概述

Cargo 用于 Rust 的包管理。Cargo 下载 Rust 包的依赖项，编译包，制作可分发的包，并将它们上传到 crates.io--Rust 社区的包注册表。

安装

在 linux 上使用以下命令进行安装：

```
curl https://sh.rustup.rs -sSf | sh
```

在 windows 上在 https://win.rustup.rs/下载 rustup-init.exe，进行安装。

General Commands

cargo

Rust 语言的包管理器和构建工具。

使用

cargo 的使用方式有如下一些：

- cargo [options] command [args]
- cargo [options] --version
- cargo [options] --list
- cargo [options] --help
- cargo [options] --explain code

选项

有以下一些选项：

- -V，--version：打印版本信息并退出。 如果与--verbose 一起使用，则打印额外信息。
- -v，--verbose：尽可能详细的输出。
- --list：列出所有已安装的 Cargo 子命令。 如果与--verbose 一起使用，则打印额外信息。
- --explain code：打印出错误消息的详细解释（例如，E0004）。
- -q，--quiet：没有输出打印到标准输出。
- --color when：控制何时使用彩色输出。 有效值：
  - auto（默认）：自动检测终端是否支持颜色。
  - always：始终显示颜色。
  - never：从不显示颜色。
- --frozen，--locked：要求 Cargo.lock 文件是最新的。如果锁文件丢失，或者需要更新，Cargo 会报错退出。
- --offline：防止 Cargo 出于任何原因访问网络。
- +toolchain：如果 Cargo 的第一个参数以 + 开头，它将被解释为 rustup 工具链名称（例如 +stable 或 +nightly）。
- -h，--help：打印帮助信息。
- -Z flag：不稳定（仅限+nightly）标志。 运行 cargo -Z help 了解详情。

示例

cargo help

打印给定子命令的帮助消息。

使用

cargo help [subcommand]

选项

见 cargo 选项。

示例

获取 build 命令的帮助信息：

cargo help build

cargo version

显示 Cargo 的版本。

使用

```
cargo version [options]
```

选项

见 cargo 选项。

示例

显示版本信息：

```
cargo version
```

Package Commands

cargo init

在现有目录中创建一个新的 Cargo 包。

使用

```
cargo init [options] [path]
```

选项

有以下一些选项：

- --bin：创建一个带有二进制目标的包 (src/main.rs)。 这是默认行为。
- --lib：创建一个带有库目标 (src/lib.rs) 的包。
- --edition edition：指定要使用的 Rust 版本。 默认值为 2018。可能的值：2015、2018、2021。
- --name name：设置包名。 默认为目录名称。
- --vcs vcs：为给定的版本控制系统初始化一个新的 VCS 存储库，或者根本不初始化任何版本控制。如果未指定，则默认为 git。
- --registry registry：将 Cargo.toml 中的 package.publish 字段设置为给定的注册表名称，这将限制仅发布到该注册表。

示例

在当前目录下创建一个二进制的 Cargo 包：

```
cargo init
```

cargo new

创建一个新的目录，并在其中创建新的 Cargo package。

使用

```
cargo new [options] path
```

选项

有以下一些选项：

- --bin：创建一个带有二进制目标的包 (src/main.rs)。 这是默认行为。
- --lib：创建一个带有库目标 (src/lib.rs) 的包。
- --edition edition：指定要使用的 Rust 版本。 默认值为 2018。可能的值：2015、2018、2021。
- --name name：设置包名。 默认为目录名称。
- --vcs vcs：为给定的版本控制系统初始化一个新的 VCS 存储库，或者根本不初始化任何版本控制。如果未指定，则默认为 git。
- --registry registry：将 Cargo.toml 中的 package.publish 字段设置为给定的注册表名称，这将限制仅发布到该注册表。

示例

在当前目录下创建一个二进制的 Cargo 包：

```
cargo new helloworld
```

cargo install

管理 Cargo 的本地安装的二进制包。 只能安装具有可执行文件 [[bin]] 或 [[example]] 目标的软件包，并且所有可执行文件都安装到安装根目录的 bin 文件夹中。安装根目录选择如下：

- 选项--root
- 环境变量 CARGO_INSTALL_ROOT
- Cargo 配置值 install.root
- 环境变量 CARGO_HOME
- $HOME/.cargo

