不适用)	`yarn install --pure-lockfile`
`npm install [package]`	`yarn add [package]`
`npm install [package] --save-dev`	`yarn add [package] --dev`
(不适用)	`yarn add [package] --peer`
`npm install [package] --save-optional`	`yarn add [package] --optional`
`npm install [package] --save-exact`	`yarn add [package] --exact`
(不适用)	`yarn add [package] --tilde`
`npm install [package] --global`	`yarn global add [package]`

`npm update –global`	`yarn global upgrade`
`npm rebuild`	`yarn install –force`
`npm uninstall [package]`	`yarn remove [package]`
`npm cache clean`	`yarn cache clean [package]`
`rm -rf node_modules && npm install`	`yarn upgrade`

> 原文: *https://yarnpkg.com/zh-Hans/docs/migrating-from-npm*

包 由项目代码和一个向Yarn提供包信息的 `package.json` 文件组成。

大部分的包都使用了版本控制系统，最常见的是git。但 Yarn 对此并不做要求，你可随意选择。本指南以git为例。

> 注意： 如果打算照着本指南实践，请先安装 *git* 和 *Yarn*。

创建你的第一个包

想要创建你的第一个包，打开系统终端/控制台并运行以下命令：

```
1. git init my-new-project
2. cd my-new-project
3. yarn init
```

这些命令会创建一个新的 git 仓库，并切换其为当前工作目录，然后显示一个包含以下问题的交互式表单，用于创建一个新的 yarn 项目：

```
1. name (my-new-project):
2. version (1.0.0):
3. description:
4. entry point (index.js):
5. git repository:
6. author:
7. license (MIT):
```

你既可以回答这些问题，也可以直接敲回车键（enter/return）使用默认配置或者留空。

> 提示：如果打算所有的问题都使用默认配置，你也可以运行 *yarn init –yes*，这会跳过所有问题。

package.json

现在应该创建了一个和下面文件内容类似的 `package.json`：

```
1. {
2.   "name": "my-new-project",
3.   "version": "1.0.0",
4.   "description": "My New Project description.",
5.   "main": "index.js",
6.   "repository": {
7.     "url": "https://example.com/your-username/my-new-project",
8.     "type": "git"
9.   },
10.   "author": "Your Name <you@example.com>",
11.   "license": "MIT"
12. }
```

`package.json` 里的字段含义如下：

- **name** 是包的标识，如果你打算把它发布到全局registry，请确保这个标识是唯一的。
- **version** 是语义版本号（semver），包可以被发布任意多次，但每次发布必须包含新的版本号。
- **description**是包的描述，用以让其他 Yarn 用户搜索并了解你的项目，这个字段非必须，但推荐填写。
- **main** 用来定义会被 Node.js 这类程序使用的代码入口，默认值为 `index.js`。
- **repository** 可以帮助其他用户找到包的代码托管处，并为其做贡献，这同样是一个可选但推荐填写的字段。
- **author** 是包的创建者或维护者，遵循 `"Your Name <you@example.com> (http://your-website.com)&#34;` 这样的格式。
- **license** 是包发布的法律条款，以及什么是包代码的许可用法。
  `yarn init` 的运行结果除了创建此文件之外，没有任何副作用，你可以自由编辑此文件。

附加字段

我们来看看 `package.json` 有哪些附加字段：

```
{
  "name": "my-new-project",
  "...": "...",
  "keywords": ["cool", "useful", "stuff"],
  "homepage": "https://my-new-project-website.com",
  "bugs": "https://github.com/you/my-new-project/issues",
  "contributors": [
    "Your Friend <their-email@example.com> (http://their-website.com)",
    "Another Friend <another-email@example.com> (https://another-website.org)"
  ],
  "files": ["index.js", "lib/*.js", "bin/*.js"],
  "bin": {
    "my-new-project-cli": "bin/my-new-project-cli.js"
  }
}
```

- **keywords** 是关键字列表，帮助其他开发者进行搜索。
- **homepage** 是项目的主页，包含包的简介、文档和其他附加资源链接。
- **bugs** 是帮助用户了解包现有问题的URL链接。
- **contributors** 是包的贡献者列表，如果有别人参与你的项目，你可以在这里指明。
- **files** 是包发布和安装时应该包含的文件列表，如果不指定，Yarn 会列出项目中的所有文件。
- **bin** 是一个让 Yarn 在包安装时给包创建 cli 命令（二进制）的映射表。
  要得到 `package.json` 所有字段以及上面那些字段的更详细信息，请参阅 `package.json` 文档。

许可证和开源

通常来说，Yarn 鼓励将包 开源，但要注意的是，开源指的不是简单的发布。

开源许可证是开源代码必需的，有很多许可证可供选择，以下是最常见的几个：

- MIT License
- Apache License 2.0
- GNU General Public License 3.0
  这个链接里有更多的选项。

选择好开源许可证以后，请在包的根目录下添加包含许可证文本的 `LICENSE` 文件，并更新 `package.json` 文件的 `license` 字段。

> 注意：如果不想开源，你应该指明授权方式。

共享代码

你可以将代码托管在以下几个流行的网站，以方便其他用户访问以及报告问题：

- GitHub
- GitLab
- Bitbucket
  用户通过上述网站可以查看代码、报告问题、贡献力量。代码被托管妥当后，请更新 `package.json` 文件的以下字段：

```
{
  "homepage": "https://github.com/username/my-new-project",
  "bugs": "https://github.com/username/my-new-project/issues",
  "repository": {
    "url": "https://github.com/username/my-new-project",
    "type": "git"
  }
}
```

