The Cargo Book

Cargo is the Rust *package manager*. Cargo downloads your Rust package's dependencies, compiles your packages, makes distributable packages, and uploads them to crates.io, the Rust community's *package registry*. You can contribute to this book on GitHub.

Sections

Getting Started

To get started with Cargo, install Cargo (and Rust) and set up your first crate.

Cargo Guide

The guide will give you all you need to know about how to use Cargo to develop Rust packages.

Cargo Reference

The reference covers the details of various areas of Cargo.

Frequently Asked Questions

Getting Started

To get started with Cargo, install Cargo (and Rust) and set up your first crate.

- Installation
- First steps with Cargo

Installation

Install Rust and Cargo

The easiest way to get Cargo is to install the current stable release of Rust by using `rustup`. Installing Rust using `rustup` will also install `cargo`.

On Linux and macOS systems, this is done as follows:

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

It will download a script, and start the installation. If everything goes well, you'll see this appear:

```
Rust is installed now. Great!
```

On Windows, download and run rustup-init.exe. It will start the installation in a console and present the above message on success.

After this, you can use the `rustup` command to also install `beta` or `nightly` channels for Rust and Cargo.

For other installation options and information, visit the install page of the Rust website.

Build and Install Cargo from Source

Alternatively, you can build Cargo from source.

First Steps with Cargo

To start a new package with Cargo, use `cargo new`:

```
$ cargo new hello_world
```

Cargo defaults to `--bin` to make a binary program. To make a library, we'd pass `--lib`.

Let's check out what Cargo has generated for us:

```
$ cd hello_world
$ tree .
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

This is all we need to get started. First, let's check out `Cargo.toml`:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[dependencies]
```

This is called a **manifest**, and it contains all of the metadata that Cargo needs to compile your package.

Here's what's in `src/main.rs`:

```
fn main() {
    println!("Hello, world!");
}
```

Cargo generated a "hello world" for us. Let's compile it:

```
$ cargo build
   Compiling hello_world v0.1.0 (file:///path/to/package/hello_world)
```

And then run it:

```
$ ./target/debug/hello_world
Hello, world!
```

We can also use `cargo run` to compile and then run it, all in one step:

```
$ cargo run
     Fresh hello_world v0.1.0 (file:///path/to/package/hello_world)
   Running `target/hello_world`
Hello, world!
```

Going further

For more details on using Cargo, check out the Cargo Guide

Cargo Guide

This guide will give you all that you need to know about how to use Cargo to develop Rust packages.

Why Cargo Exists

Cargo is a tool that allows Rust packages to declare their various dependencies and ensure that you'll always get a repeatable build.

To accomplish this goal, Cargo does four things:

- Introduces two metadata files with various bits of package information.
- Fetches and builds your package's dependencies.
- Invokes `rustc` or another build tool with the correct parameters to build your package.
- Introduces conventions to make working with Rust packages easier.

Creating a New Package

To start a new package with Cargo, use `cargo new`:

```
$ cargo new hello_world --bin
```

We're passing `--bin` because we're making a binary program: if we were making a library, we'd pass `--lib`. This also initializes a new `git` repository by default. If you don't want it to do that, pass `--vcs none`.

Let's check out what Cargo has generated for us:

```
$ cd hello_world
$ tree .
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

Let's take a closer look at `Cargo.toml`:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[dependencies]
```

This is called a **manifest**, and it contains all of the metadata that Cargo needs to compile your package.

Here's what's in `src/main.rs`:

```
fn main() {
    println!("Hello, world!");
}
```

Cargo generated a "hello world" for us. Let's compile it:

```
$ cargo build
   Compiling hello_world v0.1.0 (file:///path/to/package/hello_world)
```

And then run it:

```
$ ./target/debug/hello_world
Hello, world!
```

We can also use `cargo run` to compile and then run it, all in one step (You won't see the `Compiling` line if you have not made any changes since you last compiled):

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///path/to/package/hello_world)
     Running `target/debug/hello_world`
Hello, world!
```

You'll now notice a new file, `Cargo.lock`. It contains information about our dependencies. Since we don't have any yet, it's not very interesting.

Once you're ready for release, you can use `cargo build --release` to compile your files with optimizations turned on:

```
$ cargo build --release
   Compiling hello_world v0.1.0 (file:///path/to/package/hello_world)
```

`cargo build --release` puts the resulting binary in `target/release` instead of `target/debug`.

Compiling in debug mode is the default for development-- compilation time is shorter since the compiler doesn't do optimizations, but the code will run slower. Release mode takes longer to compile, but the code will run faster.

Working on an Existing Cargo Package

If you download an existing package that uses Cargo, it's really easy to get going.

First, get the package from somewhere. In this example, we'll use `rand` cloned from its repository on GitHub:

```
$ git clone https://github.com/rust-lang-nursery/rand.git
$ cd rand
```

To build, use `cargo build`:

```
$ cargo build
   Compiling rand v0.1.0 (file:///path/to/package/rand)
```

This will fetch all of the dependencies and then build them, along with the package.

Dependencies

crates.io is the Rust community's central package registry that serves as a location to discover and download packages. `cargo` is configured to use it by default to find requested packages.

To depend on a library hosted on crates.io, add it to your `Cargo.toml`.

Adding a dependency

If your `Cargo.toml` doesn't already have a `[dependencies]` section, add that, then list the crate name and version that you would like to use. This example adds a dependency of the `time` crate:

```
[dependencies]
time = "0.1.12"
```

The version string is a semver version requirement. The specifying dependencies docs have more information about the options you have here.

If we also wanted to add a dependency on the `regex` crate, we would not need to add `[dependencies]` for each crate listed. Here's what your whole `Cargo.toml` file would look like with dependencies on the `time` and `regex` crates:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[dependencies]
time = "0.1.12"
regex = "0.1.41"
```

Re-run `cargo build`, and Cargo will fetch the new dependencies and all of their dependencies, compile them all, and update the `Cargo.lock`:

```
$ cargo build
      Updating crates.io index
   Downloading memchr v0.1.5
   Downloading libc v0.1.10
   Downloading regex-syntax v0.2.1
   Downloading memchr v0.1.5
   Downloading aho-corasick v0.3.0
   Downloading regex v0.1.41
     Compiling memchr v0.1.5
     Compiling libc v0.1.10
     Compiling regex-syntax v0.2.1
     Compiling memchr v0.1.5
     Compiling aho-corasick v0.3.0
     Compiling regex v0.1.41
     Compiling hello_world v0.1.0 (file:///path/to/package/hello_world)
```

Our `Cargo.lock` contains the exact information about which revision of all of these dependencies we used.

Now, if `regex` gets updated, we will still build with the same revision until we choose to `cargo update`.

You can now use the `regex` library in `main.rs`.

```
use regex::Regex;

fn main() {
    let re = Regex::new(r"^\d{4}-\d{2}-\d{2}$").unwrap();
    println!("Did our date match? {}", re.is_match("2014-01-01"));
}
```

Running it will show:

```
$ cargo run
   Running `target/hello_world`
Did our date match? true
```

Package Layout

Cargo uses conventions for file placement to make it easy to dive into a new Cargo package:

```
.
├── Cargo.lock
├── Cargo.toml
├── benches
│   └── large-input.rs
├── examples
│   └── simple.rs
├── src
│   ├── bin
│   │   └── another_executable.rs
│   ├── lib.rs
│   └── main.rs
└── tests
    └── some-integration-tests.rs
```

- `Cargo.toml` and `Cargo.lock` are stored in the root of your package (*package root*).
- Source code goes in the `src` directory.
- The default library file is `src/lib.rs`.
- The default executable file is `src/main.rs`.
- Other executables can be placed in `src/bin/*.rs`.
- Integration tests go in the `tests` directory (unit tests go in each file they're testing).
- Examples go in the `examples` directory.
- Benchmarks go in the `benches` directory.

These are explained in more detail in the manifest description.

Cargo.toml vs Cargo.lock

`Cargo.toml` and `Cargo.lock` serve two different purposes. Before we talk about them, here's a summary:

- `Cargo.toml` is about describing your dependencies in a broad sense, and is written by you.
- `Cargo.lock` contains exact information about your dependencies. It is maintained by Cargo and should not be manually edited.

If you're building a non-end product, such as a rust library that other rust packages will depend on, put `Cargo.lock` in your `.gitignore`. If you're building an end product, which

are executable like command-line tool or an application, or a system library with crate-type of `staticlib` or `cdylib`, check `Cargo.lock` into `git`. If you're curious about why that is, see "Why do binaries have `Cargo.lock` in version control, but not libraries?" in the FAQ.

Let's dig in a little bit more.

`Cargo.toml` is a **manifest** file in which we can specify a bunch of different metadata about our package. For example, we can say that we depend on another package:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
rand = { git = "https://github.com/rust-lang-nursery/rand.git" }
```

This package has a single dependency, on the `rand` library. We've stated in this case that we're relying on a particular Git repository that lives on GitHub. Since we haven't specified any other information, Cargo assumes that we intend to use the latest commit on the `master` branch to build our package.

Sound good? Well, there's one problem: If you build this package today, and then you send a copy to me, and I build this package tomorrow, something bad could happen. There could be more commits to `rand` in the meantime, and my build would include new commits while yours would not. Therefore, we would get different builds. This would be bad because we want reproducible builds.

We could fix this problem by putting a `rev` line in our `Cargo.toml`:

```
[dependencies]
rand = { git = "https://github.com/rust-lang-nursery/rand.git", rev = "9f35b8e"
}
```

Now our builds will be the same. But there's a big drawback: now we have to manually think about SHA-1s every time we want to update our library. This is both tedious and error prone.

Enter the `Cargo.lock`. Because of its existence, we don't need to manually keep track of the exact revisions: Cargo will do it for us. When we have a manifest like this:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
rand = { git = "https://github.com/rust-lang-nursery/rand.git" }
```

Cargo will take the latest commit and write that information out into our `Cargo.lock` when we build for the first time. That file will look like this:

```
[[package]]
name = "hello_world"
version = "0.1.0"
dependencies = [
 "rand 0.1.0 (git+https://github.com/rust-lang-
nursery/rand.git#9f35b8e439eeedd60b9414c58f389bdc6a3284f9)",
]

[[package]]
name = "rand"
version = "0.1.0"
source = "git+https://github.com/rust-lang-
nursery/rand.git#9f35b8e439eeedd60b9414c58f389bdc6a3284f9"
```

You can see that there's a lot more information here, including the exact revision we used to build. Now when you give your package to someone else, they'll use the exact same SHA, even though we didn't specify it in our `Cargo.toml`.

When we're ready to opt in to a new version of the library, Cargo can re-calculate the dependencies and update things for us:

```
$ cargo update           # updates all dependencies
$ cargo update -p rand   # updates just “rand”
```

This will write out a new `Cargo.lock` with the new version information. Note that the argument to `cargo update` is actually a Package ID Specification and `rand` is just a short specification.

Tests

Cargo can run your tests with the `cargo test` command. Cargo looks for tests to run in two places: in each of your `src` files and any tests in `tests/`. Tests in your `src` files should be unit tests, and tests in `tests/` should be integration-style tests. As such, you'll need to import your crates into the files in `tests`.

Here's an example of running `cargo test` in our package, which currently has no tests:

```
$ cargo test
   Compiling rand v0.1.0 (https://github.com/rust-lang-nursery/rand.git#9f35b8e)
   Compiling hello_world v0.1.0 (file:///path/to/package/hello_world)
     Running target/test/hello_world-9c2b65bbb79eabce

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

If our package had tests, we would see more output with the correct number of tests.

You can also run a specific test by passing a filter:

```
$ cargo test foo
```

This will run any test with `foo` in its name.

`cargo test` runs additional checks as well. For example, it will compile any examples you've included and will also test the examples in your documentation. Please see the testing guide in the Rust documentation for more details.

Continuous Integration

Travis CI

To test your package on Travis CI, here is a sample `.travis.yml` file:

```
language: rust
rust:
  - stable
  - beta
  - nightly
matrix:
  allow_failures:
    - rust: nightly
```

This will test all three release channels, but any breakage in nightly will not fail your overall build. Please see the Travis CI Rust documentation for more information.

GitLab CI

To test your package on GitLab CI, here is a sample `.gitlab-ci.yml` file:

```
stages:
  - build

rust-latest:
  stage: build
  image: rust:latest
  script:
    - cargo build --verbose
    - cargo test --verbose

rust-nightly:
  stage: build
  image: rustlang/rust:nightly
  script:
    - cargo build --verbose
    - cargo test --verbose
  allow_failure: true
```

This will test on the stable channel and nightly channel, but any breakage in nightly will not fail your overall build. Please see the GitLab CI for more information.

builds.sr.ht

To test your package on sr.ht, here is a sample `.build.yml` file. Be sure to change `<your repo>` and `<your project>` to the repo to clone and the directory where it was cloned.

```
image: archlinux
packages:
  - rustup
sources:
  - <your repo>
tasks:
  - setup: |
      rustup toolchain install nightly stable
      cd <your project>/
      rustup run stable cargo fetch
  - stable: |
      rustup default stable
      cd <your project>/
      cargo build --verbose
      cargo test --verbose
  - nightly: |
      rustup default nightly
      cd <your project>/
      cargo build --verbose ||:
      cargo test --verbose  ||:
  - docs: |
      cd <your project>/
      rustup run stable cargo doc --no-deps
      rustup run nightly cargo doc --no-deps ||:
```

This will test and build documentation on the stable channel and nightly channel, but any breakage in nightly will not fail your overall build. Please see the builds.sr.ht documentation for more information.

Cargo Home

The "Cargo home" functions as a download and source cache. When building a crate, Cargo stores downloaded build dependencies in the Cargo home. You can alter the location of the Cargo home by setting the `CARGO_HOME` environmental variable. The home crate provides an API for getting this location if you need this information inside your Rust crate. By default, the Cargo home is located in `$HOME/.cargo/`.

Please note that the internal structure of the Cargo home is not stabilized and may be subject to change at any time.

The Cargo home consists of following components:

Files:

- `config` Cargo's global configuration file, see the config entry in the reference.
- `credentials` Private login credentials from `cargo login` in order to log in to a registry.
- `.crates.toml` This hidden file contains package information of crates installed via `cargo install`. Do NOT edit by hand!

Directories:

- `bin` The bin directory contains executables of crates that were installed via `cargo install` or `rustup`. To be able to make these binaries accessible, add the path of the directory to your `$PATH` environment variable.
- `git` Git sources are stored here:
  - `git/db` When a crate depends on a git repository, Cargo clones the repo as a bare repo into this directory and updates it if necessary.
  - `git/checkouts` If a git source is used, the required commit of the repo is checked out from the bare repo inside `git/db` into this directory. This provides the compiler with the actual files contained in the repo of the commit specified for that dependency. Multiple checkouts of different commits of the same repo are possible.
- `registry` Packages and metadata of crate registries (such as crates.io) are located here.
  - `registry/index` The index is a bare git repository which contains the metadata (versions, dependencies etc) of all available crates of a registry.
  - `registry/cache` Downloaded dependencies are stored in the cache. The crates are compressed gzip archives named with a `.crate` extension.
  - `registry/src` If a downloaded `.crate` archive is required by a package, it is unpacked into `registry/src` folder where rustc will find the `.rs` files.

Caching the Cargo home in CI

To avoid redownloading all crate dependencies during continuous integration, you can cache the `$CARGO_HOME` directory. However, caching the entire directory is often inefficient as it will contain downloaded sources twice. If we depend on a crate such as `serde 1.0.92` and cache the entire `$CARGO_HOME` we would actually cache the sources twice, the `serde-1.0.92.crate` inside `registry/cache` and the extracted `.rs` files of serde inside `registry/src`. The can unnecessarily slow down the build as downloading, extracting, recompressing and reuploading the cache to the CI servers can take some time.

It should be sufficient to only cache the following directories across builds:

- `bin/`

- `registry/index/`
- `registry/cache/`
- `git/db/`

Vendoring all dependencies of a project

See the `cargo vendor` subcommand.

Clearing the cache

In theory, you can always remove any part of the cache and Cargo will do its best to restore sources if a crate needs them either by reextracting an archive or checking out a bare repo or by simply redownloading the sources from the web.

Alternatively, the cargo-cache crate provides a simple CLI tool to only clear selected parts of the cache or show sizes of its components in your command-line.

Build cache

Cargo shares build artifacts among all the packages of a single workspace. Today, Cargo does not share build results across different workspaces, but a similar result can be achieved by using a third party tool, sccache.

To setup `sccache`, install it with `cargo install sccache` and set `RUSTC_WRAPPER` environmental variable to `sccache` before invoking Cargo. If you use bash, it makes sense to add `export RUSTC_WRAPPER=sccache` to `.bashrc`. Alternatively, you can set `build.rustc-wrapper` in the Cargo configuration. Refer to sccache documentation for more details.

Cargo Reference

The reference covers the details of various areas of Cargo.

- Specifying Dependencies
- The Manifest Format
- Configuration
- Environment Variables
- Build Scripts
- Publishing on crates.io

Specifying Dependencies

Your crates can depend on other libraries from crates.io or other registries, `git` repositories, or subdirectories on your local file system. You can also temporarily override the location of a dependency — for example, to be able to test out a bug fix in the dependency that you are working on locally. You can have different dependencies for different platforms, and dependencies that are only used during development. Let's take a look at how to do each of these.

Specifying dependencies from crates.io

Cargo is configured to look for dependencies on crates.io by default. Only the name and a version string are required in this case. In the cargo guide, we specified a dependency on the `time` crate:

```
[dependencies]
time = "0.1.12"
```

The string `"0.1.12"` is a semver version requirement. Since this string does not have any operators in it, it is interpreted the same way as if we had specified `"^0.1.12"`, which is called a caret requirement.

Caret requirements

Caret requirements allow SemVer compatible updates to a specified version. An update is allowed if the new version number does not modify the left-most non-zero digit in the major, minor, patch grouping. In this case, if we ran `cargo update -p time`, cargo should update us to version `0.1.13` if it is the latest `0.1.z` release, but would not update us to `0.2.0`. If instead we had specified the version string as `^1.0`, cargo should update to `1.1` if it is the latest `1.y` release, but not `2.0`. The version `0.0.x` is not considered compatible with any other version.

Here are some more examples of caret requirements and the versions that would be allowed with them:

```
^1.2.3  :=  >=1.2.3, <2.0.0
^1.2    :=  >=1.2.0, <2.0.0
^1      :=  >=1.0.0, <2.0.0
^0.2.3  :=  >=0.2.3, <0.3.0
^0.2    :=  >=0.2.0, <0.3.0
^0.0.3  :=  >=0.0.3, <0.0.4
^0.0    :=  >=0.0.0, <0.1.0
^0      :=  >=0.0.0, <1.0.0
```

This compatibility convention is different from SemVer in the way it treats versions before 1.0.0. While SemVer says there is no compatibility before 1.0.0, Cargo considers `0.x.y` to be compatible with `0.x.z`, where `y ≥ z` and `x > 0`.

Tilde requirements

Tilde requirements specify a minimal version with some ability to update. If you specify a major, minor, and patch version or only a major and minor version, only patch-level changes are allowed. If you only specify a major version, then minor- and patch-level changes are allowed.

`~1.2.3` is an example of a tilde requirement.

```
~1.2.3  := >=1.2.3, <1.3.0
~1.2    := >=1.2.0, <1.3.0
~1      := >=1.0.0, <2.0.0
```

Wildcard requirements

Wildcard requirements allow for any version where the wildcard is positioned.

`*`, `1.*` and `1.2.*` are examples of wildcard requirements.

```
*     := >=0.0.0
1.*   := >=1.0.0, <2.0.0
1.2.* := >=1.2.0, <1.3.0
```

Comparison requirements

Comparison requirements allow manually specifying a version range or an exact version to depend on.

Here are some examples of comparison requirements:

```
>= 1.2.0
> 1
< 2
= 1.2.3
```

Multiple requirements

As shown in the examples above, multiple version requirements can be separated with a comma, e.g., `>= 1.2, < 1.5`.

Specifying dependencies from other registries

To specify a dependency from a registry other than crates.io, first the registry must be configured in a `.cargo/config` file. See the registries documentation for more information. In the dependency, set the `registry` key to the name of the registry to use.

```
[dependencies]
some-crate = { version = "1.0", registry = "my-registry" }
```

Specifying dependencies from `git` repositories

To depend on a library located in a `git` repository, the minimum information you need to specify is the location of the repository with the `git` key:

```
[dependencies]
rand = { git = "https://github.com/rust-lang-nursery/rand" }
```

Cargo will fetch the `git` repository at this location then look for a `Cargo.toml` for the requested crate anywhere inside the `git` repository (not necessarily at the root - for example, specifying a member crate name of a workspace and setting `git` to the repository containing the workspace).

Since we haven't specified any other information, Cargo assumes that we intend to use the latest commit on the `master` branch to build our package. You can combine the `git` key with the `rev`, `tag`, or `branch` keys to specify something else. Here's an example of specifying that you want to use the latest commit on a branch named `next`:

```
[dependencies]
rand = { git = "https://github.com/rust-lang-nursery/rand", branch = "next" }
```

Specifying path dependencies

Over time, our `hello_world` package from the guide has grown significantly in size! It's gotten to the point that we probably want to split out a separate crate for others to use. To do this Cargo supports **path dependencies** which are typically sub-crates that live within one repository. Let's start off by making a new crate inside of our `hello_world` package:

```
# inside of hello_world/
$ cargo new hello_utils
```

This will create a new folder `hello_utils` inside of which a `Cargo.toml` and `src` folder are ready to be configured. In order to tell Cargo about this, open up `hello_world/Cargo.toml` and add `hello_utils` to your dependencies:

```
[dependencies]
hello_utils = { path = "hello_utils" }
```

This tells Cargo that we depend on a crate called `hello_utils` which is found in the `hello_utils` folder (relative to the `Cargo.toml` it's written in).

And that's it! The next `cargo build` will automatically build `hello_utils` and all of its own dependencies, and others can also start using the crate as well. However, crates that use dependencies specified with only a path are not permitted on crates.io. If we wanted to publish our `hello_world` crate, we would need to publish a version of `hello_utils` to crates.io and specify its version in the dependencies line as well:

```
[dependencies]
hello_utils = { path = "hello_utils", version = "0.1.0" }
```

Overriding dependencies

There are a number of methods in Cargo to support overriding dependencies and otherwise controlling the dependency graph. These options are typically, though, only available at the workspace level and aren't propagated through dependencies. In other words, "applications" have the ability to override dependencies but "libraries" do not.

The desire to override a dependency or otherwise alter some dependencies can arise through a number of scenarios. Most of them, however, boil down to the ability to work with a crate before it's been published to crates.io. For example:

- A crate you're working on is also used in a much larger application you're working on, and you'd like to test a bug fix to the library inside of the larger application.
- An upstream crate you don't work on has a new feature or a bug fix on the master branch of its git repository which you'd like to test out.
- You're about to publish a new major version of your crate, but you'd like to do integration testing across an entire package to ensure the new major version works.
- You've submitted a fix to an upstream crate for a bug you found, but you'd like to immediately have your application start depending on the fixed version of the crate to avoid blocking on the bug fix getting merged.

These scenarios are currently all solved with the `[patch]` manifest section. Historically some of these scenarios have been solved with the `[replace]` section, but we'll document the `[patch]` section here.

Testing a bugfix

Let's say you're working with the `uuid` crate but while you're working on it you discover a bug. You are, however, quite enterprising so you decide to also try to fix the bug! Originally your manifest will look like:

```
[package]
name = "my-library"
version = "0.1.0"
authors = ["..."]

[dependencies]
uuid = "1.0"
```

First thing we'll do is to clone the `uuid` repository locally via:

```
$ git clone https://github.com/rust-lang-nursery/uuid
```

Next we'll edit the manifest of `my-library` to contain:

```
[patch.crates-io]
uuid = { path = "../path/to/uuid" }
```

Here we declare that we're *patching* the source `crates-io` with a new dependency. This will effectively add the local checked out version of `uuid` to the crates.io registry for our local package.

Next up we need to ensure that our lock file is updated to use this new version of `uuid` so our package uses the locally checked out copy instead of one from crates.io. The way `[patch]` works is that it'll load the dependency at `../path/to/uuid` and then whenever crates.io is queried for versions of `uuid` it'll *also* return the local version.

This means that the version number of the local checkout is significant and will affect whether the patch is used. Our manifest declared `uuid = "1.0"` which means we'll only resolve to `>= 1.0.0, < 2.0.0`, and Cargo's greedy resolution algorithm also means that we'll resolve to the maximum version within that range. Typically this doesn't matter as the version of the git repository will already be greater or match the maximum version published on crates.io, but it's important to keep this in mind!

In any case, typically all you need to do now is:

```
$ cargo build
   Compiling uuid v1.0.0 (.../uuid)
   Compiling my-library v0.1.0 (.../my-library)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

And that's it! You're now building with the local version of `uuid` (note the path in parentheses in the build output). If you don't see the local path version getting built then you may need to run `cargo update -p uuid --precise $version` where `$version` is the version of the locally checked out copy of `uuid`.

Once you've fixed the bug you originally found the next thing you'll want to do is to likely submit that as a pull request to the `uuid` crate itself. Once you've done this then you can also update the `[patch]` section. The listing inside of `[patch]` is just like the `[dependencies]` section, so once your pull request is merged you could change your `path` dependency to:

```
[patch.crates-io]
uuid = { git = 'https://github.com/rust-lang-nursery/uuid' }
```

Working with an unpublished minor version

Let's now shift gears a bit from bug fixes to adding features. While working on `my-library` you discover that a whole new feature is needed in the `uuid` crate. You've implemented this feature, tested it locally above with `[patch]`, and submitted a pull request. Let's go over how you continue to use and test it before it's actually published.

Let's also say that the current version of `uuid` on crates.io is `1.0.0`, but since then the master branch of the git repository has updated to `1.0.1`. This branch includes your new feature you submitted previously. To use this repository we'll edit our `Cargo.toml` to look like

```
[package]
name = "my-library"
version = "0.1.0"
authors = ["..."]

[dependencies]
uuid = "1.0.1"

[patch.crates-io]
uuid = { git = 'https://github.com/rust-lang-nursery/uuid' }
```

Note that our local dependency on `uuid` has been updated to `1.0.1` as it's what we'll actually require once the crate is published. This version doesn't exist on crates.io, though, so we provide it with the `[patch]` section of the manifest.

Now when our library is built it'll fetch `uuid` from the git repository and resolve to 1.0.1 inside the repository instead of trying to download a version from crates.io. Once 1.0.1 is published on crates.io the `[patch]` section can be deleted.

It's also worth noting that `[patch]` applies *transitively*. Let's say you use `my-library` in a larger package, such as:

```
[package]
name = "my-binary"
version = "0.1.0"
authors = ["..."]

[dependencies]
my-library = { git = 'https://example.com/git/my-library' }
uuid = "1.0"

[patch.crates-io]
uuid = { git = 'https://github.com/rust-lang-nursery/uuid' }
```

Remember that `[patch]` is applicable *transitively* but can only be defined at the *top level* so we consumers of `my-library` have to repeat the `[patch]` section if necessary. Here, though, the new `uuid` crate applies to *both* our dependency on `uuid` and the `my-library -> uuid` dependency. The `uuid` crate will be resolved to one version for this entire crate graph, 1.0.1, and it'll be pulled from the git repository.

Overriding repository URL

In case the dependency you want to override isn't loaded from `crates.io`, you'll have to change a bit how you use `[patch]`:

```
[patch."https://github.com/your/repository"]
my-library = { path = "../my-library/path" }
```

And that's it!

Prepublishing a breaking change

As a final scenario, let's take a look at working with a new major version of a crate, typically accompanied with breaking changes. Sticking with our previous crates, this means that we're going to be creating version 2.0.0 of the `uuid` crate. After we've submitted all changes upstream we can update our manifest for `my-library` to look like:

```
[dependencies]
uuid = "2.0"

[patch.crates-io]
uuid = { git = "https://github.com/rust-lang-nursery/uuid", branch = "2.0.0" }
```

And that's it! Like with the previous example the 2.0.0 version doesn't actually exist on crates.io but we can still put it in through a git dependency through the usage of the `[patch]` section. As a thought exercise let's take another look at the `my-binary` manifest from above again as well:

```
[package]
name = "my-binary"
version = "0.1.0"
authors = ["..."]

[dependencies]
my-library = { git = 'https://example.com/git/my-library' }
uuid = "1.0"

[patch.crates-io]
uuid = { git = 'https://github.com/rust-lang-nursery/uuid', branch = '2.0.0' }
```

Note that this will actually resolve to two versions of the `uuid` crate. The `my-binary` crate will continue to use the 1.x.y series of the `uuid` crate but the `my-library` crate will use the 2.0.0 version of `uuid`. This will allow you to gradually roll out breaking changes to a crate through a dependency graph without being force to update everything all at once.

Overriding with local dependencies

Sometimes you're only temporarily working on a crate and you don't want to have to modify `Cargo.toml` like with the `[patch]` section above. For this use case Cargo offers a much more limited version of overrides called **path overrides**.

Path overrides are specified through `.cargo/config` instead of `Cargo.toml`, and you can find more documentation about this configuration. Inside of `.cargo/config` you'll specify a key called `paths`:

```
paths = ["/path/to/uuid"]
```

This array should be filled with directories that contain a `Cargo.toml`. In this instance, we're just adding `uuid`, so it will be the only one that's overridden. This path can be either absolute or relative to the directory that contains the `.cargo` folder.

Path overrides are more restricted than the `[patch]` section, however, in that they cannot change the structure of the dependency graph. When a path replacement is used then the previous set of dependencies must all match exactly to the new `Cargo.toml` specification. For example this means that path overrides cannot be used to test out adding a dependency to a crate, instead `[patch]` must be used in that situation. As a result usage of a path override is typically isolated to quick bug fixes rather than larger changes.

Note: using a local configuration to override paths will only work for crates that have been published to crates.io. You cannot use this feature to tell Cargo how to find local unpublished crates.

Platform specific dependencies

Platform-specific dependencies take the same format, but are listed under a `target` section. Normally Rust-like `#[cfg]` syntax will be used to define these sections:

```
[target.'cfg(windows)'.dependencies]
winhttp = "0.4.0"

[target.'cfg(unix)'.dependencies]
openssl = "1.0.1"

[target.'cfg(target_arch = "x86")'.dependencies]
native = { path = "native/i686" }

[target.'cfg(target_arch = "x86_64")'.dependencies]
native = { path = "native/x86_64" }
```

Like with Rust, the syntax here supports the `not`, `any`, and `all` operators to combine various cfg name/value pairs.

If you want to know which cfg targets are available on your platform, run `rustc --print=cfg` from the command line. If you want to know which `cfg` targets are available for another platform, such as 64-bit Windows, run `rustc --print=cfg --target=x86_64-pc-windows-msvc`.

Unlike in your Rust source code, you cannot use `[target.'cfg(feature = "my_crate")'.dependencies]` to add dependencies based on optional crate features. Use the `[features]` section instead.

In addition to `#[cfg]` syntax, Cargo also supports listing out the full target the dependencies would apply to:

```
[target.x86_64-pc-windows-gnu.dependencies]
winhttp = "0.4.0"

[target.i686-unknown-linux-gnu.dependencies]
openssl = "1.0.1"
```

If you're using a custom target specification (such as `--target foo/bar.json`), use the base filename without the `.json` extension:

```
[target.bar.dependencies]
winhttp = "0.4.0"

[target.my-special-i686-platform.dependencies]
openssl = "1.0.1"
native = { path = "native/i686" }
```

Development dependencies

You can add a `[dev-dependencies]` section to your `Cargo.toml` whose format is equivalent to `[dependencies]`:

```
[dev-dependencies]
tempdir = "0.3"
```

Dev-dependencies are not used when compiling a package for building, but are used for compiling tests, examples, and benchmarks.

These dependencies are *not* propagated to other packages which depend on this package.

You can also have target-specific development dependencies by using `dev-dependencies` in the target section header instead of `dependencies`. For example:

```
[target.'cfg(unix)'.dev-dependencies]
mio = "0.0.1"
```

Build dependencies

You can depend on other Cargo-based crates for use in your build scripts. Dependencies are declared through the `build-dependencies` section of the manifest:

```
[build-dependencies]
cc = "1.0.3"
```

The build script **does not** have access to the dependencies listed in the `dependencies` or `dev-dependencies` section. Build dependencies will likewise not be available to the package itself unless listed under the `dependencies` section as well. A package itself and its build script are built separately, so their dependencies need not coincide. Cargo is kept simpler and cleaner by using independent dependencies for independent purposes.

Choosing features

If a package you depend on offers conditional features, you can specify which to use:

```
[dependencies.awesome]
version = "1.3.5"
default-features = false # do not include the default features, and optionally
                         # cherry-pick individual features
features = ["secure-password", "civet"]
```

More information about features can be found in the manifest documentation.

Renaming dependencies in `Cargo.toml`

When writing a `[dependencies]` section in `Cargo.toml` the key you write for a dependency typically matches up to the name of the crate you import from in the code. For some

projects, though, you may wish to reference the crate with a different name in the code regardless of how it's published on crates.io. For example you may wish to:

- Avoid the need to `use foo as bar` in Rust source.
- Depend on multiple versions of a crate.
- Depend on crates with the same name from different registries.

To support this Cargo supports a `package` key in the `[dependencies]` section of which package should be depended on:

```
[package]
name = "mypackage"
version = "0.0.1"

[dependencies]
foo = "0.1"
bar = { git = "https://github.com/example/project", package = "foo" }
baz = { version = "0.1", registry = "custom", package = "foo" }
```

In this example, three crates are now available in your Rust code:

```
extern crate foo; // crates.io
extern crate bar; // git repository
extern crate baz; // registry `custom`
```

All three of these crates have the package name of `foo` in their own `Cargo.toml`, so we're explicitly using the `package` key to inform Cargo that we want the `foo` package even though we're calling it something else locally. The `package` key, if not specified, defaults to the name of the dependency being requested.

Note that if you have an optional dependency like:

```
[dependencies]
foo = { version = "0.1", package = 'bar', optional = true }
```

you're depending on the crate `bar` from crates.io, but your crate has a `foo` feature instead of a `bar` feature. That is, names of features take after the name of the dependency, not the package name, when renamed.

Enabling transitive dependencies works similarly, for example we could add the following to the above manifest:

```
[features]
log-debug = ['foo/log-debug'] # using 'bar/log-debug' would be an error!
```

The Manifest Format

The `Cargo.toml` file for each package is called its *manifest*. Every manifest file consists of one or more sections.

The `[package]` section

The first section in a `Cargo.toml` is `[package]`.

```
[package]
name = "hello_world" # the name of the package
version = "0.1.0"    # the current version, obeying semver
authors = ["Alice <a@example.com>", "Bob <b@example.com>"]
```

The `name` field

The package name is an identifier used to refer to the package. It is used when listed as a dependency in another package, and as the default name of inferred lib and bin targets.

The name must not be empty, use only alphanumeric characters or `-` or `_`. Note that `cargo new` and `cargo init` impose some additional restrictions on the package name, such as enforcing that it is a valid Rust identifier and not a keyword. crates.io imposes even more restrictions, such as enforcing only ASCII characters, not a reserved name, not a special Windows name such as "nul", is not too long, etc.