使用

```
cargo install [options] crate...
cargo install [options] --path path
cargo install [options] --git url [crate...]
cargo install [options] --list
```

选项

有以下一些选项：

- --vers version，--version version：指定要安装的版本。支持语义化版本，例如~1.2。
- --git url：用于安装指定 crate 的 Git URL。
- --branch branch：从 git 安装时使用的分支。
- --tag tag：从 git 安装时使用的 tag。
- --rev sha：从 git 安装时使用的修订版本号。
- --path path：要安装的本地 crate 的文件系统路径。
- --list：列出所有已安装的软件包及其版本。
- -f，--force：强制覆盖现有的 crate 或二进制文件。
- --no-track：默认情况下，Cargo 使用存储在安装根目录中的元数据文件跟踪已安装的包。该标志告诉 Cargo 不要使用或创建该文件。
- --bin name...：仅安装指定的二进制文件。
- --bins：仅安装二进制文件。
- --example name...：仅安装指定的示例文件。
- --examples：仅安装示例文件。
- --root dir：指定包安装到的目录。
- --registry registry：要使用的注册表的名称。
- --index index：要使用的注册表索引的 URL。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --target triple：为给定的架构安装。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --debug：使用开发配置文件而不是发布配置文件进行构建。

示例

从 crates.io 安装或升级包：

```
cargo install ripgrep
```

在当前目录安装或重新安装包：

```
cargo install --path .
```

查看已安装的软件包列表：

```
cargo install --list
```

cargo uninstall

删除使用 cargo install 安装的包。默认情况下，删除所有安装的二进制文件。

使用

```
cargo uninstall [options] [spec...]
```

选项

有以下一些选项：

- -p，--package spec...：需要卸载的包。

- --bin name...：只卸载指定的二进制。
- --root dir：卸载软件包的目录。

示例

卸载以前安装的软件包：
```
cargo uninstall ripgrep
```

cargo search

在 https://crates.io 上对 crates 执行搜索。

使用

```
cargo search [options] [query...]
```

选项

有以下一些选项：
- --limit limit：限制结果数量（默认：10，最大：100）。
- --index index：要使用的注册表索引的 URL。
- --registry registry：要使用的注册表的名称。

示例

从 crates.io 搜索包：
```
cargo search serde
```

Build Commands

cargo build

编译本地包及其所有依赖项。

使用

```
cargo build [options]
```

选项

有以下一些选项：
- -p spec...，--package spec...：仅构建指定的包。
- --workspace：构建工作区中的所有成员。
- --exclude SPEC...：排除指定的包。 必须与 --workspace 标志一起使用。
- --lib：构建库。
- --bin name...：构建指定的二进制文件。该标志可以多次指定。
- --bins：构建所有二进制文件。
- --example name...：构建指定的示例。这个标志可以被多次指定。
- --examples：构建所有示例。
- --test name...：构建指定的集成测试。这个标志可以被多次指定。
- --tests：构建在 cargo.toml 中设置了 test=true 的目标的单元测试，以及集成测试。
- --bench name...：构建指定的基准测试。这个标志可以被多次指定。
- --benches：构建在 cargo.toml 中设置了 bench=true 的目标的基准测试。
- --all-targets：构建所有目标。 这相当于指定 --lib --bins --tests --benches --examples。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --target triple：为给定的架构构建。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --release：使用发布配置文件而不是开发配置文件进行构建。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --out-dir directory：将最终编译文件复制到此目录。
- --build-plan：将一系列 JSON 消息输出到 stdout，指示运行构建的命令。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

构建本地包及其所有依赖项：
```
cargo build
```

使用优化构建：
```
cargo build --release
```

cargo run

运行本地包的二进制文件或示例，并传递额外的运行参数给运行的文件。

使用

```
cargo run [options] [-- run_args]
```

选项

有以下一些选项：
- -p spec...，--package spec...：仅运行指定的包。
- --bin name：运行指定的二进制文件。
- --example name：运行指定的示例。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --target triple：为给定的架构运行。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --release：使用发布配置文件而不是开发配置文件进行构建。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