文档

理想情况下，你应该在包发布之前撰写文档。 起码应该在项目根目录下放置 `README.md` 文件，包含包的简介和对外的API描述。

良好的文档应该为用户指明上手方法，以及如何深度使用。 让自己站在小白用户的立场上思考。 文档要详细准确，同时尽可能简单易懂。 有高质量文档的项目更容易成功。

越小越好

我们鼓励创建小巧简单的Yarn包。 只要可以，就尽可能把较大的包拆分成较小的包。 拆分的原因是，Yarn可以高效地安装成百上千数量的包。

“大量小包”是包管理的最佳实践之一，这会使下载大小更小，因为这样做避免了打包体积巨大的代码，却仅使用到了其中很小一部分。

还要记得留意包里的文件内容， 不要无意间发布用于测试的代码，或者其他与包的正常使用无关的内容，比如构建脚本、图片等。

同时留意包的依赖，能少则少。不要无意间引入大型依赖项目。

> 原文：*https://yarnpkg.com/zh-Hans/docs/creating-a-package*

为了通过 Yarn 共享你的包给全世界的其他开发者，你需要先发布它。

通过 Yarn 发布的包会托管在 npm registry 上，用于全球分发。

登录到 npm

如果你从未登录过，那就先创建一个 npm 账号。创建完成后，在 Yarn 里设置用户名和email。

```
1. yarn login
```

这个命令会提示你输入用户名和 email， 但不需要输入密码。 Yarn 不保存你的密码，也不保持任何 Session。当你真正发布或者修改 npm 上的包时，才需要输入密码。

发布你的包

在你写完代码、测试通过、准备发布的时候，你可以执行：

```
1. yarn publish
```

首先，需要输入新的版本号：

```
1. [1/4] Bumping version...
2. info Current version: 1.0.0
3. question New version: _____
```

之后，输入 npm 密码：

```
1. [2/4] Logging in...
2. info npm username: your-npm-username
3. info npm username: you@example.com
4. question npm password: ____________
```

最后，Yarn 将发布包并且注销你的会话。

```
1. [3/4] Publishing...
2. success Published.
3. [4/4] Revoking token...
4. success Revoked login token.
5.   Done in 10.53s.
```

每次发布都可以遵循上述流程。

访问你的包

你应该可以通过 https://www.npmjs.com/package/my-new-project 来访问新上传的包，也可以通过以下命令安装：

```
yarn add my-new-project
```

你也可以看到 npm registry 里的全部信息：

```
yarn info my-new-project
```

```
{ name: 'my-new-project',
  description: 'My New Project description.',
  'dist-tags': { latest: '1.0.0' },
  versions: [ '1.0.0' ],
  maintainers: [ { name: 'Your Name', email: 'you@example.com' } ],
  time:
  { modified: '2018-12-05T09:28:54+00:00',
    created: '2018-12-05T09:28:54+00:00',
    '1.0.0': '2018-12-05T09:28:54+00:00' },
  homepage: 'https://my-new-project-website.com/',
  keywords: [ 'cool', 'useful', 'stuff' ],
  repository:
   { url: 'https://example.com/your-username/my-new-project',
     type: 'git' },
  contributors:
   [ { name: 'Your Friend',
       email: 'their-email@example.com',
       url: 'http://their-website.com' },
     { name: 'Another Friend',
       email: 'another-email@example.com',
       url: 'https://another-website.org' } ],
  author: { name: 'Your Name', email: 'you@example.com' },
  bugs: { url: 'https://github.com/you/my-new-project/issues' },
  license: 'MIT',
  readmeFilename: 'README.md',
  version: '1.0.0',
  main: 'index.js',
  files: [ 'index.js', 'lib/*.js', 'bin/*.js' ],
  bin: { 'my-new-project-cli': 'bin/my-new-project-cli.js' },
  dist:
   { shasum: '908bc9a06fa4421e96ceda243c1ee1789b0dc763',
     tarball: 'https://registry.npmjs.org/my-new-project/-/my-new-project-1.0.0.tgz' },
  directories: {} }
```

> 原文：*https://yarnpkg.com/zh-Hans/docs/publishing-a-package*

包依赖对包的成功至关重要。 你很可能会用到其他包里已有的代码来开发自己包的功能。 那些“其他包”称为项目依赖。

所以依赖都在 `package.json` 文件里声明，包含开发依赖、运行依赖、可选依赖等。 每个依赖都需要指明依赖名称和最低可用版本。

`yarn.lock` 里保存了每个依赖的安装版本，这可以确保你的包每次安装的一致性。

> 原文: *https://yarnpkg.com/zh-Hans/docs/dependencies*

不同的依赖有着不同的目的。 开发构建时需要一部分，程序运行时也需要一部分。 因此有着不同的依赖类型（比如 `dependencies` 、 `devDependencies` 和 `peerDependencies` ）。

`package.json` 可以包含以下依赖类型：

```
{
  "name": "my-project",
  "dependencies": {
    "package-a": "^1.0.0"
  },
  "devDependencies": {
    "package-b": "^1.2.1"
  },
  "peerDependencies": {
    "package-c": "^2.5.4"
  },
  "optionalDependencies": {
    "package-d": "^3.1.0"
  }
}
```

大多数情况下只会用到 `dependencies` 和 `devDependencies` ，但这些类型都很重要。

dependencies

这是所谓的常规依赖，确切地说，是代码运行时所需要的（比如 React 和 immutableJS）。

devDependencies

这是开发依赖，就是那些只在开发过程中需要，而运行时不需要的依赖（比如 Babel 和 Flow）。

peerDependencies

这是“同伴依赖”，一种特殊的依赖，在发布包的时候需要。

有这种依赖意味着安装包的用户也需要和包同样的依赖。 这对于像 `react` 这样也被人安装的、需要单一 `react-dom` 副本的包很有用。

optionalDependencies

这是可选依赖，意味着依赖是……可选的。这种依赖即便安装失败，Yarn也会认为整个依赖安装过程是成功的。

这种类型适用于那些即便没有成功安装可选依赖，也有后备方案的情况（比如 Watchman）。

bundledDependencies

这是“打包依赖”，在发布包时，这个数组里的包都会被打包（Bundle）。

这种类型的依赖应该在项目内部使用，基本上和普通依赖相同。执行 `yarn pack` 同样会进行打包。

普通依赖通常从 npm registry 安装，这些情况下，打包依赖比普通依赖更好用：

- 当你想使用一个不在 npm registry 里的，或者被修改过的第三方库时；

- 当你想把自己的项目作为模块来重用时；
- 当你想和你的模块一起发布一些文件时。

> 原文：*https://yarnpkg.com/zh-Hans/docs/dependency-types*

语义化版本

Yarn 里的包遵守 语义化版本，也叫 “semver”。 当你从资源库安装一个新包，它会和语义版本范围一起被添加到你的 `package.json` 。

版本号可以划分为 `主版本号.次版本号.修订号` ，类似这些： `3.14.1` 、 `0.42.0` 、 `2.7.18` 。 不同的情况对应不同的版本号增长：