The `version` field

Cargo bakes in the concept of Semantic Versioning, so make sure you follow some basic rules:

- Before you reach 1.0.0, anything goes, but if you make breaking changes, increment the minor version. In Rust, breaking changes include adding fields to structs or variants to enums.
- After 1.0.0, only make breaking changes when you increment the major version. Don't break the build.
- After 1.0.0, don't add any new public API (no new `pub` anything) in patch-level versions. Always increment the minor version if you add any new `pub` structs, traits, fields, types, functions, methods or anything else.
- Use version numbers with three numeric parts such as 1.0.0 rather than 1.0.

The `authors` field (optional)

The `authors` field lists people or organizations that are considered the "authors" of the package. The exact meaning is open to interpretation — it may list the original or primary authors, current maintainers, or owners of the package. These names will be listed on the crate's page on crates.io. An optional email address may be included within angled brackets at the end of each author.

The `edition` field (optional)

You can opt in to a specific Rust Edition for your package with the `edition` key in `Cargo.toml`. If you don't specify the edition, it will default to 2015.

```
[package]
# ...
edition = '2018'
```

The `edition` key affects which edition your package is compiled with. Cargo will always generate packages via `cargo new` with the `edition` key set to the latest edition. Setting the `edition` key in `[package]` will affect all targets/crates in the package, including test suites, benchmarks, binaries, examples, etc.

The `build` field (optional)

This field specifies a file in the package root which is a build script for building native code. More information can be found in the build script guide.

```
[package]
# ...
build = "build.rs"
```

The default is `"build.rs"`, which loads the script from a file named `build.rs` in the root of the package. Use `build = "custom_build_name.rs"` to specify a path to a different file or `build = false` to disable automatic detection of the build script.

The `links` field (optional)

This field specifies the name of a native library that is being linked to. More information can be found in the `links` section of the build script guide.

```
[package]
# ...
links = "foo"
```

The `documentation` field (optional)

This field specifies a URL to a website hosting the crate's documentation. If no URL is specified in the manifest file, crates.io will automatically link your crate to the corresponding docs.rs page.

Documentation links from specific hosts are blacklisted. Hosts are added to the blacklist if they are known to not be hosting documentation and are possibly of malicious intent e.g., ad tracking networks. URLs from the following hosts are blacklisted:

- rust-ci.org

Documentation URLs from blacklisted hosts will not appear on crates.io, and may be replaced by docs.rs links.

The `exclude` and `include` fields (optional)

You can explicitly specify that a set of file patterns should be ignored or included for the purposes of packaging. The patterns specified in the `exclude` field identify a set of files that are not included, and the patterns in `include` specify files that are explicitly included.

The patterns should be gitignore-style patterns. Briefly:

- `foo` matches any file or directory with the name `foo` anywhere in the package. This is equivalent to the pattern `**/foo`.
- `/foo` matches any file or directory with the name `foo` only in the root of the package.
- `foo/` matches any *directory* with the name `foo` anywhere in the package.
- Common glob patterns like `*`, `?`, and `[]` are supported:
  - `*` matches zero or more characters except `/`. For example, `*.html` matches any file or directory with the `.html` extension anywhere in the package.
  - `?` matches any character except `/`. For example, `foo?` matches `food`, but not `foo`.
  - `[]` allows for matching a range of characters. For example, `[ab]` matches either `a` or `b`. `[a-z]` matches letters a through z.
- `**/` prefix matches in any directory. For example, `**/foo/bar` matches the file or directory `bar` anywhere that is directly under directory `foo`.
- `/**` suffix matches everything inside. For example, `foo/**` matches all files inside directory `foo`, including all files in subdirectories below `foo`.
- `/**/` matches zero or more directories. For example, `a/**/b` matches `a/b`, `a/x/b`, `a/x/y/b`, and so on.
- `!` prefix negates a pattern. For example, a pattern of `src/**.rs` and `!foo.rs` would match all files with the `.rs` extension inside the `src` directory, except for any file named `foo.rs`.

If git is being used for a package, the `exclude` field will be seeded with the `gitignore` settings from the repository.

```
[package]
# ...
exclude = ["build/**/*.o", "doc/**/*.html"]
```

```
[package]
# ...
include = ["src/**/*", "Cargo.toml"]
```

The options are mutually exclusive: setting `include` will override an `exclude`. Note that `include` must be an exhaustive list of files as otherwise necessary source files may not be included. The package's `Cargo.toml` is automatically included.

The include/exclude list is also used for change tracking in some situations. For targets built with `rustdoc`, it is used to determine the list of files to track to determine if the target should be rebuilt. If the package has a build script that does not emit any `rerun-if-*` directives, then the include/exclude list is used for tracking if the build script should be re-run if any of those files change.

The `publish` field (optional)

The `publish` field can be used to prevent a package from being published to a package registry (like *crates.io*) by mistake, for instance to keep a package private in a company.

```
[package]
# ...
publish = false
```

The value may also be an array of strings which are registry names that are allowed to be published to.

```
[package]
# ...
publish = ["some-registry-name"]
```

The `workspace` field (optional)

The `workspace` field can be used to configure the workspace that this package will be a member of. If not specified this will be inferred as the first Cargo.toml with `[workspace]` upwards in the filesystem.

```
[package]
# ...
workspace = "path/to/workspace/root"
```

For more information, see the documentation for the workspace table below.

Package metadata

There are a number of optional metadata fields also accepted under the `[package]` section:

```toml
[package]
# ...

# A short blurb about the package. This is not rendered in any format when
# uploaded to crates.io (aka this is not markdown).
description = "..."

# These URLs point to more information about the package. These are
# intended to be webviews of the relevant data, not necessarily compatible
# with VCS tools and the like.
documentation = "..."
homepage = "..."
repository = "..."

# This points to a file under the package root (relative to this `Cargo.toml`).
# The contents of this file are stored and indexed in the registry.
# crates.io will render this file and place the result on the crate's page.
readme = "..."

# This is a list of up to five keywords that describe this crate. Keywords
# are searchable on crates.io, and you may choose any words that would
# help someone find this crate.
keywords = ["...", "..."]

# This is a list of up to five categories where this crate would fit.
# Categories are a fixed list available at crates.io/category_slugs, and
# they must match exactly.
categories = ["...", "..."]

# This is an SPDX 2.1 license expression for this package. Currently
# crates.io will validate the license provided against a whitelist of
# known license and exception identifiers from the SPDX license list
# 3.6. Parentheses are not currently supported.
#
# Multiple licenses can be separated with a `/`, although that usage
# is deprecated. Instead, use a license expression with AND and OR
# operators to get more explicit semantics.
license = "..."

# If a package is using a nonstandard license, then this key may be specified in
# lieu of the above key and must point to a file relative to this manifest
# (similar to the readme key).
license-file = "..."

# Optional specification of badges to be displayed on crates.io.
#
# - The badges pertaining to build status that are currently available are
#   Appveyor, CircleCI, Cirrus CI, GitLab, Azure DevOps and TravisCI.
# - Available badges pertaining to code test coverage are Codecov and
#   Coveralls.
# - There are also maintenance-related badges based on isitmaintained.com
#   which state the issue resolution time, percent of open issues, and future
#   maintenance intentions.
#
# If a `repository` key is required, this refers to a repository in
# `user/repo` format.
[badges]
```

```
# Appveyor: `repository` is required. `branch` is optional; default is `master`
# `service` is optional; valid values are `github` (default), `bitbucket`, and
# `gitlab`; `id` is optional; you can specify the appveyor project id if you
# want to use that instead. `project_name` is optional; use when the repository
# name differs from the appveyor project name.
appveyor = { repository = "...", branch = "master", service = "github" }

# Circle CI: `repository` is required. `branch` is optional; default is `master`
circle-ci = { repository = "...", branch = "master" }

# Cirrus CI: `repository` is required. `branch` is optional; default is `master`
cirrus-ci = { repository = "...", branch = "master" }

# GitLab: `repository` is required. `branch` is optional; default is `master`
gitlab = { repository = "...", branch = "master" }

# Azure DevOps: `project` is required. `pipeline` is required. `build` is
optional; default is `1`
# Note: project = `organization/project`, pipeline = `name_of_pipeline`, build =
`definitionId`
azure-devops = { project = "...", pipeline = "...", build="2" }

# Travis CI: `repository` in format "<user>/<project>" is required.
# `branch` is optional; default is `master`
travis-ci = { repository = "...", branch = "master" }

# Codecov: `repository` is required. `branch` is optional; default is `master`
# `service` is optional; valid values are `github` (default), `bitbucket`, and
# `gitlab`.
codecov = { repository = "...", branch = "master", service = "github" }

# Coveralls: `repository` is required. `branch` is optional; default is `master`
# `service` is optional; valid values are `github` (default) and `bitbucket`.
coveralls = { repository = "...", branch = "master", service = "github" }

# Is it maintained resolution time: `repository` is required.
is-it-maintained-issue-resolution = { repository = "..." }

# Is it maintained percentage of open issues: `repository` is required.
is-it-maintained-open-issues = { repository = "..." }

# Maintenance: `status` is required. Available options are:
# - `actively-developed`: New features are being added and bugs are being fixed.
# - `passively-maintained`: There are no plans for new features, but the
maintainer intends to
#   respond to issues that get filed.
# - `as-is`: The crate is feature complete, the maintainer does not intend to
continue working on
#   it or providing support, but it works for the purposes it was designed for.
# - `experimental`: The author wants to share it with the community but is not
intending to meet
#   anyone's particular use case.
# - `looking-for-maintainer`: The current maintainer would like to transfer the
crate to someone
#   else.
# - `deprecated`: The maintainer does not recommend using this crate (the
description of the crate
```

```
#   can describe why, there could be a better solution available or there could
be problems with
#   the crate that the author does not want to fix).
# - `none`: Displays no badge on crates.io, since the maintainer has not chosen
to specify
#   their intentions, potential crate users will need to investigate on their
own.
maintenance = { status = "..." }
```

The crates.io registry will render the description, display the license, link to the three URLs and categorize by the keywords. These keys provide useful information to users of the registry and also influence the search ranking of a crate. It is highly discouraged to omit everything in a published crate.

SPDX 2.1 license expressions are documented here. The current version of the license list is available here, and version 3.6 is available here.

The `metadata` table (optional)

Cargo by default will warn about unused keys in `Cargo.toml` to assist in detecting typos and such. The `package.metadata` table, however, is completely ignored by Cargo and will not be warned about. This section can be used for tools which would like to store package configuration in `Cargo.toml`. For example:

```
[package]
name = "..."
# ...

# Metadata used when generating an Android APK, for example.
[package.metadata.android]
package-name = "my-awesome-android-app"
assets = "path/to/static"
```

The `default-run` field

The `default-run` field in the `[package]` section of the manifest can be used to specify a default binary picked by `cargo run`. For example, when there is both `src/bin/a.rs` and `src/bin/b.rs`:

```
[package]
default-run = "a"
```

Dependency sections

See the specifying dependencies page for information on the `[dependencies]`, `[dev-dependencies]`, `[build-dependencies]`, and target-specific `[target.*.dependencies]` sections.

The `[profile.*]` sections

The `[profile]` tables provide a way to customize compiler settings such as optimizations and debug settings. See the Profiles chapter for more detail.

The `[features]` section

Cargo supports features to allow expression of:

- conditional compilation options (usable through `cfg` attributes);
- optional dependencies, which enhance a package, but are not required; and
- clusters of optional dependencies, such as `postgres`, that would include the `postgres` package, the `postgres-macros` package, and possibly other packages (such as development-time mocking libraries, debugging tools, etc.).

A feature of a package is either an optional dependency, or a set of other features. The format for specifying features is:

```toml
[package]
name = "awesome"

[features]
# The default set of optional packages. Most people will want to use these
# packages, but they are strictly optional. Note that `session` is not a package
# but rather another feature listed in this manifest.
default = ["jquery", "uglifier", "session"]

# A feature with no dependencies is used mainly for conditional compilation,
# like `#[cfg(feature = "go-faster")]`.
go-faster = []

# The `secure-password` feature depends on the bcrypt package. This aliasing
# will allow people to talk about the feature in a higher-level way and allow
# this package to add more requirements to the feature in the future.
secure-password = ["bcrypt"]

# Features can be used to reexport features of other packages. The `session`
# feature of package `awesome` will ensure that the `session` feature of the
# package `cookie` is also enabled.
session = ["cookie/session"]

[dependencies]
# These packages are mandatory and form the core of this package's distribution.
cookie = "1.2.0"
oauth = "1.1.0"
route-recognizer = "=2.1.0"

# A list of all of the optional dependencies, some of which are included in the
# above `features`. They can be opted into by apps.
jquery = { version = "1.0.2", optional = true }
uglifier = { version = "1.5.3", optional = true }
bcrypt = { version = "*", optional = true }
civet = { version = "*", optional = true }
```

To use the package `awesome`:

```toml
[dependencies.awesome]
version = "1.3.5"
default-features = false # do not include the default features, and optionally
                         # cherry-pick individual features
features = ["secure-password", "civet"]
```

Rules

The usage of features is subject to a few rules:

- Feature names must not conflict with other package names in the manifest. This is because they are opted into via `features = [...]`, which only has a single namespace.
- With the exception of the `default` feature, all features are opt-in. To opt out of the default feature, use `default-features = false` and cherry-pick individual features.

- Feature groups are not allowed to cyclically depend on one another.
- Dev-dependencies cannot be optional.
- Features groups can only reference optional dependencies.
- When a feature is selected, Cargo will call `rustc` with `--cfg feature="${feature_name}"`. If a feature group is included, it and all of its individual features will be included. This can be tested in code via `#[cfg(feature = "foo")]`.

Note that it is explicitly allowed for features to not actually activate any optional dependencies. This allows packages to internally enable/disable features without requiring a new dependency.

Usage in end products

One major use-case for this feature is specifying optional features in end-products. For example, the Servo package may want to include optional features that people can enable or disable when they build it.

In that case, Servo will describe features in its `Cargo.toml` and they can be enabled using command-line flags:

```
$ cargo build --release --features "shumway pdf"
```

Default features could be excluded using `--no-default-features`.

Usage in packages

In most cases, the concept of *optional dependency* in a library is best expressed as a separate package that the top-level application depends on.

However, high-level packages, like Iron or Piston, may want the ability to curate a number of packages for easy installation. The current Cargo system allows them to curate a number of mandatory dependencies into a single package for easy installation.

In some cases, packages may want to provide additional curation for optional dependencies:

- grouping a number of low-level optional dependencies together into a single high-level feature;
- specifying packages that are recommended (or suggested) to be included by users of the package; and
- including a feature (like `secure-password` in the motivating example) that will only work if an optional dependency is available, and would be difficult to implement as a separate package (for example, it may be overly difficult to design an IO package to be completely decoupled from OpenSSL, with opt-in via the inclusion of a separate package).

In almost all cases, it is an antipattern to use these features outside of high-level packages that are designed for curation. If a feature is optional, it can almost certainly be expressed

as a separate package.

The `[workspace]` section

Packages can define a workspace which is a set of crates that will all share the same `Cargo.lock` and output directory. The `[workspace]` table can be defined as:

```
[workspace]

# Optional key, inferred from path dependencies if not present.
# Additional non-path dependencies that should be included must be given here.
# In particular, for a virtual manifest, all members have to be listed.
members = ["path/to/member1", "path/to/member2", "path/to/member3/*"]

# Optional key, empty if not present.
exclude = ["path1", "path/to/dir2"]
```

Workspaces were added to Cargo as part of RFC 1525 and have a number of properties:

- A workspace can contain multiple crates where one of them is the *root crate*.
- The *root crate*'s `Cargo.toml` contains the `[workspace]` table, but is not required to have other configuration.
- Whenever any crate in the workspace is compiled, output is placed in the *workspace root* (i.e., next to the *root crate*'s `Cargo.toml`).
- The lock file for all crates in the workspace resides in the *workspace root*.
- The `[patch]`, `[replace]` and `[profile.*]` sections in `Cargo.toml` are only recognized in the *root crate*'s manifest, and ignored in member crates' manifests.

The *root crate* of a workspace, indicated by the presence of `[workspace]` in its manifest, is responsible for defining the entire workspace. All `path` dependencies residing in the workspace directory become members. You can add additional packages to the workspace by listing them in the `members` key. Note that members of the workspaces listed explicitly will also have their path dependencies included in the workspace. Sometimes a package may have a lot of workspace members and it can be onerous to keep up to date. The `members` list can also use globs to match multiple paths. Finally, the `exclude` key can be used to blacklist paths from being included in a workspace. This can be useful if some path dependencies aren't desired to be in the workspace at all.

The `package.workspace` manifest key (described above) is used in member crates to point at a workspace's root crate. If this key is omitted then it is inferred to be the first crate whose manifest contains `[workspace]` upwards in the filesystem.

A crate may either specify `package.workspace` or specify `[workspace]`. That is, a crate cannot both be a root crate in a workspace (contain `[workspace]`) and also be a member crate of another workspace (contain `package.workspace`).

Most of the time workspaces will not need to be dealt with as `cargo new` and `cargo init` will handle workspace configuration automatically.

Virtual Manifest

In workspace manifests, if the `package` table is present, the workspace root crate will be treated as a normal package, as well as a workspace. If the `package` table is not present in a workspace manifest, it is called a *virtual manifest*.

Package selection

In a workspace, package-related cargo commands like `cargo build` apply to packages selected by `-p` / `--package` or `--workspace` command-line parameters. When neither is specified, the optional `default-members` configuration is used:

```
[workspace]
members = ["path/to/member1", "path/to/member2", "path/to/member3/*"]
default-members = ["path/to/member2", "path/to/member3/foo"]
```

When specified, `default-members` must expand to a subset of `members`.

When `default-members` is not specified, the default is the root manifest if it is a package, or every member manifest (as if `--workspace` were specified on the command-line) for virtual workspaces.

The project layout

If your package is an executable, name the main source file `src/main.rs`. If it is a library, name the main source file `src/lib.rs`.

Cargo will also treat any files located in `src/bin/*.rs` as executables. If your executable consists of more than just one source file, you might also use a directory inside `src/bin` containing a `main.rs` file which will be treated as an executable with a name of the parent directory.

Your package can optionally contain folders named `examples`, `tests`, and `benches`, which Cargo will treat as containing examples, integration tests, and benchmarks respectively. Analogous to `bin` targets, they may be composed of single files or directories with a `main.rs` file.

```
▾ src/           # directory containing source files
  lib.rs         # the main entry point for libraries and packages
  main.rs        # the main entry point for packages producing executables
  ▾ bin/         # (optional) directory containing additional executables
    *.rs
  ▾ */           # (optional) directories containing multi-file executables
    main.rs
▾ examples/      # (optional) examples
  *.rs
  ▾ */           # (optional) directories containing multi-file examples
    main.rs
▾ tests/         # (optional) integration tests
  *.rs
  ▾ */           # (optional) directories containing multi-file tests
    main.rs
▾ benches/       # (optional) benchmarks
  *.rs
  ▾ */           # (optional) directories containing multi-file benchmarks
    main.rs
```

To structure your code after you've created the files and folders for your package, you should remember to use Rust's module system, which you can read about in the book.

See Configuring a target below for more details on manually configuring target settings. See Target auto-discovery below for more information on controlling how Cargo automatically infers targets.

Examples

Files located under `examples` are example uses of the functionality provided by the library. When compiled, they are placed in the `target/examples` directory.

They can compile either as executables (with a `main()` function) or libraries and pull in the library by using `extern crate <library-name>`. They are compiled when you run your tests to protect them from bitrotting.

You can run individual executable examples with the command `cargo run --example <example-name>`.

Specify `crate-type` to make an example be compiled as a library (additional information about crate types is available in The Rust Reference):

```
[[example]]
name = "foo"
crate-type = ["staticlib"]
```

You can build individual library examples with the command `cargo build --example <example-name>`.

Tests

When you run `cargo test`, Cargo will:

- compile and run your library's unit tests, which are in the files reachable from `lib.rs` (naturally, any sections marked with `#[cfg(test)]` will be considered at this stage);
- compile and run your library's documentation tests, which are embedded inside of documentation blocks;
- compile and run your library's integration tests; and
- compile your library's examples.

Integration tests

Each file in `tests/*.rs` is an integration test. When you run `cargo test`, Cargo will compile each of these files as a separate crate. The crate can link to your library by using `extern crate <library-name>`, like any other code that depends on it.

Cargo will not automatically compile files inside subdirectories of `tests`, but an integration test can import modules from these directories as usual. For example, if you want several integration tests to share some code, you can put the shared code in `tests/common/mod.rs` and then put `mod common;` in each of the test files.

Configuring a target

All of the `[[bin]]`, `[lib]`, `[[bench]]`, `[[test]]`, and `[[example]]` sections support similar configuration for specifying how a target should be built. The double-bracket sections like `[[bin]]` are array-of-table of TOML, which means you can write more than one `[[bin]]` section to make several executables in your crate.

The example below uses `[lib]`, but it also applies to all other sections as well. All values listed are the defaults for that option unless otherwise specified.

```toml
[package]
# ...

[lib]
# The name of a target is the name of the library that will be generated. This
# is defaulted to the name of the package, with any dashes replaced
# with underscores. (Rust `extern crate` declarations reference this name;
# therefore the value must be a valid Rust identifier to be usable.)
name = "foo"

# This field points at where the crate is located, relative to the `Cargo.toml`.
path = "src/lib.rs"

# A flag for enabling unit tests for this target. This is used by `cargo test`.
test = true

# A flag for enabling documentation tests for this target. This is only relevant
# for libraries, it has no effect on other sections. This is used by
# `cargo test`.
doctest = true

# A flag for enabling benchmarks for this target. This is used by `cargo bench`.
bench = true

# A flag for enabling documentation of this target. This is used by `cargo doc`.
doc = true

# If the target is meant to be a compiler plugin, this field must be set to true
# for Cargo to correctly compile it and make it available for all dependencies.
plugin = false

# If the target is meant to be a "macros 1.1" procedural macro, this field must
# be set to true.
proc-macro = false

# If set to false, `cargo test` will omit the `--test` flag to rustc, which
# stops it from generating a test harness. This is useful when the binary being
# built manages the test runner itself.
harness = true

# If set then a target can be configured to use a different edition than the
# `[package]` is configured to use, perhaps only compiling a library with the
# 2018 edition or only compiling one unit test with the 2015 edition. By default
# all targets are compiled with the edition specified in `[package]`.
edition = '2015'

# Here's an example of a TOML "array of tables" section, in this case specifying
# a binary target name and path.
[[bin]]
name = "my-cool-binary"
path = "src/my-cool-binary.rs"
```

Target auto-discovery

By default, Cargo automatically determines the targets to build based on the layout of the files on the filesystem. The target configuration tables, such as `[lib]`, `[[bin]]`, `[[test]]`, `[[bench]]`, or `[[example]]`, can be used to add additional targets that don't follow the standard directory layout.

The automatic target discovery can be disabled so that only manually configured targets will be built. Setting the keys `autobins`, `autoexamples`, `autotests`, or `autobenches` to `false` in the `[package]` section will disable auto-discovery of the corresponding target type.

Disabling automatic discovery should only be needed for specialized situations. For example, if you have a library where you want a *module* named `bin`, this would present a problem because Cargo would usually attempt to compile anything in the `bin` directory as an executable. Here is a sample layout of this scenario:

```
├── Cargo.toml
└── src
    ├── lib.rs
    └── bin
        └── mod.rs
```

To prevent Cargo from inferring `src/bin/mod.rs` as an executable, set `autobins = false` in `Cargo.toml` to disable auto-discovery:

```toml
[package]
# …
autobins = false
```

> **Note**: For packages with the 2015 edition, the default for auto-discovery is `false` if at least one target is manually defined in `Cargo.toml`. Beginning with the 2018 edition, the default is always `true`.

The `required-features` field (optional)

The `required-features` field specifies which features the target needs in order to be built. If any of the required features are not selected, the target will be skipped. This is only relevant for the `[[bin]]`, `[[bench]]`, `[[test]]`, and `[[example]]` sections, it has no effect on `[lib]`.

```toml
[features]
# ...
postgres = []
sqlite = []
tools = []

[[bin]]
# ...
required-features = ["postgres", "tools"]
```

Building dynamic or static libraries

If your package produces a library, you can specify which kind of library to build by explicitly listing the library in your `Cargo.toml`:

```
# ...

[lib]
name = "..."
crate-type = ["dylib"] # could be `staticlib` as well
```

The available options are `dylib`, `rlib`, `staticlib`, `cdylib`, and `proc-macro`.

You can read more about the different crate types in the Rust Reference Manual

The `[patch]` Section

This section of Cargo.toml can be used to override dependencies with other copies. The syntax is similar to the `[dependencies]` section:

```
[patch.crates-io]
foo = { git = 'https://github.com/example/foo' }
bar = { path = 'my/local/bar' }

[dependencies.baz]
git = 'https://github.com/example/baz'

[patch.'https://github.com/example/baz']
baz = { git = 'https://github.com/example/patched-baz', branch = 'my-branch' }
```

The `[patch]` table is made of dependency-like sub-tables. Each key after `[patch]` is a URL of the source that is being patched, or the name of a registry. The name `crates-io` may be used to override the default registry crates.io. The first `[patch]` in the example above demonstrates overriding crates.io, and the second `[patch]` demonstrates overriding a git source.

Each entry in these tables is a normal dependency specification, the same as found in the `[dependencies]` section of the manifest. The dependencies listed in the `[patch]` section are resolved and used to patch the source at the URL specified. The above manifest snippet patches the `crates-io` source (e.g. crates.io itself) with the `foo` crate and `bar` crate. It also patches the `https://github.com/example/baz` source with a `my-branch` that comes from elsewhere.

Sources can be patched with versions of crates that do not exist, and they can also be patched with versions of crates that already exist. If a source is patched with a crate version that already exists in the source, then the source's original crate is replaced.

More information about overriding dependencies can be found in the overriding dependencies section of the documentation and RFC 1969 for the technical specification of

this feature.

Using `[patch]` with multiple versions

You can patch in multiple versions of the same crate with the `package` key used to rename dependencies. For example let's say that the `serde` crate has a bugfix that we'd like to use to its 1.* series but we'd also like to prototype using a 2.0.0 version of serde we have in our git repository. To configure this we'd do:

```
[patch.crates-io]
serde = { git = 'https://github.com/serde-rs/serde' }
serde2 = { git = 'https://github.com/example/serde', package = 'serde', branch =
'v2' }
```

The first `serde = ...` directive indicates that serde 1.* should be used from the git repository (pulling in the bugfix we need) and the second `serde2 = ...` directive indicates that the `serde` package should also be pulled from the `v2` branch of `https://github.com/example/serde`. We're assuming here that `Cargo.toml` on that branch mentions version 2.0.0.

Note that when using the `package` key the `serde2` identifier here is actually ignored. We simply need a unique name which doesn't conflict with other patched crates.

The `[replace]` Section

This section of Cargo.toml can be used to override dependencies with other copies. The syntax is similar to the `[dependencies]` section:

```
[replace]
"foo:0.1.0" = { git = 'https://github.com/example/foo' }
"bar:1.0.2" = { path = 'my/local/bar' }
```

Each key in the `[replace]` table is a package ID specification, which allows arbitrarily choosing a node in the dependency graph to override. The value of each key is the same as the `[dependencies]` syntax for specifying dependencies, except that you can't specify features. Note that when a crate is overridden the copy it's overridden with must have both the same name and version, but it can come from a different source (e.g., git or a local path).

More information about overriding dependencies can be found in the overriding dependencies section of the documentation.

Profiles

Profiles provide a way to alter the compiler settings, influencing things like optimizations and debugging symbols.

Cargo has 4 built-in profiles: `dev`, `release`, `test`, and `bench`. It automatically chooses the profile based on which command is being run, the package and target that is being built, and command-line flags like `--release`. The selection process is described below.

Profile settings can be changed in `Cargo.toml` with the `[profile]` table. Within each named profile, individual settings can be changed with key/value pairs like this:

```
[profile.dev]
opt-level = 1               # Use slightly better optimizations.
overflow-checks = false     # Disable integer overflow checks.
```

Cargo only looks at the profile settings in the `Cargo.toml` manifest at the root of the workspace. Profile settings defined in dependencies will be ignored.

Profile settings

The following is a list of settings that can be controlled in a profile.

opt-level

The `opt-level` setting controls the `-C opt-level` flag which controls the level of optimization. Higher optimization levels may produce faster runtime code at the expense of longer compiler times. Higher levels may also change and rearrange the compiled code which may make it harder to use with a debugger.

The valid options are:

- `0`: no optimizations, also turns on `cfg(debug_assertions)`.
- `1`: basic optimizations
- `2`: some optimizations
- `3`: all optimizations
- `"s"`: optimize for binary size
- `"z"`: optimize for binary size, but also turn off loop vectorization.

It is recommended to experiment with different levels to find the right balance for your project. There may be surprising results, such as level `3` being slower than `2`, or the `"s"` and `"z"` levels not being necessarily smaller. You may also want to reevaluate your settings over time as newer versions of `rustc` changes optimization behavior.

See also Profile Guided Optimization for more advanced optimization techniques.

debug

The `debug` setting controls the `-C debuginfo` flag which controls the amount of debug information included in the compiled binary.

The valid options are:

- `0` or `false`: no debug info at all
- `1`: line tables only
- `2` or `true`: full debug info

debug-assertions

The `debug-assertions` setting controls the `-C debug-assertions` flag which turns `cfg(debug_assertions)` conditional compilation on or off. Debug assertions are intended to include runtime validation which is only available in debug/development builds. These may be things that are too expensive or otherwise undesirable in a release build. Debug assertions enables the `debug_assert!` macro in the standard library.

The valid options are:

- `true`: enabled
- `false`: disabled

overflow-checks

The `overflow-checks` setting controls the `-C overflow-checks` flag which controls the behavior of runtime integer overflow. When overflow-checks are enabled, a panic will occur on overflow.

The valid options are:

- `true`: enabled
- `false`: disabled

lto

The `lto` setting controls the `-C lto` flag which controls LLVM's link time optimizations. LTO can produce better optimized code, using whole-program analysis, at the cost of longer linking time.

The valid options are:

- `false`: Performs "thin local LTO" which performs "thin" LTO on the local crate only across its codegen units. No LTO is performed if codegen units is 1 or opt-level is 0.
- `true` or `"fat"`: Performs "fat" LTO which attempts to perform optimizations across all crates within the dependency graph.
- `"thin"`: Performs "thin" LTO. This is similar to "fat", but takes substantially less time to run while still achieving performance gains similar to "fat".
- `"off"`: Disables LTO.

See also the `-C linker-plugin-lto` `rustc` flag for cross-language LTO.

panic

The `panic` setting controls the `-C panic` flag which controls which panic strategy to use.

The valid options are:

- `"unwind"`: Unwind the stack upon panic.
- `"abort"`: Terminate the process upon panic.

When set to `"unwind"`, the actual value depends on the default of the target platform. For example, the NVPTX platform does not support unwinding, so it always uses `"abort"`.

Tests, benchmarks, build scripts, and proc macros ignore the `panic` setting. The `rustc` test harness currently requires `unwind` behavior. See the `panic-abort-tests` unstable flag which enables `abort` behavior.

Additionally, when using the `abort` strategy and building a test, all of the dependencies will also be forced to built with the `unwind` strategy.

incremental

The `incremental` setting controls the `-C incremental` flag which controls whether or not incremental compilation is enabled. Incremental compilation causes `rustc` to to save additional information to disk which will be reused when recompiling the crate, improving re-compile times. The additional information is stored in the `target` directory.

The valid options are:

- `true`: enabled
- `false`: disabled

Incremental compilation is only used for workspace members and "path" dependencies.

The incremental value can be overridden globally with the `CARGO_INCREMENTAL` environment variable or the `build.incremental` config variable.

codegen-units

The `codegen-units` setting controls the `-C codegen-units` flag which controls how many "code generation units" a crate will be split into. More code generation units allows more of a crate to be processed in parallel possibly reducing compile time, but may produce slower code.

This option takes an integer greater than 0.

This option is ignored if incremental is enabled, in which case `rustc` uses an internal heuristic to split the crate.

rpath

The `rpath` setting controlls the `-C rpath` flag which controls whether or not `rpath` is enabled.

Default profiles

dev

The `dev` profile is used for normal development and debugging. It is the default for build commands like `cargo build`.

The default settings for the `dev` profile are:

```
[profile.dev]
opt-level = 0
debug = true
debug-assertions = true
overflow-checks = true
lto = false
panic = 'unwind'
incremental = true
codegen-units = 16  # Note: ignored because `incremental` is enabled.
rpath = false
```

release

The `release` profile is intended for optimized artifacts used for releases and in production. This profile is used when the `--release` flag is used, and is the default for `cargo install`.

The default settings for the `release` profile are:

```
[profile.release]
opt-level = 3
debug = false
debug-assertions = false
overflow-checks = false
lto = false
panic = 'unwind'
incremental = false
codegen-units = 16
rpath = false
```

test

The `test` profile is used for building tests, or when benchmarks are built in debug mode with `cargo build`.

The default settings for the `test` profile are:

```
[profile.test]
opt-level = 0
debug = 2
debug-assertions = true
overflow-checks = true
lto = false
panic = 'unwind'    # This setting is always ignored.
incremental = true
codegen-units = 16  # Note: ignored because `incremental` is enabled.
rpath = false
```

bench

The `bench` profile is used for building benchmarks, or when tests are built with the `--release` flag.

The default settings for the `bench` profile are:

```
[profile.bench]
opt-level = 3
debug = false
debug-assertions = false
overflow-checks = false
lto = false
panic = 'unwind'    # This setting is always ignored.
incremental = false
codegen-units = 16
rpath = false
```

Profile selection

The profile used depends on the command, the package, the Cargo target, and command-line flags like `--release`.

Build commands like `cargo build`, `cargo rustc`, `cargo check`, and `cargo run` default to using the `dev` profile. The `--release` flag may be used to switch to the `release` profile.

The `cargo install` command defaults to the `release` profile, and may use the `--debug` flag to switch to the `dev` profile.

Test targets are built with the `test` profile by default. The `--release` flag switches tests to the `bench` profile.

Bench targets are built with the `bench` profile by default. The `cargo build` command can be used to build a bench target with the `test` profile to enable debugging.

Note that when using the `cargo test` and `cargo bench` commands, the `test`/`bench` profiles only apply to the final test executable. Dependencies will continue to use the

`dev` / `release` profiles. Also note that when a library is built for unit tests, then the library is built with the `test` profile. However, when building an integration test target, the library target is built with the `dev` profile and linked into the integration test executable.

Overrides

Profile settings can be overridden for specific packages and build-time crates. To override the settings for a specific package, use the `package` table to change the settings for the named package:

```
# The `foo` package will use the -Copt-level=3 flag.
[profile.dev.package.foo]
opt-level = 3
```

The package name is actually a Package ID Spec, so you can target individual versions of a package with syntax such as `[profile.dev.package."foo:2.1.0"]`.

To override the settings for all dependencies (but not any workspace member), use the `"*"` package name:

```
# Set the default for dependencies.
[profile.dev.package."*"]
opt-level = 2
```

To override the settings for build scripts, proc macros, and their dependencies, use the `build-override` table:

```
# Set the settings for build scripts and proc-macros.
[profile.dev.build-override]
opt-level = 3
```

> Note: When a dependency is both a normal dependency and a build dependency, Cargo will try to only build it once when `--target` is not specified. When using `build-override`, the dependency may need to be built twice, once as a normal dependency and once with the overridden build settings. This may increase initial build times.