构建本地包并运行其主要目标（假设只有一个二进制文件）：
```
cargo run
```

运行带有额外参数的示例：
```
cargo run --example exname -- --exoption exarg1 exarg2
```

cargo rustc

编译当前包，并将额外的选项传递给编译器

使用

```
cargo rustc [options] [-- args]
```

选项

有以下一些选项：
- -p spec...，--package spec...：仅构建指定的包。
- --lib：构建库。
- --bin name...：构建指定的二进制文件。该标志可以多次指定。
- --bins：构建所有二进制文件。
- --example name...：构建指定的示例。这个标志可以被多次指定。
- --examples：构建所有示例。
- --test name...：构建指定的集成测试。这个标志可以被多次指定。
- --tests：构建在 cargo.toml 中设置了 test=true 的目标的单元测试，以及集成测试。
- --bench name...：构建指定的基准测试。这个标志可以被多次指定。
- --benches：构建在 cargo.toml 中设置了 bench=true 的目标的基准测试。
- --all-targets：构建所有目标。 这相当于指定 --lib --bins --tests --benches --examples。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --target triple：为给定的架构构建。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --release：使用发布配置文件而不是开发配置文件进行构建。
- --target-dir directory：生成的编译文件和中间文件的目录。

- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

检查包（不包括依赖项）是否使用了不安全的代码：
```
cargo rustc --lib -- -D unsafe-code
```

在夜间频道编译器上尝试一个实验性标志，例如打印每种类型的大小：
```
cargo rustc --lib -- -Z print-type-sizes
```

cargo test

编译并执行包的单元和集成测试。

使用

```
cargo test [options] [testname] [-- test-options]
```

选项

有以下一些选项：
- --no-run：编译，但不运行测试。
- --no-fail-fast：无论是否失败，都运行所有测试。 如果没有这个标志，Cargo 将在第一个可执行文件失败后退出。
- -p spec...，--package spec...：仅测试指定的包。
- --workspace：测试工作区中的所有成员。
- --exclude SPEC...：排除指定的包。 必须与 --workspace 标志一起使用。
- --lib：测试库。
- --bin name...：测试指定的二进制文件。该标志可以多次指定。
- --bins：测试所有二进制文件。
- --example name...：测试指定的示例。这个标志可以被多次指定。
- --examples：测试所有示例。
- --test name...：测试指定的集成测试。这个标志可以被多次指定。
- --tests：测试在 cargo.toml 中设置了 test=true 的目标的单元测试，以及集成测试。
- --bench name...：测试指定的基准测试。这个标志可以被多次指定。
- --benches：测试在 cargo.toml 中设置了 bench=true 的目标的基准测试。
- --all-targets：测试所有目标。 这相当于指定 --lib --bins --tests --benches --examples。
- --doc：仅测试库的文档。 这不能与其他目标选项混合使用。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --target triple：为给定的架构测试 。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --release：使用发布配置文件而不是开发配置文件进行测试。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

执行当前包的所有单元和集成测试：
```
cargo test
```

仅运行名称与过滤器字符串匹配的测试：
```
cargo test name_filter
```

仅在特定集成测试中运行特定测试：
```
cargo test --test int_test_name -- modname::test_name
```

cargo bench

编译并执行基准测试。

使用

```
cargo bench [options] [benchname] [-- bench-options]
```

选项

有以下一些选项：

- --no-run：编译，但不运行基准测试。
- --no-fail-fast：无论是否失败，都运行所有基准测试。 如果没有这个标志，Cargo 将在第一个可执行文件失败后退出。
- -p spec...，--package spec...：仅测试指定的包。
- --workspace：测试工作区中的所有成员。
- --exclude SPEC...：排除指定的包。 必须与 --workspace 标志一起使用。
- --lib：测试库。
- --bin name...：测试指定的二进制文件。该标志可以多次指定。
- --bins：测试所有二进制文件。
- --example name...：测试指定的示例。这个标志可以被多次指定。
- --examples：测试所有示例。
- --test name...：测试指定的集成测试。这个标志可以被多次指定。
- --tests：测试在 cargo.toml 中设置了 test=true 的目标的单元测试，以及集成测试。
- --bench name...：测试指定的基准测试。这个标志可以被多次指定。
- --benches：测试在 cargo.toml 中设置了 bench=true 的目标的基准测试。
- --all-targets：测试所有目标。 这相当于指定 --lib --bins --tests --benches --examples。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --target triple：为给定的架构测试 。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