- `主版本号` ：新的版本不兼容老版本的 API
- `次版本号` ：新的版本新增了部分功能，并向下兼容
- `修订号` ：新的版本修复了部分bug，并向下兼容

> 注意： 有时语义版本也会包含一些“标签”或者“扩展”，用于标记预发布版本或者测试版本，比如 `2.0.0-beta.3` 。

开发者所说的“兼容”通常表示新的版本“向下兼容”(`次版本号` 和 `修订号`)。

版本范围

请在 `package.json` 文件里同时使用依赖名称和其版本范围来指明所需要的依赖：

```
{
  "dependencies": {
    "package-1": ">=2.0.0 <3.1.4",
    "package-2": "^0.4.2",
    "package-3": "~2.7.1"
  }
}
```

你一定注意到了除版本号外的那些特殊字符。 这些字符有 `>=` ， `<` ， `^` 和 `~` ，它们是运算符，用来指定版本范围。

版本范围的用处是标明依赖的哪个版本会在代码中起作用。

比较器

每个版本范围组成 比较器。这些比较器是简单 运算符 后面跟着一个 版本号。以下是一些基本的运算符：

比较器	描述
`<2.0.0`	任何小于 `2.0.0` 的版本
`<=3.1.4`	任何小于或等于 `3.1.4` 的版本
`>0.4.2`	任何大于 `0.4.2` 的版本
`>=2.7.1`	任何大于或等于 `2.7.1` 的版本
`=4.6.6`	任何等于 `4.6.6` 的版本

> 注意：如果没有明确指定运算符，那么版本范围默认为 `=` ，也就是说 `=` 是可选的。

交集

用空格连接若干比较器可以创建 比较器集合。 最终版本范围是它包含的比较器的交集。 例如，比较器集合 `>=2.0.0 <3.1.4` 表示“大于或等于 `2.0.0` 并小于 `3.1.4` ”。

并集

完整的版本范围可以包含多个用 `||` 连接的比较器集合的并集。 如果并集的任何一边满足条件，整个版本范围就满足条件。 例如，版本范围 `<2.0.0 || >3.1.4` 意味着“小于 `2.0.0` 或者大于 `3.1.4` ”。

预发布标签

版本号也可以包含预发布标签（比如 `3.1.4-beta.2` ）。 如果一个比较器包含有预发布标签的版本，它将只匹配有相同 `major.minor.patch` 的版本。

例如，版本范围 `>=3.1.4-beta.2` 将匹配 `3.1.4-beta.2` 或 `3.1.4-beta.12` ，但不会匹配 `3.1.5-beta.1` ，即使在理论上 `3.1.5-beta.1` 版本 “大于或等于” （ `>=` ） <0>3.1.4-beta.2</0> 版本。

预发布版本通常包含一些不兼容的修改，而且通常你也不会愿意匹配到指定版本之外的预发布版本，所以上述匹配规则很有用。

版本范围进阶

连字符范围

连字符范围（例如 `2.0.0 - 3.1.4` ）标明了一个包含集合。数字 `0` 会被用来填充版本号中缺少的那些部分（例如 `0.4` 或 `2` ）。

版本范围	扩展的版本范围
`2.0.0 - 3.1.4`	`>=2.0.0 <=3.1.4`
`0.4 - 2`	`>=0.4.0 <=2.0.0`

X 范围

字符 `X` 、 `x` 或者 `*` 都可以作为通配符，用于填充部分或全部版本号。

版本范围	扩展的版本范围
`*`	`>=0.0.0` （任意版本）
`2.x`	`>=2.0.0 <3.0.0` （匹配主要版本）
`3.1.x`	`> = 3.1.0 < 3.2.0` （匹配主要和次要版本）

被省略的那部分版本号默认为 `x` 范围。

版本范围	扩展的版本范围
`` (empty string)	`*` 或 `> = 0.0.0`
`2`	`2.x.x` 或 `> = 2.0.0 < 3.0.0`
`3.1`	`3.1.x` 或 `> = 3.1.0 < 3.2.0`

~ 字符范围

同时使用字符 `~` 和次版本号，表明允许 `修订号` 变更。同时使用字符 `~` 和主版本号，表明允许 `次版本号` 变更。

版本范围	扩展的版本范围
`~3.1.4`	`>=3.1.4 <3.2.0`
`~3.1`	`3.1.x` 或 `> = 3.1.0 < 3.2.0`
`~3`	`3.x` 或 `> = 3.0.0 < 4.0.0`

> 注意：在波浪号范围中指定预发布版本将只匹配和它完整版本号相同的预发布版本。 例如，版本范围 `~3.1.4-beta.2` 会匹配 `3.1.4-beta.4` 但不匹配 `3.1.5-beta.2`，因为 `major.minor.patch` 版本不同。

^ 字符范围

字符 `^` 表明不会修改版本号中的第一个非零数字，`3.1.4` 里的 `3` 或者 `0.4.2` 里的 `4`。

版本范围	扩展的版本范围
`^3.1.4`	`>=3.1.4 <4.0.0`
`^0.4.2`	`>=0.4.2 <0.5.0`
`^0.0.2`	`>=0.0.2 <0.0.3`

> 注意：`yarn add [package-name]` 命令默认使用 `^` 范围。

版本号中缺少的部分将被 `0` 填充，且在匹配时这些位置允许改变。

版本范围	扩展的版本范围
`^0.0.x`	`>=0.0.0 <0.1.0`
`^0.0`	`>=0.0.0 <0.1.0`
`^0.x`	`>=0.0.0 <1.0.0`
`^0`	`>=0.0.0 <1.0.0`

更多信息

- 语义版本系统的完整规范：`node-semver` README。
- 用真实的包测试语义版本：npm semver calculator。

> 原文：*https://yarnpkg.com/zh-Hans/docs/dependency-versions*

`package.json` 文件里的 `resolutions` 字段用于解析选择性版本，可以通过此功能自定义依赖版本。 这通常需要手动编辑 `yarn.lock` 文件。