The precedence for which value is used is done in the following order (first match wins):

1. `[profile.dev.package.name]` — A named package.
2. `[profile.dev.package."*"]` — For any non-workspace member.
3. `[profile.dev.build-override]` — Only for build scripts, proc macros, and their dependencies.
4. `[profile.dev]` — Settings in `Cargo.toml`.
5. Default values built-in to Cargo.

Overrides cannot specify the `panic`, `lto`, or `rpath` settings.

Overrides and generics

The location where generic code is instantiated will influence the optimization settings used for that generic code. This can cause subtle interactions when using profile overrides to change the optimization level of a specific crate. If you attempt to raise the optimization level of a dependency which defines generic functions, those generic functions may not be optimized when used in your local crate. This is because the code may be generated in the crate where it is instantiated, and thus may use the optimization settings of that crate.

For example, nalgebra is a library which defines vectors and matrices making heavy use of generic parameters. If your local code defines concrete nalgebra types like `Vector4<f64>` and uses their methods, the corresponding nalgebra code will be instantiated and built within your crate. Thus, if you attempt to increase the optimization level of `nalgebra` using a profile override, it may not result in faster performance.

Further complicating the issue, `rustc` has some optimizations where it will attempt to share monomorphized generics between crates. If the opt-level is 2 or 3, then a crate will not use monomorphized generics from other crates, nor will it export locally defined monomorphized items to be shared with other crates. When experimenting with optimizing dependencies for development, consider trying opt-level 1, which will apply some optimizations while still allowing monomorphized items to be shared.

Configuration

This document explains how Cargo's configuration system works, as well as available keys or configuration. For configuration of a package through its manifest, see the manifest format.

Hierarchical structure

Cargo allows local configuration for a particular package as well as global configuration. It looks for configuration files in the current directory and all parent directories. If, for example, Cargo were invoked in `/projects/foo/bar/baz`, then the following configuration files would be probed for and unified in this order:

- `/projects/foo/bar/baz/.cargo/config`
- `/projects/foo/bar/.cargo/config`
- `/projects/foo/.cargo/config`
- `/projects/.cargo/config`
- `/.cargo/config`
- `$CARGO_HOME/config` which defaults to:
  - Windows: `%USERPROFILE%\.cargo\config`
  - Unix: `$HOME/.cargo/config`

With this structure, you can specify configuration per-package, and even possibly check it into version control. You can also specify personal defaults with a configuration file in your home directory.

If a key is specified in multiple config files, the values will get merged together. Numbers, strings, and booleans will use the value in the deeper config directory taking precedence over ancestor directories, where the home directory is the lowest priority. Arrays will be joined together.

Configuration format

Configuration files are written in the TOML format (like the manifest), with simple key-value pairs inside of sections (tables). The following is a quick overview of all settings, with detailed descriptions found below.

```
paths = ["/path/to/override"] # path dependency overrides

[alias]     # command aliases
b = "build"
c = "check"
t = "test"
r = "run"
rr = "run --release"
space_example = ["run", "--release", "--", "\"command list\""]

[build]
jobs = 1                  # number of parallel jobs, defaults to # of CPUs
rustc = "rustc"           # the rust compiler tool
rustc-wrapper = "…"       # run this wrapper instead of `rustc`
rustdoc = "rustdoc"       # the doc generator tool
target = "triple"         # build for the target triple (ignored by `cargo
install`)
target-dir = "target"     # path of where to place all generated artifacts
rustflags = ["…", "…"]    # custom flags to pass to all compiler invocations
rustdocflags = ["…", "…"] # custom flags to pass to rustdoc
incremental = true        # whether or not to enable incremental compilation
dep-info-basedir = "…"    # path for the base directory for targets in depfiles
pipelining = true         # rustc pipelining

[cargo-new]
name = "Your Name"        # name to use in `authors` field
email = "you@example.com" # email address to use in `authors` field
vcs = "none"              # VCS to use ('git', 'hg', 'pijul', 'fossil', 'none')

[http]
debug = false               # HTTP debugging
proxy = "host:port"         # HTTP proxy in libcurl format
ssl-version = "tlsv1.3"     # TLS version to use
ssl-version.max = "tlsv1.3" # maximum TLS version
ssl-version.min = "tlsv1.1" # minimum TLS version
timeout = 30                # timeout for each HTTP request, in seconds
low-speed-limit = 10        # network timeout threshold (bytes/sec)
cainfo = "cert.pem"         # path to Certificate Authority (CA) bundle
check-revoke = true         # check for SSL certificate revocation
multiplexing = true         # HTTP/2 multiplexing
user-agent = "…"            # the user-agent header

[install]
root = "/some/path"         # `cargo install` destination directory

[net]
retry = 2                   # network retries
git-fetch-with-cli = true   # use the `git` executable for git operations
offline = false             # do not access the network

[registries.<name>]  # registries other than crates.io
index = "…"          # URL of the registry index
token = "…"          # authentication token for the registry

[registry]
default = "…"        # name of the default registry
token = "…"          # authentication token for crates.io
```

```
[source.<name>]      # source definition and replacement
replace-with = "…"   # replace this source with the given named source
directory = "…"      # path to a directory source
registry = "…"       # URL to a registry source
local-registry = "…" # path to a local registry source
git = "…"            # URL of a git repository source
branch = "…"         # branch name for the git repository
tag = "…"            # tag name for the git repository
rev = "…"            # revision for the git repository

[target.<triple>]
linker = "…"             # linker to use
runner = "…"             # wrapper to run executables
rustflags = ["…", "…"]  # custom flags for `rustc`

[target.<cfg>]
runner = "…"             # wrapper to run executables
rustflags = ["…", "…"]  # custom flags for `rustc`

[target.<triple>.<links>] # `links` build script override
rustc-link-lib = ["foo"]
rustc-link-search = ["/path/to/foo"]
rustc-flags = ["-L", "/some/path"]
rustc-cfg = ['key="value"']
rustc-env = {key = "value"}
rustc-cdylib-link-arg = ["…"]
metadata_key1 = "value"
metadata_key2 = "value"

[term]
verbose = false        # whether cargo provides verbose output
color = 'auto'         # whether cargo colorizes output
```

Environment variables

Cargo can also be configured through environment variables in addition to the TOML configuration files. For each configuration key of the form `foo.bar` the environment variable `CARGO_FOO_BAR` can also be used to define the value. Keys are converted to uppercase, dots and dashes are converted to underscores. For example the `target.x86_64-unknown-linux-gnu.runner` key can also be defined by the `CARGO_TARGET_X86_64_UNKNOWN_LINUX_GNU_RUNNER` environment variable.

Environment variables will take precedence over TOML configuration files. Currently only integer, boolean, string and some array values are supported to be defined by environment variables. Descriptions below indicate which keys support environment variables.

In addition to the system above, Cargo recognizes a few other specific environment variables.

Config-relative paths

Paths in config files may be absolute, relative, or a bare name without any path separators. Paths for executables without a path separator will use the `PATH` environment variable to search for the executable. Paths for non-executables will be relative to where the config value is defined. For config files, that is relative to the parent directory of the `.cargo` directory where the value was defined. For environment variables it is relative to the current working directory.

```
# Relative path examples.

[target.x86_64-unknown-linux-gnu]
runner = "foo"  # Searches `PATH` for `foo`.

[source.vendored-sources]
# Directory is relative to the parent where `.cargo/config` is located.
# For example, `/my/project/.cargo/config` would result in `/my/project/vendor`.
directory = "vendor"
```

Credentials

Configuration values with sensitive information are stored in the `$CARGO_HOME/credentials` file. This file is automatically created and updated by `cargo login`. It follows the same format as Cargo config files.

```
[registry]
token = "…"   # Access token for crates.io

[registries.<name>]
token = "…"   # Access token for the named registry
```

Tokens are used by some Cargo commands such as `cargo publish` for authenticating with remote registries. Care should be taken to protect the tokens and to keep them secret.

As with most other config values, tokens may be specified with environment variables. The token for crates.io may be specified with the `CARGO_REGISTRY_TOKEN` environment variable. Tokens for other registries may be specified with environment variables of the form `CARGO_REGISTRIES_<name>_TOKEN` where `<name>` is the name of the registry in all capital letters.

Configuration keys

This section documents all configuration keys. The description for keys with variable parts are annotated with angled brackets like `target.<triple>` where the `<triple>` part can be any target triple like `target.x86_64-pc-windows-msvc`.

`paths`

- Type: array of strings (paths)

- Default: none
- Environment: not supported

An array of paths to local packages which are to be used as overrides for dependencies. For more information see the Specifying Dependencies guide.

`[alias]`

- Type: string or array of strings
- Default: see below
- Environment: `CARGO_ALIAS_<name>`

The `[alias]` table defines CLI command aliases. For example, running `cargo b` is an alias for running `cargo build`. Each key in the table is the subcommand, and the value is the actual command to run. The value may be an array of strings, where the first element is the command and the following are arguments. It may also be a string, which will be split on spaces into subcommand and arguments. The following aliases are built-in to Cargo:

```
[alias]
b = "build"
c = "check"
t = "test"
r = "run"
```

Aliases are not allowed to redefine existing built-in commands.

`[build]`

The `[build]` table controls build-time operations and compiler settings.

`build.jobs`

- Type: integer
- Default: number of logical CPUs
- Environment: `CARGO_BUILD_JOBS`

Sets the maximum number of compiler processes to run in parallel.

Can be overridden with the `--jobs` CLI option.

`build.rustc`

- Type: string (program path)
- Default: "rustc"
- Environment: `CARGO_BUILD_RUSTC` or `RUSTC`

Sets the executable to use for `rustc`.

`build.rustc-wrapper`

- Type: string (program path)
- Default: none
- Environment: `CARGO_BUILD_RUSTC_WRAPPER` or `RUSTC_WRAPPER`

Sets a wrapper to execute instead of `rustc`. The first argument passed to the wrapper is the path to the actual `rustc`.

`build.rustdoc`

- Type: string (program path)
- Default: "rustdoc"
- Environment: `CARGO_BUILD_RUSTDOC` or `RUSTDOC`

Sets the executable to use for `rustdoc`.

`build.target`

- Type: string
- Default: host platform
- Environment: `CARGO_BUILD_TARGET`

The default target platform triple to compile to.

This may also be a relative path to a `.json` target spec file.

Can be overridden with the `--target` CLI option.

`build.target-dir`

- Type: string (path)
- Default: "target"
- Environment: `CARGO_BUILD_TARGET_DIR` or `CARGO_TARGET_DIR`

The path to where all compiler output is placed. The default if not specified is a directory named `target` located at the root of the workspace.

Can be overridden with the `--target-dir` CLI option.

`build.rustflags`

- Type: string or array of strings
- Default: none
- Environment: `CARGO_BUILD_RUSTFLAGS` or `RUSTFLAGS`

Extra command-line flags to pass to `rustc`. The value may be a array of strings or a space-separated string.

There are three mutually exclusive sources of extra flags. They are checked in order, with the first one being used:

1. `RUSTFLAGS` environment variable.
2. All matching `target.<triple>.rustflags` and `target.<cfg>.rustflags` config entries joined together.
3. `build.rustflags` config value.

Additional flags may also be passed with the `cargo rustc` command.

If the `--target` flag (or `build.target`) is used, then the flags will only be passed to the compiler for the target. Things being built for the host, such as build scripts or proc macros, will not receive the args. Without `--target`, the flags will be passed to all compiler invocations (including build scripts and proc macros) because dependencies are shared. If you have args that you do not want to pass to build scripts or proc macros and are building for the host, pass `--target` with the host triple.

`build.rustdocflags`

- Type: string or array of strings
- Default: none
- Environment: `CARGO_BUILD_RUSTDOCFLAGS` or `RUSTDOCFLAGS`

Extra command-line flags to pass to `rustdoc`. The value may be a array of strings or a space-separated string.

There are two mutually exclusive sources of extra flags. They are checked in order, with the first one being used:

1. `RUSTDOCFLAGS` environment variable.
2. `build.rustdocflags` config value.

Additional flags may also be passed with the `cargo rustdoc` command.

`build.incremental`

- Type: bool
- Default: from profile
- Environment: `CARGO_BUILD_INCREMENTAL` or `CARGO_INCREMENTAL`

Whether or not to perform incremental compilation. The default if not set is to use the value from the profile. Otherwise this overrides the setting of all profiles.

The `CARGO_INCREMENTAL` environment variable can be set to `1` to force enable incremental compilation for all profiles, or `0` to disable it. This env var overrides the config setting.

`build.dep-info-basedir`

- Type: string (path)

- Default: none
- Environment: `CARGO_BUILD_DEP_INFO_BASEDIR`

Strips the given path prefix from dep info file paths.

Cargo saves a "dep info" file with a `.d` suffix which is a Makefile-like syntax that indicates all of the file dependencies required to rebuild the artifact. These are intended to be used with external build systems so that they can detect if Cargo needs to be re-executed. The paths in the file are absolute by default. This config setting can be set to strip the given prefix from all of the paths for tools that require relative paths.

The setting itself is a config-relative path. So, for example, a value of `"."` would strip all paths starting with the parent directory of the `.cargo` directory.

`build.pipelining`

- Type: boolean
- Default: true
- Environment: `CARGO_BUILD_PIPELINING`

Controls whether or not build pipelining is used. This allows Cargo to schedule overlapping invocations of `rustc` in parallel when possible.

`[cargo-new]`

The `[cargo-new]` table defines defaults for the `cargo new` command.

`cargo-new.name`

- Type: string
- Default: from environment
- Environment: `CARGO_NAME` or `CARGO_CARGO_NEW_NAME`

Defines the name to use in the `authors` field when creating a new `Cargo.toml` file. If not specified in the config, Cargo searches the environment or your `git` configuration as described in the `cargo new` documentation.

`cargo-new.email`

- Type: string
- Default: from environment
- Environment: `CARGO_EMAIL` or `CARGO_CARGO_NEW_EMAIL`

Defines the email address used in the `authors` field when creating a new `Cargo.toml` file. If not specified in the config, Cargo searches the environment or your `git` configuration as described in the `cargo new` documentation. The `email` value may be set to an empty string to prevent Cargo from placing an address in the authors field.

`cargo-new.vcs`

- Type: string
- Default: "git" or "none"
- Environment: `CARGO_CARGO_NEW_VCS`

Specifies the source control system to use for initializing a new repository. Valid values are `git`, `hg` (for Mercurial), `pijul`, `fossil` or `none` to disable this behavior. Defaults to `git`, or `none` if already inside a VCS repository. Can be overridden with the `--vcs` CLI option.

`[http]`

The `[http]` table defines settings for HTTP behavior. This includes fetching crate dependencies and accessing remote git repositories.

`http.debug`

- Type: boolean
- Default: false
- Environment: `CARGO_HTTP_DEBUG`

If `true`, enables debugging of HTTP requests. The debug information can be seen by setting the `CARGO_LOG=cargo::ops::registry=debug` environment variable (or use `trace` for even more information).

Be wary when posting logs from this output in a public location. The output may include headers with authentication tokens which you don't want to leak! Be sure to review logs before posting them.

`http.proxy`

- Type: string
- Default: none
- Environment: `CARGO_HTTP_PROXY` or `HTTPS_PROXY` or `https_proxy` or `http_proxy`

Sets an HTTP and HTTPS proxy to use. The format is in libcurl format as in `[protocol://]host[:port]`. If not set, Cargo will also check the `http.proxy` setting in your global git configuration. If none of those are set, the `HTTPS_PROXY` or `https_proxy` environment variables set the proxy for HTTPS requests, and `http_proxy` sets it for HTTP requests.

`http.timeout`

- Type: integer
- Default: 30
- Environment: `CARGO_HTTP_TIMEOUT` or `HTTP_TIMEOUT`

Sets the timeout for each HTTP request, in seconds.

`http.cainfo`

- Type: string (path)
- Default: none
- Environment: `CARGO_HTTP_CAINFO`

Path to a Certificate Authority (CA) bundle file, used to verify TLS certificates. If not specified, Cargo attempts to use the system certificates.

`http.check-revoke`

- Type: boolean
- Default: true (Windows) false (all others)
- Environment: `CARGO_HTTP_CHECK_REVOKE`

This determines whether or not TLS certificate revocation checks should be performed. This only works on Windows.

`http.ssl-version`

- Type: string or min/max table
- Default: none
- Environment: `CARGO_HTTP_SSL_VERSION`

This sets the minimum TLS version to use. It takes a string, with one of the possible values of "default", "tlsv1", "tlsv1.0", "tlsv1.1", "tlsv1.2", or "tlsv1.3".

This may alternatively take a table with two keys, `min` and `max`, which each take a string value of the same kind that specifies the minimum and maximum range of TLS versions to use.

The default is a minimum version of "tlsv1.0" and a max of the newest version supported on your platform, typically "tlsv1.3".

`http.low-speed-limit`

- Type: integer
- Default: 10
- Environment: `CARGO_HTTP_LOW_SPEED_LIMIT`

This setting controls timeout behavior for slow connections. If the average transfer speed in bytes per second is below the given value for `http.timeout` seconds (default 30 seconds), then the connection is considered too slow and Cargo will abort and retry.

`http.multiplexing`

- Type: boolean
- Default: true
- Environment: `CARGO_HTTP_MULTIPLEXING`

When `true`, Cargo will attempt to use the HTTP2 protocol with multiplexing. This allows multiple requests to use the same connection, usually improving performance when fetching multiple files. If `false`, Cargo will use HTTP 1.1 without pipelining.

`http.user-agent`

- Type: string
- Default: Cargo's version
- Environment: `CARGO_HTTP_USER_AGENT`

Specifies a custom user-agent header to use. The default if not specified is a string that includes Cargo's version.

`[install]`

The `[install]` table defines defaults for the `cargo install` command.

`install.root`

- Type: string (path)
- Default: Cargo's home directory
- Environment: `CARGO_INSTALL_ROOT`

Sets the path to the root directory for installing executables for `cargo install`. Executables go into a `bin` directory underneath the root.

The default if not specified is Cargo's home directory (default `.cargo` in your home directory).

Can be overridden with the `--root` command-line option.

`[net]`

The `[net]` table controls networking configuration.

`net.retry`

- Type: integer
- Default: 2
- Environment: `CARGO_NET_RETRY`

Number of times to retry possibly spurious network errors.

`net.git-fetch-with-cli`

- Type: boolean
- Default: false

- Environment: `CARGO_NET_GIT_FETCH_WITH_CLI`

If this is `true`, then Cargo will use the `git` executable to fetch registry indexes and git dependencies. If `false`, then it uses a built-in `git` library.

Setting this to `true` can be helpful if you have special authentication requirements that Cargo does not support.

`net.offline`

- Type: boolean
- Default: false
- Environment: `CARGO_NET_OFFLINE`

If this is `true`, then Cargo will avoid accessing the network, and attempt to proceed with locally cached data. If `false`, Cargo will access the network as needed, and generate an error if it encounters a network error.

Can be overridden with the `--offline` command-line option.

`[registries]`

The `[registries]` table is used for specifying additional registries. It consists of a sub-table for each named registry.

`registries.<name>.index`

- Type: string (url)
- Default: none
- Environment: `CARGO_REGISTRIES_<name>_INDEX`

Specifies the URL of the git index for the registry.

`registries.<name>.token`

- Type: string
- Default: none
- Environment: `CARGO_REGISTRIES_<name>_TOKEN`

Specifies the authentication token for the given registry. This value should only appear in the credentials file. This is used for registry commands like `cargo publish` that require authentication.

Can be overridden with the `--token` command-line option.

`[registry]`

The `[registry]` table controls the default registry used when one is not specified.

`registry.index`

This value is deprecated and should not be used.

`registry.default`

- Type: string
- Default: `"crates-io"`
- Environment: `CARGO_REGISTRY_DEFAULT`

The name of the registry (from the `registries` table) to use by default for registry commands like `cargo publish`.

Can be overridden with the `--registry` command-line option.

`registry.token`

- Type: string
- Default: none
- Environment: `CARGO_REGISTRY_TOKEN`

Specifies the authentication token for crates.io. This value should only appear in the credentials file. This is used for registry commands like `cargo publish` that require authentication.

Can be overridden with the `--token` command-line option.

`[source]`

The `[source]` table defines the registry sources available. See Source Replacement for more information. It consists of a sub-table for each named source. A source should only define one kind (directory, registry, local-registry, or git).

`source.<name>.replace-with`

- Type: string
- Default: none
- Environment: not supported

If set, replace this source with the given named source.

`source.<name>.directory`

- Type: string (path)
- Default: none
- Environment: not supported

Sets the path to a directory to use as a directory source.

`source.<name>.registry`

- Type: string (url)
- Default: none
- Environment: not supported

Sets the URL to use for a registry source.

`source.<name>.local-registry`

- Type: string (path)
- Default: none
- Environment: not supported

Sets the path to a directory to use as a local registry source.

`source.<name>.git`

- Type: string (url)
- Default: none
- Environment: not supported

Sets the URL to use for a git repository source.

`source.<name>.branch`

- Type: string
- Default: none
- Environment: not supported

Sets the branch name to use for a git repository.

If none of `branch`, `tag`, or `rev` is set, defaults to the `master` branch.

`source.<name>.tag`

- Type: string
- Default: none
- Environment: not supported

Sets the tag name to use for a git repository.

If none of `branch`, `tag`, or `rev` is set, defaults to the `master` branch.

`source.<name>.rev`

- Type: string
- Default: none
- Environment: not supported

Sets the revision to use for a git repository.

If none of `branch`, `tag`, or `rev` is set, defaults to the `master` branch.

`[target]`

The `[target]` table is used for specifying settings for specific platform targets. It consists of a sub-table which is either a platform triple or a `cfg()` expression. The given values will be used if the target platform matches either the `<triple>` value or the `<cfg>` expression.

```
[target.thumbv7m-none-eabi]
linker = "arm-none-eabi-gcc"
runner = "my-emulator"
rustflags = ["…", "…"]

[target.'cfg(all(target_arch = "arm", target_os = "none"))']
runner = "my-arm-wrapper"
rustflags = ["…", "…"]
```

`cfg` values come from those built-in to the compiler (run `rustc --print=cfg` to view), values set by build scripts, and extra `--cfg` flags passed to `rustc` (such as those defined in `RUSTFLAGS`). Do not try to match on `debug_assertions` or Cargo features like `feature="foo"`.

If using a target spec JSON file, the `<triple>` value is the filename stem. For example `--target foo/bar.json` would match `[target.bar]`.

`target.<triple>.ar`

This option is deprecated and unused.

`target.<triple>.linker`

- Type: string (program path)
- Default: none
- Environment: `CARGO_TARGET_<triple>_LINKER`

Specifies the linker which is passed to `rustc` (via `-C linker`) when the `<triple>` is being compiled for. By default, the linker is not overridden.

`target.<triple>.runner`

- Type: string or array of strings (program path and args)
- Default: none
- Environment: `CARGO_TARGET_<triple>_RUNNER`

If a runner is provided, executables for the target `<triple>` will be executed by invoking the specified runner with the actual executable passed as an argument. This applies to `cargo`

`run`, `cargo test` and `cargo bench` commands. By default, compiled executables are executed directly.

The value may be an array of strings like `['/path/to/program', 'somearg']` or a space-separated string like `'/path/to/program somearg'`. The arguments will be passed to the runner with the executable to run as the last argument. If the runner program does not have path separators, it will search `PATH` for the runner executable.

`target.<cfg>.runner`

This is similar to the target runner, but using a `cfg()` expression. If both a `<triple>` and `<cfg>` runner match, the `<triple>` will take precedence. It is an error if more than one `<cfg>` runner matches the current target.

`target.<triple>.rustflags`

- Type: string or array of strings
- Default: none
- Environment: `CARGO_TARGET_<triple>_RUSTFLAGS`

Passes a set of custom flags to the compiler for this `<triple>`. The value may be a array of strings or a space-separated string.

See `build.rustflags` for more details on the different ways to specific extra flags.

`target.<cfg>.rustflags`

This is similar to the target rustflags, but using a `cfg()` expression. If several `<cfg>` and `<triple>` entries match the current target, the flags are joined together.

`target.<triple>.<links>`

The links sub-table provides a way to override a build script. When specified, the build script for the given `links` library will not be run, and the given values will be used instead.

```
[target.x86_64-unknown-linux-gnu.foo]
rustc-link-lib = ["foo"]
rustc-link-search = ["/path/to/foo"]
rustc-flags = "-L /some/path"
rustc-cfg = ['key="value"']
rustc-env = {key = "value"}
rustc-cdylib-link-arg = ["…"]
metadata_key1 = "value"
metadata_key2 = "value"
```

`[term]`

The `[term]` table controls terminal output and interaction.

`term.verbose`

- Type: boolean
- Default: false
- Environment: `CARGO_TERM_VERBOSE`

Controls whether or not extra detailed messages are displayed by Cargo.

Specifying the `--quiet` flag will override and disable verbose output. Specifying the `--verbose` flag will override and force verbose output.

`term.color`

- Type: string
- Default: "auto"
- Environment: `CARGO_TERM_COLOR`

Controls whether or not colored output is used in the terminal. Possible values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

Can be overridden with the `--color` command-line option.

Environment Variables

Cargo sets and reads a number of environment variables which your code can detect or override. Here is a list of the variables Cargo sets, organized by when it interacts with them:

Environment variables Cargo reads

You can override these environment variables to change Cargo's behavior on your system:

- `CARGO_HOME` — Cargo maintains a local cache of the registry index and of git checkouts of crates. By default these are stored under `$HOME/.cargo` (`%USERPROFILE%\.cargo` on Windows), but this variable overrides the location of this directory. Once a crate is cached it is not removed by the clean command. For more details refer to the guide.
- `CARGO_TARGET_DIR` — Location of where to place all generated artifacts, relative to the current working directory. See `build.target-dir` to set via config.
- `RUSTC` — Instead of running `rustc`, Cargo will execute this specified compiler instead. See `build.rustc` to set via config.
- `RUSTC_WRAPPER` — Instead of simply running `rustc`, Cargo will execute this specified wrapper instead, passing as its commandline arguments the rustc invocation, with the first argument being `rustc`. Useful to set up a build cache tool such as `sccache`. See `build.rustc-wrapper` to set via config.

- `RUSTDOC` — Instead of running `rustdoc`, Cargo will execute this specified `rustdoc` instance instead. See `build.rustdoc` to set via config.
- `RUSTDOCFLAGS` — A space-separated list of custom flags to pass to all `rustdoc` invocations that Cargo performs. In contrast with [`cargo rustdoc`], this is useful for passing a flag to *all* `rustdoc` instances. See `build.rustdocflags` for some more ways to set flags.
- `RUSTFLAGS` — A space-separated list of custom flags to pass to all compiler invocations that Cargo performs. In contrast with `cargo rustc`, this is useful for passing a flag to *all* compiler instances. See `build.rustflags` for some more ways to set flags.
- `CARGO_INCREMENTAL` — If this is set to 1 then Cargo will force incremental compilation to be enabled for the current compilation, and when set to 0 it will force disabling it. If this env var isn't present then cargo's defaults will otherwise be used. See also `build.incremental` config value.
- `CARGO_CACHE_RUSTC_INFO` — If this is set to 0 then Cargo will not try to cache compiler version information.
- `CARGO_NAME` — The author name to use for `cargo new`.
- `CARGO_EMAIL` — The author email to use for `cargo new`.
- `HTTPS_PROXY` or `https_proxy` or `http_proxy` — The HTTP proxy to use, see `http.proxy` for more detail.
- `HTTP_TIMEOUT` — The HTTP timeout in seconds, see `http.timeout` for more detail.
- `TERM` — If this is set to `dumb`, it disables the progress bar.
- `BROWSER` — The web browser to execute to open documentation with `cargo doc`'s' `--open` flag.

Configuration environment variables

Cargo reads environment variables for configuration values. See the configuration chapter for more details. In summary, the supported environment variables are:

- `CARGO_ALIAS_<name>` — Command aliases, see `alias`.
- `CARGO_BUILD_JOBS` — Number of parallel jobs, see `build.jobs`.
- `CARGO_BUILD_RUSTC` — The `rustc` executable, see `build.rustc`.
- `CARGO_BUILD_RUSTC_WRAPPER` — The `rustc` wrapper, see `build.rustc-wrapper`.
- `CARGO_BUILD_RUSTDOC` — The `rustdoc` executable, see `build.rustdoc`.
- `CARGO_BUILD_TARGET` — The default target platform, see `build.target`.
- `CARGO_BUILD_TARGET_DIR` — The default output directory, see `build.target-dir`.
- `CARGO_BUILD_RUSTFLAGS` — Extra `rustc` flags, see `build.rustflags`.
- `CARGO_BUILD_RUSTDOCFLAGS` — Extra `rustdoc` flags, see `build.rustdocflags`.
- `CARGO_BUILD_INCREMENTAL` — Incremental compilation, see `build.incremental`.
- `CARGO_BUILD_DEP_INFO_BASEDIR` — Dep-info relative directory, see `build.dep-info-basedir`.
- `CARGO_BUILD_PIPELINING` — Whether or not to use `rustc` pipelining, see `build.pipelining`.
- `CARGO_CARGO_NEW_NAME` — The author name to use with `cargo new`, see `cargo-new.name`.
- `CARGO_CARGO_NEW_EMAIL` — The author email to use with `cargo new`, see `cargo-new.email`.

- `CARGO_CARGO_NEW_VCS` — The default source control system with `cargo new`, see `cargo-new.vcs`.
- `CARGO_HTTP_DEBUG` — Enables HTTP debugging, see `http.debug`.
- `CARGO_HTTP_PROXY` — Enables HTTP proxy, see `http.proxy`.
- `CARGO_HTTP_TIMEOUT` — The HTTP timeout, see `http.timeout`.
- `CARGO_HTTP_CAINFO` — The TLS certificate Certificate Authority file, see `http.cainfo`.
- `CARGO_HTTP_CHECK_REVOKE` — Disables TLS certificate revocation checks, see `http.check-revoke`.
- `CARGO_HTTP_SSL_VERSION` — The TLS version to use, see `http.ssl-version`.
- `CARGO_HTTP_LOW_SPEED_LIMIT` — The HTTP low-speed limit, see `http.low-speed-limit`.
- `CARGO_HTTP_MULTIPLEXING` — Whether HTTP/2 multiplexing is used, see `http.multiplexing`.
- `CARGO_HTTP_USER_AGENT` — The HTTP user-agent header, see `http.user-agent`.
- `CARGO_INSTALL_ROOT` — The default directory for `cargo install`, see `install.root`.
- `CARGO_NET_RETRY` — Number of times to retry network errors, see `net.retry`.
- `CARGO_NET_GIT_FETCH_WITH_CLI` — Enables the use of the `git` executable to fetch, see `net.git-fetch-with-cli`.
- `CARGO_NET_OFFLINE` — Offline mode, see `net.offline`.
- `CARGO_REGISTRIES_<name>_INDEX` — URL of a registry index, see `registries.<name>.index`.
- `CARGO_REGISTRIES_<name>_TOKEN` — Authentication token of a registry, see `registries.<name>.token`.
- `CARGO_REGISTRY_DEFAULT` — Default registry for the `--registry` flag, see `registry.default`.
- `CARGO_REGISTRY_TOKEN` — Authentication token for crates.io, see `registry.token`.
- `CARGO_TARGET_<triple>_LINKER` — The linker to use, see `target.<triple>.linker`.
- `CARGO_TARGET_<triple>_RUNNER` — The executable runner, see `target.<triple>.runner`.
- `CARGO_TARGET_<triple>_RUSTFLAGS` — Extra `rustc` flags for a target, see `target.<triple>.rustflags`.
- `CARGO_TERM_VERBOSE` — The default terminal verbosity, see `term.verbose`.
- `CARGO_TERM_COLOR` — The default color mode, see `term.color`.

Environment variables Cargo sets for crates

Cargo exposes these environment variables to your crate when it is compiled. Note that this applies for running binaries with `cargo run` and `cargo test` as well. To get the value of any of these variables in a Rust program, do this:

```
let version = env!("CARGO_PKG_VERSION");
```

`version` will now contain the value of `CARGO_PKG_VERSION`.

- `CARGO` - Path to the `cargo` binary performing the build.

- `CARGO_MANIFEST_DIR` - The directory containing the manifest of your package.
- `CARGO_PKG_VERSION` - The full version of your package.
- `CARGO_PKG_VERSION_MAJOR` - The major version of your package.
- `CARGO_PKG_VERSION_MINOR` - The minor version of your package.
- `CARGO_PKG_VERSION_PATCH` - The patch version of your package.
- `CARGO_PKG_VERSION_PRE` - The pre-release version of your package.
- `CARGO_PKG_AUTHORS` - Colon separated list of authors from the manifest of your package.
- `CARGO_PKG_NAME` - The name of your package.
- `CARGO_PKG_DESCRIPTION` - The description from the manifest of your package.
- `CARGO_PKG_HOMEPAGE` - The home page from the manifest of your package.
- `CARGO_PKG_REPOSITORY` - The repository from the manifest of your package.
- `OUT_DIR` - If the package has a build script, this is set to the folder where the build script should place its output. See below for more information. (Only set during compilation.)

Dynamic library paths

Cargo also sets the dynamic library path when compiling and running binaries with commands like `cargo run` and `cargo test`. This helps with locating shared libraries that are part of the build process. The variable name depends on the platform:

- Windows: `PATH`
- macOS: `DYLD_FALLBACK_LIBRARY_PATH`
- Unix: `LD_LIBRARY_PATH`

The value is extended from the existing value when Cargo starts. macOS has special consideration where if `DYLD_FALLBACK_LIBRARY_PATH` is not already set, it will add the default `$HOME/lib:/usr/local/lib:/usr/lib`.

Cargo includes the following paths:

- Search paths included from any build script with the `rustc-link-search` instruction. Paths outside of the `target` directory are removed. It is the responsibility of the user running Cargo to properly set the environment if additional libraries on the system are needed in the search path.
- The base output directory, such as `target/debug`, and the "deps" directory. This is mostly for legacy support of `rustc` compiler plugins.
- The rustc sysroot library path. This generally is not important to most users.

Environment variables Cargo sets for build scripts

Cargo sets several environment variables when build scripts are run. Because these variables are not yet set when the build script is compiled, the above example using `env!` won't work and instead you'll need to retrieve the values when the build script is run:

```
use std::env;
let out_dir = env::var("OUT_DIR").unwrap();
```

`out_dir` will now contain the value of `OUT_DIR`.