构建并执行当前包的所有基准测试：

```
cargo bench
```

仅在特定基准目标内运行特定基准：

```
cargo bench --bench bench_name -- modname::some_benchmark
```

cargo check

检查本地包及其所有依赖项是否有错误。

使用

```
cargo check [options]
```

选项

有以下一些选项：

- -p spec...，--package spec...：仅检查指定的包。
- --workspace：检查工作区中的所有成员。
- --exclude SPEC...：排除指定的包。 必须与 --workspace 标志一起使用。
- --lib：检查库。
- --bin name...：检查指定的二进制文件。该标志可以多次指定。
- --bins：检查所有二进制文件。
- --example name...：检查指定的示例。这个标志可以被多次指定。
- --examples：检查所有示例。
- --test name...：检查指定的集成测试。这个标志可以被多次指定。
- --tests：检查在 cargo.toml 中设置了 test=true 的目标的单元测试，以及集成测试。
- --bench name...：检查指定的基准测试。这个标志可以被多次指定。
- --benches：检查在 cargo.toml 中设置了 bench=true 的目标的基准测试。
- --all-targets：检查所有目标。 这相当于指定 --lib --bins --tests --benches --examples。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --target triple：为给定的架构检查。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --release：使用发布配置文件而不是开发配置文件进行检查。
- --profile name：更改检查行为。目前仅支持测试，这将在启用 #[cfg(test)] 属性的情况下进行检查。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

检查本地包是否有错误：
```
cargo check
```

检查所有目标，包括单元测试：
```
cargo check --all-targets --profile=test
```

cargo fix

自动从警告等诊断中获取 rustc 的建议，并将它们应用到您的源代码中。

使用

```
cargo fix [options]
```

选项

有以下一些选项：

- --broken-code：修复代码，即使存在有编译器错误。
- --edition：应用会将代码更新到下一版本的更改。
- --edition-idioms：应用会将代码更新为当前版本的首选样式的建议。
- --allow-no-vcs：即使未检测到 VCS，也要修复代码。
- --allow-dirty：即使工作目录发生变化，也要修复代码。
- --allow-staged：即使工作目录已暂存更改，也要修复代码。
- -p spec...，--package spec...：仅修复指定的包。
- --workspace：修复工作区中的所有成员。
- --exclude SPEC...：排除指定的包。 必须与 --workspace 标志一起使用。
- --lib：修复库。
- --bin name...：修复指定的二进制文件。该标志可以多次指定。
- --bins：修复所有二进制文件。
- --example name...：修复指定的示例。这个标志可以被多次指定。
- --examples：修复所有示例。
- --test name...：修复指定的集成测试。这个标志可以被多次指定。
- --tests：修复在 cargo.toml 中设置了 test=true 的目标的单元测试，以及集成测试。
- --bench name...：修复指定的基准测试。这个标志可以被多次指定。
- --benches：修复在 cargo.toml 中设置了 bench=true 的目标的基准测试。
- --all-targets：修复所有目标。 这相当于指定 --lib --bins --tests --benches --examples。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --target triple：为给定的架构修复。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --release：使用发布配置文件而不是开发配置文件进行修复。
- --profile name：更改修复行为。目前仅支持测试，这将在启用 #[cfg(test)] 属性的情况下进行检查。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

将编译器建议应用于本地包：
```
cargo fix
```

更新一个包，为下一个版本做准备：
```
cargo fix --edition
```

为当前版本应用建议：
```
cargo fix --edition-idioms
```

cargo fetch

下载包的依赖项。如果 Cargo.lock 文件可用，此命令将确保所有 git 依赖项和/或注册表依赖项都已下载并在本地可用。如果 Cargo.lock 文件不可用，则此命令将在获取依赖项之前生成 Cargo.lock 文件。

使用

```
cargo fetch [options]
```

选项

有以下一些选项：
- --target triple：为给定的架构拉取依赖。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

获取所有依赖项：
```
cargo fetch
```

cargo clean

删除生成的文件。

使用

```
cargo clean [options]
```

选项

有以下一些选项：
- -p spec...，--package spec...：仅移除指定的包的生成的文件。
- --doc：此选项将导致 cargo clean 仅删除目标目录中的 doc 目录。
- --release：清除使用 release 或 benchmark profile 构建的所有文件。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --target triple：为给定的架构清除。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