你为什么要这么做？

- 有些时候，项目会依赖一个不常更新的包，但这个包又依赖另一个需要立即升级的包。 这时候，如果这个（不常更新的）包的依赖列表里不包含需要升级的包的新版本，那就只能等待作者升级，没别的办法。
- 项目的子依赖（依赖的依赖）需要紧急安全更新，来不及等待直接依赖更新。
- 项目的直接依赖还可以正常工作但已经停止维护，这时子依赖需要更新。 同时，你清楚子依赖的更新不会影响现有系统，但是又不想通过 `fork` 的方式来升级直接依赖。
- 项目的直接依赖定义了过于宽泛的子依赖版本范围，恰巧这其中的某个版本有问题，这时你想要把子依赖限制在某些正常工作的版本范围里。

如何使用它？

在 `package.json` 文件里添加 `resolutions` 字段，用于覆盖版本定义：

package.json

```
{
  "name": "project",
  "version": "1.0.0",
  "dependencies": {
    "left-pad": "1.0.0",
    "c": "file:../c-1",
    "d2": "file:../d2-1"
  },
  "resolutions": {
    "d2/left-pad": "1.1.1",
    "c/**/left-pad": "1.1.2"
  }
}
```

之后执行 `yarn install`。

提示和技巧

- 如果定义了无效的版本解析规则，比如错写了无效的包名，会收到警告。
- 如果定义的版本解析的版本号、版本范围无效，也会收到警告。
- 如果定义的版本解析规则的版本号、版本范围与原始版本范围不兼容，同样会收到警告。

限制和警告

- 嵌套包机制（`Nested packages`）可能会挂。
- 这是一项新功能，因此会在某些极端情况下挂掉。

> 原文：*https://yarnpkg.com/zh-Hans/docs/selective-version-resolutions*

配置你的包

Yarn 使用 `package.json` 文件来标识每个包，并配置 yarn 在那个包里的运行方式。

`pet-kitten` 包的配置可以在 `pet-kitten/package.json` 找到：

```
{
  "name": "pet-kitten",
  "version": "0.1.0",
  "main": "pet.js",
  "dependencies": {
    "hand": "1.0.0"
  }
}
```

使用 yarn.lock 文件来固化依赖

除了 `package.json` 文件，yarn 也使用 <0>yarn.lock</0> 文件来确保依赖解析又快又稳。 你无需编辑这个文件，yarn自己搞定。

为了保证你应用的行为保持一致，你应该把 `yarn.lock` 文件提交到代码仓库。

> 原文: *https://yarnpkg.com/zh-Hans/docs/configuration*

重要字段

`name` 和 `version` 是 `package.json` 文件里最重要的两个字段，没有它们你的包无法被安装。 `name` 和 `version` 字段一起用来创建一个唯一 id。

name

```
{
  "name": "my-awesome-package"
}
```

这是你的包的名字。它在 URL 中、作为命令行参数、作为 `node_modules` 里的目录名使用。

```
yarn add [name]
```

```
node_modules/[name]


https://registry.npmjs.org/[name]/-/[name]-[version].tgz
```

规则

- 必须少于或等于 214 个字符（对于限定域的包来说包括 `@scope/`）。
- 不能以句点（`.`）或者下划线（`_`）开头。
- 名字里不能有大写字母。

- 必须只使用 URL 安全的字符。
  Tips

- 不要使用和 Node.js 核心模块相同的名字。

- 不要在名字里包含 `js` 或者 `node` 单词。
- 短小精悍，让人看到名字就大概了解包的功能，记住它也会被用在 `require()` 调用里。
- 保证名字在 registry 里是唯一的。

version

```
{
  "version": "1.0.0"
}
```

包的当前版本号。

信息类字段

description

```
{
  "description": "我的包的简短描述"
}
```

Description 是帮助使用者了解包的功能的字符串，包管理器也会把这个字符串作为搜索关键词。

keywords

```
{
  "keywords": ["short", "relevant", "keywords", "for", "searching"]
}
```

关键字是一个字符串数组，当在包管理器里搜索包时很有用。

license

```
{
  "license": "MIT",
  "license": "(MIT or GPL-3.0)",
  "license": "SEE LICENSE IN LICENSE_FILENAME.txt",
  "license": "UNLICENSED"
}
```

所有包都应该指定许可证，以便让用户了解他们是在什么授权下使用此包，以及此包还有哪些附加限制。

鼓励使用开源（OSI-approved）许可证，除非你有特别的原因不用它。 如果你开发的包是你工作的一部分，最好和公司讨论后再做决定。

license字段必须是以下之一：

- 如果你使用标准的许可证，需要一个有效地 SPDX 许可证标识。
- 如果你用多种标准许可证，需要有效的 SPDX 许可证表达式2.0语法表达式。
- 如果你使用非标准的许可证，一个 `SEE LICENSE IN <文件名>` 字符串指向你的包里顶级目录的一个 `<文件名>`。
- 如果你不想在任何条款下授权其他人使用你的私有或未公开的包，一个 `UNLICENSED` 字符串。

链接类字段

各种指向项目文档、issues 上报，以及代码托管网站的链接字段。

homepage

```
{
  "homepage": "https://your-package.org"
}
```

homepage 是包的项目主页或者文档首页。

bugs

```
{
  "bugs": "https://github.com/user/repo/issues"
}
```

问题反馈系统的 URL，或者是 email 地址之类的链接。用户通过该途径向你反馈问题。

repository

```
{
  "repository": { "type": "git", "url": "https://github.com/user/repo.git" },
  "repository": "github:user/repo",
  "repository": "gitlab:user/repo",
  "repository": "bitbucket:user/repo",
  "repository": "gist:a1b2c3d4e5f"
}
```

repository 是代码托管的位置。

项目维护类字段

项目的维护者。

author

```
{
  "author": {
    "name": "Your Name",
    "email": "you@example.com",
    "url": "http://your-website.com"
  },
  "author": "Your Name <you@example.com> (http://your-website.com)"
}
```

作者信息，一个人。

contributors

```
{
  "contributors": [
    { "name": "Your Friend", "email": "friend@example.com", "url": "http://friends-website.com" }
    { "name": "Other Friend", "email": "other@example.com", "url": "http://other-website.com" }
```

```
  ],
  "contributors": [
    "Your Friend <friend@example.com> (http://friends-website.com)",
    "Other Friend <other@example.com> (http://other-website.com)"
  ]
}
```