- `CARGO` - Path to the `cargo` binary performing the build.
- `CARGO_MANIFEST_DIR` - The directory containing the manifest for the package being built (the package containing the build script). Also note that this is the value of the current working directory of the build script when it starts.
- `CARGO_MANIFEST_LINKS` - the manifest `links` value.
- `CARGO_FEATURE_<name>` - For each activated feature of the package being built, this environment variable will be present where `<name>` is the name of the feature uppercased and having `-` translated to `_`.
- `CARGO_CFG_<cfg>` - For each configuration option of the package being built, this environment variable will contain the value of the configuration, where `<cfg>` is the name of the configuration uppercased and having `-` translated to `_`. Boolean configurations are present if they are set, and not present otherwise. Configurations with multiple values are joined to a single variable with the values delimited by `,`. This includes values built-in to the compiler (which can be seen with `rustc --print=cfg`) and values set by build scripts and extra flags passed to `rustc` (such as those defined in `RUSTFLAGS`). Some examples of what these variables are:
  - `CARGO_CFG_UNIX` — Set on unix-like platforms.
  - `CARGO_CFG_WINDOWS` — Set on windows-like platforms.
  - `CARGO_CFG_TARGET_FAMILY=unix` — The target family, either `unix` or `windows`.
  - `CARGO_CFG_TARGET_OS=macos` — The target operating system.
  - `CARGO_CFG_TARGET_ARCH=x86_64` — The CPU target architecture.
  - `CARGO_CFG_TARGET_VENDOR=apple` — The target vendor.
  - `CARGO_CFG_TARGET_ENV=gnu` — The target environment ABI.
  - `CARGO_CFG_TARGET_POINTER_WIDTH=64` — The CPU pointer width.
  - `CARGO_CFG_TARGET_ENDIAN=little` — The CPU target endianess.
  - `CARGO_CFG_TARGET_FEATURE=mmx,sse` — List of CPU target features enabled.
- `OUT_DIR` - the folder in which all output should be placed. This folder is inside the build directory for the package being built, and it is unique for the package in question.
- `TARGET` - the target triple that is being compiled for. Native code should be compiled for this triple. See the Target Triple description for more information.
- `HOST` - the host triple of the rust compiler.
- `NUM_JOBS` - the parallelism specified as the top-level parallelism. This can be useful to pass a `-j` parameter to a system like `make`. Note that care should be taken when interpreting this environment variable. For historical purposes this is still provided but recent versions of Cargo, for example, do not need to run `make -j` as it'll automatically happen. Cargo implements its own jobserver and will allow build scripts to inherit this information, so programs compatible with GNU make jobservers will already have appropriately configured parallelism.
- `OPT_LEVEL`, `DEBUG` - values of the corresponding variables for the profile currently being built.
- `PROFILE` - `release` for release builds, `debug` for other builds.

- `DEP_<name>_<key>` - For more information about this set of environment variables, see build script documentation about `links`.
- `RUSTC`, `RUSTDOC` - the compiler and documentation generator that Cargo has resolved to use, passed to the build script so it might use it as well.
- `RUSTC_LINKER` - The path to the linker binary that Cargo has resolved to use for the current target, if specified. The linker can be changed by editing `.cargo/config`; see the documentation about cargo configuration for more information.

Environment variables Cargo sets for 3rd party subcommands

Cargo exposes this environment variable to 3rd party subcommands (ie. programs named `cargo-foobar` placed in `$PATH`):

- `CARGO` - Path to the `cargo` binary performing the build.

Build Scripts

Some packages need to compile third-party non-Rust code, for example C libraries. Other packages need to link to C libraries which can either be located on the system or possibly need to be built from source. Others still need facilities for functionality such as code generation before building (think parser generators).

Cargo does not aim to replace other tools that are well-optimized for these tasks, but it does integrate with them with custom build scripts. Placing a file named `build.rs` in the root of a package will cause Cargo to compile that script and execute it just before building the package.

```
// Example custom build script.
fn main() {
    // Tell Cargo that if the given file changes, to rerun this build script.
    println!("cargo:rerun-if-changed=src/hello.c");
    // Use the `cc` crate to build a C file and statically link it.
    cc::Build::new()
        .file("src/hello.c")
        .compile("hello");
}
```

Some example use cases of build scripts are:

- Building a bundled C library.
- Finding a C library on the host system.
- Generating a Rust module from a specification.
- Performing any platform-specific configuration needed for the crate.

The sections below describe how build scripts work, and the examples chapter shows a variety of examples on how to write scripts.

> Note: The `package.build` manifest key can be used to change the name of the build script, or disable it entirely.

Life Cycle of a Build Script

Just before a package is built, Cargo will compile a build script into an executable (if it has not already been built). It will then run the script, which may perform any number of tasks. The script may communicate with Cargo by printing specially formated commands prefixed with `cargo:` to stdout.

The build script will be rebuilt if any of its source files or dependencies change.

By default, Cargo will re-run the build script if any of the files in the package changes. Typically it is best to use the `rerun-if` commands, described in the change detection section below, to narrow the focus of what triggers a build script to run again.

Once the build script successfully finishes executing, the rest of the package will be compiled. Scripts should exit with a non-zero exit code to halt the build if there is an error, in which case the build script's output will be displayed on the terminal.

Inputs to the Build Script

When the build script is run, there are a number of inputs to the build script, all passed in the form of environment variables.

In addition to environment variables, the build script's current directory is the source directory of the build script's package.

Outputs of the Build Script

Build scripts may save any output files in the directory specified in the `OUT_DIR` environment variable. Scripts should not modify any files outside of that directory.

Build scripts communicate with Cargo by printing to stdout. Cargo will interpret each line that starts with `cargo:` as an instruction that will influence compilation of the package. All other lines are ignored.

The output of the script is hidden from the terminal during normal compilation. If you would like to see the output directly in your terminal, invoke Cargo as "very verbose" with the `-vv` flag. This only happens when the build script is run. If Cargo determines nothing has changed, it will not re-run the script, see change detection below for more.

All the lines printed to stdout by a build script are written to a file like `target/debug/build/<pkg>/output` (the precise location may depend on your

configuration). The stderr output is also saved in that same directory.

The following is a summary of the instructions that Cargo recognizes, with each one detailed below.

- `cargo:rerun-if-changed=PATH` — Tells Cargo when to re-run the script.
- `cargo:rerun-if-env-changed=VAR` — Tells Cargo when to re-run the script.
- `cargo:rustc-link-lib=[KIND=]NAME` — Adds a library to link.
- `cargo:rustc-link-search=[KIND=]PATH` — Adds to the library search path.
- `cargo:rustc-flags=FLAGS` — Passes certain flags to the compiler.
- `cargo:rustc-cfg=KEY[="VALUE"]` — Enables compile-time `cfg` settings.
- `cargo:rustc-env=VAR=VALUE` — Sets an environment variable.
- `cargo:rustc-cdylib-link-arg=FLAG` — Passes custom flags to a linker for cdylib crates.
- `cargo:warning=MESSAGE` — Displays a warning on the terminal.
- `cargo:KEY=VALUE` — Metadata, used by `links` scripts.

`cargo:rustc-link-lib=[KIND=]NAME`

The `rustc-link-lib` instruction tells Cargo to link the given library using the compiler's `-l` flag. This is typically used to link a native library using FFI.

The `-l` flag is only passed to the library target of the package, unless there is no library target, in which case it is passed to all targets. This is done because all other targets have an implicit dependency on the library target, and the given library to link should only be included once. This means that if a package has both a library and a binary target, the *library* has access to the symbols from the given lib, and the binary should access them through the library target's public API.

The optional `KIND` may be one of `dylib`, `static`, or `framework`. See the rustc book for more detail.

`cargo:rustc-link-search=[KIND=]PATH`

The `rustc-link-search` instruction tells Cargo to pass the `-L` flag to the compiler to add a directory to the library search path.

The optional `KIND` may be one of `dependency`, `crate`, `native`, `framework`, or `all`. See the rustc book for more detail.

These paths are also added to the dynamic library search path environment variable if they are within the `OUT_DIR`. Depending on this behavior is discouraged since this makes it difficult to use the resulting binary. In general, it is best to avoid creating dynamic libraries in a build script (using existing system libraries is fine).

`cargo:rustc-flags=FLAGS`

The `rustc-flags` instruction tells Cargo to pass the given space-separated flags to the compiler. This only allows the `-l` and `-L` flags, and is equivalent to using `rustc-link-lib` and `rustc-link-search`.

`cargo:rustc-cfg=KEY[="VALUE"]`

The `rustc-cfg` instruction tells Cargo to pass the given value to the `--cfg` flag to the compiler. This may be used for compile-time detection of features to enable conditional compilation.

Note that this does *not* affect Cargo's dependency resolution. This cannot be used to enable an optional dependency, or enable other Cargo features.

Be aware that Cargo features use the form `feature="foo"`. `cfg` values passed with this flag are not restricted to that form, and may provide just a single identifier, or any arbitrary key/value pair. For example, emitting `cargo:rustc-cfg=abc` will then allow code to use `#[cfg(abc)]` (note the lack of `feature=`). Or an arbitrary key/value pair may be used with an `=` symbol like `cargo:rustc-cfg=my_component="foo"`. The key should be a Rust identifier, the value should be a string.

`cargo:rustc-env=VAR=VALUE`

The `rustc-env` instruction tells Cargo to set the given environment variable when compiling the package. The value can be then retrieved by the `env!` macro in the compiled crate. This is useful for embedding additional metadata in crate's code, such as the hash of git HEAD or the unique identifier of a continuous integration server.

See also the environment variables automatically included by Cargo.

`cargo:rustc-cdylib-link-arg=FLAG`

The `rustc-cdylib-link-arg` instruction tells Cargo to pass the `-C link-arg=FLAG` option to the compiler, but only when building a `cdylib` library target. Its usage is highly platform specific. It is useful to set the shared library version or the runtime-path.

`cargo:warning=MESSAGE`

The `warning` instruction tells Cargo to display a warning after the build script has finished running. Warnings are only shown for `path` dependencies (that is, those you're working on locally), so for example warnings printed out in crates.io crates are not emitted by default. The `-vv` "very verbose" flag may be used to have Cargo display warnings for all crates.

Build Dependencies

Build scripts are also allowed to have dependencies on other Cargo-based crates. Dependencies are declared through the `build-dependencies` section of the manifest.

```
[build-dependencies]
cc = "1.0.46"
```

The build script **does not** have access to the dependencies listed in the `dependencies` or `dev-dependencies` section (they're not built yet!). Also, build dependencies are not available to the package itself unless also explicitly added in the `[dependencies]` table.

It is recommended to carefully consider each dependency you add, weighing against the impact on compile time, licensing, maintenance, etc. Cargo will attempt to reuse a dependency if it is shared between build dependencies and normal dependencies. However, this is not always possible, for example when cross-compiling, so keep that in consideration of the impact on compile time.

Change Detection

When rebuilding a package, Cargo does not necessarily know if the build script needs to be run again. By default, it takes a conservative approach of always re-running the build script if any file within the package is changed. For most cases, this is not a good choice, so it is recommended that every build script emit at least one of the `rerun-if` instructions (described below). If these are emitted, then Cargo will only re-run the script if the given value has changed.

`cargo:rerun-if-changed=PATH`

The `rerun-if-changed` instruction tells Cargo to re-run the build script if the file at the given path has changed. Currently, Cargo only uses the filesystem last-modified "mtime" timestamp to determine if the file has changed. It compares against an internal cached timestamp of when the build script last ran.

If the path points to a directory, it does *not* automatically traverse the directory for changes. Only the mtime change of the directory itself is considered (which corresponds to some types of changes within the directory, depending on platform). To request a re-run on any changes within an entire directory, print a line for the directory and separate lines for everything inside it, recursively.

If the build script inherently does not need to re-run under any circumstance, then emitting `cargo:rerun-if-changed=build.rs` is a simple way to prevent it from being re-run. Cargo automatically handles whether or not the script itself needs to be recompiled, and of course the script will be re-run after it has been recompiled. Otherwise, specifying `build.rs` is redundant and unnecessary.

`cargo:rerun-if-env-changed=NAME`

The `rerun-if-env-changed` instruction tells Cargo to re-run the build script if the value of an environment variable of the given name has changed.

Note that the environment variables here are intended for global environment variables like `CC` and such, it is not necessary to use this for environment variables like `TARGET` that Cargo sets.

The `links` Manifest Key

The `package.links` key may be set in the `Cargo.toml` manifest to declare that the package links with the given native library. The purpose of this manifest key is to give Cargo an understanding about the set of native dependencies that a package has, as well as providing a principled system of passing metadata between package build scripts.

```
[package]
# ...
links = "foo"
```

This manifest states that the package links to the `libfoo` native library. When using the `links` key, the package must have a build script, and the build script should use the `rustc-link-lib` instruction to link the library.

Primarily, Cargo requires that there is at most one package per `links` value. In other words, it is forbidden to have two packages link to the same native library. This helps prevent duplicate symbols between crates. Note, however, that there are conventions in place to alleviate this.

As mentioned above in the output format, each build script can generate an arbitrary set of metadata in the form of key-value pairs. This metadata is passed to the build scripts of **dependent** packages. For example, if the package `bar` depends on `foo`, then if `foo` generates `key=value` as part of its build script metadata, then the build script of `bar` will have the environment variables `DEP_FOO_KEY=value`. See the "Using another `sys` crate" for an example of how this can be used.

Note that metadata is only passed to immediate dependents, not transitive dependents.

`*-sys` Packages

Some Cargo packages that link to system libraries have a naming convention of having a `-sys` suffix. Any package named `foo-sys` should provide two major pieces of functionality:

- The library crate should link to the native library `libfoo`. This will often probe the current system for `libfoo` before resorting to building from source.
- The library crate should provide **declarations** for functions in `libfoo`, but **not** bindings or higher-level abstractions.

The set of `*-sys` packages provides a common set of dependencies for linking to native libraries. There are a number of benefits earned from having this convention of native-library-related packages:

- Common dependencies on `foo-sys` alleviates the rule about one package per value of `links`.
- Other `-sys` packages can take advantage of the `DEP_NAME_KEY=value` environment variables to better integrate with other packages. See the "Using another `sys` crate" example.
- A common dependency allows centralizing logic on discovering `libfoo` itself (or building it from source).
- These dependencies are easily overridable.

It is common to have a companion package without the `-sys` suffix that provides a safe, high-level abstractions on top of the sys package. For example, the `git2` crate provides a high-level interface to the `libgit2-sys` crate.

Overriding Build Scripts

If a manifest contains a `links` key, then Cargo supports overriding the build script specified with a custom library. The purpose of this functionality is to prevent running the build script in question altogether and instead supply the metadata ahead of time.

To override a build script, place the following configuration in any acceptable Cargo configuration location.

```
[target.x86_64-unknown-linux-gnu.foo]
rustc-link-lib = ["foo"]
rustc-link-search = ["/path/to/foo"]
rustc-flags = "-L /some/path"
rustc-cfg = ['key="value"']
rustc-env = {key = "value"}
rustc-cdylib-link-arg = ["…"]
metadata_key1 = "value"
metadata_key2 = "value"
```

With this configuration, if a package declares that it links to `foo` then the build script will **not** be compiled or run, and the metadata specified will be used instead.

The `warning`, `rerun-if-changed`, and `rerun-if-env-changed` keys should not be used and will be ignored.

Jobserver

Cargo and `rustc` use the jobserver protocol, developed for GNU make, to coordinate concurrency across processes. It is essentially a semaphore that controls the number of

jobs running concurrently. The concurrency may be set with the `--jobs` flag, which defaults to the number of logical CPUs.

Each build script inherits one job slot from Cargo, and should endeavor to only use one CPU while it runs. If the script wants to use more CPUs in parallel, it should use the `jobserver` crate to coordinate with Cargo.

As an example, the `cc` crate may enable the optional `parallel` feature which will use the jobserver protocol to attempt to build multiple C files at the same time.

Build Script Examples

The following sections illustrate some examples of writing build scripts.

Some common build script functionality can be found via crates on crates.io. Check out the `build-dependencies` keyword to see what is available. The following is a sample of some popular crates[1]:

- `bindgen` — Automatically generate Rust FFI bindings to C libraries.
- `cc` — Compiles C/C++/assembly.
- `pkg-config` — Detect system libraries using the `pkg-config` utility.
- `cmake` — Runs the `cmake` build tool to build a native library.
- `autocfg`, `rustc_version`, `version_check` — These crates provide ways to implement conditional compilation based on the current `rustc` such as the version of the compiler.

[1] This list is not an endorsement. Evaluate your dependencies to see which is right for your project.

Code generation

Some Cargo packages need to have code generated just before they are compiled for various reasons. Here we'll walk through a simple example which generates a library call as part of the build script.

First, let's take a look at the directory structure of this package:

```
.
├── Cargo.toml
├── build.rs
└── src
    └── main.rs

1 directory, 3 files
```

Here we can see that we have a `build.rs` build script and our binary in `main.rs`. This package has a basic manifest:

```
# Cargo.toml

[package]
name = "hello-from-generated-code"
version = "0.1.0"
```

Let's see what's inside the build script:

```
// build.rs

use std::env;
use std::fs;
use std::path::Path;

fn main() {
    let out_dir = env::var_os("OUT_DIR").unwrap();
    let dest_path = Path::new(&out_dir).join("hello.rs");
    fs::write(
        &dest_path,
        "pub fn message() -> &'static str {
            \"Hello, World!\"
        }
        "
    ).unwrap();
    println!("cargo:rerun-if-changed=build.rs");
}
```

There's a couple of points of note here:

- The script uses the `OUT_DIR` environment variable to discover where the output files should be located. It can use the process' current working directory to find where the input files should be located, but in this case we don't have any input files.
- In general, build scripts should not modify any files outside of `OUT_DIR`. It may seem fine on the first blush, but it does cause problems when you use such crate as a dependency, because there's an *implicit* invariant that sources in `.cargo/registry` should be immutable. `cargo` won't allow such scripts when packaging.
- This script is relatively simple as it just writes out a small generated file. One could imagine that other more fanciful operations could take place such as generating a Rust module from a C header file or another language definition, for example.
- The `rerun-if-changed` instruction tells Cargo that the build script only needs to re-run if the build script itself changes. Without this line, Cargo will automatically run the build script if any file in the package changes. If your code generation uses some input files, this is where you would print a list of each of those files.

Next, let's peek at the library itself:

```
// src/main.rs

include!(concat!(env!("OUT_DIR"), "/hello.rs"));

fn main() {
    println!("{}", message());
}
```

This is where the real magic happens. The library is using the rustc-defined `include!` macro in combination with the `concat!` and `env!` macros to include the generated file (`hello.rs`) into the crate's compilation.

Using the structure shown here, crates can include any number of generated files from the build script itself.

Building a native library

Sometimes it's necessary to build some native C or C++ code as part of a package. This is another excellent use case of leveraging the build script to build a native library before the Rust crate itself. As an example, we'll create a Rust library which calls into C to print "Hello, World!".

Like above, let's first take a look at the package layout:

```
.
├── Cargo.toml
├── build.rs
└── src
    ├── hello.c
    └── main.rs

1 directory, 4 files
```

Pretty similar to before! Next, the manifest:

```
# Cargo.toml

[package]
name = "hello-world-from-c"
version = "0.1.0"
edition = "2018"
```

For now we're not going to use any build dependencies, so let's take a look at the build script now:

```
// build.rs

use std::process::Command;
use std::env;
use std::path::Path;

fn main() {
    let out_dir = env::var("OUT_DIR").unwrap();

    // Note that there are a number of downsides to this approach, the comments
    // below detail how to improve the portability of these commands.
    Command::new("gcc").args(&["src/hello.c", "-c", "-fPIC", "-o"])
                       .arg(&format!("{}/hello.o", out_dir))
                       .status().unwrap();
    Command::new("ar").args(&["crus", "libhello.a", "hello.o"])
                      .current_dir(&Path::new(&out_dir))
                      .status().unwrap();

    println!("cargo:rustc-link-search=native={}", out_dir);
    println!("cargo:rustc-link-lib=static=hello");
    println!("cargo:rerun-if-changed=src/hello.c");
}
```

This build script starts out by compiling our C file into an object file (by invoking `gcc`) and then converting this object file into a static library (by invoking `ar`). The final step is feedback to Cargo itself to say that our output was in `out_dir` and the compiler should link the crate to `libhello.a` statically via the `-l static=hello` flag.

Note that there are a number of drawbacks to this hard-coded approach:

- The `gcc` command itself is not portable across platforms. For example it's unlikely that Windows platforms have `gcc`, and not even all Unix platforms may have `gcc`. The `ar` command is also in a similar situation.
- These commands do not take cross-compilation into account. If we're cross compiling for a platform such as Android it's unlikely that `gcc` will produce an ARM executable.

Not to fear, though, this is where a `build-dependencies` entry would help! The Cargo ecosystem has a number of packages to make this sort of task much easier, portable, and standardized. Let's try the `cc` crate from crates.io. First, add it to the `build-dependencies` in `Cargo.toml`:

```
[build-dependencies]
cc = "1.0"
```

And rewrite the build script to use this crate:

```
// build.rs

fn main() {
    cc::Build::new()
        .file("src/hello.c")
        .compile("hello");
    println!("cargo:rerun-if-changed=src/hello.c");
}
```

The `cc` crate abstracts a range of build script requirements for C code:

- It invokes the appropriate compiler (MSVC for windows, `gcc` for MinGW, `cc` for Unix platforms, etc.).
- It takes the `TARGET` variable into account by passing appropriate flags to the compiler being used.
- Other environment variables, such as `OPT_LEVEL`, `DEBUG`, etc., are all handled automatically.
- The stdout output and `OUT_DIR` locations are also handled by the `cc` library.

Here we can start to see some of the major benefits of farming as much functionality as possible out to common build dependencies rather than duplicating logic across all build scripts!

Back to the case study though, let's take a quick look at the contents of the `src` directory:

```
// src/hello.c

#include <stdio.h>

void hello() {
    printf("Hello, World!\n");
}
```

```
// src/main.rs

// Note the lack of the `#[link]` attribute. We're delegating the responsibility
// of selecting what to link over to the build script rather than hard-coding
// it in the source file.
extern { fn hello(); }

fn main() {
    unsafe { hello(); }
}
```

And there we go! This should complete our example of building some C code from a Cargo package using the build script itself. This also shows why using a build dependency can be crucial in many situations and even much more concise!

We've also seen a brief example of how a build script can use a crate as a dependency purely for the build process and not for the crate itself at runtime.

Linking to system libraries

This example demonstrates how to link a system library and how the build script is used to support this use case.

Quite frequently a Rust crate wants to link to a native library provided on the system to bind its functionality or just use it as part of an implementation detail. This is quite a nuanced problem when it comes to performing this in a platform-agnostic fashion. It is best, if possible, to farm out as much of this as possible to make this as easy as possible for consumers.

For this example, we will be creating a binding to the system's zlib library. This is a library that is commonly found on most Unix-like systems that provides data compression. This is already wrapped up in the `libz-sys` crate, but for this example, we'll do an extremely simplified version. Check out the source code for the full example.

To make it easy to find the location of the library, we will use the `pkg-config` crate. This crate uses the system's `pkg-config` utility to discover information about a library. It will automatically tell Cargo what is needed to link the library. This will likely only work on Unix-like systems with `pkg-config` installed. Let's start by setting up the manifest:

```
# Cargo.toml

[package]
name = "libz-sys"
version = "0.1.0"
edition = "2018"
links = "z"

[build-dependencies]
pkg-config = "0.3.16"
```

Take note that we included the `links` key in the `package` table. This tells Cargo that we are linking to the `libz` library. See "Using another sys crate" for an example that will leverage this.

The build script is fairly simple:

```
// build.rs

fn main() {
    pkg_config::Config::new().probe("zlib").unwrap();
    println!("cargo:rerun-if-changed=build.rs");
}
```

Let's round out the example with a basic FFI binding:

```
// src/lib.rs

use std::os::raw::{c_uint, c_ulong};

extern "C" {
    pub fn crc32(crc: c_ulong, buf: *const u8, len: c_uint) -> c_ulong;
}

#[test]
fn test_crc32() {
    let s = "hello";
    unsafe {
        assert_eq!(crc32(0, s.as_ptr(), s.len() as c_uint), 0x3610a686);
    }
}
```

Run `cargo build -vv` to see the output from the build script. On a system with `libz` already installed, it may look something like this:

```
[libz-sys 0.1.0] cargo:rustc-link-search=native=/usr/lib
[libz-sys 0.1.0] cargo:rustc-link-lib=z
[libz-sys 0.1.0] cargo:rerun-if-changed=build.rs
```

Nice! `pkg-config` did all the work of finding the library and telling Cargo where it is.

It is not unusual for packages to include the source for the library, and build it statically if it is not found on the system, or if a feature or environment variable is set. For example, the real `libz-sys` crate checks the environment variable `LIBZ_SYS_STATIC` or the `static` feature to build it from source instead of using the system library. Check out the source for a more complete example.

Using another `sys` crate

When using the `links` key, crates may set metadata that can be read by other crates that depend on it. This provides a mechanism to communicate information between crates. In this example, we'll be creating a C library that makes use of zlib from the real `libz-sys` crate.

If you have a C library that depends on zlib, you can leverage the `libz-sys` crate to automatically find it or build it. This is great for cross-platform support, such as Windows where zlib is not usually installed. `libz-sys` sets the `include` metadata to tell other packages where to find the header files for zlib. Our build script can read that metadata with the `DEP_Z_INCLUDE` environment variable. Here's an example:

```
# Cargo.toml

[package]
name = "zuser"
version = "0.1.0"
edition = "2018"

[dependencies]
libz-sys = "1.0.25"

[build-dependencies]
cc = "1.0.46"
```

Here we have included `libz-sys` which will ensure that there is only one `libz` used in the final library, and give us access to it from our build script:

```
// build.rs

fn main() {
    let mut cfg = cc::Build::new();
    cfg.file("src/zuser.c");
    if let Some(include) = std::env::var_os("DEP_Z_INCLUDE") {
        cfg.include(include);
    }
    cfg.compile("zuser");
    println!("cargo:rerun-if-changed=src/zuser.c");
}
```

With `libz-sys` doing all the heavy lifting, the C source code may now include the zlib header, and it should find the header, even on systems where it isn't already installed.

```
// src/zuser.c

#include "zlib.h"

// … rest of code that makes use of zlib.
```

Conditional compilation

A build script may emit `rustc-cfg` instructions which can enable conditions that can be checked at compile time. In this example, we'll take a look at how the `openssl` crate uses this to support multiple versions of the OpenSSL library.

The `openssl-sys` crate implements building and linking the OpenSSL library. It supports multiple different implementations (like LibreSSL) and multiple versions. It makes use of the `links` key so that it may pass information to other build scripts. One of the things it passes is the `version_number` key, which is the version of OpenSSL that was detected. The code in the build script looks something like this:

```
println!("cargo:version_number={:x}", openssl_version);
```

This instruction causes the `DEP_OPENSSL_VERSION_NUMBER` environment variable to be set in any crates that directly depend on `openssl-sys`.

The `openssl` crate, which provides the higher-level interface, specifies `openssl-sys` as a dependency. The `openssl` build script can read the version information generated by the `openssl-sys` build script with the `DEP_OPENSSL_VERSION_NUMBER` environment variable. It uses this to generate some `cfg` values:

```
// (portion of build.rs)

if let Ok(version) = env::var("DEP_OPENSSL_VERSION_NUMBER") {
    let version = u64::from_str_radix(&version, 16).unwrap();

    if version >= 0x1_00_01_00_0 {
        println!("cargo:rustc-cfg=ossl101");
    }
    if version >= 0x1_00_02_00_0 {
        println!("cargo:rustc-cfg=ossl102");
    }
    if version >= 0x1_01_00_00_0 {
        println!("cargo:rustc-cfg=ossl110");
    }
    if version >= 0x1_01_00_07_0 {
        println!("cargo:rustc-cfg=ossl110g");
    }
    if version >= 0x1_01_01_00_0 {
        println!("cargo:rustc-cfg=ossl111");
    }
}
```

These `cfg` values can then be used with the `cfg` attribute or the `cfg` macro to conditionally include code. For example, SHA3 support was added in OpenSSL 1.1.1, so it is conditionally excluded for older versions:

```
// (portion of openssl crate)

#[cfg(ossl111)]
pub fn sha3_224() -> MessageDigest {
    unsafe { MessageDigest(ffi::EVP_sha3_224()) }
}
```

Of course, one should be careful when using this, since it makes the resulting binary even more dependent on the build environment. In this example, if the binary is distributed to another system, it may not have the exact same shared libraries, which could cause problems.

Publishing on crates.io

Once you've got a library that you'd like to share with the world, it's time to publish it on crates.io! Publishing a crate is when a specific version is uploaded to be hosted on crates.io.

Take care when publishing a crate, because a publish is **permanent**. The version can never be overwritten, and the code cannot be deleted. There is no limit to the number of versions which can be published, however.

Before your first publish

First thing's first, you'll need an account on crates.io to acquire an API token. To do so, visit the home page and log in via a GitHub account (required for now). After this, visit your Account Settings page and run the `cargo login` command specified.

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

This command will inform Cargo of your API token and store it locally in your `~/.cargo/credentials`. Note that this token is a **secret** and should not be shared with anyone else. If it leaks for any reason, you should regenerate it immediately.

Before publishing a new crate

Keep in mind that crate names on crates.io are allocated on a first-come-first- serve basis. Once a crate name is taken, it cannot be used for another crate.

Check out the metadata you can specify in `Cargo.toml` to ensure your crate can be discovered more easily! Before publishing, make sure you have filled out the following fields:

- `authors`
- `license` or `license-file`
- `description`
- `homepage`
- `documentation`
- `repository`

It would also be a good idea to include some `keywords` and `categories`, though they are not required.

If you are publishing a library, you may also want to consult the Rust API Guidelines.

Packaging a crate

The next step is to package up your crate and upload it to crates.io. For this we'll use the `cargo publish` subcommand. This command performs the following steps:

1. Perform some verification checks on your package.
2. Compress your source code into a `.crate` file.
3. Extract the `.crate` file into a temporary directory and verify that it compiles.
4. Upload the `.crate` file to crates.io.

5. The registry will perform some additional checks on the uploaded package before adding it.

It is recommended that you first run `cargo publish --dry-run` (or `cargo package` which is equivalent) to ensure there aren't any warnings or errors before publishing. This will perform the first three steps listed above.

```
$ cargo publish --dry-run
```

You can inspect the generated `.crate` file in the `target/package` directory. crates.io currently has a 10MB size limit on the `.crate` file. You may want to check the size of the `.crate` file to ensure you didn't accidentally package up large assets that are not required to build your package, such as test data, website documentation, or code generation. You can check which files are included with the following command:

```
$ cargo package --list
```

Cargo will automatically ignore files ignored by your version control system when packaging, but if you want to specify an extra set of files to ignore you can use the `exclude` key in the manifest:

```
[package]
# ...
exclude = [
    "public/assets/*",
    "videos/*",
]
```

If you'd rather explicitly list the files to include, Cargo also supports an `include` key, which if set, overrides the `exclude` key:

```
[package]
# ...
include = [
    "**/*.rs",
    "Cargo.toml",
]
```

Uploading the crate

When you are ready to publish, use the `cargo publish` command to upload to crates.io:

```
$ cargo publish
```

And that's it, you've now published your first crate!

Publishing a new version of an existing crate

In order to release a new version, change the `version` value specified in your `Cargo.toml` manifest. Keep in mind the semver rules, and consult RFC 1105 for what constitutes a semver-breaking change. Then run `cargo publish` as described above to upload the new version.

Managing a crates.io-based crate

Management of crates is primarily done through the command line `cargo` tool rather than the crates.io web interface. For this, there are a few subcommands to manage a crate.

`cargo yank`

Occasions may arise where you publish a version of a crate that actually ends up being broken for one reason or another (syntax error, forgot to include a file, etc.). For situations such as this, Cargo supports a "yank" of a version of a crate.

```
$ cargo yank --vers 1.0.1
$ cargo yank --vers 1.0.1 --undo
```

A yank **does not** delete any code. This feature is not intended for deleting accidentally uploaded secrets, for example. If that happens, you must reset those secrets immediately.

The semantics of a yanked version are that no new dependencies can be created against that version, but all existing dependencies continue to work. One of the major goals of crates.io is to act as a permanent archive of crates that does not change over time, and allowing deletion of a version would go against this goal. Essentially a yank means that all packages with a `Cargo.lock` will not break, while any future `Cargo.lock` files generated will not list the yanked version.

`cargo owner`

A crate is often developed by more than one person, or the primary maintainer may change over time! The owner of a crate is the only person allowed to publish new versions of the crate, but an owner may designate additional owners.

```
$ cargo owner --add github-handle
$ cargo owner --remove github-handle
$ cargo owner --add github:rust-lang:owners
$ cargo owner --remove github:rust-lang:owners
```

The owner IDs given to these commands must be GitHub user names or GitHub teams.

If a user name is given to `--add`, that user is invited as a "named" owner, with full rights to the crate. In addition to being able to publish or yank versions of the crate, they have the ability to add or remove owners, *including* the owner that made *them* an owner. Needless to say, you shouldn't make people you don't fully trust into a named owner. In order to become a named owner, a user must have logged into crates.io previously.

If a team name is given to `--add`, that team is invited as a "team" owner, with restricted right to the crate. While they have permission to publish or yank versions of the crate, they *do not* have the ability to add or remove owners. In addition to being more convenient for managing groups of owners, teams are just a bit more secure against owners becoming malicious.

The syntax for teams is currently `github:org:team` (see examples above). In order to invite a team as an owner one must be a member of that team. No such restriction applies to removing a team as an owner.

GitHub permissions

Team membership is not something GitHub provides simple public access to, and it is likely for you to encounter the following message when working with them:

> It looks like you don't have permission to query a necessary property from GitHub to complete this request. You may need to re-authenticate on crates.io to grant permission to read GitHub org memberships. Just go to https://crates.io/login.

This is basically a catch-all for "you tried to query a team, and one of the five levels of membership access control denied this". That is not an exaggeration. GitHub's support for team access control is Enterprise Grade.

The most likely cause of this is simply that you last logged in before this feature was added. We originally requested *no* permissions from GitHub when authenticating users, because we didn't actually ever use the user's token for anything other than logging them in. However to query team membership on your behalf, we now require the `read:org` scope.

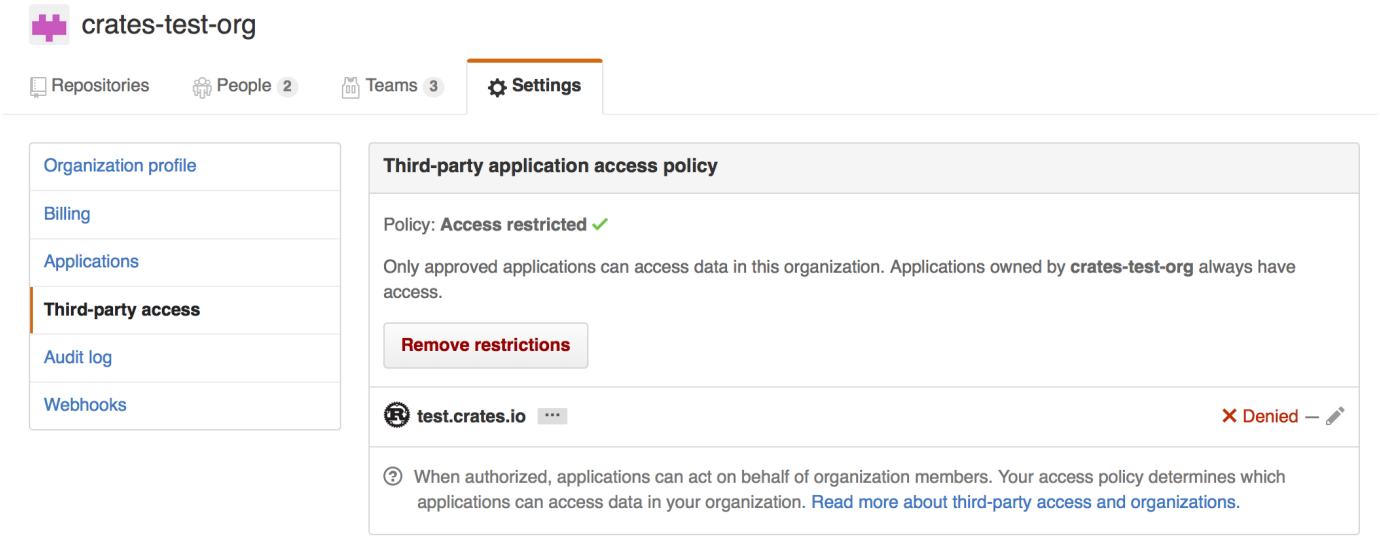
You are free to deny us this scope, and everything that worked before teams were introduced will keep working. However you will never be able to add a team as an owner, or publish a crate as a team owner. If you ever attempt to do this, you will get the error above. You may also see this error if you ever try to publish a crate that you don't own at all, but otherwise happens to have a team.