删除整个目标目录：
```
cargo clean
```

仅删除使用 release 或 benchmark profile 构建的文件：
```
argo clean --release
```

cargo doc

构建包的文档

使用

```
cargo doc [options]
```

选项

有以下一些选项：
- --open：构建后在浏览器中打开文档。 这将使用您的默认浏览器，除非您在 BROWSER 环境变量中定义另一个浏览器。
- --no-deps：不要为依赖构建文档。
- --document-private-items：在文档中包含非公开项目。
- -p spec...，--package spec...：仅为指定的包生成文档。
- --workspace：为工作区中的所有成员生成文档。
- --exclude SPEC...：排除指定的包。 必须与 --workspace 标志一起使用。
- --lib：为库生成文档。
- --bin name...：为指定的二进制文件生成文档。该标志可以多次指定。
- --bins：为所有二进制文件生成文档。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --target triple：为给定的架构生成文档 。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --release：使用发布配置文件而不是开发配置文件进行文档生成。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

构建本地包文档及其依赖项并输出到目标/文档：
```
cargo doc
```

cargo rustdoc

使用指定的自定义标志构建包的文档。

使用

```
cargo rustdoc [options] [-- args]
```

选项

有以下一些选项：
- --open：构建后在浏览器中打开文档。 这将使用您的默认浏览器，除非您在 BROWSER 环境变量中定义另一个浏览器。
- -p spec...，--package spec...：仅为指定的包生成文档。
- --lib：为库生成文档。
- --bin name...：为指定的二进制文件生成文档。该标志可以多次指定。
- --bins：为所有二进制文件生成文档。
- --example name...：为指定的示例生成文档。这个标志可以被多次指定。
- --examples：为所有示例生成文档。
- --test name...：为指定的集成测试生成文档。这个标志可以被多次指定。
- --tests：为在 cargo.toml 中设置了 test=true 的目标的单元测试以及集成测试生成文档。
- --bench name...：为指定的基准测试生成文档。这个标志可以被多次指定。
- --benches：为在 cargo.toml 中设置了 bench=true 的目标的基准测试生成文档。
- --all-targets：为所有目标生成文档。 这相当于指定 --lib --bins --tests --benches --examples。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --target triple：为给定的架构生成文档。 默认为主机架构。 三元组的一般格式是<arch><sub>-<vendor>-<sys>-<abi>。
- --release：使用发布配置文件而不是开发配置文件进行生成文档。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

使用包含在给定文件中的自定义 CSS 构建文档：
```
cargo rustdoc --lib -- --extend-css extra.css
```

Manifest Commands

cargo update

更新本地锁文件中记录的依赖项。

使用

```
cargo update [options]
```

选项

有以下一些选项：
- -p spec...，--package spec...：仅更新指定的包。
- --aggressive：当与 -p 一起使用时，spec 的依赖项也被强制更新。不能与 --precise 一起使用。
- --precise precise：与 -p 一起使用时，允许您指定要设置包的特定版本号。 如果包来自 git 存储库，则这可以是 git 修订版。
- -w，--workspace：尝试更新工作区中定义的包。
- --dry-run：显示将更新的内容，但实际上并不写入锁定文件。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

更新锁文件中的所有依赖项：

```
cargo update
```

仅更新特定的依赖项：
```
cargo update -p foo -p bar
```

将特定依赖项设置为特定版本：
```
cargo update -p foo --precise 1.2.3
```

cargo generate-lockfile

为当前包或工作空间创建 Cargo.lock 锁文件。 如果锁文件已经存在，它将使用每个包的最新可用版本重建。

使用

```
cargo generate-lockfile [options]
```

选项

有以下一些选项：
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

创建或更新当前包或工作区的锁文件：
```
cargo generate-lockfile
```

cargo locate-project

使用 Cargo.toml 清单的完整路径将 JSON 对象打印到标准输出。

使用

```
cargo locate-project [options]
```

选项

有以下一些选项：
- --workspace：在工作空间的根目录中找到 Cargo.toml，而不是当前工作空间成员。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