贡献者信息，可能很多人。

文件类信息

指定包含在项目中的文件，以及项目的入口文件。

files

```
{
  "files": ["filename.js", "directory/", "glob/*.{js,json}"]
}
```

项目包含的文件，可以是单独的文件、整个文件夹，或者通配符匹配到的文件。

main

```
{
  "main": "filename.js"
}
```

项目的入口文件。

bin

```
{
  "bin": "bin.js",
  "bin": {
    "command-name": "bin/command-name.js",
    "other-command": "bin/other-command"
  }
}
```

随着项目一起被安装的可执行文件。

man

```
{
  "man": "./man/doc.1",
```

```
  "man": ["./man/doc.1", "./man/doc.2"]
}
```

和项目相关的文档页面（man page）。

directories

```
{
  "directories": {
    "lib": "path/to/lib/",
    "bin": "path/to/bin/",
    "man": "path/to/man/",
    "doc": "path/to/doc/",
    "example": "path/to/example/"
  }
}
```

当你的包安装时，你可以指定确切的位置来放二进制文件、man pages、文档、例子等。

任务类字段

包里还可以包含一些可执行脚本或者其他配置信息。

scripts

```
{
  "scripts": {
    "build-project": "node build-project.js"
  }
}
```

脚本是定义自动化开发相关任务的好方法，比如使用一些简单的构建过程或开发工具。 在 `"scripts"` 字段里定义的脚本，可以通过 `yarn run <script>` 命令来执行。 例如，上述 `build-project` 脚本可以通过 `yarn run build-project` 调用，并执行 `node build-project.js` 。

有一些特殊的脚本名称。 如果定义了 `preinstall` 脚本，它会在包安装前被调用。 出于兼容性考虑， `install` 、 `postinstall` 和 `prepublish` 脚本会在包完成安装后被调用。

`start` 脚本的默认值为 `node server.js` 。

config

```
{
  "config": {
    "port": "8080"
  }
}
```

配置你的脚本的选项或参数。

依赖描述类字段

你的包很可能依赖其他包。你可以在你的 `package.json` 文件里指定那些依赖。

dependencies

```
{
  "dependencies": {
    "package-1": "^3.1.4"
  }
}
```

这些是你的包的开发版和发布版都需要的依赖。

> 你可以指定一个确切的版本、一个最小的版本（比如 `>=`）或者一个版本范围（比如 `>= … <`）。

devDependencies

```
{
  "devDependencies": {
    "package-2": "^0.4.2"
  }
}
```

这些是只在你的包开发期间需要，但是生产环境不会被安装的包。

peerDependencies

```
{
  "peerDependencies": {
    "package-3": "^2.7.18"
  }
}
```

平行依赖允许你说明你的包和其他包版本的兼容性。

optionalDependencies

```
{
  "optionalDependencies": {
    "package-5": "^1.6.1"
  }
}
```

可选依赖可以用于你的包，但不是必需的。如果可选包没有找到，安装还可以继续。

bundledDependencies

```
{
  "bundledDependencies": ["package-4"]
}
```

打包依赖是发布你的包时将会一起打包的一个包名数组。

flat

```
{
  "flat": true
}
```

如果你的包只允许给定依赖的一个版本，你想强制和命令行上 `yarn install —flat` 相同的行为，把这个值设为 `true` 。

请注意，如果你的 `package.json` 包含 `"flat": true` 并且其它包依赖你的包（比如你在构建一个库，而不是应用)， 其它那些包也需要在它们的 `package.json` 加上 `"flat": true` ，或者在命令行上用 `yarn install —flat` 安装。

resolutions

```
{
  "resolutions": {
    "transitive-package-1": "0.0.29",
    "transitive-package-2": "file:./local-forks/transitive-package-2",
    "dependencies-package-1/transitive-package-3": "^2.1.1"
  }
}
```

允许您覆盖特定嵌套依赖项的版本。 有关完整规范，请参见选择性版本解析 RFC。

注意， `yarn install —flat` 命令将会自动在 `package.json` 文件里加入 `resolutions` 字段。

系统

你可以提供和你的包关联的系统级的信息，比如操作系统兼容性之类。

engines

```
{
  "engines": {
```

```
    "node": ">=4.4.7 <7.0.0",
    "zlib": "^1.2.8",
    "yarn": "^0.14.0"
  }
}
```

engines 指定使用你的包客户必须使用的版本，这将检查 `process.versions` 以及当前 yarn 版本。

This check follows normal semver rules with one exception. It allows prerelease versions to match semvers that do not explicitly specify a prerelease. For example, `1.4.0-rc.0` matches `>=1.3.0`, while it would not match a typical semver check.

os

```
{
  "os": ["darwin", "linux"],
  "os": ["!win32"]
}
```

此选项指定你的包的操作系统兼容性，它会检查 `process.platform` 。

cpu

```
{
  "cpu": ["x64", "ia32"],
  "cpu": ["!arm", "!mips"]
}
```

使用这个选项指定你的包将只能在某些 CPU 体系架构上运行，这会检查 `process.arch` 。

发布

private

```
{
  "private": true
}
```

如果你不想你的包发布到包管理器，设置为 `true` 。

publishConfig

```
{
  "publishConfig": {
    ...
```

```
  }
}
```

这些配置值将在你的包发布时使用。比如，你可以给包打标签。

> 原文：*https://yarnpkg.com/zh-Hans/docs/package-json*

在 `process.env` 中定义的环境变量允许您配置 Yarn 的附加功能。

CHILD_CONCURRENCY

```
process.env.CHILD_CONCURRENCY=#number#
```

控制并行执行的子进程数以构建节点模块。

将此数字设置为 1，将会按顺序构建节点模块，这样可以避免在 Windows 中使用 node-gyp 时出现链接器错误。

> 原文：*https://yarnpkg.com/zh-Hans/docs/envvars*

`.yarnrc` 文件允许你配置更多的 Yarn 功能。 也可以用 `config` 命令来配置这些选项。 Yarn 会把你的 `.yarnrc` merge 进文件树里。

yarn-offline-mirror

```
1. yarn-offline-mirror "./packages-cache"
```

离线维护你的包，这样可以让你的构建过程更加稳定。在这里看更多信息。

必须是个相对路径，或者用 `false` 来禁用镜像（默认值）。

yarn-offline-mirror-pruning

```
1. yarn-offline-mirror-pruning true
```

控制离线镜像的自动删除。在此获取更多信息。

值必须是一个布尔值，默认值为 `false` 。

yarn-path

```
1. yarn-path "./bin/yarn"
```

为了执行 yarn，可以让 yarn 指向另一个 Yarn 的二进制文件。 如果把 Yarn 绑定到存储库中，并且让每个人都使用同样的的版本以保持一致性的话，这是非常有用的。 这会在 Yarn 1.0 中介绍，所以所有开发者必须安装 Yarn >= 1.0 的版本。