If you ever change your mind, or just aren't sure if crates.io has sufficient permission, you can always go to https://crates.io/login, which will prompt you for permission if crates.io doesn't have all the scopes it would like to.

An additional barrier to querying GitHub is that the organization may be actively denying third party access. To check this, you can go to:

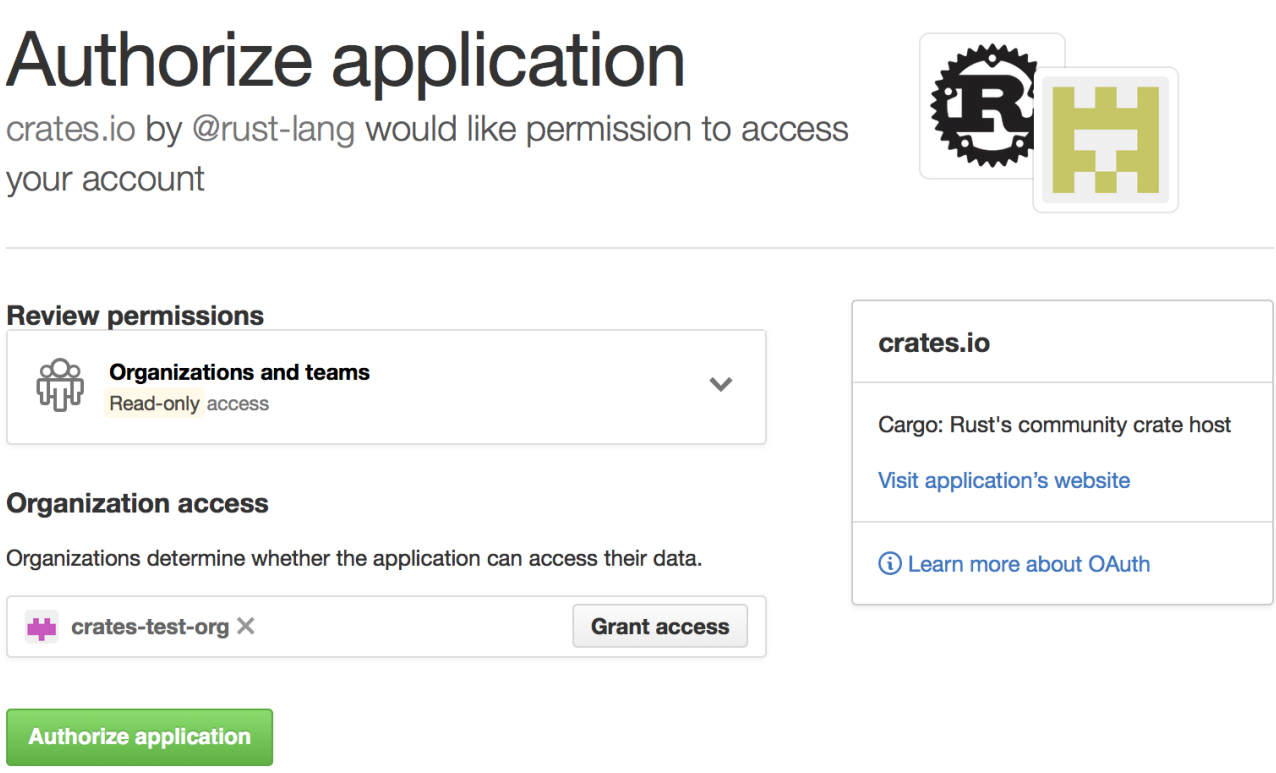
```
https://github.com/organizations/:org/settings/oauth_application_policy
```

where `:org` is the name of the organization (e.g., `rust-lang`). You may see something like:



Where you may choose to explicitly remove crates.io from your organization's blacklist, or simply press the "Remove Restrictions" button to allow all third party applications to access this data.

Alternatively, when crates.io requested the `read:org` scope, you could have explicitly whitelisted crates.io querying the org in question by pressing the "Grant Access" button next to its name:



Package ID Specifications

Package ID specifications

Subcommands of Cargo frequently need to refer to a particular package within a dependency graph for various operations like updating, cleaning, building, etc. To solve this problem, Cargo supports Package ID Specifications. A specification is a string which is used to uniquely refer to one package within a graph of packages.

Specification grammar

The formal grammar for a Package Id Specification is:

```
pkgid := pkgname
       | [ proto "://" ] hostname-and-path [ "#" ( pkgname | semver ) ]
pkgname := name [ ":" semver ]

proto := "http" | "git" | ...
```

Here, brackets indicate that the contents are optional.

Example specifications

These could all be references to a package `foo` version `1.2.3` from the registry at `crates.io`

pkgid	name	version	url
`foo`	`foo`	`*`	`*`
`foo:1.2.3`	`foo`	`1.2.3`	`*`
`crates.io/foo`	`foo`	`*`	`*://crates.io/foo`
`crates.io/foo#1.2.3`	`foo`	`1.2.3`	`*://crates.io/foo`
`crates.io/bar#foo:1.2.3`	`foo`	`1.2.3`	`*://crates.io/bar`
`https://crates.io/foo#1.2.3`	`foo`	`1.2.3`	`https://crates.io/foo`

Brevity of specifications

The goal of this is to enable both succinct and exhaustive syntaxes for referring to packages in a dependency graph. Ambiguous references may refer to one or more packages. Most commands generate an error if more than one package could be referred to with the same specification.

Source Replacement

This document is about replacing the crate index. You can read about overriding dependencies in the overriding dependencies section of this documentation.

A *source* is a provider that contains crates that may be included as dependencies for a package. Cargo supports the ability to **replace one source with another** to express strategies such as:

- Vendoring - custom sources can be defined which represent crates on the local filesystem. These sources are subsets of the source that they're replacing and can be checked into packages if necessary.
- Mirroring - sources can be replaced with an equivalent version which acts as a cache for crates.io itself.

Cargo has a core assumption about source replacement that the source code is exactly the same from both sources. Note that this also means that a replacement source is not allowed to have crates which are not present in the original source.

As a consequence, source replacement is not appropriate for situations such as patching a dependency or a private registry. Cargo supports patching dependencies through the usage of the `[replace]` key, and private registry support is described in Registries.

Configuration

Configuration of replacement sources is done through `.cargo/config` and the full set of available keys are:

```
# The `source` table is where all keys related to source-replacement
# are stored.
[source]

# Under the `source` table are a number of other tables whose keys are a
# name for the relevant source. For example this section defines a new
# source, called `my-vendor-source`, which comes from a directory
# located at `vendor` relative to the directory containing this `.cargo/config`
# file
[source.my-vendor-source]
directory = "vendor"

# The crates.io default source for crates is available under the name
# "crates-io", and here we use the `replace-with` key to indicate that it's
# replaced with our source above.
[source.crates-io]
replace-with = "my-vendor-source"

# Each source has its own table where the key is the name of the source
[source.the-source-name]

# Indicate that `the-source-name` will be replaced with `another-source`,
# defined elsewhere
replace-with = "another-source"

# Several kinds of sources can be specified (described in more detail below):
registry = "https://example.com/path/to/index"
local-registry = "path/to/registry"
directory = "path/to/vendor"

# Git sources can optionally specify a branch/tag/rev as well
git = "https://example.com/path/to/repo"
# branch = "master"
# tag = "v1.0.1"
# rev = "313f44e8"
```

Registry Sources

A "registry source" is one that is the same as crates.io itself. That is, it has an index served in a git repository which matches the format of the crates.io index. That repository then has configuration indicating where to download crates from.

Currently there is not an already-available project for setting up a mirror of crates.io. Stay tuned though!

Local Registry Sources

A "local registry source" is intended to be a subset of another registry source, but available on the local filesystem (aka vendoring). Local registries are downloaded ahead of time, typically sync'd with a `Cargo.lock`, and are made up of a set of `*.crate` files and an index like the normal registry is.

The primary way to manage and create local registry sources is through the `cargo-local-registry` subcommand, available on crates.io and can be installed with `cargo install cargo-local-registry`.

Local registries are contained within one directory and contain a number of `*.crate` files downloaded from crates.io as well as an `index` directory with the same format as the crates.io-index project (populated with just entries for the crates that are present).

Directory Sources

A "directory source" is similar to a local registry source where it contains a number of crates available on the local filesystem, suitable for vendoring dependencies. Directory sources are primarily managed the `cargo vendor` subcommand.

Directory sources are distinct from local registries though in that they contain the unpacked version of `*.crate` files, making it more suitable in some situations to check everything into source control. A directory source is just a directory containing a number of other directories which contain the source code for crates (the unpacked version of `*.crate` files). Currently no restriction is placed on the name of each directory.

Each crate in a directory source also has an associated metadata file indicating the checksum of each file in the crate to protect against accidental modifications.

External tools

One of the goals of Cargo is simple integration with third-party tools, like IDEs and other build systems. To make integration easier, Cargo has several facilities:

- a `cargo metadata` command, which outputs package structure and dependencies information in JSON,
- a `--message-format` flag, which outputs information about a particular build, and
- support for custom subcommands.

Information about package structure

You can use `cargo metadata` command to get information about package structure and dependencies. See the `cargo metadata` documentation for details on the format of the output.

The format is stable and versioned. When calling `cargo metadata`, you should pass `--format-version` flag explicitly to avoid forward incompatibility hazard.

If you are using Rust, the cargo_metadata crate can be used to parse the output.

JSON messages

When passing `--message-format=json`, Cargo will output the following information during the build:

- compiler errors and warnings,
- produced artifacts,
- results of the build scripts (for example, native dependencies).

The output goes to stdout in the JSON object per line format. The `reason` field distinguishes different kinds of messages.

The `--message-format` option can also take additional formatting values which alter the way the JSON messages are computed and rendered. See the description of the `--message-format` option in the build command documentation for more details.

Compiler messages

The "compiler-message" message includes output from the compiler, such as warnings and errors. See the rustc JSON chapter for details on `rustc`'s message format, which is embedded in the following structure:

```
{
    /* The "reason" indicates the kind of message. */
    "reason": "compiler-message",
    /* The Package ID, a unique identifier for referring to the package. */
    "package_id": "my-package 0.1.0 (path+file:///path/to/my-package)",
    /* The Cargo target (lib, bin, example, etc.) that generated the message. */
    "target": {
        /* Array of target kinds.
           - lib targets list the `crate-type` values from the
             manifest such as "lib", "rlib", "dylib",
             "proc-macro", etc. (default ["lib"])
           - binary is ["bin"]
           - example is ["example"]
           - integration test is ["test"]
           - benchmark is ["bench"]
           - build script is ["custom-build"]
        */
        "kind": [
            "lib"
        ],
        /* Array of crate types.
           - lib and example libraries list the `crate-type` values
             from the manifest such as "lib", "rlib", "dylib",
             "proc-macro", etc. (default ["lib"])
           - all other target kinds are ["bin"]
        */
        "crate_types": [
            "lib"
        ],
        /* The name of the target. */
        "name": "my-package",
        /* Absolute path to the root source file of the target. */
        "src_path": "/path/to/my-package/src/lib.rs",
        /* The Rust edition of the target.
           Defaults to the package edition.
        */
        "edition": "2018",
        /* Array of required features.
           This property is not included if no required features are set.
        */
        "required-features": ["feat1"],
        /* Whether or not this target has doc tests enabled, and
           the target is compatible with doc testing.
        */
        "doctest": true
    },
    /* The message emitted by the compiler.

    See https://doc.rust-lang.org/rustc/json.html for details.
    */
    "message": {
        /* ... */
    }
}
```

Artifact messages

For every compilation step, a "compiler-artifact" message is emitted with the following structure:

```
{
    /* The "reason" indicates the kind of message. */
    "reason": "compiler-artifact",
    /* The Package ID, a unique identifier for referring to the package. */
    "package_id": "my-package 0.1.0 (path+file:///path/to/my-package)",
    /* The Cargo target (lib, bin, example, etc.) that generated the artifacts.
       See the definition above for `compiler-message` for details.
    */
    "target": {
        "kind": [
            "lib"
        ],
        "crate_types": [
            "lib"
        ],
        "name": "my-package",
        "src_path": "/path/to/my-package/src/lib.rs",
        "edition": "2018",
        "doctest": true
    },
    /* The profile indicates which compiler settings were used. */
    "profile": {
        /* The optimization level. */
        "opt_level": "0",
        /* The debug level, an integer of 0, 1, or 2. If `null`, it implies
           rustc's default of 0.
        */
        "debuginfo": 2,
        /* Whether or not debug assertions are enabled. */
        "debug_assertions": true,
        /* Whether or not overflow checks are enabled. */
        "overflow_checks": true,
        /* Whether or not the `--test` flag is used. */
        "test": false
    },
    /* Array of features enabled. */
    "features": ["feat1", "feat2"],
    /* Array of files generated by this step. */
    "filenames": [
        "/path/to/my-package/target/debug/libmy_package.rlib",
        "/path/to/my-package/target/debug/deps/libmy_package-
be9f3faac0a26ef0.rmeta"
    ],
    /* A string of the path to the executable that was created, or null if
       this step did not generate an executable.
    */
    "executable": null,
    /* Whether or not this step was actually executed.
       When `true`, this means that the pre-existing artifacts were
       up-to-date, and `rustc` was not executed. When `false`, this means that
       `rustc` was run to generate the artifacts.
    */
    "fresh": true
}
```

Build script output

The "build-script-executed" message includes the parsed output of a build script. Note that this is emitted even if the build script is not run; it will display the previously cached value. More details about build script output may be found in the chapter on build scripts.

```
{
    /* The "reason" indicates the kind of message. */
    "reason": "build-script-executed",
    /* The Package ID, a unique identifier for referring to the package. */
    "package_id": "my-package 0.1.0 (path+file:///path/to/my-package)",
    /* Array of libraries to link, as indicated by the `cargo:rustc-link-lib`
       instruction. Note that this may include a "KIND=" prefix in the string
       where KIND is the library kind.
    */
    "linked_libs": ["foo", "static=bar"],
    /* Array of paths to include in the library search path, as indicated by
       the `cargo:rustc-link-search` instruction. Note that this may include a
       "KIND=" prefix in the string where KIND is the library kind.
    */
    "linked_paths": ["/some/path", "native=/another/path"],
    /* Array of cfg values to enable, as indicated by the `cargo:rustc-cfg`
       instruction.
    */
    "cfgs": ["cfg1", "cfg2=\"string\""],
    /* Array of [KEY, VALUE] arrays of environment variables to set, as
       indicated by the `cargo:rustc-env` instruction.
    */
    "env": [
        ["SOME_KEY", "some value"],
        ["ANOTHER_KEY", "another value"]
    ],
    /* A path which is used as a value of `OUT_DIR` environmental variable
       when compiling current package.
    */
    "out_dir": "/some/path/in/target/dir"
}
```

Custom subcommands

Cargo is designed to be extensible with new subcommands without having to modify Cargo itself. This is achieved by translating a cargo invocation of the form `cargo (?<command>[^ ]+)` into an invocation of an external tool `cargo-${command}`. The external tool must be present in one of the user's `$PATH` directories.

When Cargo invokes a custom subcommand, the first argument to the subcommand will be the filename of the custom subcommand, as usual. The second argument will be the subcommand name itself. For example, the second argument would be `${command}` when invoking `cargo-${command}`. Any additional arguments on the command line will be forwarded unchanged.

Cargo can also display the help output of a custom subcommand with `cargo help ${command}`. Cargo assumes that the subcommand will print a help message if its third argument is `--help`. So, `cargo help ${command}` would invoke `cargo-${command} ${command} --help`.

Custom subcommands may use the `CARGO` environment variable to call back to Cargo. Alternatively, it can link to `cargo` crate as a library, but this approach has drawbacks:

- Cargo as a library is unstable: the API may change without deprecation
- versions of the linked Cargo library may be different from the Cargo binary

Registries

Cargo installs crates and fetches dependencies from a "registry". The default registry is crates.io. A registry contains an "index" which contains a searchable list of available crates. A registry may also provide a web API to support publishing new crates directly from Cargo.

Note: If you are interested in mirroring or vendoring an existing registry, take a look at Source Replacement.

Using an Alternate Registry

To use a registry other than crates.io, the name and index URL of the registry must be added to a `.cargo/config` file. The `registries` table has a key for each registry, for example:

```
[registries]
my-registry = { index = "https://my-intranet:8080/git/index" }
```

The `index` key should be a URL to a git repository with the registry's index. A crate can then depend on a crate from another registry by specifying the `registry` key and a value of the registry's name in that dependency's entry in `Cargo.toml`:

```
# Sample Cargo.toml
[package]
name = "my-project"
version = "0.1.0"

[dependencies]
other-crate = { version = "1.0", registry = "my-registry" }
```

As with most config values, the index may be specified with an environment variable instead of a config file. For example, setting the following environment variable will accomplish the same thing as defining a config file:

```
CARGO_REGISTRIES_MY_REGISTRY_INDEX=https://my-intranet:8080/git/index
```

Note: crates.io does not accept packages that depend on crates from other registries.

Publishing to an Alternate Registry

If the registry supports web API access, then packages can be published directly to the registry from Cargo. Several of Cargo's commands such as `cargo publish` take a `--registry` command-line flag to indicate which registry to use. For example, to publish the package in the current directory:

1. `cargo login --registry=my-registry`
   This only needs to be done once. You must enter the secret API token retrieved from the registry's website. Alternatively the token may be passed directly to the `publish` command with the `--token` command-line flag or an environment variable with the name of the registry such as `CARGO_REGISTRIES_MY_REGISTRY_TOKEN`.
2. `cargo publish --registry=my-registry`

Instead of always passing the `--registry` command-line option, the default registry may be set in `.cargo/config` with the `registry.default` key.

Setting the `package.publish` key in the `Cargo.toml` manifest restricts which registries the package is allowed to be published to. This is useful to prevent accidentally publishing a closed-source package to crates.io. The value may be a list of registry names, for example:

```
[package]
# ...
publish = ["my-registry"]
```

The `publish` value may also be `false` to restrict all publishing, which is the same as an empty list.

The authentication information saved by `cargo login` is stored in the `credentials` file in the Cargo home directory (default `$HOME/.cargo`). It has a separate table for each registry, for example:

```
[registries.my-registry]
token = "854DvwSlUwEHtIo3kWy6x7UCPKHfzCmy"
```

Running a Registry

A minimal registry can be implemented by having a git repository that contains an index, and a server that contains the compressed `.crate` files created by `cargo package`. Users

won't be able to use Cargo to publish to it, but this may be sufficient for closed environments.

A full-featured registry that supports publishing will additionally need to have a web API service that conforms to the API used by Cargo. The web API is documented below.

At this time, there is no widely used software for running a custom registry. There is interest in documenting projects that implement registry support, or existing package caches that add support for Cargo.

Index Format

The following defines the format of the index. New features are occasionally added, which are only understood starting with the version of Cargo that introduced them. Older versions of Cargo may not be able to use packages that make use of new features. However, the format for older packages should not change, so older versions of Cargo should be able to use them.

The index is stored in a git repository so that Cargo can efficiently fetch incremental updates to the index. In the root of the repository is a file named `config.json` which contains JSON information used by Cargo for accessing the registry. This is an example of what the crates.io config file looks like:

```
{
    "dl": "https://crates.io/api/v1/crates",
    "api": "https://crates.io"
}
```

The keys are:

- `dl`: This is the URL for downloading crates listed in the index. The value may have the markers `{crate}` and `{version}` which are replaced with the name and version of the crate to download. If the markers are not present, then the value `/{crate}/{version}/download` is appended to the end.
- `api`: This is the base URL for the web API. This key is optional, but if it is not specified, commands such as `cargo publish` will not work. The web API is described below.

The download endpoint should send the `.crate` file for the requested package. Cargo supports https, http, and file URLs, HTTP redirects, HTTP1 and HTTP2. The exact specifics of TLS support depend on the platform that Cargo is running on, the version of Cargo, and how it was compiled.

The rest of the index repository contains one file for each package, where the filename is the name of the package in lowercase. Each version of the package has a separate line in the file. The files are organized in a tier of directories:

- Packages with 1 character names are placed in a directory named `1`.
- Packages with 2 character names are placed in a directory named `2`.

- Packages with 3 character names are placed in the directory `3/{first-character}` where `{first-character}` is the first character of the package name.
- All other packages are stored in directories named `{first-two}/{second-two}` where the top directory is the first two characters of the package name, and the next subdirectory is the third and fourth characters of the package name. For example, `cargo` would be stored in a file named `ca/rg/cargo`.

Note: Although the index filenames are in lowercase, the fields that contain package names in `Cargo.toml` and the index JSON data are case-sensitive and may contain upper and lower case characters.

Registries should consider enforcing limitations on package names added to their index. Cargo itself allows names with any alphanumeric, `-`, or `_` characters. crates.io imposes its own limitations, including the following:

- Only allows ASCII characters.
- Only alphanumeric, `-`, and `_` characters.
- First character must be alphabetic.
- Case-insensitive collision detection.
- Prevent differences of `-` vs `_`.
- Under a specific length (max 64).
- Rejects reserved names, such as Windows special filenames like "nul".

Registries should consider incorporating similar restrictions, and consider the security implications, such as IDN homograph attacks and other concerns in UTR36 and UTS39.

Each line in a package file contains a JSON object that describes a published version of the package. The following is a pretty-printed example with comments explaining the format of the entry.

```
{
    // The name of the package.
    // This must only contain alphanumeric, `-`, or `_` characters.
    "name": "foo",
    // The version of the package this row is describing.
    // This must be a valid version number according to the Semantic
    // Versioning 2.0.0 spec at https://semver.org/.
    "vers": "0.1.0",
    // Array of direct dependencies of the package.
    "deps": [
        {
            // Name of the dependency.
            // If the dependency is renamed from the original package name,
            // this is the new name. The original package name is stored in
            // the `package` field.
            "name": "rand",
            // The semver requirement for this dependency.
            // This must be a valid version requirement defined at
            // https://github.com/steveklabnik/semver#requirements.
            "req": "^0.6",
            // Array of features (as strings) enabled for this dependency.
            "features": ["i128_support"],
            // Boolean of whether or not this is an optional dependency.
            "optional": false,
            // Boolean of whether or not default features are enabled.
            "default_features": true,
            // The target platform for the dependency.
            // null if not a target dependency.
            // Otherwise, a string such as "cfg(windows)".
            "target": null,
            // The dependency kind.
            // "dev", "build", or "normal".
            // Note: this is a required field, but a small number of entries
            // exist in the crates.io index with either a missing or null
            // `kind` field due to implementation bugs.
            "kind": "normal",
            // The URL of the index of the registry where this dependency is
            // from as a string. If not specified or null, it is assumed the
            // dependency is in the current registry.
            "registry": null,
            // If the dependency is renamed, this is a string of the actual
            // package name. If not specified or null, this dependency is not
            // renamed.
            "package": null,
        }
    ],
    // A SHA256 checksum of the `.crate` file.
    "cksum": "d867001db0e2b6e0496f9fac96930e2d42233ecd3ca0413e0753d4c7695d289c",
    // Set of features defined for the package.
    // Each feature maps to an array of features or dependencies it enables.
    "features": {
        "extras": ["rand/simd_support"]
    },
    // Boolean of whether or not this version has been yanked.
    "yanked": false,
    // The `links` string value from the package's manifest, or null if not
    // specified. This field is optional and defaults to null.
```

```
    "links": null
}
```

The JSON objects should not be modified after they are added except for the `yanked` field whose value may change at any time.

Web API

A registry may host a web API at the location defined in `config.json` to support any of the actions listed below.

Cargo includes the `Authorization` header for requests that require authentication. The header value is the API token. The server should respond with a 403 response code if the token is not valid. Users are expected to visit the registry's website to obtain a token, and Cargo can store the token using the `cargo login` command, or by passing the token on the command-line.

Responses use a 200 response code for both success and errors. Cargo looks at the JSON response to determine if there was success or failure. Failure responses have a JSON object with the following structure:

```
{
    // Array of errors to display to the user.
    "errors": [
        {
            // The error message as a string.
            "detail": "error message text"
        }
    ]
}
```

Servers may also respond with a 404 response code to indicate the requested resource is not found (for example, an unknown crate name). However, using a 200 response with an `errors` object allows a registry to provide a more detailed error message if desired.

For backwards compatibility, servers should ignore any unexpected query parameters or JSON fields. If a JSON field is missing, it should be assumed to be null. The endpoints are versioned with the `v1` component of the path, and Cargo is responsible for handling backwards compatibility fallbacks should any be required in the future.

Cargo sets the following headers for all requests:

- `Content-Type`: `application/json`
- `Accept`: `application/json`
- `User-Agent`: The Cargo version such as `cargo 1.32.0 (8610973aa 2019-01-02)`. This may be modified by the user in a configuration value. Added in 1.29.

Publish

- Endpoint: `/api/v1/crates/new`
- Method: PUT
- Authorization: Included

The publish endpoint is used to publish a new version of a crate. The server should validate the crate, make it available for download, and add it to the index.

The body of the data sent by Cargo is:

- 32-bit unsigned little-endian integer of the length of JSON data.
- Metadata of the package as a JSON object.
- 32-bit unsigned little-endian integer of the length of the `.crate` file.
- The `.crate` file.

The following is a commented example of the JSON object. Some notes of some restrictions imposed by crates.io are included only to illustrate some suggestions on types of validation that may be done, and should not be considered as an exhaustive list of restrictions crates.io imposes.

```
{
    // The name of the package.
    "name": "foo",
    // The version of the package being published.
    "vers": "0.1.0",
    // Array of direct dependencies of the package.
    "deps": [
        {
            // Name of the dependency.
            // If the dependency is renamed from the original package name,
            // this is the original name. The new package name is stored in
            // the `explicit_name_in_toml` field.
            "name": "rand",
            // The semver requirement for this dependency.
            "version_req": "^0.6",
            // Array of features (as strings) enabled for this dependency.
            "features": ["i128_support"],
            // Boolean of whether or not this is an optional dependency.
            "optional": false,
            // Boolean of whether or not default features are enabled.
            "default_features": true,
            // The target platform for the dependency.
            // null if not a target dependency.
            // Otherwise, a string such as "cfg(windows)".
            "target": null,
            // The dependency kind.
            // "dev", "build", or "normal".
            "kind": "normal",
            // The URL of the index of the registry where this dependency is
            // from as a string. If not specified or null, it is assumed the
            // dependency is in the current registry.
            "registry": null,
            // If the dependency is renamed, this is a string of the new
            // package name. If not specified or null, this dependency is not
            // renamed.
            "explicit_name_in_toml": null,
        }
    ],
    // Set of features defined for the package.
    // Each feature maps to an array of features or dependencies it enables.
    // Cargo does not impose limitations on feature names, but crates.io
    // requires alphanumeric ASCII, `_` or `-` characters.
    "features": {
        "extras": ["rand/simd_support"]
    },
    // List of strings of the authors.
    // May be empty. crates.io requires at least one entry.
    "authors": ["Alice <a@example.com>"],
    // Description field from the manifest.
    // May be null. crates.io requires at least some content.
    "description": null,
    // String of the URL to the website for this package's documentation.
    // May be null.
    "documentation": null,
    // String of the URL to the website for this package's home page.
    // May be null.
    "homepage": null,
```

```
    // String of the content of the README file.
    // May be null.
    "readme": null,
    // String of a relative path to a README file in the crate.
    // May be null.
    "readme_file": null,
    // Array of strings of keywords for the package.
    "keywords": [],
    // Array of strings of categories for the package.
    "categories": [],
    // String of the license for the package.
    // May be null. crates.io requires either `license` or `license_file` to be
set.
    "license": null,
    // String of a relative path to a license file in the crate.
    // May be null.
    "license_file": null,
    // String of the URL to the website for the source repository of this
package.
    // May be null.
    "repository": null,
    // Optional object of "status" badges. Each value is an object of
    // arbitrary string to string mappings.
    // crates.io has special interpretation of the format of the badges.
    "badges": {
        "travis-ci": {
            "branch": "master",
            "repository": "rust-lang/cargo"
        }
    },
    // The `links` string value from the package's manifest, or null if not
    // specified. This field is optional and defaults to null.
    "links": null,
}
```

A successful response includes the JSON object:

```
{
    // Optional object of warnings to display to the user.
    "warnings": {
        // Array of strings of categories that are invalid and ignored.
        "invalid_categories": [],
        // Array of strings of badge names that are invalid and ignored.
        "invalid_badges": [],
        // Array of strings of arbitrary warnings to display to the user.
        "other": []
    }
}
```

Yank

- Endpoint: `/api/v1/crates/{crate_name}/{version}/yank`
- Method: DELETE
- Authorization: Included

The yank endpoint will set the `yank` field of the given version of a crate to `true` in the index.

A successful response includes the JSON object:

```
{
    // Indicates the delete succeeded, always true.
    "ok": true,
}
```

Unyank

- Endpoint: `/api/v1/crates/{crate_name}/{version}/unyank`
- Method: PUT
- Authorization: Included

The unyank endpoint will set the `yank` field of the given version of a crate to `false` in the index.

A successful response includes the JSON object:

```
{
    // Indicates the delete succeeded, always true.
    "ok": true,
}
```

Owners

Cargo does not have an inherent notion of users and owners, but it does provide the `owner` command to assist managing who has authorization to control a crate. It is up to the registry to decide exactly how users and owners are handled. See the publishing documentation for a description of how crates.io handles owners via GitHub users and teams.

Owners: List

- Endpoint: `/api/v1/crates/{crate_name}/owners`
- Method: GET
- Authorization: Included

The owners endpoint returns a list of owners of the crate.

A successful response includes the JSON object:

```
{
    // Array of owners of the crate.
    "users": [
        {
            // Unique unsigned 32-bit integer of the owner.
            "id": 70,
            // The unique username of the owner.
            "login": "github:rust-lang:core",
            // Name of the owner.
            // This is optional and may be null.
            "name": "Core",
        }
    ]
}
```

Owners: Add

- Endpoint: `/api/v1/crates/{crate_name}/owners`
- Method: PUT
- Authorization: Included

A PUT request will send a request to the registry to add a new owner to a crate. It is up to the registry how to handle the request. For example, crates.io sends an invite to the user that they must accept before being added.

The request should include the following JSON object:

```
{
    // Array of `login` strings of owners to add.
    "users": ["login_name"]
}
```

A successful response includes the JSON object:

```
{
    // Indicates the add succeeded, always true.
    "ok": true,
    // A string to be displayed to the user.
    "msg": "user ehuss has been invited to be an owner of crate cargo"
}
```

Owners: Remove

- Endpoint: `/api/v1/crates/{crate_name}/owners`
- Method: DELETE
- Authorization: Included

A DELETE request will remove an owner from a crate. The request should include the following JSON object:

```
{
    // Array of `login` strings of owners to remove.
    "users": ["login_name"]
}
```

A successful response includes the JSON object:

```
{
    // Indicates the remove succeeded, always true.
    "ok": true
}
```

Search

- Endpoint: `/api/v1/crates`
- Method: GET
- Query Parameters:
  - `q`: The search query string.
  - `per_page`: Number of results, default 10, max 100.

The search request will perform a search for crates, using criteria defined on the server.

A successful response includes the JSON object:

```
{
    // Array of results.
    "crates": [
        {
            // Name of the crate.
            "name": "rand",
            // The highest version available.
            "max_version": "0.6.1",
            // Textual description of the crate.
            "description": "Random number generators and other randomness
functionality.\n",
        }
    ],
    "meta": {
        // Total number of results available on the server.
        "total": 119
    }
}
```

Login

- Endpoint: `/me`

The "login" endpoint is not an actual API request. It exists solely for the `cargo login` command to display a URL to instruct a user to visit in a web browser to log in and retrieve an API token.

Unstable Features

Experimental Cargo features are only available on the nightly channel. You typically use one of the `-Z` flags to enable them. Run `cargo -Z help` to see a list of flags available.

`-Z unstable-options` is a generic flag for enabling other unstable command-line flags. Options requiring this will be called out below.

Some unstable features will require you to specify the `cargo-features` key in `Cargo.toml`.

no-index-update

- Original Issue: #3479
- Tracking Issue: #7404

The `-Z no-index-update` flag ensures that Cargo does not attempt to update the registry index. This is intended for tools such as Crater that issue many Cargo commands, and you want to avoid the network latency for updating the index each time.

mtime-on-use

- Original Issue: #6477
- Cache usage meta tracking issue: #7150

The `-Z mtime-on-use` flag is an experiment to have Cargo update the mtime of used files to make it easier for tools like cargo-sweep to detect which files are stale. For many workflows this needs to be set on *all* invocations of cargo. To make this more practical setting the `unstable.mtime_on_use` flag in `.cargo/config` or the corresponding ENV variable will apply the `-Z mtime-on-use` to all invocations of nightly cargo. (the config flag is ignored by stable)

avoid-dev-deps

- Original Issue: #4988
- Stabilization Issue: #5133

When running commands such as `cargo install` or `cargo build`, Cargo currently requires dev-dependencies to be downloaded, even if they are not used. The `-Z avoid-dev-deps` flag allows Cargo to avoid downloading dev-dependencies if they are not needed. The `Cargo.lock` file will not be generated if dev-dependencies are skipped.

minimal-versions

- Original Issue: #4100

- Tracking Issue: #5657

> Note: It is not recommended to use this feature. Because it enforces minimal versions for all transitive dependencies, its usefulness is limited since not all external dependencies declare proper lower version bounds. It is intended that it will be changed in the future to only enforce minimal versions for direct dependencies.

When a `Cargo.lock` file is generated, the `-Z minimal-versions` flag will resolve the dependencies to the minimum semver version that will satisfy the requirements (instead of the greatest version).