根据当前目录显示清单的路径：
```
cargo locate-project
```

cargo metadata

显示有关当前包的机器可读元数据。

使用

```
cargo metadata [options]
```

选项

有以下一些选项：
- --no-deps：仅输出有关工作区成员的信息，不获取依赖项。
- --format-version version：指定要使用的输出格式的版本。 目前 1 是唯一可能的值。
- --filter-platform triple：过滤解析输出以仅包含给定目标三元组的依赖项。 如果没有此标志，则解析包括所有目标。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

输出关于当前包的 JSON：
```
cargo metadata --format-version=1
```

cargo pkgid

获取包的完全合格的包 ID。

使用

```
cargo pkgid [options] [spec]
```

选项

有以下一些选项：
- -p spec...，--package spec...：获取指定包的包 ID。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

检索 foo 包的包规范：
```
cargo pkgid foo
```

检索 foo 版本 1.0.0 的包规范：
```
cargo pkgid foo:1.0.0
```

从 crates.io 检索 foo 的包规范：
```
cargo pkgid https://github.com/rust-lang/crates.io-index#foo
```

从本地包中检索 foo 的包规范：
```
cargo pkgid file:///path/to/local/package#foo
```

cargo tree

显示可视化的树形依赖图。

使用

```
cargo tree [options]
```

选项

有以下一些选项：
- -i spec，--invert spec：显示给定包的反向依赖关系。 此标志将反转树并显示依赖于给定包的包。
- --no-dedupe：不要删除重复的依赖项。
- -d，--duplicates：仅显示具有多个版本的依赖项。
- -e kinds，--edges kinds：要显示的依赖项类型。 采用逗号分隔的值列表：
  - all — 显示所有边类型。
  - normal — 显示正常的依赖关系。
  - build — 显示构建依赖项。
  - dev — 显示开发依赖项。
  - features - 显示每个依赖项启用的特性。如果这是唯一给出的种类，那么它将自动包含其他依赖项种类。
  - no-normal — 不包括正常的依赖项。
  - no-build — 不包括构建依赖项。
  - no-dev — 不包含开发依赖项。
- --target triple：过滤与给定目标三元组匹配的依赖项。 默认为主机平台。 使用值 all 包括所有目标。
- --charset charset：选择用于树的字符集。 有效值为“utf8”或“ascii”。 默认为“utf8”。
- -f format，--format format：为每个包设置格式字符串。 默认值为“{p}”。以下字符串将替换为相应的值：
  - {p} — 包名称。
  - {l} — 软件包许可证。
  - {r} — 包存储库 URL。
  - {f} — 已启用的软件包功能的逗号分隔列表。
- --prefix prefix：设置每行的显示方式。 前缀值可以是以下之一：
  - indent（默认）——显示每一行缩进为树。
  - depth — 显示为列表，在每个条目之前打印数字深度。

    - none — 显示为平面列表。
- -p spec...，--package spec...：只显示指定的包
- --workspace：显示工作区中的所有成员。
- --exclude SPEC...：排除指定的包。 必须与 --workspace 标志一起使用。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

显示当前目录中包的树：
```
cargo tree
```

显示所有依赖于 syn 包的包：
```
cargo tree -i syn
```

显示每个包上启用的特性：
```
cargo tree --format "{p} {f}"
```

显示多次构建的所有包。 如果树中出现多个与 semver 不兼容的版本（如 1.0.0 和 2.0.0），就会发生这种情况：
```
cargo tree -d
```

描述为什么为 syn 包启用特性：
```
cargo tree -e features -i syn
```

cargo vendor

将项目的所有 crates.io 和 git 依赖项供应到 <path> 的指定目录中。 此命令完成后，<path> 指定的供应商目录将包含来自指定依赖项的所有远程源。

使用

```
cargo vendor [options] [path]
```

选项

有以下一些选项：
- -s manifest，--sync manifest：为工作区指定额外的 Cargo.toml 清单，这些清单也应该被供应商提供并同步到输出。
- --no-delete：在 vendoring 时不要删除“vendor”目录，而是保留 vendor 目录的所有现有内容
- --respect-source-config：在.cargo/config.toml 中读取它并在从 crates.io 下载 crates 时使用它，而不是忽略 [source] 配置。
- --versioned-dirs：通常添加版本只是为了消除同一包的多个版本的歧义。此选项会导致“供应商”目录中的所有目录都进行版本控制。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