该值必须是相对路径，或者用 `false` 来禁用它（默认）。

disable-self-update-check

```
1. disable-self-update-check true
```

在安装包时，如果你的 CLI 命令过时了，Yarn 会提供更新命令。可以在这里禁用该检查。

值必须是一个布尔值，默认值为 `false` 。

child-concurrency

```
1. child-concurrency #number#
```

控制并行执行的子进程数以构建节点模块。

将此数字设置为 1，将会按顺序构建节点模块，这样可以避免在 Windows 中使用 node-gyp 时出现链接器错误。

CLI 参数

在 .yarnrc 设置 `–<command>.<flag> <value>` 和执行 `yarn <command> –<flag> <value>` 一样。

示例：

```
$> cat .yarnrc
--install.check-files true
```

和执行 `yarn install –check-files` 一样

示例 2:

```
$> cat .yarnrc
--cache-folder /tmp/yarn-cache/

$> yarn cache dir
/tmp/yarn-cache/v1
```

> 原文: *https://yarnpkg.com/zh-Hans/docs/yarnrc*

为了跨机器安装得到一致的结果，Yarn 需要比你配置在 `package.json` 中的依赖列表更多的信息。 Yarn 需要准确存储每个安装的依赖是哪个版本。

为了做到这样，Yarn 使用一个你项目根目录里的 `yarn.lock` 文件。这些 “lockfile” 看起来像这样的：

```
# THIS IS AN AUTOGENERATED FILE. DO NOT EDIT THIS FILE DIRECTLY.
# yarn lockfile v1
package-1@^1.0.0:
  version "1.0.3"
  resolved "https://registry.npmjs.org/package-1/-/package-1-1.0.3.tgz#a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0"
package-2@^2.0.0:
  version "2.0.1"
  resolved "https://registry.npmjs.org/package-2/-/package-2-2.0.1.tgz#a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0"
  dependencies:
    package-4 "^4.0.0"
package-3@^3.0.0:
  version "3.1.9"
  resolved "https://registry.npmjs.org/package-3/-/package-3-3.1.9.tgz#a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0"
  dependencies:
    package-4 "^4.5.0"
package-4@^4.0.0, package-4@^4.5.0:
  version "4.6.3"
  resolved "https://registry.npmjs.org/package-4/-/package-4-2.6.3.tgz#a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0"
```

这可以媲美其他像 Bundler 或 Cargo 这样的包管理器的 lockfiles。它类似于 npm 的 `npm-shrinkwrap.json`，然而他并不是有损的并且它能创建可重现的结果。

用 Yarn 管理

你的 `yarn.lock` 文件是自动产生的，而且应该完全被 Yarn 管理。 当你用 Yarn CLI 增加 / 升级 / 删除依赖，它将自动更新你的 `yarn.lock` 文件。 不要直接编辑这个文件，那样很容易弄坏某些东西。

只用于当前包

安装期间 Yarn 将只使用顶级 `yarn.lock` 文件，并会忽略任何依赖里面的 `yarn.lock` 文件。 顶级 `yarn.lock` 包含 Yarn 需要锁定的整个依赖树里全部包版本的所有信息。

提交到版本控制系统

所有 `yarn.lock` 文件应该被提交到版本控制系统（例如 git 或者 mercurial）。 这允许 Yarn 跨所有机器安装相同的依赖树，无论它是你同事的笔记本还是 CI 服务器。

Framework and library authors should also check `yarn.lock` into source control. 别担心发布 `yarn.lock` 文件，因为它对库的用户不会有任何作用。

See https://yarnpkg.com/blog/2016/11/24/lockfiles-for-all/.

> 原文：*https://yarnpkg.com/zh-Hans/docs/yarn-lock*

照着这篇博文来创建离线镜像。

> 原文：*https://yarnpkg.com/zh-Hans/docs/offline-mirror*

After you configure your offline mirror, Yarn will automatically add new package tarballs to the mirror. 但是，它不会自动移除不再在 `yarn.lock` 中提到的压缩包。 For example, if you `$ yarn remove` a dependency, the tarball will remain in the mirror, even if no other dependencies have it as a sub-dependency. This behavior can be desirable in a setting where many projects share the same mirror, but when that is not the case, you may want to have Yarn remove unnecessary tarballs.

To turn on automating pruning, set `yarn-offline-mirror-pruning` to `true` in your `.yarnrc`:

```
$ yarn config set yarn-offline-mirror-pruning true
```

Now, tarballs will be removed when appropriate. The end result is that `package.json`, `node_modules`, `yarn.lock`, and the offline mirror should all remain perfectly in sync whenever you change your project’s dependencies.

> 原文: *https://yarnpkg.com/zh-Hans/docs/prune-offline-mirror*

工作区是设置你的软件包体系结构的一种新方式，默认情况下从 Yarn 1.0 开始使用。它允许你可以使用这种方式安装多个软件包， 就是只需要运行一次 `yarn install` 便可将所有依赖包全部安装。

你为什么要这么做？

- 你的依赖包可以链接在一起，这意味着你的工作区可以相互依赖，同时始终使用最新的可用代码。 这也是一个比 `yarn link` 更好的机制，因为它只影响你工作区的依赖树，而不会影响整个系统。
- 你所有的项目依赖将被安装在一起，这样可以让 Yarn 来更好地优化它们。
- Yarn 将使用一个单一的 lock 文件，而不是每个项目多有一个，这意味着更少的冲突和更容易进行代码检查。

如何使用它？

在 `package.json` 文件中添加以下内容，从现在开始，我们将此目录称为 “工作区根目录”：

package.json

```
{
  "private": true,
  "workspaces": ["workspace-a", "workspace-b"]
}
```