The intended use-case of this flag is to check, during continuous integration, that the versions specified in Cargo.toml are a correct reflection of the minimum versions that you are actually using. That is, if Cargo.toml says `foo = "1.0.0"` that you don't accidentally depend on features added only in `foo 1.5.0`.

out-dir

- Original Issue: #4875
- Tracking Issue: #6790

This feature allows you to specify the directory where artifacts will be copied to after they are built. Typically artifacts are only written to the `target/release` or `target/debug` directories. However, determining the exact filename can be tricky since you need to parse JSON output. The `--out-dir` flag makes it easier to predictably access the artifacts. Note that the artifacts are copied, so the originals are still in the `target` directory. Example:

```
cargo +nightly build --out-dir=out -Z unstable-options
```

doctest-xcompile

- Tracking Issue: #7040
- Tracking Rustc Issue: #64245

This flag changes `cargo test`'s behavior when handling doctests when a target is passed. Currently, if a target is passed that is different from the host cargo will simply skip testing doctests. If this flag is present, cargo will continue as normal, passing the tests to doctest, while also passing it a `--target` option, as well as enabling `-Zunstable-features --enable-per-target-ignores` and passing along information from `.cargo/config`. See the rustc issue for more information.

```
cargo test --target foo -Zdoctest-xcompile
```

Custom named profiles

- Tracking Issue: rust-lang/cargo#6988
- RFC: #2678

With this feature you can define custom profiles having new names. With the custom profile enabled, build artifacts can be emitted by default to directories other than `release` or `debug`, based on the custom profile's name.

For example:

```
cargo-features = ["named-profiles"]

[profile.release-lto]
inherits = "release"
lto = true
```

An `inherits` key is used in order to receive attributes from other profiles, so that a new custom profile can be based on the standard `dev` or `release` profile presets. Cargo emits errors in case `inherits` loops are detected. When considering inheritance hierarchy, all profiles directly or indirectly inherit from either from `release` or from `dev`.

Valid profile names are: must not be empty, use only alphanumeric characters or `-` or `_`.

Passing `--profile` with the profile's name to various Cargo commands, directs operations to use the profile's attributes. Overrides that are specified in the profiles from which the custom profile inherits are inherited too.

For example, using `cargo build` with `--profile` and the manifest from above:

```
cargo +nightly build --profile release-lto -Z unstable-options
```

When a custom profile is used, build artifcats go to a different target by default. In the example above, you can expect to see the outputs under `target/release-lto`.

New `dir-name` attribute

Some of the paths generated under `target/` have resulted in a de-facto "build protocol", where `cargo` is invoked as a part of a larger project build. So, to preserve the existing behavior, there is also a new attribute `dir-name`, which when left unspecified, defaults to the name of the profile. For example:

```
[profile.release-lto]
inherits = "release"
dir-name = "lto"  # Emits to target/lto instead of target/release-lto
lto = true
```

Config Profiles

- Tracking Issue: rust-lang/rust#48683
- RFC: #2282

Profiles can be specified in `.cargo/config` files. The `-Z config-profile` command-line flag is required to use this feature. The format is the same as in a `Cargo.toml` manifest. If found in multiple config files, settings will be merged using the regular config hierarchy. Config settings take precedence over manifest settings.

```
[profile.dev]
opt-level = 3
```

```
cargo +nightly build -Z config-profile
```

Namespaced features

- Original issue: #1286
- Tracking Issue: #5565

Currently, it is not possible to have a feature and a dependency with the same name in the manifest. If you set `namespaced-features` to `true`, the namespaces for features and dependencies are separated. The effect of this is that, in the feature requirements, dependencies have to be prefixed with `crate:`. Like this:

```
[package]
namespaced-features = true

[features]
bar = ["crate:baz", "foo"]
foo = []

[dependencies]
baz = { version = "0.1", optional = true }
```

To prevent unnecessary boilerplate from having to explicitly declare features for each optional dependency, implicit features get created for any optional dependencies where a feature of the same name is not defined. However, if a feature of the same name as a dependency is defined, that feature must include the dependency as a requirement, as `foo = ["crate:foo"]`.

Build-plan

- Tracking Issue: #5579

The `--build-plan` argument for the `build` command will output JSON with information about which commands would be run without actually executing anything. This can be useful when integrating with another build tool. Example:

```
cargo +nightly build --build-plan -Z unstable-options
```

Metabuild

- Tracking Issue: rust-lang/rust#49803
- RFC: #2196

Metabuild is a feature to have declarative build scripts. Instead of writing a `build.rs` script, you specify a list of build dependencies in the `metabuild` key in `Cargo.toml`. A build script is automatically generated that runs each build dependency in order. Metabuild packages can then read metadata from `Cargo.toml` to specify their behavior.

Include `cargo-features` at the top of `Cargo.toml`, a `metabuild` key in the `package`, list the dependencies in `build-dependencies`, and add any metadata that the metabuild packages require under `package.metadata`. Example:

```
cargo-features = ["metabuild"]

[package]
name = "mypackage"
version = "0.0.1"
metabuild = ["foo", "bar"]

[build-dependencies]
foo = "1.0"
bar = "1.0"

[package.metadata.foo]
extra-info = "qwerty"
```

Metabuild packages should have a public function called `metabuild` that performs the same actions as a regular `build.rs` script would perform.

public-dependency

- Tracking Issue: #44663

The 'public-dependency' feature allows marking dependencies as 'public' or 'private'. When this feature is enabled, additional information is passed to rustc to allow the 'exported_private_dependencies' lint to function properly.

This requires the appropriate key to be set in `cargo-features`:

```
cargo-features = ["public-dependency"]

[dependencies]
my_dep = { version = "1.2.3", public = true }
private_dep = "2.0.0" # Will be 'private' by default
```

build-std

- Tracking Repository: https://github.com/rust-lang/wg-cargo-std-aware

The `build-std` feature enables Cargo to compile the standard library itself as part of a crate graph compilation. This feature has also historically been known as "std-aware Cargo". This feature is still in very early stages of development, and is also a possible massive feature addition to Cargo. This is a very large feature to document, even in the minimal form that it exists in today, so if you're curious to stay up to date you'll want to follow the tracking repository and its set of issues.

The functionality implemented today is behind a flag called `-Z build-std`. This flag indicates that Cargo should compile the standard library from source code using the same profile as the main build itself. Note that for this to work you need to have the source code for the standard library available, and at this time the only supported method of doing so is to add the `rust-src` rust rustup component:

```
$ rustup component add rust-src --toolchain nightly
```

It is also required today that the `-Z build-std` flag is combined with the `--target` flag. Note that you're not forced to do a cross compilation, you're just forced to pass `--target` in one form or another.

Usage looks like:

```
$ cargo new foo
$ cd foo
$ cargo +nightly run -Z build-std --target x86_64-unknown-linux-gnu
   Compiling core v0.0.0 (...)
   ...
   Compiling foo v0.1.0 (...)
    Finished dev [unoptimized + debuginfo] target(s) in 21.00s
     Running `target/x86_64-unknown-linux-gnu/debug/foo`
Hello, world!
```

Here we recompiled the standard library in debug mode with debug assertions (like `src/main.rs` is compiled) and everything was linked together at the end.

Using `-Z build-std` will implicitly compile the stable crates `core`, `std`, `alloc`, and `proc_macro`. If you're using `cargo test` it will also compile the `test` crate. If you're working with an environment which does not support some of these crates, then you can pass an argument to `-Zbuild-std` as well:

```
$ cargo +nightly build -Z build-std=core,alloc
```

The value here is a comma-separated list of standard library crates to build.

Requirements

As a summary, a list of requirements today to use `-Z build-std` are:

- You must install libstd's source code through `rustup component add rust-src`
- You must pass `--target`
- You must use both a nightly Cargo and a nightly rustc
- The `-Z build-std` flag must be passed to all `cargo` invocations.

Reporting bugs and helping out

The `-Z build-std` feature is in the very early stages of development! This feature for Cargo has an extremely long history and is very large in scope, and this is just the beginning. If you'd like to report bugs please either report them to:

- Cargo - https://github.com/rust-lang/cargo/issues/new - for implementation bugs
- The tracking repository - https://github.com/rust-lang/wg-cargo-std-aware/issues/new - for larger design questions.

Also if you'd like to see a feature that's not yet implemented and/or if something doesn't quite work the way you'd like it to, feel free to check out the issue tracker of the tracking repository, and if it's not there please file a new issue!

timings

- Tracking Issue: #7405

The `timings` feature gives some information about how long each compilation takes, and tracks concurrency information over time.

```
cargo +nightly build -Z timings
```

The `-Ztimings` flag can optionally take a comma-separated list of the following values:

- `html` — Saves a file called `cargo-timing.html` to the current directory with a report of the compilation. Files are also saved with a timestamp in the filename if you want to look at older runs.
- `info` — Displays a message to stdout after each compilation finishes with how long it took.
- `json` — Emits some JSON information about timing information.

The default if none are specified is `html,info`.

Reading the graphs

There are two graphs in the output. The "unit" graph shows the duration of each unit over time. A "unit" is a single compiler invocation. There are lines that show which additional units are "unlocked" when a unit finishes. That is, it shows the new units that are now allowed to run because their dependencies are all finished. Hover the mouse over a unit to

highlight the lines. This can help visualize the critical path of dependencies. This may change between runs because the units may finish in different orders.

The "codegen" times are highlighted in a lavender color. In some cases, build pipelining allows units to start when their dependencies are performing code generation. This information is not always displayed (for example, binary units do not show when code generation starts).

The "custom build" units are `build.rs` scripts, which when run are highlighted in orange.

The second graph shows Cargo's concurrency over time. The three lines are:

- "Waiting" (red) — This is the number of units waiting for a CPU slot to open.
- "Inactive" (blue) — This is the number of units that are waiting for their dependencies to finish.
- "Active" (green) — This is the number of units currently running.

Note: This does not show the concurrency in the compiler itself. `rustc` coordinates with Cargo via the "job server" to stay within the concurrency limit. This currently mostly applies to the code generation phase.

Tips for addressing compile times:

- Look for slow dependencies.
  - Check if they have features that you may wish to consider disabling.
  - Consider trying to remove the dependency completely.
- Look for a crate being built multiple times with different versions. Try to remove the older versions from the dependency graph.
- Split large crates into smaller pieces.
- If there are a large number of crates bottlenecked on a single crate, focus your attention on improving that one crate to improve parallelism.

binary-dep-depinfo

- Tracking rustc issue: #63012

The `-Z binary-dep-depinfo` flag causes Cargo to forward the same flag to `rustc` which will then cause `rustc` to include the paths of all binary dependencies in the "dep info" file (with the `.d` extension). Cargo then uses that information for change-detection (if any binary dependency changes, then the crate will be rebuilt). The primary use case is for building the compiler itself, which has implicit dependencies on the standard library that would otherwise be untracked for change-detection.

panic-abort-tests

The `-Z panic-abort-tests` flag will enable nightly support to compile test harness crates with `-Cpanic=abort`. Without this flag Cargo will compile tests, and everything they depend

on, with `-Cpanic=unwind` because it's the only way `test`-the-crate knows how to operate. As of rust-lang/rust#64158, however, the `test` crate supports `-C panic=abort` with a test-per-process, and can help avoid compiling crate graphs multiple times.

It's currently unclear how this feature will be stabilized in Cargo, but we'd like to stabilize it somehow!

cargo

NAME

cargo - The Rust package manager

SYNOPSIS

```
cargo [OPTIONS] COMMAND [ARGS]
cargo [OPTIONS] --version
cargo [OPTIONS] --list
cargo [OPTIONS] --help
cargo [OPTIONS] --explain CODE
```

DESCRIPTION

This program is a package manager and build tool for the Rust language, available at https://rust-lang.org.

COMMANDS

Build Commands

cargo-bench(1)
: Execute benchmarks of a package.

cargo-build(1)
: Compile a package.

cargo-check(1)

Check a local package and all of its dependencies for errors.

cargo-clean(1)
: Remove artifacts that Cargo has generated in the past.

cargo-doc(1)
: Build a package's documentation.

cargo-fetch(1)
: Fetch dependencies of a package from the network.

cargo-fix(1)
: Automatically fix lint warnings reported by rustc.

cargo-run(1)
: Run a binary or example of the local package.

cargo-rustc(1)
: Compile a package, and pass extra options to the compiler.

cargo-rustdoc(1)
: Build a package's documentation, using specified custom flags.

cargo-test(1)
: Execute unit and integration tests of a package.

Manifest Commands

cargo-generate-lockfile(1)
: Generate `Cargo.lock` for a project.

cargo-locate-project(1)
: Print a JSON representation of a `Cargo.toml` file's location.

cargo-metadata(1)
: Output the resolved dependencies of a package, the concrete used versions including overrides, in machine-readable format.

cargo-pkgid(1)
: Print a fully qualified package specification.

cargo-update(1)
: Update dependencies as recorded in the local lock file.

cargo-verify-project(1)
: Check correctness of crate manifest.

Package Commands

cargo-init(1)
: Create a new Cargo package in an existing directory.

cargo-install(1)
: Build and install a Rust binary.

cargo-new(1)
: Create a new Cargo package.

cargo-search(1)
: Search packages in crates.io.

cargo-uninstall(1)
: Remove a Rust binary.

Publishing Commands

cargo-login(1)
: Save an API token from the registry locally.

cargo-owner(1)
: Manage the owners of a crate on the registry.

cargo-package(1)
: Assemble the local package into a distributable tarball.

cargo-publish(1)
: Upload a package to the registry.

cargo-yank(1)
: Remove a pushed crate from the index.

General Commands

cargo-help(1)
: Display help information about Cargo.

cargo-version(1)
: Show version information.

OPTIONS

Special Options

-V
--version
Print version info and exit. If used with `--verbose`, prints extra information.

--list
List all installed Cargo subcommands. If used with `--verbose`, prints extra information.

--explain *CODE*
Run `rustc --explain CODE` which will print out a detailed explanation of an error message (for example, `E0004`).

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Manifest Options

--frozen
--locked
Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline

Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

May also be specified with the `net.offline` config value.

Common Options

-h
--help
Prints help information.

-Z *FLAG*...
Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
Cargo succeeded.

101
Cargo failed to complete.

FILES

`~/.cargo/`
Default location for Cargo's "home" directory where it stores various files. The location can be changed with the `CARGO_HOME` environment variable.

`$CARGO_HOME/bin/`
Binaries installed by cargo-install(1) will be located here. If using rustup, executables distributed with Rust are also located here.

`$CARGO_HOME/config`
: The global configuration file. See the reference for more information about configuration files.

`.cargo/config`
: Cargo automatically searches for a file named `.cargo/config` in the current directory, and all parent directories. These configuration files will be merged with the global configuration file.

`$CARGO_HOME/credentials`
: Private authentication information for logging in to a registry.

`$CARGO_HOME/registry/`
: This directory contains cached downloads of the registry index and any downloaded dependencies.

`$CARGO_HOME/git/`
: This directory contains cached downloads of git dependencies.

Please note that the internal structure of the `$CARGO_HOME` directory is not stable yet and may be subject to change.

EXAMPLES

1. Build a local package and all of its dependencies:

   ```
   cargo build
   ```

2. Build a package with optimizations:

   ```
   cargo build --release
   ```

3. Run tests for a cross-compiled target:

   ```
   cargo test --target i686-unknown-linux-gnu
   ```

4. Create a new package that builds an executable:

   ```
   cargo new foobar
   ```

5. Create a package in the current directory:

   ```
   mkdir foo && cd foo
   cargo init .
   ```

6. Learn about a command's options and usage:

   ```
   cargo help clean
   ```

BUGS

See https://github.com/rust-lang/cargo/issues for issues.

SEE ALSO

rustc(1), rustdoc(1)

Build Commands

cargo bench

NAME

cargo-bench - Execute benchmarks of a package

SYNOPSIS

`cargo bench [*OPTIONS*] [BENCHNAME] [-- *BENCH-OPTIONS*]`

DESCRIPTION

Compile and execute benchmarks.

The benchmark filtering argument `BENCHNAME` and all the arguments following the two dashes (`--`) are passed to the benchmark binaries and thus to *libtest* (rustc's built in unit-test and micro-benchmarking framework). If you're passing arguments to both Cargo and the binary, the ones after `--` go to the binary, the ones before go to Cargo. For details about libtest's arguments see the output of `cargo bench -- --help`. As an example, this will run only the benchmark named `foo` (and skip other similarly named benchmarks like `foobar`):

```
cargo bench -- foo --exact
```

Benchmarks are built with the `--test` option to `rustc` which creates an executable with a `main` function that automatically runs all functions annotated with the `#[bench]` attribute. Cargo passes the `--bench` flag to the test harness to tell it to run only benchmarks.

The libtest harness may be disabled by setting `harness = false` in the target manifest settings, in which case your code will need to provide its own `main` function to handle running benchmarks.

OPTIONS

Benchmark Options

--no-run
: Compile, but don't run benchmarks.

--no-fail-fast
: Run all benchmarks regardless of failure. Without this flag, Cargo will exit after the first executable fails. The Rust test harness will run all benchmarks within the executable to completion, this flag only applies to the executable as a whole.

Package Selection

By default, when no package selection options are given, the packages selected depend on the selected manifest file (based on the current working directory if `--manifest-path` is not given). If the manifest is the root of a workspace then the workspaces default members are selected, otherwise only the package defined by the manifest will be selected.

The default members of a workspace can be set explicitly with the `workspace.default-members` key in the root manifest. If this is not set, a virtual workspace will include all workspace members (equivalent to passing `--workspace`), and a non-virtual workspace will include only the root crate itself.

-p *SPEC*...
--package *SPEC*...
: Benchmark only the specified packages. See cargo-pkgid(1) for the SPEC format. This flag may be specified multiple times.

--workspace
: Benchmark all members in the workspace.

--all
: Deprecated alias for `--workspace`.

--exclude *SPEC*...
: Exclude the specified packages. Must be used in conjunction with the `--workspace` flag. This flag may be specified multiple times.

Target Selection

When no target selection options are given, `cargo bench` will build the following targets of the selected packages:

- lib — used to link with binaries and benchmarks
- bins (only if benchmark targets are built and required features are available)
- lib as a benchmark
- bins as benchmarks
- benchmark targets

The default behavior can be changed by setting the `bench` flag for the target in the manifest settings. Setting examples to `bench = true` will build and run the example as a benchmark. Setting targets to `bench = false` will stop them from being benchmarked by default. Target selection options that take a target by name ignore the `bench` flag and will always benchmark the given target.

Passing target selection flags will benchmark only the specified targets.

--lib
Benchmark the package's library.

--bin *NAME*...
Benchmark the specified binary. This flag may be specified multiple times.

--bins
Benchmark all binary targets.

--example *NAME*...
Benchmark the specified example. This flag may be specified multiple times.

--examples
Benchmark all example targets.

--test *NAME*...
Benchmark the specified integration test. This flag may be specified multiple times.

--tests
Benchmark all targets in test mode that have the `test = true` manifest flag set. By default this includes the library and binaries built as unittests, and integration tests. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a unittest, and once as a dependency for binaries, integration tests, etc.). Targets may be enabled or disabled by setting the `test` flag in the manifest settings for the target.

--bench *NAME*...
Benchmark the specified benchmark. This flag may be specified multiple times.

--benches

Benchmark all targets in benchmark mode that have the `bench = true` manifest flag set. By default this includes the library and binaries built as benchmarks, and bench targets. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a benchmark, and once as a dependency for binaries, benchmarks, etc.). Targets may be enabled or disabled by setting the `bench` flag in the manifest settings for the target.

--all-targets
Benchmark all targets. This is equivalent to specifying `--lib --bins --tests --benches --examples`.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
Activate all available features of all selected packages.

--no-default-features
Do not activate the `default` feature of the current directory's package.

Compilation Options

--target *TRIPLE*
Benchmark for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

This may also be specified with the `build.target` config value.

Output Options

--target-dir *DIRECTORY*
Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

Display Options

By default the Rust test harness hides output from benchmark execution to keep results readable. Benchmark output can be recovered (e.g., for debugging) by passing `--nocapture` to the benchmark binaries:

```
cargo bench -- --nocapture
```

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

--message-format *FMT*
The output format for diagnostic messages. Can be specified multiple times and consists of comma-separated values. Valid values:

- `human` (default): Display in a human-readable text format.
- `short`: Emit shorter, human-readable text messages.
- `json`: Emit JSON messages to stdout. See the reference for more details.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics in in JSON messages printed, but instead Cargo itself should render the JSON diagnostics coming from rustc. Cargo's own JSON diagnostics and others coming from rustc are still emitted.

Manifest Options

--manifest-path *PATH*
Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen

--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

  These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

Miscellaneous Options

The `--jobs` argument affects the building of the benchmark executable but does not affect how many threads are used when running the benchmarks. The Rust test harness runs benchmarks serially in a single thread.

-j *N*
--jobs *N*
: Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

PROFILES

Profiles may be used to configure compiler options such as optimization levels and debug settings. See the reference for more details.

Benchmarks are always built with the `bench` profile. Binary and lib targets are built separately as benchmarks with the `bench` profile. Library targets are built with the `release` profiles when linked to binaries and benchmarks. Dependencies use the `release` profile.

If you need a debug build of a benchmark, try building it with cargo-build(1) which will use the `test` profile which is by default unoptimized and includes debug information. You can then run the debug-enabled benchmark manually.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Build and execute all the benchmarks of the current package:

   ```
   cargo bench
   ```

2. Run only a specific benchmark within a specific benchmark target:

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

SEE ALSO

cargo(1), cargo-test(1)

cargo build

NAME

cargo-build - Compile the current package

SYNOPSIS

`cargo build [*OPTIONS*]`

DESCRIPTION

Compile local packages and all of their dependencies.

OPTIONS

Package Selection

By default, when no package selection options are given, the packages selected depend on the selected manifest file (based on the current working directory if `--manifest-path` is not given). If the manifest is the root of a workspace then the workspaces default members are selected, otherwise only the package defined by the manifest will be selected.

The default members of a workspace can be set explicitly with the `workspace.default-members` key in the root manifest. If this is not set, a virtual workspace will include all workspace members (equivalent to passing `--workspace`), and a non-virtual workspace will include only the root crate itself.

-p *SPEC*...
--package *SPEC*...
: Build only the specified packages. See cargo-pkgid(1) for the SPEC format. This flag may be specified multiple times.

--workspace
: Build all members in the workspace.

--all
: Deprecated alias for `--workspace`.

--exclude *SPEC*...
: Exclude the specified packages. Must be used in conjunction with the `--workspace` flag. This flag may be specified multiple times.

Target Selection

When no target selection options are given, `cargo build` will build all binary and library targets of the selected packages. Binaries are skipped if they have `required-features` that are missing.

Passing target selection flags will build only the specified targets.

--lib
Build the package's library.

--bin *NAME*...
Build the specified binary. This flag may be specified multiple times.

--bins
Build all binary targets.

--example *NAME*...
Build the specified example. This flag may be specified multiple times.

--examples
Build all example targets.

--test *NAME*...
Build the specified integration test. This flag may be specified multiple times.

--tests
Build all targets in test mode that have the `test = true` manifest flag set. By default this includes the library and binaries built as unittests, and integration tests. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a unittest, and once as a dependency for binaries, integration tests, etc.). Targets may be enabled or disabled by setting the `test` flag in the manifest settings for the target.

--bench *NAME*...
Build the specified benchmark. This flag may be specified multiple times.

--benches
Build all targets in benchmark mode that have the `bench = true` manifest flag set. By default this includes the library and binaries built as benchmarks, and bench targets. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a benchmark, and once as a dependency for binaries, benchmarks, etc.). Targets may be enabled or disabled by setting the `bench` flag in the manifest settings for the target.

--all-targets
Build all targets. This is equivalent to specifying `--lib --bins --tests --benches --examples`.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
: Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
: Activate all available features of all selected packages.

--no-default-features
: Do not activate the `default` feature of the current directory's package.

Compilation Options

--target *TRIPLE*
: Build for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

  This may also be specified with the `build.target` config value.

--release
: Build optimized artifacts with the `release` profile. See the PROFILES section for details on how this affects profile selection.

Output Options

--target-dir *DIRECTORY*
: Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

--out-dir *DIRECTORY*
: Copy final artifacts to this directory.

  This option is unstable and available only on the nightly channel and requires the `-Z unstable-options` flag to enable. See https://github.com/rust-lang/cargo/issues/6790 for more information.

Display Options

-v

--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

--message-format *FMT*
The output format for diagnostic messages. Can be specified multiple times and consists of comma-separated values. Valid values:

- `human` (default): Display in a human-readable text format.
- `short`: Emit shorter, human-readable text messages.
- `json`: Emit JSON messages to stdout. See the reference for more details.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics in in JSON messages printed, but instead Cargo itself should render the JSON diagnostics coming from rustc. Cargo's own JSON diagnostics and others coming from rustc are still emitted.

--build-plan
Outputs a series of JSON messages to stdout that indicate the commands to run the build.

This option is unstable and available only on the nightly channel and requires the `-Z unstable-options` flag to enable. See https://github.com/rust-lang/cargo/issues/5579 for more information.

Manifest Options

--manifest-path *PATH*
Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

  These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

Miscellaneous Options

-j *N*
--jobs *N*
: Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

PROFILES

Profiles may be used to configure compiler options such as optimization levels and debug settings. See the reference for more details.

Profile selection depends on the target and crate being built. By default the `dev` or `test` profiles are used. If the `--release` flag is given, then the `release` or `bench` profiles are used.

Target	Default Profile	`--release` Profile
lib, bin, example	`dev`	`release`
test, bench, or any target in "test" or "bench" mode	`test`	`bench`

Dependencies use the `dev`/`release` profiles.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Build the local package and all of its dependencies:

   ```
   cargo build
   ```

2. Build with optimizations:

   ```
   cargo build --release
   ```

SEE ALSO

cargo(1), cargo-rustc(1)

cargo check

NAME

cargo-check - Check the current package

SYNOPSIS

`cargo check [*OPTIONS*]`

DESCRIPTION

Check a local package and all of its dependencies for errors. This will essentially compile the packages without performing the final step of code generation, which is faster than running `cargo build`. The compiler will save metadata files to disk so that future runs will reuse them if the source has not been modified.

OPTIONS

Package Selection

By default, when no package selection options are given, the packages selected depend on the selected manifest file (based on the current working directory if `--manifest-path` is not given). If the manifest is the root of a workspace then the workspaces default members are selected, otherwise only the package defined by the manifest will be selected.

The default members of a workspace can be set explicitly with the `workspace.default-members` key in the root manifest. If this is not set, a virtual workspace will include all workspace members (equivalent to passing `--workspace`), and a non-virtual workspace will include only the root crate itself.

-p *SPEC*...
--package *SPEC*...
: Check only the specified packages. See cargo-pkgid(1) for the SPEC format. This flag may be specified multiple times.

--workspace

Check all members in the workspace.

--all
Deprecated alias for `--workspace`.

--exclude *SPEC*…
Exclude the specified packages. Must be used in conjunction with the `--workspace` flag. This flag may be specified multiple times.

Target Selection

When no target selection options are given, `cargo check` will check all binary and library targets of the selected packages. Binaries are skipped if they have `required-features` that are missing.

Passing target selection flags will check only the specified targets.

--lib
Check the package's library.

--bin *NAME*…
Check the specified binary. This flag may be specified multiple times.

--bins
Check all binary targets.

--example *NAME*…
Check the specified example. This flag may be specified multiple times.

--examples
Check all example targets.

--test *NAME*…
Check the specified integration test. This flag may be specified multiple times.

--tests
Check all targets in test mode that have the `test = true` manifest flag set. By default this includes the library and binaries built as unittests, and integration tests. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a unittest, and once as a dependency for binaries, integration tests, etc.). Targets may be enabled or disabled by setting the `test` flag in the manifest settings for the target.

--bench *NAME*…
Check the specified benchmark. This flag may be specified multiple times.

--benches
Check all targets in benchmark mode that have the `bench = true` manifest flag set. By default this includes the library and binaries built as benchmarks, and bench targets.

Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a benchmark, and once as a dependency for binaries, benchmarks, etc.). Targets may be enabled or disabled by setting the `bench` flag in the manifest settings for the target.

--all-targets
Check all targets. This is equivalent to specifying `--lib --bins --tests --benches --examples`.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
Activate all available features of all selected packages.

--no-default-features
Do not activate the `default` feature of the current directory's package.

Compilation Options

--target *TRIPLE*
Check for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

This may also be specified with the `build.target` config value.

--release
Check optimized artifacts with the `release` profile. See the PROFILES section for details on how this affects profile selection.

--profile *NAME*
Changes check behavior. Currently only `test` is supported, which will check with the `#[cfg(test)]` attribute enabled. This is useful to have it check unit tests which are usually excluded via the `cfg` attribute. This does not change the actual profile used.

Output Options

--target-dir *DIRECTORY*

Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

--message-format *FMT*
The output format for diagnostic messages. Can be specified multiple times and consists of comma-separated values. Valid values:

- `human` (default): Display in a human-readable text format.
- `short`: Emit shorter, human-readable text messages.
- `json`: Emit JSON messages to stdout. See the reference for more details.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics in in JSON messages printed, but instead Cargo itself should render the JSON diagnostics coming from rustc. Cargo's own JSON diagnostics and others coming from rustc are still emitted.

Manifest Options

--manifest-path *PATH*
Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

  These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

Miscellaneous Options

-j *N*
--jobs *N*
: Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

PROFILES

Profiles may be used to configure compiler options such as optimization levels and debug settings. See the reference for more details.

Profile selection depends on the target and crate being built. By default the `dev` or `test` profiles are used. If the `--release` flag is given, then the `release` or `bench` profiles are used.

Target	Default Profile	`--release` Profile
lib, bin, example	`dev`	`release`
test, bench, or any target in "test" or "bench" mode	`test`	`bench`

Dependencies use the `dev`/`release` profiles.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Check the local package for errors:

   ```
   cargo check
   ```

2. Check all targets, including unit tests:

   ```
   cargo check --all-targets --profile=test
   ```

SEE ALSO

cargo(1), cargo-build(1)

cargo clean

NAME

cargo-clean - Remove generated artifacts

SYNOPSIS

`cargo clean [*OPTIONS*]`

DESCRIPTION

Remove artifacts from the target directory that Cargo has generated in the past.

With no options, `cargo clean` will delete the entire target directory.

OPTIONS

Package Selection

When no packages are selected, all packages and all dependencies in the workspace are cleaned.

-p *SPEC*...
--package *SPEC*...
: Clean only the specified packages. This flag may be specified multiple times. See cargo-pkgid(1) for the SPEC format.

Clean Options

--doc
: This option will cause `cargo clean` to remove only the `doc` directory in the target directory.

--release
: Clean all artifacts that were built with the `release` or `bench` profiles.

--target-dir *DIRECTORY*
: Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

--target *TRIPLE*
: Clean for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

  This may also be specified with the `build.target` config value.

Display Options

-v
--verbose
: Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
: No output printed to stdout.

--color *WHEN*
: Control when colored output is used. Valid values:

  - `auto` (default): Automatically detect if color support is available on the terminal.
  - `always`: Always display colors.
  - `never`: Never display colors.

  May also be specified with the `term.color` config value.

Manifest Options

--manifest-path *PATH*
: Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Remove the entire target directory:

   ```
   cargo clean
   ```

2. Remove only the release artifacts:

```
cargo clean --release
```

SEE ALSO

cargo(1), cargo-build(1)

cargo doc

NAME

cargo-doc - Build a package's documentation

SYNOPSIS

`cargo doc [*OPTIONS*]`

DESCRIPTION

Build the documentation for the local package and all dependencies. The output is placed in `target/doc` in rustdoc's usual format.

OPTIONS

Documentation Options

--open
: Open the docs in a browser after building them. This will use your default browser unless you define another one in the `BROWSER` environment variable.

--no-deps
: Do not build documentation for dependencies.

--document-private-items

Include non-public items in the documentation.

Package Selection

By default, when no package selection options are given, the packages selected depend on the selected manifest file (based on the current working directory if `--manifest-path` is not given). If the manifest is the root of a workspace then the workspaces default members are selected, otherwise only the package defined by the manifest will be selected.

The default members of a workspace can be set explicitly with the `workspace.default-members` key in the root manifest. If this is not set, a virtual workspace will include all workspace members (equivalent to passing `--workspace`), and a non-virtual workspace will include only the root crate itself.

-p *SPEC*…
--package *SPEC*…
Document only the specified packages. See cargo-pkgid(1) for the SPEC format. This flag may be specified multiple times.

--workspace
Document all members in the workspace.

--all
Deprecated alias for `--workspace`.

--exclude *SPEC*…
Exclude the specified packages. Must be used in conjunction with the `--workspace` flag. This flag may be specified multiple times.

Target Selection

When no target selection options are given, `cargo doc` will document all binary and library targets of the selected package. The binary will be skipped if its name is the same as the lib target. Binaries are skipped if they have `required-features` that are missing.

The default behavior can be changed by setting `doc = false` for the target in the manifest settings. Using target selection options will ignore the `doc` flag and will always document the given target.

--lib
Document the package's library.

--bin *NAME*…
Document the specified binary. This flag may be specified multiple times.

--bins
Document all binary targets.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
: Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
: Activate all available features of all selected packages.

--no-default-features
: Do not activate the `default` feature of the current directory's package.

Compilation Options

--target *TRIPLE*
: Document for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

  This may also be specified with the `build.target` config value.

--release
: Document optimized artifacts with the `release` profile. See the PROFILES section for details on how this affects profile selection.

Output Options

--target-dir *DIRECTORY*
: Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

Display Options

-v
--verbose
: Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q

--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

--message-format *FMT*
The output format for diagnostic messages. Can be specified multiple times and consists of comma-separated values. Valid values:

- `human` (default): Display in a human-readable text format.
- `short`: Emit shorter, human-readable text messages.
- `json`: Emit JSON messages to stdout. See the reference for more details.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics in in JSON messages printed, but instead Cargo itself should render the JSON diagnostics coming from rustc. Cargo's own JSON diagnostics and others coming from rustc are still emitted.

Manifest Options

--manifest-path *PATH*
Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

May also be specified with the `net.offline` config value.

Common Options

-h
--help
Prints help information.

-Z *FLAG*...
Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

Miscellaneous Options

-j *N*
--jobs *N*
Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

PROFILES

Profiles may be used to configure compiler options such as optimization levels and debug settings. See the reference for more details.

Profile selection depends on the target and crate being built. By default the `dev` or `test` profiles are used. If the `--release` flag is given, then the `release` or `bench` profiles are used.

Target	Default Profile	`--release` Profile
lib, bin, example	`dev`	`release`
test, bench, or any target in "test" or "bench" mode	`test`	`bench`

Dependencies use the `dev`/`release` profiles.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Build the local package documentation and its dependencies and output to `target/doc`.

```
cargo doc
```

SEE ALSO

cargo(1), cargo-rustdoc(1), rustdoc(1)

cargo fetch

NAME

cargo-fetch - Fetch dependencies of a package from the network

SYNOPSIS

`cargo fetch [OPTIONS]`

DESCRIPTION

If a `Cargo.lock` file is available, this command will ensure that all of the git dependencies and/or registry dependencies are downloaded and locally available. Subsequent Cargo commands never touch the network after a `cargo fetch` unless the lock file changes.

If the lock file is not available, then this command will generate the lock file before fetching the dependencies.

If `--target` is not specified, then all target dependencies are fetched.

See also the cargo-prefetch plugin which adds a command to download popular crates. This may be useful if you plan to use Cargo without a network with the `--offline` flag.

OPTIONS

Fetch options

--target *TRIPLE*
: Fetch for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

  This may also be specified with the `build.target` config value.

Display Options

-v
--verbose
: Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
: No output printed to stdout.

--color *WHEN*
: Control when colored output is used. Valid values:

  - `auto` (default): Automatically detect if color support is available on the terminal.
  - `always`: Always display colors.
  - `never`: Never display colors.

May also be specified with the `term.color` config value.