Vendor 所有依赖项到本地“vendor”文件夹中：
```
cargo vendor
```

Vendor 所有依赖项到本地“third-party/vendor”文件夹中：
```
cargo vendor third-party/vendor
```

Vendor 当前工作区以及另一个“Vendor”：
```
cargo vendor -s ../path/to/Cargo.toml
```

cargo verify-project

检查 crate 清单的正确性。

使用

```
cargo verify-project [options]
```

选项

有以下一些选项：

- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

检查当前工作区是否有错误：
```
cargo verify-project
```

Publishing Commands

cargo login

在本地保存注册表中的 API 令牌。

使用

```
cargo login [options] [token]
```

选项

有以下一些选项：
- --registry registry：要使用的注册表的名称。

示例

将 API 令牌保存到磁盘：
```
cargo login
```

cargo owner

修改注册表中 crate 的所有者。 crate 的所有者可以上传新版本和拉取旧版本。非团队所有者也可以修改所有者的设置，所以要小心！

使用

```
cargo owner [options] --add login [crate]
cargo owner [options] --remove login [crate]
cargo owner [options] --list [crate]
```

选项

有以下一些选项：
- -a，--add login...：邀请给定的用户或团队作为所有者。
- -r，--remove login...：删除给定的用户或团队的所有者权限。
- -l，--list：列出 crate 的所有者。
- --token token：身份验证时使用的 API 令牌。 这会覆盖存储在凭据文件中的令牌。
- --index index：要使用的注册表索引的 URL。
- --registry registry：要使用的注册表的名称。

示例

列出包的所有者：
```
cargo owner --list foo
```

邀请所有者加入包：
```
cargo owner --add username foo
```

从包中删除所有者：
```
cargo owner --remove username foo
```

cargo package

将本地包组装成一个可分发的 tarball。在当前目录中使用包的源代码创建一个可分发的压缩 .crate 文件。 生成的文件将存储在 target/package 目录中。

使用

```
cargo package [options]
```

选项

有以下一些选项：
- -l，--list：打印包含在包中的文件，无需进行制作。
- --no-verify：不要通过构建它们来验证内容。
- --no-metadata：忽略有关缺乏可供人类使用的元数据（例如描述或许可证）的警告。
- --allow-dirty：允许打包未提交的 VCS 更改的工作目录。
- --target triple：为给定架构的打包。默认为主机架构。三元组的一般格式为<arch><sub>-<vendor>-<sys>-<abi>。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

创建当前包的压缩 .crate 文件：
```
cargo package
```

cargo publish

将包上传到注册表。此命令将在当前目录中使用包的源代码创建一个可分发的压缩 .crate 文件，并将其上传到注册表。

使用

```
cargo publish [options]
```

选项

有以下一些选项：
- --dry-run：在不上传的情况下执行所有检查。
- --token token：身份验证时使用的 API 令牌。 这会覆盖存储在凭据文件中的令牌。
- --no-verify：不要通过构建它们来验证内容。
- --no-metadata：忽略有关缺乏可供人类使用的元数据（例如描述或许可证）的警告。
- --allow-dirty：允许打包未提交的 VCS 更改的工作目录。
- --target triple：为给定架构的打包。默认为主机架构。三元组的一般格式为<arch><sub>-<vendor>-<sys>-<abi>。
- --target-dir directory：生成的编译文件和中间文件的目录。
- --features features：要激活的特性列表，以空格或逗号分隔。
- --all-features：激活所选包的所有特性。
- --no-default-features：不要激活所选包的默认特性。
- --manifest-path path：Cargo.toml 文件的路径。默认情况下，Cargo 在当前目录或任何父目录中搜索 Cargo.toml 文件。

示例

发布当前包：
```
cargo publish
```

cargo yank

从索引中删除推送的 crate。此命令不会删除任何数据，并且仍然可以通过注册表的下载链接下载 crate。

使用

```
cargo yank [options] --vers version [crate]
```

选项

有以下一些选项：
- --vers version：要拉出或取消拉出的版本。
- --undo：撤消 yank，将版本放回索引中。
- --token token：身份验证时使用的 API 令牌。 这会覆盖存储在凭据文件中的令牌。

- --index index：要使用的注册表索引的 URL。
- --registry registry：要使用的注册表的名称。

示例

从注册表索引中撤回一个 crate：
cargo yank --vers 1.0.7 foo