请注意， `private: true` 是必需的！工作区本身不应当被发布出去，所以我们添加了这个安全措施以确保它不会被意外暴露。

创建这个文件后，再创建两个名为 `workspace-a` 和 `workspace-b` 的子文件夹。 在每个文件夹里面，创建一个具有以下内容的 `package. json` 文件：

workspace-a/package.json:

```
{
  "name": "workspace-a",
  "version": "1.0.0",

  "dependencies": {
    "cross-env": "5.0.5"
  }
}
```

workspace-b/package.json:

```
{
  "name": "workspace-b",
  "version": "1.0.0",

  "dependencies": {
    "cross-env": "5.0.5",
```

```
    "workspace-a": "1.0.0"
  }
}
```

最后，在某个地方运行 `yarn install` ，当然最好是在工作区根目录里面。如果一切正常，你现在应该有一个类似这样的文件层次结构：

```
/package.json
/yarn.lock

/node_modules
/node_modules/cross-env
/node_modules/workspace-a -> /workspace-a

/workspace-a/package.json
/workspace-b/package.json
```

就是这样 ！ `workspace-b <code>需要一个在</code> workspace-a <code>中的文件，现在将直接使用当前项目内部的文件，而不是直接去从 Github 上面取。 </code> cross-env` 包已正确去重并放在项目的根目录下，让 `workspace-a` 和 `workspace-b` 可以一起使用这个包。

它与 Lerna 的对比怎么样？

Yarn 的工作区是诸如 Lerna 这样的工具可以（并且正在）利用的底层机制。 它们将永远不会试图提供像 Lerna 那么高级的功能，但通过实现该解决方案的核心逻辑和 Yarn 内部的连接步骤，我们希望能够提供新的用法并提高性能。

提示和技巧

- `workspaces` 字段是一个数组，其中包含到每个工作区的路径。 如果用这种方式来跟踪每个工作区可能是比较乏味的，所以这个字段也接受了通配符模式！ 例如，Babel 通过这个 `packages/*` 指令引用他们的所有包。
- 工作区的稳定性足以在 large-scale 应用程序中使用，并且不应改变常规安装的运行方式，但如果你认为它们正在破坏某些东西，则可以通过将以下内容添加到你的 Yarnrc 文件中来禁用它们：

```
  workspaces-experimental false
```

限制和警告

- 包层级在你的工作区和用户得到的内容之间将有所不同（工作区依赖将提升到文件系统层次结构中）。 对这个层级的假设已经是危险的，因为提升过程不是标准化的，所以理论上没有什么新东西。
- 在上面的示例中，如果 `workspace-b` 依赖于 `workspace-a` 的包，但是引用的是不同的版本，那么依赖包将从 Github 安装，而不是从本地文件系统链接。 这是因为一些软件包实际上需要使用以前的版本，以建立新的版本（Babel 是其中之一）。

- 在工作区中发布包时要留心。 如果你正准备发布下一个版本，并且你决定引用一个新依赖但忘了在 `package.json` 中声明，你的测试仍可能在本地通过（如果其他包已经把那个引用下载到了项目根目录）。然而其他从源中拉取包的用户就不行了，由于依赖列表现在是不完整的，他们没办法下载那个新依赖。 目前没有办法在这种情况下抛出警告。
- 工作区必须是项目根目录的子目录。你不能也不应当引用位于项目目录之外的工作区。
- 目前尚不支持嵌套工作区。

> 原文：*https://yarnpkg.com/zh-Hans/docs/workspaces*

在 Yarn 贡献代码、处理请求和社区交流中的规则、指南和文档。

Yarn 是一个社区驱动型项目，同时接受来自公司的赞助。 每个人可以参与并且向 Yarn 贡献，我们承诺建立一个开放包容的社区。

社区的每个希望通过贡献代码、文档、支持或其他形式贡献的成员必须阅读和遵守 代码规范。

> 原文: *https://yarnpkg.com/zh-Hans/org*

我们的承诺

为了培养一个开放热情的环境，我们作为贡献者和维护者承诺，创造一个让参与我们项目和社区的每个人不被骚扰的体验，不管年龄、体型、残障、种族特点、性别身份和表达、经验水平、国籍、个人外表、种族、宗教信仰或性身份和取向。

我们的标准

为创建积极的环境做贡献的例子包括：

- 使用欢迎和包容的语言
- 尊重不同的观点和经验
- 优雅的接受建设性意见
- 聚焦于什么是对社区最好的

- 能与其他社区成员产生共鸣
  参与者不可接受的行为例子包括：

- 使用色情化的语言或图像和和不受欢迎的性关注或求爱

- 恶意破坏、侮辱或贬损言论和个人或政治攻击
- 公开或私下的骚扰
- 在没有明确的允许下发布他人的私人信息，如住址或电子邮件
- 在专业背景里有理由被认为不合适的其它行为。

我们的责任

项目的维护者负责澄清可接受的行为标准，和预期对任何不可接受的行为采取适当和公平的纠正行动。

项目维护者有权并有责任删除、编辑或拒绝不符合本守则的评论、提交、代码、`wiki` 编辑、`issues` 和其它贡献，或者临时或永久性禁止其他的他们认为不合适的行为、威胁、无礼、不良的行为。

范围

当个人代表该项目或其社区时，本守则对项目空间内部和公共空间都适用。 代表项目或社区的例子包括使用官方的项目 `e-mail` 地址，通过官方社交媒体账号发布东西，或者在线上或线下社交场合担任代表。 项目的代表可能由项目维护者进一步定义和澄清。

实施

辱骂、骚扰或其它不被接受的行为情况可以联系 `sebmck@gmail.com` 报告给项目团队。 所有投诉都将审查和调查，并会答复对该情况认为必要和适当的结果。 项目团队有义务对涉及到的事件报告者保密。 具体执行政策的进一步细节可能会单独发布。

善意的不遵守或执行守则的项目维护者可能面临项目的其他领导成员决定的临时或永久性后果。

发行商

此行为守则是改编自 Contributor Covenant 1.4版本, 链接 http://contributor-covenant.org/version/1/4

> 原文: *https://yarnpkg.com/zh-Hans/org/code-of-conduct*

欢迎参与贡献，无论大小。贡献之前，请先阅读 代码规则。

Find things to work on

需要帮助改进的issue除了被打上了 `help wanted` 的标签的以外，也被分为如下几类：

- cat-bug
- cat-feature
- cat-chore
- cat-performance
  以上是你可以使用的主要类别。 另外，我们还会使用 `high-priority` 和 `good first issue` 标签来分别表示issue的重要程度和容易上手的程度。 如果issue没有被打上 `triaged` 标签，或者被打上了 `needs-confirmation` 、 `needs-repro-script` 、 `needs-discussion` 之一，那么你或许可以先别急着动手修改。

你可以从以下链接开始你的贡献之路：

- 适合上手修复的缺陷
- 适合上手实现的特性
- 需要帮忙的重要问题
- 需要重现方法的问题
- Issues need triaging

If you would like to start triaging issues, one easy way to get started is to subscribe to yarn on CodeTriage.