Manifest Options

--manifest-path *PATH*
: Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

  These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Fetch all dependencies:

   ```
   cargo fetch
   ```

SEE ALSO

cargo(1), cargo-update(1), cargo-generate-lockfile(1)

cargo fix

NAME

cargo-fix - Automatically fix lint warnings reported by rustc

SYNOPSIS

`cargo fix [OPTIONS]`

DESCRIPTION

This Cargo subcommand will automatically take rustc's suggestions from diagnostics like warnings and apply them to your source code. This is intended to help automate tasks that rustc itself already knows how to tell you to fix! The `cargo fix` subcommand is also being developed for the Rust 2018 edition to provide code the ability to easily opt-in to the new edition without having to worry about any breakage.

Executing `cargo fix` will under the hood execute cargo-check(1). Any warnings applicable to your crate will be automatically fixed (if possible) and all remaining warnings will be displayed when the check process is finished. For example if you'd like to prepare for the 2018 edition, you can do so by executing:

```
cargo fix --edition
```

which behaves the same as `cargo check --all-targets`. Similarly if you'd like to fix code for different platforms you can do:

```
cargo fix --edition --target x86_64-pc-windows-gnu
```

or if your crate has optional features:

```
cargo fix --edition --no-default-features --features foo
```

If you encounter any problems with `cargo fix` or otherwise have any questions or feature requests please don't hesitate to file an issue at https://github.com/rust-lang/cargo

OPTIONS

Fix options

--broken-code
: Fix code even if it already has compiler errors. This is useful if `cargo fix` fails to apply the changes. It will apply the changes and leave the broken code in the working directory for you to inspect and manually fix.

--edition
: Apply changes that will update the code to the latest edition. This will not update the edition in the `Cargo.toml` manifest, which must be updated manually.

--edition-idioms
: Apply suggestions that will update code to the preferred style for the current edition.

--allow-no-vcs
: Fix code even if a VCS was not detected.

--allow-dirty
: Fix code even if the working directory has changes.

--allow-staged
: Fix code even if the working directory has staged changes.

Package Selection

By default, when no package selection options are given, the packages selected depend on the selected manifest file (based on the current working directory if `--manifest-path` is not given). If the manifest is the root of a workspace then the workspaces default members are selected, otherwise only the package defined by the manifest will be selected.

The default members of a workspace can be set explicitly with the `workspace.default-members` key in the root manifest. If this is not set, a virtual workspace will include all workspace members (equivalent to passing `--workspace`), and a non-virtual workspace will include only the root crate itself.

-p *SPEC*...
--package *SPEC*...
Fix only the specified packages. See cargo-pkgid(1) for the SPEC format. This flag may be specified multiple times.

--workspace
Fix all members in the workspace.

--all
Deprecated alias for `--workspace`.

--exclude *SPEC*...
Exclude the specified packages. Must be used in conjunction with the `--workspace` flag. This flag may be specified multiple times.

Target Selection

When no target selection options are given, `cargo fix` will fix all targets (`--all-targets` implied). Binaries are skipped if they have `required-features` that are missing.

Passing target selection flags will fix only the specified targets.

--lib
Fix the package's library.

--bin *NAME*...
Fix the specified binary. This flag may be specified multiple times.

--bins
Fix all binary targets.

--example *NAME*...
Fix the specified example. This flag may be specified multiple times.

--examples
Fix all example targets.

--test *NAME*...
: Fix the specified integration test. This flag may be specified multiple times.

--tests
: Fix all targets in test mode that have the `test = true` manifest flag set. By default this includes the library and binaries built as unittests, and integration tests. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a unittest, and once as a dependency for binaries, integration tests, etc.). Targets may be enabled or disabled by setting the `test` flag in the manifest settings for the target.

--bench *NAME*...
: Fix the specified benchmark. This flag may be specified multiple times.

--benches
: Fix all targets in benchmark mode that have the `bench = true` manifest flag set. By default this includes the library and binaries built as benchmarks, and bench targets. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a benchmark, and once as a dependency for binaries, benchmarks, etc.). Targets may be enabled or disabled by setting the `bench` flag in the manifest settings for the target.

--all-targets
: Fix all targets. This is equivalent to specifying `--lib --bins --tests --benches --examples`.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
: Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
: Activate all available features of all selected packages.

--no-default-features
: Do not activate the `default` feature of the current directory's package.

Compilation Options

--target *TRIPLE*
: Fix for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list`

for a list of supported targets.

This may also be specified with the `build.target` config value.

--release
Fix optimized artifacts with the `release` profile. See the PROFILES section for details on how this affects profile selection.

--profile *NAME*
Changes fix behavior. Currently only `test` is supported, which will fix with the `#[cfg(test)]` attribute enabled. This is useful to have it fix unit tests which are usually excluded via the `cfg` attribute. This does not change the actual profile used.

Output Options

--target-dir *DIRECTORY*
Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

--message-format *FMT*
The output format for diagnostic messages. Can be specified multiple times and consists of comma-separated values. Valid values:

- `human` (default): Display in a human-readable text format.
- `short`: Emit shorter, human-readable text messages.

- `json`: Emit JSON messages to stdout. See the reference for more details.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics in in JSON messages printed, but instead Cargo itself should render the JSON diagnostics coming from rustc. Cargo's own JSON diagnostics and others coming from rustc are still emitted.

Manifest Options

--manifest-path *PATH*
: Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

  These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

Miscellaneous Options

-j *N*
--jobs *N*
: Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

PROFILES

Profiles may be used to configure compiler options such as optimization levels and debug settings. See the reference for more details.

Profile selection depends on the target and crate being built. By default the `dev` or `test` profiles are used. If the `--release` flag is given, then the `release` or `bench` profiles are used.

Target	Default Profile	`--release` Profile
lib, bin, example	`dev`	`release`
test, bench, or any target in "test" or "bench" mode	`test`	`bench`

Dependencies use the `dev`/`release` profiles.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Apply compiler suggestions to the local package:

    ```
    cargo fix
    ```

2. Convert a 2015 edition to 2018:

    ```
    cargo fix --edition
    ```

3. Apply suggested idioms for the current edition:

    ```
    cargo fix --edition-idioms
    ```

SEE ALSO

cargo(1), cargo-check(1)

cargo run

NAME

cargo-run - Run the current package

SYNOPSIS

`cargo run [*OPTIONS*] [-- *ARGS*]`

DESCRIPTION

Run a binary or example of the local package.

All the arguments following the two dashes (`--`) are passed to the binary to run. If you're passing arguments to both Cargo and the binary, the ones after `--` go to the binary, the ones before go to Cargo.

OPTIONS

Package Selection

By default, the package in the current working directory is selected. The `-p` flag can be used to choose a different package in a workspace.

-p *SPEC*
--package *SPEC*
The package to run. See cargo-pkgid(1) for the SPEC format.

Target Selection

When no target selection options are given, `cargo run` will run the binary target. If there are multiple binary targets, you must pass a target flag to choose one. Or, the `default-run` field may be specified in the `[package]` section of `Cargo.toml` to choose the name of the binary to run by default.

--bin *NAME*
Run the specified binary.

--example *NAME*
Run the specified example.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
Activate all available features of all selected packages.

--no-default-features
Do not activate the `default` feature of the current directory's package.

Compilation Options

--target *TRIPLE*
Run for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

This may also be specified with the `build.target` config value.

--release

Run optimized artifacts with the `release` profile. See the PROFILES section for details on how this affects profile selection.

Output Options

--target-dir *DIRECTORY*
Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

--message-format *FMT*
The output format for diagnostic messages. Can be specified multiple times and consists of comma-separated values. Valid values:

- `human` (default): Display in a human-readable text format.
- `short`: Emit shorter, human-readable text messages.
- `json`: Emit JSON messages to stdout. See the reference for more details.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics in in JSON messages printed, but instead Cargo itself should render the JSON diagnostics coming from rustc. Cargo's own JSON diagnostics and others coming from rustc are still emitted.

Manifest Options

--manifest-path *PATH*
: Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

  These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

Miscellaneous Options

-j *N*
--jobs *N*
: Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

PROFILES

Profiles may be used to configure compiler options such as optimization levels and debug settings. See the reference for more details.

Profile selection depends on the target and crate being built. By default the `dev` or `test` profiles are used. If the `--release` flag is given, then the `release` or `bench` profiles are used.

Target	Default Profile	`--release` Profile
lib, bin, example	`dev`	`release`
test, bench, or any target in "test" or "bench" mode	`test`	`bench`

Dependencies use the `dev`/`release` profiles.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Build the local package and run its main target (assuming only one binary):

   ```
   cargo run
   ```

2. Run an example with extra arguments:

   ```
   cargo run --example exname -- --exoption exarg1 exarg2
   ```

SEE ALSO

cargo(1), cargo-build(1)

cargo rustc

NAME

cargo-rustc - Compile the current package, and pass extra options to the compiler

SYNOPSIS

`cargo rustc [*OPTIONS*] [-- *ARGS*]`

DESCRIPTION

The specified target for the current package (or package specified by `-p` if provided) will be compiled along with all of its dependencies. The specified *ARGS* will all be passed to the final compiler invocation, not any of the dependencies. Note that the compiler will still unconditionally receive arguments such as `-L`, `--extern`, and `--crate-type`, and the specified *ARGS* will simply be added to the compiler invocation.

See https://doc.rust-lang.org/rustc/index.html for documentation on rustc flags.

This command requires that only one target is being compiled when additional arguments are provided. If more than one target is available for the current package the filters of `--lib`, `--bin`, etc, must be used to select which target is compiled. To pass flags to all compiler processes spawned by Cargo, use the `RUSTFLAGS` environment variable or the `build.rustflags` config value.

OPTIONS

Package Selection

By default, the package in the current working directory is selected. The `-p` flag can be used to choose a different package in a workspace.

-p *SPEC*
--package *SPEC*
The package to build. See cargo-pkgid(1) for the SPEC format.

Target Selection

When no target selection options are given, `cargo rustc` will build all binary and library targets of the selected package.

Passing target selection flags will build only the specified targets.

--lib
Build the package's library.

--bin *NAME*...
Build the specified binary. This flag may be specified multiple times.

--bins
Build all binary targets.

--example *NAME*...
Build the specified example. This flag may be specified multiple times.

--examples
Build all example targets.

--test *NAME*...
Build the specified integration test. This flag may be specified multiple times.

--tests
Build all targets in test mode that have the `test = true` manifest flag set. By default this includes the library and binaries built as unittests, and integration tests. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a unittest, and once as a dependency for binaries, integration tests, etc.). Targets may be enabled or disabled by setting the `test` flag in the manifest settings for the target.

--bench *NAME*...
Build the specified benchmark. This flag may be specified multiple times.

--benches
Build all targets in benchmark mode that have the `bench = true` manifest flag set. By default this includes the library and binaries built as benchmarks, and bench targets. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a benchmark, and once as a dependency for binaries, benchmarks, etc.). Targets may be enabled or disabled by setting the `bench` flag in the manifest settings for the target.

--all-targets

Build all targets. This is equivalent to specifying `--lib --bins --tests --benches --examples`.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
Activate all available features of all selected packages.

--no-default-features
Do not activate the `default` feature of the current directory's package.

Compilation Options

--target *TRIPLE*
Build for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

This may also be specified with the `build.target` config value.

--release
Build optimized artifacts with the `release` profile. See the PROFILES section for details on how this affects profile selection.

Output Options

--target-dir *DIRECTORY*
Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

Display Options

-v
--verbose

Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

--message-format *FMT*
The output format for diagnostic messages. Can be specified multiple times and consists of comma-separated values. Valid values:

- `human` (default): Display in a human-readable text format.
- `short`: Emit shorter, human-readable text messages.
- `json`: Emit JSON messages to stdout. See the reference for more details.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics in in JSON messages printed, but instead Cargo itself should render the JSON diagnostics coming from rustc. Cargo's own JSON diagnostics and others coming from rustc are still emitted.

Manifest Options

--manifest-path *PATH*
Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

Miscellaneous Options

-j *N*
--jobs *N*
: Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

PROFILES

Profiles may be used to configure compiler options such as optimization levels and debug settings. See the reference for more details.

Profile selection depends on the target and crate being built. By default the `dev` or `test` profiles are used. If the `--release` flag is given, then the `release` or `bench` profiles are used.

Target	Default Profile	`--release` Profile
lib, bin, example	`dev`	`release`

Target	Default Profile	`--release` Profile
test, bench, or any target in "test" or "bench" mode	`test`	`bench`

Dependencies use the `dev`/`release` profiles.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Check if your package (not including dependencies) uses unsafe code:

   ```
   cargo rustc --lib -- -D unsafe-code
   ```

2. Try an experimental flag on the nightly compiler, such as this which prints the size of every type:

   ```
   cargo rustc --lib -- -Z print-type-sizes
   ```

SEE ALSO

cargo(1), cargo-build(1), rustc(1)

cargo rustdoc

NAME

cargo-rustdoc - Build a package's documentation, using specified custom flags

SYNOPSIS

`cargo rustdoc [*OPTIONS*] [-- *ARGS*]`

DESCRIPTION

The specified target for the current package (or package specified by `-p` if provided) will be documented with the specified *ARGS* being passed to the final rustdoc invocation. Dependencies will not be documented as part of this command. Note that rustdoc will still unconditionally receive arguments such as `-L`, `--extern`, and `--crate-type`, and the specified *ARGS* will simply be added to the rustdoc invocation.

See https://doc.rust-lang.org/rustdoc/index.html for documentation on rustdoc flags.

This command requires that only one target is being compiled when additional arguments are provided. If more than one target is available for the current package the filters of `--lib`, `--bin`, etc, must be used to select which target is compiled. To pass flags to all rustdoc processes spawned by Cargo, use the `RUSTDOCFLAGS` environment variable or the `build.rustdocflags` configuration option.

OPTIONS

Documentation Options

--open
: Open the docs in a browser after building them. This will use your default browser unless you define another one in the `BROWSER` environment variable.

Package Selection

By default, the package in the current working directory is selected. The `-p` flag can be used to choose a different package in a workspace.

-p *SPEC*
--package *SPEC*

The package to document. See cargo-pkgid(1) for the SPEC format.

Target Selection

When no target selection options are given, `cargo rustdoc` will document all binary and library targets of the selected package. The binary will be skipped if its name is the same as the lib target. Binaries are skipped if they have `required-features` that are missing.

Passing target selection flags will document only the specified targets.

--lib
Document the package's library.

--bin *NAME*...
Document the specified binary. This flag may be specified multiple times.

--bins
Document all binary targets.

--example *NAME*...
Document the specified example. This flag may be specified multiple times.

--examples
Document all example targets.

--test *NAME*...
Document the specified integration test. This flag may be specified multiple times.

--tests
Document all targets in test mode that have the `test = true` manifest flag set. By default this includes the library and binaries built as unittests, and integration tests. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a unittest, and once as a dependency for binaries, integration tests, etc.). Targets may be enabled or disabled by setting the `test` flag in the manifest settings for the target.

--bench *NAME*...
Document the specified benchmark. This flag may be specified multiple times.

--benches
Document all targets in benchmark mode that have the `bench = true` manifest flag set. By default this includes the library and binaries built as benchmarks, and bench targets. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a benchmark, and once as a dependency for binaries, benchmarks, etc.). Targets may be enabled or disabled by setting the `bench` flag in the manifest settings for the target.

--all-targets

Document all targets. This is equivalent to specifying `--lib --bins --tests --benches --examples`.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
Activate all available features of all selected packages.

--no-default-features
Do not activate the `default` feature of the current directory's package.

Compilation Options

--target *TRIPLE*
Document for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

This may also be specified with the `build.target` config value.

--release
Document optimized artifacts with the `release` profile. See the PROFILES section for details on how this affects profile selection.

Output Options

--target-dir *DIRECTORY*
Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

Display Options

-v
--verbose

Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

--message-format *FMT*
The output format for diagnostic messages. Can be specified multiple times and consists of comma-separated values. Valid values:

- `human` (default): Display in a human-readable text format.
- `short`: Emit shorter, human-readable text messages.
- `json`: Emit JSON messages to stdout. See the reference for more details.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics in in JSON messages printed, but instead Cargo itself should render the JSON diagnostics coming from rustc. Cargo's own JSON diagnostics and others coming from rustc are still emitted.

Manifest Options

--manifest-path *PATH*
Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

Miscellaneous Options

-j *N*
--jobs *N*
: Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

PROFILES

Profiles may be used to configure compiler options such as optimization levels and debug settings. See the reference for more details.

Profile selection depends on the target and crate being built. By default the `dev` or `test` profiles are used. If the `--release` flag is given, then the `release` or `bench` profiles are used.

Target	Default Profile	`--release` Profile
lib, bin, example	`dev`	`release`

Target	Default Profile	`--release` Profile
test, bench, or any target in "test" or "bench" mode	`test`	`bench`

Dependencies use the `dev` / `release` profiles.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Build documentation with custom CSS included from a given file:

```
cargo rustdoc --lib -- --extend-css extra.css
```

SEE ALSO

cargo(1), cargo-doc(1), rustdoc(1)

cargo test

NAME

cargo-test - Execute unit and integration tests of a package

SYNOPSIS

`cargo test [*OPTIONS*] [TESTNAME] [-- *TEST-OPTIONS*]`

DESCRIPTION

Compile and execute unit and integration tests.

The test filtering argument `TESTNAME` and all the arguments following the two dashes (`--`) are passed to the test binaries and thus to *libtest* (rustc's built in unit-test and micro-benchmarking framework). If you're passing arguments to both Cargo and the binary, the ones after `--` go to the binary, the ones before go to Cargo. For details about libtest's arguments see the output of `cargo test -- --help`. As an example, this will run all tests with `foo` in their name on 3 threads in parallel:

```
cargo test foo -- --test-threads 3
```

Tests are built with the `--test` option to `rustc` which creates an executable with a `main` function that automatically runs all functions annotated with the `#[test]` attribute in multiple threads. `#[bench]` annotated functions will also be run with one iteration to verify that they are functional.

The libtest harness may be disabled by setting `harness = false` in the target manifest settings, in which case your code will need to provide its own `main` function to handle running tests.

Documentation tests are also run by default, which is handled by `rustdoc`. It extracts code samples from documentation comments and executes them. See the rustdoc book for more information on writing doc tests.

OPTIONS

Test Options

--no-run
: Compile, but don't run tests.

--no-fail-fast
: Run all tests regardless of failure. Without this flag, Cargo will exit after the first executable fails. The Rust test harness will run all tests within the executable to completion, this flag only applies to the executable as a whole.

Package Selection

By default, when no package selection options are given, the packages selected depend on the selected manifest file (based on the current working directory if `--manifest-path` is not given). If the manifest is the root of a workspace then the workspaces default members are selected, otherwise only the package defined by the manifest will be selected.

The default members of a workspace can be set explicitly with the `workspace.default-members` key in the root manifest. If this is not set, a virtual workspace will include all workspace members (equivalent to passing `--workspace`), and a non-virtual workspace will include only the root crate itself.

-p *SPEC*...
--package *SPEC*...
Test only the specified packages. See cargo-pkgid(1) for the SPEC format. This flag may be specified multiple times.

--workspace
Test all members in the workspace.

--all
Deprecated alias for `--workspace`.

--exclude *SPEC*...
Exclude the specified packages. Must be used in conjunction with the `--workspace` flag. This flag may be specified multiple times.

Target Selection

When no target selection options are given, `cargo test` will build the following targets of the selected packages:

- lib — used to link with binaries, examples, integration tests, and doc tests
- bins (only if integration tests are built and required features are available)
- examples — to ensure they compile
- lib as a unit test
- bins as unit tests
- integration tests
- doc tests for the lib target

The default behavior can be changed by setting the `test` flag for the target in the manifest settings. Setting examples to `test = true` will build and run the example as a test. Setting targets to `test = false` will stop them from being tested by default. Target selection options that take a target by name ignore the `test` flag and will always test the given target.

Doc tests for libraries may be disabled by setting `doctest = false` for the library in the manifest.

Passing target selection flags will test only the specified targets.

--lib
: Test the package's library.

--bin *NAME*...
: Test the specified binary. This flag may be specified multiple times.

--bins
: Test all binary targets.

--example *NAME*...
: Test the specified example. This flag may be specified multiple times.

--examples
: Test all example targets.

--test *NAME*...
: Test the specified integration test. This flag may be specified multiple times.

--tests
: Test all targets in test mode that have the `test = true` manifest flag set. By default this includes the library and binaries built as unittests, and integration tests. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a unittest, and once as a dependency for binaries, integration tests, etc.). Targets may be enabled or disabled by setting the `test` flag in the manifest settings for the target.

--bench *NAME*...
: Test the specified benchmark. This flag may be specified multiple times.

--benches
: Test all targets in benchmark mode that have the `bench = true` manifest flag set. By default this includes the library and binaries built as benchmarks, and bench targets. Be aware that this will also build any required dependencies, so the lib target may be built twice (once as a benchmark, and once as a dependency for binaries, benchmarks, etc.). Targets may be enabled or disabled by setting the `bench` flag in the manifest settings for the target.

--all-targets
: Test all targets. This is equivalent to specifying `--lib --bins --tests --benches --examples`.

--doc
: Test only the library's documentation. This cannot be mixed with other target options.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
: Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
: Activate all available features of all selected packages.

--no-default-features
: Do not activate the `default` feature of the current directory's package.

Compilation Options

--target *TRIPLE*
: Test for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

  This may also be specified with the `build.target` config value.

--release
: Test optimized artifacts with the `release` profile. See the PROFILES section for details on how this affects profile selection.

Output Options

--target-dir *DIRECTORY*
: Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

Display Options

By default the Rust test harness hides output from test execution to keep results readable. Test output can be recovered (e.g., for debugging) by passing `--nocapture` to the test binaries:

```
cargo test -- --nocapture
```

-v
--verbose

Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

--message-format *FMT*
The output format for diagnostic messages. Can be specified multiple times and consists of comma-separated values. Valid values:

- `human` (default): Display in a human-readable text format.
- `short`: Emit shorter, human-readable text messages.
- `json`: Emit JSON messages to stdout. See the reference for more details.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics in in JSON messages printed, but instead Cargo itself should render the JSON diagnostics coming from rustc. Cargo's own JSON diagnostics and others coming from rustc are still emitted.

Manifest Options

--manifest-path *PATH*
Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

Miscellaneous Options

The `--jobs` argument affects the building of the test executable but does not affect how many threads are used when running the tests. The Rust test harness includes an option to control the number of threads used:

```
cargo test -j 2 -- --test-threads=2
```

-j *N*
--jobs *N*
: Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

PROFILES

Profiles may be used to configure compiler options such as optimization levels and debug settings. See the reference for more details.

Profile selection depends on the target and crate being built. By default the `dev` or `test` profiles are used. If the `--release` flag is given, then the `release` or `bench` profiles are used.

Target	Default Profile	`--release` Profile
lib, bin, example	`dev`	`release`
test, bench, or any target in "test" or "bench" mode	`test`	`bench`

Dependencies use the `dev` / `release` profiles.

Unit tests are separate executable artifacts which use the `test` / `bench` profiles. Example targets are built the same as with `cargo build` (using the `dev` / `release` profiles) unless you are building them with the test harness (by setting `test = true` in the manifest or using the `--example` flag) in which case they use the `test` / `bench` profiles. Library targets are built with the `dev` / `release` profiles when linked to an integration test, binary, or doctest.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Execute all the unit and integration tests of the current package:

   ```
   cargo test
   ```

2. Run only a specific test within a specific integration test:

   ```
   cargo test --test int_test_name -- modname::test_name
   ```

SEE ALSO

cargo(1), cargo-bench(1)

Manifest Commands

cargo generate-lockfile

NAME

cargo-generate-lockfile - Generate the lockfile for a package

SYNOPSIS

`cargo generate-lockfile [OPTIONS]`

DESCRIPTION

This command will create the `Cargo.lock` lockfile for the current package or workspace. If the lockfile already exists, it will be rebuilt if there are any manifest changes or dependency updates.

See also cargo-update(1) which is also capable of creating a `Cargo.lock` lockfile and has more options for controlling update behavior.

OPTIONS

Display Options

-v
--verbose
: Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
: No output printed to stdout.

--color *WHEN*
: Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Manifest Options

--manifest-path *PATH*
: Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Create or update the lockfile for the current package or workspace:

   ```
   cargo generate-lockfile
   ```

SEE ALSO

cargo(1), cargo-update(1)

cargo locate-project

NAME

cargo-locate-project - Print a JSON representation of a Cargo.toml file's location

SYNOPSIS

`cargo locate-project [*OPTIONS*]`

DESCRIPTION

This command will print a JSON object to stdout with the full path to the `Cargo.toml` manifest.

See also cargo-metadata(1) which is capable of returning the path to a workspace root.

OPTIONS

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Manifest Options

--manifest-path *PATH*
Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

Common Options

-h
--help
Prints help information.

-Z *FLAG*...
Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Display the path to the manifest based on the current directory:

   ```
   cargo locate-project
   ```

SEE ALSO

cargo(1), cargo-metadata(1)

cargo metadata

NAME

cargo-metadata - Machine-readable metadata about the current package

SYNOPSIS

`cargo metadata [*OPTIONS*]`

DESCRIPTION

Output the resolved dependencies of a package, the concrete used versions including overrides, in JSON to stdout.

It is recommended to include the `--format-version` flag to future-proof your code to ensure the output is in the format you are expecting.

See the cargo_metadata crate for a Rust API for reading the metadata.

OUTPUT FORMAT

The output has the following format:

```
{
    /* Array of all packages in the workspace.
       It also includes all feature-enabled dependencies unless --no-deps is
used.
    */
    "packages": [
        {
            /* The name of the package. */
            "name": "my-package",
            /* The version of the package. */
            "version": "0.1.0",
            /* The Package ID, a unique identifier for referring to the
package. */
            "id": "my-package 0.1.0 (path+file:///path/to/my-package)",
            /* The license value from the manifest, or null. */
            "license": "MIT/Apache-2.0",
            /* The license-file value from the manifest, or null. */
            "license_file": "LICENSE",
            /* The description value from the manifest, or null. */
            "description": "Package description.",
            /* The source ID of the package. This represents where
               a package is retrieved from.
               This is null for path dependencies and workspace members.
               For other dependencies, it is a string with the format:
               - "registry+URL" for registry-based dependencies.
                 Example: "registry+https://github.com/rust-lang/crates.io-
index"
               - "git+URL" for git-based dependencies.
                 Example: "git+https://github.com/rust-lang/cargo?
rev=5e85ba14aaa20f8133863373404cb0af69eeef2c#5e85ba14aaa20f8133863373404cb0af6

            */
            "source": null,
            /* Array of dependencies declared in the package's manifest. */
            "dependencies": [
                {
                    /* The name of the dependency. */
                    "name": "bitflags",
                    /* The source ID of the dependency. May be null, see
                       description for the package source.
                    */
                    "source": "registry+https://github.com/rust-
lang/crates.io-index",
                    /* The version requirement for the dependency.
                       Dependencies without a version requirement have a
value of "*".
                    */
                    "req": "^1.0",
                    /* The dependency kind.
                       "dev", "build", or null for a normal dependency.
                    */
                    "kind": null,
                    /* If the dependency is renamed, this is the new name
for
                       the dependency as a string.  null if it is not
renamed.
                    */
```

```
                    "rename": null,
                    /* Boolean of whether or not this is an optional
dependency. */
                    "optional": false,
                    /* Boolean of whether or not default features are
enabled. */
                    "uses_default_features": true,
                    /* Array of features enabled. */
                    "features": [],
                    /* The target platform for the dependency.
                       null if not a target dependency.
                    */
                    "target": "cfg(windows)",
                    /* A string of the URL of the registry this dependency
is from.
                       If not specified or null, the dependency is from the
default
                       registry (crates.io).
                    */
                    "registry": null
                }
            ],
            /* Array of Cargo targets. */
            "targets": [
                {
                    /* Array of target kinds.
                       - lib targets list the `crate-type` values from the
                         manifest such as "lib", "rlib", "dylib",
                         "proc-macro", etc. (default ["lib"])
                       - binary is ["bin"]
                       - example is ["example"]
                       - integration test is ["test"]
                       - benchmark is ["bench"]
                       - build script is ["custom-build"]
                    */
                    "kind": [
                        "bin"
                    ],
                    /* Array of crate types.
                       - lib and example libraries list the `crate-type`
values
                         from the manifest such as "lib", "rlib", "dylib",
                         "proc-macro", etc. (default ["lib"])
                       - all other target kinds are ["bin"]
                    */
                    "crate_types": [
                        "bin"
                    ],
                    /* The name of the target. */
                    "name": "my-package",
                    /* Absolute path to the root source file of the target.
*/
                    "src_path": "/path/to/my-package/src/main.rs",
                    /* The Rust edition of the target.
                       Defaults to the package edition.
                    */
                    "edition": "2018",
                    /* Array of required features.
```

```
                       This property is not included if no required features
are set.
                    */
                    "required-features": ["feat1"],
                    /* Whether or not this target has doc tests enabled, and
                       the target is compatible with doc testing.
                    */
                    "doctest": false
                }
            ],
            /* Set of features defined for the package.
               Each feature maps to an array of features or dependencies it
               enables.
            */
            "features": {
                "default": [
                    "feat1"
                ],
                "feat1": [],
                "feat2": []
            },
            /* Absolute path to this package's manifest. */
            "manifest_path": "/path/to/my-package/Cargo.toml",
            /* Package metadata.
               This is null if no metadata is specified.
            */
            "metadata": {
                "docs": {
                    "rs": {
                        "all-features": true
                    }
                }
            },
            /* List of registries to which this package may be published.
               Publishing is unrestricted if null, and forbidden if an empty
array. */
            "publish": [
                "crates-io"
            ],
            /* Array of authors from the manifest.
               Empty array if no authors specified.
            */
            "authors": [
                "Jane Doe <user@example.com>"
            ],
            /* Array of categories from the manifest. */
            "categories": [
                "command-line-utilities"
            ],
            /* Array of keywords from the manifest. */
            "keywords": [
                "cli"
            ],
            /* The readme value from the manifest or null if not specified.
*/
            "readme": "README.md",
            /* The repository value from the manifest or null if not
specified. */
```

```
            "repository": "https://github.com/rust-lang/cargo",
            /* The default edition of the package.
               Note that individual targets may have different editions.
            */
            "edition": "2018",
            /* Optional string that is the name of a native library the
package
               is linking to.
            */
            "links": null,
        }
    ],
    /* Array of members of the workspace.
       Each entry is the Package ID for the package.
    */
    "workspace_members": [
        "my-package 0.1.0 (path+file:///path/to/my-package)",
    ],
    // The resolved dependency graph, with the concrete versions and
features
    // selected. The set depends on the enabled features.
    //
    // This is null if --no-deps is specified.
    //
    // By default, this includes all dependencies for all target platforms.
    // The `--filter-platform` flag may be used to narrow to a specific
    // target triple.
    "resolve": {
        /* Array of nodes within the dependency graph.
           Each node is a package.
        */
        "nodes": [
            {
                /* The Package ID of this node. */
                "id": "my-package 0.1.0 (path+file:///path/to/my-package)",
                /* The dependencies of this package, an array of Package
IDs. */
                "dependencies": [
                    "bitflags 1.0.4 (registry+https://github.com/rust-
lang/crates.io-index)"
                ],
                /* The dependencies of this package. This is an alternative
to
                   "dependencies" which contains additional information. In
                   particular, this handles renamed dependencies.
                */
                "deps": [
                    {
                        /* The name of the dependency's library target.
                           If this is a renamed dependency, this is the new
                           name.
                        */
                        "name": "bitflags",
                        /* The Package ID of the dependency. */
                        "pkg": "bitflags 1.0.4
(registry+https://github.com/rust-lang/crates.io-index)",
                        /* Array of dependency kinds. Added in Cargo 1.40.
*/
```

```
                        "dep_kinds": [
                            {
                                /* The dependency kind.
                                   "dev", "build", or null for a normal
dependency.
                                */
                                "kind": null,
                                /* The target platform for the dependency.
                                   null if not a target dependency.
                                */
                                "target": "cfg(windows)"
                            }
                        ]
                    }
                ],
                /* Array of features enabled on this package. */
                "features": [
                    "default"
                ]
            }
        ],
        /* The root package of the workspace.
           This is null if this is a virtual workspace. Otherwise it is
           the Package ID of the root package.
        */
        "root": "my-package 0.1.0 (path+file:///path/to/my-package)"
    },
    /* The absolute path to the build directory where Cargo places its
output. */
    "target_directory": "/path/to/my-package/target",
    /* The version of the schema for this metadata structure.
       This will be changed if incompatible changes are ever made.
    */
    "version": 1,
    /* The absolute path to the root of the workspace. */
    "workspace_root": "/path/to/my-package"
}
```

OPTIONS

Output Options

--no-deps
: Output information only about the workspace members and don't fetch dependencies.

--format-version *VERSION*
: Specify the version of the output format to use. Currently `1` is the only possible value.

--filter-platform *TRIPLE*

This filters the `resolve` output to only include dependencies for the given target triple. Without this flag, the resolve includes all targets.

Note that the dependencies listed in the "packages" array still includes all dependencies. Each package definition is intended to be an unaltered reproduction of the information within `Cargo.toml`.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
Activate all available features of all selected packages.

--no-default-features
Do not activate the `default` feature of the current directory's package.

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Manifest Options

--manifest-path *PATH*
: Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

  These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Output JSON about the current package:

```
cargo metadata --format-version=1
```

SEE ALSO

cargo(1)

cargo pkgid

NAME

cargo-pkgid - Print a fully qualified package specification

SYNOPSIS

`cargo pkgid [*OPTIONS*] [*SPEC*]`

DESCRIPTION

Given a *SPEC* argument, print out the fully qualified package ID specifier for a package or dependency in the current workspace. This command will generate an error if *SPEC* is ambiguous as to which package it refers to in the dependency graph. If no *SPEC* is given, then the specifier for the local package is printed.

This command requires that a lockfile is available and dependencies have been fetched.

A package specifier consists of a name, version, and source URL. You are allowed to use partial specifiers to succinctly match a specific package as long as it matches only one package. The format of a *SPEC* can be one of the following:

Table 1. SPEC Query Format

SPEC Structure	Example SPEC
NAME	`bitflags`
NAME `:` *VERSION*	`bitflags:1.0.4`
URL	`https://github.com/rust-lang/cargo`
URL `#` *VERSION*	`https://github.com/rust-lang/cargo#0.33.0`
URL `#` *NAME*	`https://github.com/rust-lang/crates.io-index#bitflags`
URL `#` *NAME* `:` *VERSION*	`https://github.com/rust-lang/cargo#crates-io:0.21.0`

OPTIONS

Package Selection

-p *SPEC*
--package *SPEC*
Get the package ID for the given package instead of the current package.

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Manifest Options

--manifest-path *PATH*
Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

May also be specified with the `net.offline` config value.

Common Options

-h
--help
Prints help information.

-Z *FLAG*...
Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Retrieve package specification for `foo` package:

   ```
   cargo pkgid foo
   ```

2. Retrieve package specification for version 1.0.0 of `foo`:

   ```
   cargo pkgid foo:1.0.0
   ```

3. Retrieve package specification for `foo` from crates.io:

   ```
   cargo pkgid https://github.com/rust-lang/crates.io-index#foo
   ```

SEE ALSO

cargo(1), cargo-generate-lockfile(1), cargo-metadata(1)

cargo update

NAME

cargo-update - Update dependencies as recorded in the local lock file

SYNOPSIS

`cargo update [*OPTIONS*]`

DESCRIPTION

This command will update dependencies in the `Cargo.lock` file to the latest version. It requires that the `Cargo.lock` file already exists as generated by commands such as cargo-build(1) or cargo-generate-lockfile(1).