构建

```
1. yarn run build
```

```
1. yarn run watch
```

使用本地构建版本

```
1. alias yarn="node /path/to/yarn/lib/cli/index.js"
```

测试

```
1. yarn run test
```

```
1. yarn run lint
```

合并请求

我们积极欢迎您的合并请求。

- Fork 仓库并从 `master` 创建你的分支。
- 如果你添加了一些需要测试的代码，请同时添加一些测试方法、用例。
- 如果你修改了 API，请更新文档。
- 确保测试集运行通过。
- 确保你的代码通过代码风格检查（Code Lint）。

授权协议

要贡献给 Yarn，你要同意你的贡献在 BSD协议下授权。

> 原文: *https://yarnpkg.com/zh-Hans/org/contributing*

原文档采用英文撰写，由 Github 托管，译文通过 Crowdin 管理。

如果想为翻译做贡献，你可以在这里加入我们的团队。

只要配置好了账户，你就可以点击要翻译的语言开始翻译。

一定要记住， 在Crowdin所做的一切都要遵守 行为守则。

> 原文: *https://yarnpkg.com/zh-Hans/org/translations*

发布新版本

根据您推送的发布类型（主要、次要或补丁），步骤稍有不同

要发布新的修补程序版本（例如 从 0.28.1 到 0.28.2）

- 将所有需要的修改Cherry-pick到 `-stable` 分支（例如，`0.28-stable`）
- 确保切换到本地的`-stable`分支上
- 运行 `npm version patch` 递增版本号并创建 Git Commit和Tag
- 运行 `git push origin 0.xx-stable –follow-tags` （将 `0.xx-stable` 换为正确的分支名称）

To release a new minor or major version (eg. from 0.28.x to 0.29.0)

- 确保当前 master 分支在 Circle, Travis 和 AppVeyor 上是绿色的。
- 确保位于 `master` 分支上，且本地 Yarn 是最新版本。
- 运行 `./scripts/release-branch.sh`。这将：
  - 创建 `0. xx-stable` 分支和 `0. 0` 标签
  - Bump `master` to the next minor version (eg. after releasing `0.29.0`, master will be bumped to `0.30.0`)
  - 将其全部推送到 `origin`

将 RC 版本设置为稳定版

Once an RC has been tested by the community for a while and all major bugs have been ironed out, it can be marked as stable. To do this, go to https://release.yarnpkg.com/ and click the “Promote RC to stable” button.

注意：有一个能够访问该页面的白名单用户列表。 If a maintainer is missing from the whitelist, you can modify it here.

调试发布版本

有时会出错，这里是一些常见问题的调试方法：

我已经提交了标签，但站点依然指向旧版本

检查任何缺失制品的 GitHub 发布版本。 The release scripts do not bump the version number on the site until both the Linux **and** the Windows artifacts have been attached to the release.

缺少 Linux 制品（.tar.gz、.deb 等）

如果它已经失败，检查 CircleCI build 并重新运行它。 如果构建成功，则检查 webhook 日志 中的任何错误

信息。

缺失 Windows 制品（.msi）

Check the AppVeyor build and re-run it if it has failed. If the build succeeded, check the webhook logs for any errors.

所有制品都关联到发布了，但站点依然指向旧版本

检查 yarn-version 的 Jenkins 构建任务 ， 看看它是否失败。

如何手动执行

Most of the release has been automated and is fairly straightforward. 通常来说，您只需阅读到这里。 However, if the release tooling ever breaks (or if you like doing things the hard way), you can manually perform the release steps.

创建新发布

- 运行 `yarn build-dist && yarn build-deb` 生成发布包、Debian 软件包和 RPM 包
- 在 Windows 上运行 `yarn build-dist && yarn build-win-installer` 来构建 Windows 安装程序
- GPG 签名 `.tar.gz` 和 `.js` 制品 sh gpg -u 9D41F3C3 –armor –detach-sign yarn-0.xx.xx.tar.gz 这将生成

`.asc` files that you should also attach to the release

- Authenticode sign the `.msi` 制品 sh osslsigncode sign -t http://timestamp.digicert.com -n “Yarn Installer” -i https://yarnpkg.com/ -pkcs12 yarn-20161122.pfx -readpass yarn-20161122.key -h sha1 -in yarn-0.xx.xx-unsigned.msi -out yarn-0.xx.xx.msi osslsigncode sign -t http://timestamp.digicert.com -n “Yarn Installer” -i https://yarnpkg.com/ -pkcs12 yarn-20161122.pfx -readpass yarn-20161122.key -nest -h sha2 -in yarn-0.xx.xx.msi -out yarn-0.xx.xx.msi

- Create new release on GitHub, and attach all artifacts. For the MSI, ensure you attach the **signed** version!

- Publish the tarball to npm: `npm publish ./artifacts/yarn-v0.xx.xx.tar.gz`
- 执行下面的 post-release 步骤

RC 转入稳定版

- 修改 GitHub 发布，将其标为稳定版
- 运行 `npm dist-tag add yarn@0.xx.xx latest` （其中 `0.xx.xx` 是要发布的版本号）
- 运行下面的 post-release 步骤

Post-release

- Bump version number in _config.yml on the website
- Run `./scripts/build-chocolatey.ps1` to push to Chocolatey
- Run `./scripts/update-homebrew.sh` to push to Homebrew
- Debian 和 CentOS 仓库应该在5分钟内自动更新到最新版本（盯住那些提交）

> 原文: *https://yarnpkg.com/zh-Hans/org/release-process*

这个目前正在 yarnpkg/yarn#274 讨论。

> 原文：*https://yarnpkg.com/zh-Hans/org/governance*