OPTIONS

Update Options

-p *SPEC*...
--package *SPEC*...
: Update only the specified packages. This flag may be specified multiple times. See cargo-pkgid(1) for the SPEC format.

  If packages are specified with the `-p` flag, then a conservative update of the lockfile will be performed. This means that only the dependency specified by SPEC will be updated. Its transitive dependencies will be updated only if SPEC cannot be updated without updating dependencies. All other dependencies will remain locked at their currently recorded versions.

  If `-p` is not specified, all dependencies are updated.

--aggressive
: When used with `-p`, dependencies of *SPEC* are forced to update as well. Cannot be used with `--precise`.

--precise *PRECISE*
: When used with `-p`, allows you to specify a specific version number to set the package to. If the package comes from a git repository, this can be a git revision (such as a SHA hash or tag).

--dry-run
: Displays what would be updated, but doesn't actually write the lockfile.

Display Options

-v
--verbose
: Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q

--quiet
: No output printed to stdout.

--color *WHEN*
: Control when colored output is used. Valid values:

  - `auto` (default): Automatically detect if color support is available on the terminal.
  - `always`: Always display colors.
  - `never`: Never display colors.

  May also be specified with the `term.color` config value.

Manifest Options

--manifest-path *PATH*
: Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

  These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...

Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Update all dependencies in the lockfile:

   ```
   cargo update
   ```

2. Update only specific dependencies:

   ```
   cargo update -p foo -p bar
   ```

3. Set a specific dependency to a specific version:

   ```
   cargo update -p foo --precise 1.2.3
   ```

SEE ALSO

cargo(1), cargo-generate-lockfile(1)

cargo vendor

NAME

cargo-vendor - Vendor all dependencies locally

SYNOPSIS

`cargo vendor [OPTIONS] [PATH]`

DESCRIPTION

This cargo subcommand will vendor all crates.io and git dependencies for a project into the specified directory at `<path>`. After this command completes the vendor directory specified by `<path>` will contain all remote sources from dependencies specified. Additional manifests beyond the default one can be specified with the `-s` option.

The `cargo vendor` command will also print out the configuration necessary to use the vendored sources, which you will need to add to `.cargo/config`.

OPTIONS

Owner Options

-s *MANIFEST*
--sync *MANIFEST*
: Specify extra `Cargo.toml` manifests to workspaces which should also be vendored and synced to the output.

--no-delete
: Don't delete the "vendor" directory when vendoring, but rather keep all existing contents of the vendor directory

--respect-source-config
: Instead of ignoring `[source]` configuration by default in `.cargo/config` read it and use it when downloading crates from crates.io, for example

Manifest Options

--manifest-path *PATH*
: Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

Display Options

-v

--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Common Options

-h
--help
Prints help information.

-Z *FLAG*...
Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

--frozen
--locked
Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

May also be specified with the `net.offline` config value.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Vendor all dependencies into a local "vendor" folder

   ```
   cargo vendor
   ```

2. Vendor all dependencies into a local "third-party/vendor" folder

   ```
   cargo vendor third-party/vendor
   ```

3. Vendor the current workspace as well as another to "vendor"

   ```
   cargo vendor -s ../path/to/Cargo.toml
   ```

SEE ALSO

cargo(1)

cargo verify-project

NAME

cargo-verify-project - Check correctness of crate manifest

SYNOPSIS

`cargo verify-project [OPTIONS]`

DESCRIPTION

This command will parse the local manifest and check its validity. It emits a JSON object with the result. A successful validation will display:

```
{"success":"true"}
```

An invalid workspace will display:

```
{"invalid":"human-readable error message"}
```

OPTIONS

Display Options

-v
--verbose
: Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
: No output printed to stdout.

--color *WHEN*
: Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Manifest Options

--manifest-path *PATH*
: Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

  These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: The workspace is OK.

1
: The workspace is invalid.

EXAMPLES

1. Check the current workspace for errors:

    ```
    cargo verify-project
    ```

SEE ALSO

cargo(1), cargo-package(1)

Package Commands

cargo init

NAME

cargo-init - Create a new Cargo package in an existing directory

SYNOPSIS

`cargo init [OPTIONS] [PATH]`

DESCRIPTION

This command will create a new Cargo manifest in the current directory. Give a path as an argument to create in the given directory.

If there are typically-named Rust source files already in the directory, those will be used. If not, then a sample `src/main.rs` file will be created, or `src/lib.rs` if `--lib` is passed.

If the directory is not already in a VCS repository, then a new repository is created (see `--vcs` below).

The "authors" field in the manifest is determined from the environment or configuration settings. A name is required and is determined from (first match wins):

- `cargo-new.name` Cargo config value

- `CARGO_NAME` environment variable
- `GIT_AUTHOR_NAME` environment variable
- `GIT_COMMITTER_NAME` environment variable
- `user.name` git configuration value
- `USER` environment variable
- `USERNAME` environment variable
- `NAME` environment variable

The email address is optional and is determined from:

- `cargo-new.email` Cargo config value
- `CARGO_EMAIL` environment variable
- `GIT_AUTHOR_EMAIL` environment variable
- `GIT_COMMITTER_EMAIL` environment variable
- `user.email` git configuration value
- `EMAIL` environment variable

See the reference for more information about configuration files.

See cargo-new(1) for a similar command which will create a new package in a new directory.

OPTIONS

Init Options

--bin
: Create a package with a binary target (`src/main.rs`). This is the default behavior.

--lib
: Create a package with a library target (`src/lib.rs`).

--edition *EDITION*
: Specify the Rust edition to use. Default is 2018. Possible values: 2015, 2018

--name *NAME*
: Set the package name. Defaults to the directory name.

--vcs *VCS*
: Initialize a new VCS repository for the given version control system (git, hg, pijul, or fossil) or do not initialize any version control at all (none). If not specified, defaults to `git` or the configuration value `cargo-new.vcs`, or `none` if already inside a VCS repository.

--registry *REGISTRY*
: This sets the `publish` field in `Cargo.toml` to the given registry name which will restrict publishing only to that registry.

Registry names are defined in Cargo config files. If not specified, the default registry defined by the `registry.default` config key is used. If the default registry is not set and `--registry` is not used, the `publish` field will not be set which means that publishing will not be restricted.

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Common Options

-h
--help
Prints help information.

-Z *FLAG*...
Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0

Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Create a binary Cargo package in the current directory:

```
cargo init
```

SEE ALSO

cargo(1), cargo-new(1)

cargo install

NAME

cargo-install - Build and install a Rust binary

SYNOPSIS

`cargo install [OPTIONS] CRATE…`
`cargo install [OPTIONS] --path PATH`
`cargo install [OPTIONS] --git URL [CRATE…]`
`cargo install [OPTIONS] --list`

DESCRIPTION

This command manages Cargo's local set of installed binary crates. Only packages which have executable `[[bin]]` or `[[example]]` targets can be installed, and all executables are installed into the installation root's `bin` folder.

The installation root is determined, in order of precedence:

- `--root` option
- `CARGO_INSTALL_ROOT` environment variable
- `install.root` Cargo config value
- `CARGO_HOME` environment variable
- `$HOME/.cargo`

There are multiple sources from which a crate can be installed. The default location is crates.io but the `--git`, `--path`, and `--registry` flags can change this source. If the source contains more than one package (such as crates.io or a git repository with multiple crates) the *CRATE* argument is required to indicate which crate should be installed.

Crates from crates.io can optionally specify the version they wish to install via the `--version` flags, and similarly packages from git repositories can optionally specify the branch, tag, or revision that should be installed. If a crate has multiple binaries, the `--bin` argument can selectively install only one of them, and if you'd rather install examples the `--example` argument can be used as well.

If the package is already installed, Cargo will reinstall it if the installed version does not appear to be up-to-date. If any of the following values change, then Cargo will reinstall the package:

- The package version and source.
- The set of binary names installed.
- The chosen features.
- The release mode (`--debug`).
- The target (`--target`).

Installing with `--path` will always build and install, unless there are conflicting binaries from another package. The `--force` flag may be used to force Cargo to always reinstall the package.

If the source is crates.io or `--git` then by default the crate will be built in a temporary target directory. To avoid this, the target directory can be specified by setting the `CARGO_TARGET_DIR` environment variable to a relative path. In particular, this can be useful for caching build artifacts on continuous integration systems.

By default, the `Cargo.lock` file that is included with the package will be ignored. This means that Cargo will recompute which versions of dependencies to use, possibly using newer versions that have been released since the package was published. The `--locked` flag can be used to force Cargo to use the packaged `Cargo.lock` file if it is available. This may be useful for ensuring reproducible builds, to use the exact same set of dependencies that were available when the package was published. It may also be useful if a newer version of a dependency is published that no longer builds on your system, or has other problems. The downside to using `--locked` is that you will not receive any fixes or updates to any dependency. Note that Cargo did not start publishing `Cargo.lock` files until version 1.37, which means packages published with prior versions will not have a `Cargo.lock` file available.

OPTIONS

Install Options

--vers *VERSION*
--version *VERSION*
: Specify a version to install. This may be a version requirement, like `~1.2`, to have Cargo select the newest version from the given requirement. If the version does not have a requirement operator (such as `^` or `~`), then it must be in the form *MAJOR.MINOR.PATCH*, and will install exactly that version; it is **not** treated as a caret requirement like Cargo dependencies are.

--git *URL*
: Git URL to install the specified crate from.

--branch *BRANCH*
: Branch to use when installing from git.

--tag *TAG*
: Tag to use when installing from git.

--rev *SHA*
: Specific commit to use when installing from git.

--path *PATH*
: Filesystem path to local crate to install.

--list
: List all installed packages and their versions.

-f
--force
: Force overwriting existing crates or binaries. This can be used if a package has installed a binary with the same name as another package. This is also useful if something has changed on the system that you want to rebuild with, such as a newer version of `rustc`.

--no-track
: By default, Cargo keeps track of the installed packages with a metadata file stored in the installation root directory. This flag tells Cargo not to use or create that file. With this flag, Cargo will refuse to overwrite any existing files unless the `--force` flag is used. This also disables Cargo's ability to protect against multiple concurrent invocations of Cargo installing at the same time.

--bin *NAME*...
: Install only the specified binary.

--bins

Install all binaries.

--example *NAME*...
Install only the specified example.

--examples
Install all examples.

--root *DIR*
Directory to install packages into.

--registry *REGISTRY*
Name of the registry to use. Registry names are defined in Cargo config files. If not specified, the default registry is used, which is defined by the `registry.default` config key which defaults to `crates-io`.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
Activate all available features of all selected packages.

--no-default-features
Do not activate the `default` feature of the current directory's package.

Compilation Options

--target *TRIPLE*
Install for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

This may also be specified with the `build.target` config value.

--debug
Build with the `dev` profile instead the `release` profile.

Manifest Options

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

  These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
: Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

  Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

  May also be specified with the `net.offline` config value.

Miscellaneous Options

-j *N*
--jobs *N*
: Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

Display Options

-v
--verbose
: Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
: No output printed to stdout.

--color *WHEN*
: Control when colored output is used. Valid values:

  - `auto` (default): Automatically detect if color support is available on the terminal.
  - `always`: Always display colors.
  - `never`: Never display colors.

May also be specified with the `term.color` config value.

Common Options

-h
--help
Prints help information.

-Z *FLAG*...
Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
Cargo succeeded.

101
Cargo failed to complete.

EXAMPLES

1. Install or upgrade a package from crates.io:

   ```
   cargo install ripgrep
   ```

2. Install or reinstall the package in the current directory:

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

3. View the list of installed packages:

   ```
   cargo install --list
   ```

SEE ALSO

cargo(1), cargo-uninstall(1), cargo-search(1), cargo-publish(1)

cargo new

NAME

cargo-new - Create a new Cargo package

SYNOPSIS

`cargo new [OPTIONS] PATH`

DESCRIPTION

This command will create a new Cargo package in the given directory. This includes a simple template with a `Cargo.toml` manifest, sample source file, and a VCS ignore file. If the directory is not already in a VCS repository, then a new repository is created (see `--vcs` below).

The "authors" field in the manifest is determined from the environment or configuration settings. A name is required and is determined from (first match wins):

- `cargo-new.name` Cargo config value
- `CARGO_NAME` environment variable
- `GIT_AUTHOR_NAME` environment variable
- `GIT_COMMITTER_NAME` environment variable
- `user.name` git configuration value
- `USER` environment variable
- `USERNAME` environment variable
- `NAME` environment variable

The email address is optional and is determined from:

- `cargo-new.email` Cargo config value
- `CARGO_EMAIL` environment variable
- `GIT_AUTHOR_EMAIL` environment variable
- `GIT_COMMITTER_EMAIL` environment variable
- `user.email` git configuration value
- `EMAIL` environment variable

See the reference for more information about configuration files.

See cargo-init(1) for a similar command which will create a new manifest in an existing directory.

OPTIONS

New Options

--bin
: Create a package with a binary target (`src/main.rs`). This is the default behavior.

--lib
: Create a package with a library target (`src/lib.rs`).

--edition *EDITION*
: Specify the Rust edition to use. Default is 2018. Possible values: 2015, 2018

--name *NAME*
: Set the package name. Defaults to the directory name.

--vcs *VCS*
: Initialize a new VCS repository for the given version control system (git, hg, pijul, or fossil) or do not initialize any version control at all (none). If not specified, defaults to `git` or the configuration value `cargo-new.vcs`, or `none` if already inside a VCS repository.

--registry *REGISTRY*
: This sets the `publish` field in `Cargo.toml` to the given registry name which will restrict publishing only to that registry.

  Registry names are defined in Cargo config files. If not specified, the default registry defined by the `registry.default` config key is used. If the default registry is not set and `--registry` is not used, the `publish` field will not be set which means that publishing will not be restricted.

Display Options

-v
--verbose
: Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet

No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Common Options

-h
--help
Prints help information.

-Z *FLAG*...
Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
Cargo succeeded.

101
Cargo failed to complete.

EXAMPLES

1. Create a binary Cargo package in the given directory:

```
cargo new foo
```

SEE ALSO

cargo(1), cargo-init(1)

cargo search

NAME

cargo-search - Search packages in crates.io

SYNOPSIS

`cargo search [OPTIONS] [QUERY…]`

DESCRIPTION

This performs a textual search for crates on https://crates.io. The matching crates will be displayed along with their description in TOML format suitable for copying into a `Cargo.toml` manifest.

OPTIONS

Search Options

--limit *LIMIT*
: Limit the number of results (default: 10, max: 100).

--index *INDEX*
: The URL of the registry index to use.

--registry *REGISTRY*
: Name of the registry to use. Registry names are defined in Cargo config files. If not specified, the default registry is used, which is defined by the `registry.default` config key which defaults to `crates-io`.

Display Options

-v

--verbose
: Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
: No output printed to stdout.

--color *WHEN*
: Control when colored output is used. Valid values:

  - `auto` (default): Automatically detect if color support is available on the terminal.
  - `always`: Always display colors.
  - `never`: Never display colors.

  May also be specified with the `term.color` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Search for a package from crates.io:

    ```
    cargo search serde
    ```

SEE ALSO

cargo(1), cargo-install(1), cargo-publish(1)

cargo uninstall

NAME

cargo-uninstall - Remove a Rust binary

SYNOPSIS

`cargo uninstall [*OPTIONS*] [*SPEC…*]`

DESCRIPTION

This command removes a package installed with cargo-install(1). The *SPEC* argument is a package ID specification of the package to remove (see cargo-pkgid(1)).

By default all binaries are removed for a crate but the `--bin` and `--example` flags can be used to only remove particular binaries.

The installation root is determined, in order of precedence:

- `--root` option
- `CARGO_INSTALL_ROOT` environment variable
- `install.root` Cargo config value
- `CARGO_HOME` environment variable
- `$HOME/.cargo`

OPTIONS

Install Options

-p
--package *SPEC*...
Package to uninstall.

--bin *NAME*...
Only uninstall the binary *NAME*.

--root *DIR*
Directory to uninstall packages from.

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Common Options

-h
--help
Prints help information.

-Z *FLAG*...
Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Uninstall a previously installed package.

```
cargo uninstall ripgrep
```

SEE ALSO

cargo(1), cargo-install(1)

Publishing Commands

cargo login

NAME

cargo-login - Save an API token from the registry locally

SYNOPSIS

`cargo login [OPTIONS] [TOKEN]`

DESCRIPTION

This command will save the API token to disk so that commands that require authentication, such as cargo-publish(1), will be automatically authenticated. The token is saved in `$CARGO_HOME/credentials`. `CARGO_HOME` defaults to `.cargo` in your home directory.

If the *TOKEN* argument is not specified, it will be read from stdin.

The API token for crates.io may be retrieved from https://crates.io/me.

Take care to keep the token secret, it should not be shared with anyone else.

OPTIONS

Login Options

--registry *REGISTRY*
: Name of the registry to use. Registry names are defined in Cargo config files. If not specified, the default registry is used, which is defined by the `registry.default` config key which defaults to `crates-io`.

Display Options

-v
--verbose
: Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
: No output printed to stdout.

--color *WHEN*
: Control when colored output is used. Valid values:

  - `auto` (default): Automatically detect if color support is available on the terminal.
  - `always`: Always display colors.
  - `never`: Never display colors.

  May also be specified with the `term.color` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...

Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Save the API token to disk:

   ```
   cargo login
   ```

SEE ALSO

cargo(1), cargo-publish(1)

cargo owner

NAME

cargo-owner - Manage the owners of a crate on the registry

SYNOPSIS

```
cargo owner [OPTIONS] --add LOGIN [CRATE]
cargo owner [OPTIONS] --remove LOGIN [CRATE]
```

```
cargo owner [OPTIONS] --list [CRATE]
```

DESCRIPTION

This command will modify the owners for a crate on the registry. Owners of a crate can upload new versions and yank old versions. Non-team owners can also modify the set of owners, so take care!

This command requires you to be authenticated with either the `--token` option or using cargo-login(1).

If the crate name is not specified, it will use the package name from the current directory.

See the reference for more information about owners and publishing.

OPTIONS

Owner Options

-a
--add *LOGIN*...
Invite the given user or team as an owner.

-r
--remove *LOGIN*...
Remove the given user or team as an owner.

-l
--list
List owners of a crate.

--token *TOKEN*
API token to use when authenticating. This overrides the token stored in the credentials file (which is created by cargo-login(1)).

Cargo config environment variables can be used to override the tokens stored in the credentials file. The token for crates.io may be specified with the `CARGO_REGISTRY_TOKEN` environment variable. Tokens for other registries may be specified with environment variables of the form `CARGO_REGISTRIES_NAME_TOKEN` where `NAME` is the name of the registry in all capital letters.

--index *INDEX*
The URL of the registry index to use.

--registry *REGISTRY*
: Name of the registry to use. Registry names are defined in Cargo config files. If not specified, the default registry is used, which is defined by the `registry.default` config key which defaults to `crates-io`.

Display Options

-v
--verbose
: Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
: No output printed to stdout.

--color *WHEN*
: Control when colored output is used. Valid values:

  - `auto` (default): Automatically detect if color support is available on the terminal.
  - `always`: Always display colors.
  - `never`: Never display colors.

  May also be specified with the `term.color` config value.

Common Options

-h
--help
: Prints help information.

-Z *FLAG*...
: Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0

Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. List owners of a package:

```
cargo owner --list foo
```

2. Invite an owner to a package:

```
cargo owner --add username foo
```

3. Remove an owner from a package:

```
cargo owner --remove username foo
```

SEE ALSO

cargo(1), cargo-login(1), cargo-publish(1)

cargo package

NAME

cargo-package - Assemble the local package into a distributable tarball

SYNOPSIS

`cargo package [OPTIONS]`

DESCRIPTION

This command will create a distributable, compressed `.crate` file with the source code of the package in the current directory. The resulting file will be stored in the `target/package` directory. This performs the following steps:

1. Load and check the current workspace, performing some basic checks.
    - Path dependencies are not allowed unless they have a version key. Cargo will ignore the path key for dependencies in published packages. `dev-dependencies` do not have this restriction.
2. Create the compressed `.crate` file.
    - The original `Cargo.toml` file is rewritten and normalized.
    - `[patch]`, `[replace]`, and `[workspace]` sections are removed from the manifest.
    - `Cargo.lock` is automatically included if the package contains an executable binary or example target. cargo-install(1) will use the packaged lock file if the `--locked` flag is used.
    - A `.cargo_vcs_info.json` file is included that contains information about the current VCS checkout hash if available (not included with `--allow-dirty`).
3. Extract the `.crate` file and build it to verify it can build.
4. Check that build scripts did not modify any source files.

The list of files included can be controlled with the `include` and `exclude` fields in the manifest.

See the reference for more details about packaging and publishing.

OPTIONS

Package Options

-l
--list
: Print files included in a package without making one.

--no-verify
: Don't verify the contents by building them.

--no-metadata
: Ignore warnings about a lack of human-usable metadata (such as the description or the license).

--allow-dirty
: Allow working directories with uncommitted VCS changes to be packaged.

Compilation Options

--target *TRIPLE*
: Package for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print`

`target-list` for a list of supported targets.

This may also be specified with the `build.target` config value.

--target-dir *DIRECTORY*
Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
Activate all available features of all selected packages.

--no-default-features
Do not activate the `default` feature of the current directory's package.

Manifest Options

--manifest-path *PATH*
Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline
Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

May also be specified with the `net.offline` config value.

Miscellaneous Options

-j *N*
--jobs *N*
Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Common Options

-h
--help
Prints help information.

-Z *FLAG*...
Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Create a compressed `.crate` file of the current package:

   ```
   cargo package
   ```

SEE ALSO

cargo(1), cargo-publish(1)

cargo publish

NAME

cargo-publish - Upload a package to the registry

SYNOPSIS

`cargo publish [*OPTIONS*]`

DESCRIPTION

This command will create a distributable, compressed `.crate` file with the source code of the package in the current directory and upload it to a registry. The default registry is https://crates.io. This performs the following steps:

1. Performs a few checks, including:
    - Checks the `package.publish` key in the manifest for restrictions on which registries you are allowed to publish to.
2. Create a `.crate` file by following the steps in cargo-package(1).
3. Upload the crate to the registry. Note that the server will perform additional checks on the crate.

This command requires you to be authenticated with either the `--token` option or using cargo-login(1).

See the reference for more details about packaging and publishing.

OPTIONS

Publish Options

--dry-run
: Perform all checks without uploading.

--token *TOKEN*
: API token to use when authenticating. This overrides the token stored in the credentials file (which is created by cargo-login(1)).

  Cargo config environment variables can be used to override the tokens stored in the credentials file. The token for crates.io may be specified with the `CARGO_REGISTRY_TOKEN` environment variable. Tokens for other registries may be specified with environment variables of the form `CARGO_REGISTRIES_NAME_TOKEN` where `NAME` is the name of the registry in all capital letters.

--no-verify
: Don't verify the contents by building them.

--allow-dirty
: Allow working directories with uncommitted VCS changes to be packaged.

--index *INDEX*
: The URL of the registry index to use.

--registry *REGISTRY*
: Name of the registry to use. Registry names are defined in Cargo config files. If not specified, the default registry is used, which is defined by the `registry.default` config key which defaults to `crates-io`.

Compilation Options

--target *TRIPLE*
: Publish for the given architecture. The default is the host architecture. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets.

  This may also be specified with the `build.target` config value.

--target-dir *DIRECTORY*
: Directory for all generated artifacts and intermediate files. May also be specified with the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config value. Defaults to `target` in the root of the workspace.

Feature Selection

When no feature options are given, the `default` feature is activated for every selected package.

--features *FEATURES*
: Space or comma separated list of features to activate. These features only apply to the current directory's package. Features of direct dependencies may be enabled with `<dep-name>/<feature-name>` syntax.

--all-features
: Activate all available features of all selected packages.

--no-default-features
: Do not activate the `default` feature of the current directory's package.

Manifest Options

--manifest-path *PATH*
: Path to the `Cargo.toml` file. By default, Cargo searches for the `Cargo.toml` file in the current directory or any parent directory.

--frozen
--locked
: Either of these flags requires that the `Cargo.lock` file is up-to-date. If the lock file is missing, or it needs to be updated, Cargo will exit with an error. The `--frozen` flag also prevents Cargo from attempting to access the network to determine if it is out-of-date.

  These may be used in environments where you want to assert that the `Cargo.lock` file is up-to-date (such as a CI build) or want to avoid network access.

--offline

Prevents Cargo from accessing the network for any reason. Without this flag, Cargo will stop with an error if it needs to access the network and the network is not available. With this flag, Cargo will attempt to proceed without the network if possible.

Beware that this may result in different dependency resolution than online mode. Cargo will restrict itself to crates that are downloaded locally, even if there might be a newer version as indicated in the local copy of the index. See the cargo-fetch(1) command to download dependencies before going offline.

May also be specified with the `net.offline` config value.

Miscellaneous Options

-j *N*
--jobs *N*
Number of parallel jobs to run. May also be specified with the `build.jobs` config value. Defaults to the number of CPUs.

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Common Options

-h
--help
Prints help information.

-Z *FLAG...*

Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
: Cargo succeeded.

101
: Cargo failed to complete.

EXAMPLES

1. Publish the current package:

```
cargo publish
```

SEE ALSO

cargo(1), cargo-package(1), cargo-login(1)

cargo yank

NAME

cargo-yank - Remove a pushed crate from the index

SYNOPSIS

`cargo yank [OPTIONS] --vers VERSION [CRATE]`

DESCRIPTION

The yank command removes a previously published crate's version from the server's index. This command does not delete any data, and the crate will still be available for download via the registry's download link.

Note that existing crates locked to a yanked version will still be able to download the yanked version to use it. Cargo will, however, not allow any new crates to be locked to any yanked version.

This command requires you to be authenticated with either the `--token` option or using cargo-login(1).

If the crate name is not specified, it will use the package name from the current directory.

OPTIONS

Owner Options

--vers *VERSION*
: The version to yank or un-yank.

--undo
: Undo a yank, putting a version back into the index.

--token *TOKEN*
: API token to use when authenticating. This overrides the token stored in the credentials file (which is created by cargo-login(1)).

  Cargo config environment variables can be used to override the tokens stored in the credentials file. The token for crates.io may be specified with the `CARGO_REGISTRY_TOKEN` environment variable. Tokens for other registries may be specified with environment variables of the form `CARGO_REGISTRIES_NAME_TOKEN` where `NAME` is the name of the registry in all capital letters.

--index *INDEX*
: The URL of the registry index to use.

--registry *REGISTRY*
: Name of the registry to use. Registry names are defined in Cargo config files. If not specified, the default registry is used, which is defined by the `registry.default` config key which defaults to `crates-io`.

Display Options

-v
--verbose
Use verbose output. May be specified twice for "very verbose" output which includes extra output such as dependency warnings and build script output. May also be specified with the `term.verbose` config value.

-q
--quiet
No output printed to stdout.

--color *WHEN*
Control when colored output is used. Valid values:

- `auto` (default): Automatically detect if color support is available on the terminal.
- `always`: Always display colors.
- `never`: Never display colors.

May also be specified with the `term.color` config value.

Common Options

-h
--help
Prints help information.

-Z *FLAG*...
Unstable (nightly-only) flags to Cargo. Run `cargo -Z help` for details.

ENVIRONMENT

See the reference for details on environment variables that Cargo reads.

Exit Status

0
Cargo succeeded.

101
Cargo failed to complete.

EXAMPLES

1. Yank a crate from the index:

    ```
    cargo yank --vers 1.0.7 foo
    ```

SEE ALSO

cargo(1), cargo-login(1), cargo-publish(1)

General Commands

cargo help

NAME

cargo-help - Get help for a Cargo command

SYNOPSIS

`cargo help [SUBCOMMAND]`

DESCRIPTION

Prints a help message for the given command.

EXAMPLES

1. Get help for a command:

    ```
    cargo help build
    ```

2. Help is also available with the `--help` flag:

    ```
    cargo build --help
    ```

SEE ALSO

cargo(1)

cargo version

NAME

cargo-version - Show version information

SYNOPSIS

```
cargo version [OPTIONS]
```

DESCRIPTION

Displays the version of Cargo.

OPTIONS

-v
--verbose
: Display additional version information.

EXAMPLES

1. Display the version:

   ```
   cargo version
   ```

2. The version is also available via flags:

   ```
   cargo --version
   cargo -V
   ```

3. Display extra version information:

   ```
   cargo -Vv
   ```

SEE ALSO

cargo(1)

Frequently Asked Questions

Is the plan to use GitHub as a package repository?

No. The plan for Cargo is to use crates.io, like npm or Rubygems do with npmjs.org and rubygems.org.

We plan to support git repositories as a source of packages forever, because they can be used for early development and temporary patches, even when people use the registry as the primary source of packages.

Why build crates.io rather than use GitHub as a registry?

We think that it's very important to support multiple ways to download packages, including downloading from GitHub and copying packages into your package itself.

That said, we think that crates.io offers a number of important benefits, and will likely become the primary way that people download packages in Cargo.

For precedent, both Node.js's npm and Ruby's bundler support both a central registry model as well as a Git-based model, and most packages are downloaded through the registry in those ecosystems, with an important minority of packages making use of git-based packages.

Some of the advantages that make a central registry popular in other languages include:

- **Discoverability**. A central registry provides an easy place to look for existing packages. Combined with tagging, this also makes it possible for a registry to provide ecosystem-wide information, such as a list of the most popular or most-depended-on packages.
- **Speed**. A central registry makes it possible to easily fetch just the metadata for packages quickly and efficiently, and then to efficiently download just the published package, and not other bloat that happens to exist in the repository. This adds up to a significant improvement in the speed of dependency resolution and fetching. As dependency graphs scale up, downloading all of the git repositories bogs down fast. Also remember that not everybody has a high-speed, low-latency Internet connection.

Will Cargo work with C code (or other languages)?

Yes!

Cargo handles compiling Rust code, but we know that many Rust packages link against C code. We also know that there are decades of tooling built up around compiling languages other than Rust.

Our solution: Cargo allows a package to specify a script (written in Rust) to run before invoking `rustc`. Rust is leveraged to implement platform-specific configuration and refactor out common build functionality among packages.

Can Cargo be used inside of `make` (or `ninja`, or ...)

Indeed. While we intend Cargo to be useful as a standalone way to compile Rust packages at the top-level, we know that some people will want to invoke Cargo from other build tools.

We have designed Cargo to work well in those contexts, paying attention to things like error codes and machine-readable output modes. We still have some work to do on those fronts, but using Cargo in the context of conventional scripts is something we designed for from the beginning and will continue to prioritize.

Does Cargo handle multi-platform packages or cross-compilation?

Rust itself provides facilities for configuring sections of code based on the platform. Cargo also supports platform-specific dependencies, and we plan to support more per-platform configuration in `Cargo.toml` in the future.

In the longer-term, we're looking at ways to conveniently cross-compile packages using Cargo.

Does Cargo support environments, like `production` or `test`?

We support environments through the use of profiles to support:

- environment-specific flags (like `-g --opt-level=0` for development and `--opt-level=3` for production).
- environment-specific dependencies (like `hamcrest` for test assertions).
- environment-specific `#[cfg]`
- a `cargo test` command

Does Cargo work on Windows?

Yes!

All commits to Cargo are required to pass the local test suite on Windows. If, however, you find a Windows issue, we consider it a bug, so please file an issue.

Why do binaries have `Cargo.lock` in version control, but not libraries?

The purpose of a `Cargo.lock` is to describe the state of the world at the time of a successful build. It is then used to provide deterministic builds across whatever machine is building the package by ensuring that the exact same dependencies are being compiled.

This property is most desirable from applications and packages which are at the very end of the dependency chain (binaries). As a result, it is recommended that all binaries check in their `Cargo.lock`.

For libraries the situation is somewhat different. A library is not only used by the library developers, but also any downstream consumers of the library. Users dependent on the library will not inspect the library's `Cargo.lock` (even if it exists). This is precisely because a library should **not** be deterministically recompiled for all users of the library.

If a library ends up being used transitively by several dependencies, it's likely that just a single copy of the library is desired (based on semver compatibility). If Cargo used all of the dependencies' `Cargo.lock` files, then multiple copies of the library could be used, and perhaps even a version conflict.

In other words, libraries specify semver requirements for their dependencies but cannot see the full picture. Only end products like binaries have a full picture to decide what versions of dependencies should be used.

Can libraries use `*` as a version for their dependencies?

As of January 22nd, 2016, crates.io rejects all packages (not just libraries) with wildcard dependency constraints.

While libraries *can*, strictly speaking, they should not. A version requirement of `*` says "This will work with every version ever," which is never going to be true. Libraries should always specify the range that they do work with, even if it's something as general as "every 1.x.y version."

Why `Cargo.toml`?

As one of the most frequent interactions with Cargo, the question of why the configuration file is named `Cargo.toml` arises from time to time. The leading capital-`C` was chosen to ensure that the manifest was grouped with other similar configuration files in directory listings. Sorting files often puts capital letters before lowercase letters, ensuring files like `Makefile` and `Cargo.toml` are placed together. The trailing `.toml` was chosen to emphasize the fact that the file is in the TOML configuration format.

Cargo does not allow other names such as `cargo.toml` or `Cargofile` to emphasize the ease of how a Cargo repository can be identified. An option of many possible names has historically led to confusion where one case was handled but others were accidentally forgotten.

How can Cargo work offline?

Cargo is often used in situations with limited or no network access such as airplanes, CI environments, or embedded in large production deployments. Users are often surprised when Cargo attempts to fetch resources from the network, and hence the request for Cargo to work offline comes up frequently.

Cargo, at its heart, will not attempt to access the network unless told to do so. That is, if no crates comes from crates.io, a git repository, or some other network location, Cargo will never attempt to make a network connection. As a result, if Cargo attempts to touch the network, then it's because it needs to fetch a required resource.

Cargo is also quite aggressive about caching information to minimize the amount of network activity. It will guarantee, for example, that if `cargo build` (or an equivalent) is run to completion then the next `cargo build` is guaranteed to not touch the network so long as `Cargo.toml` has not been modified in the meantime. This avoidance of the network boils down to a `Cargo.lock` existing and a populated cache of the crates reflected in the lock file. If either of these components are missing, then they're required for the build to succeed and must be fetched remotely.

As of Rust 1.11.0 Cargo understands a new flag, `--frozen`, which is an assertion that it shouldn't touch the network. When passed, Cargo will immediately return an error if it would otherwise attempt a network request. The error should include contextual information about why the network request is being made in the first place to help debug as well. Note that this flag *does not change the behavior of Cargo*, it simply asserts that Cargo shouldn't touch the network as a previous command has been run to ensure that network activity shouldn't be necessary.

For more information about vendoring, see documentation on source replacement.

Glossary

Artifact

An *artifact* is the file or set of files created as a result of the compilation process. This includes linkable libraries and executable binaries.

Crate

Every target in a package is a *crate*. Crates are either libraries or executable binaries. It may loosely refer to either the source code of the target, or the compiled artifact that the target produces. A crate may also refer to a compressed package fetched from a registry.

Edition

A *Rust edition* is a developmental landmark of the Rust language. The edition of a package is specified in the `Cargo.toml` manifest, and individual targets can specify which edition they use. See the Edition Guide for more information.

Feature

The meaning of *feature* depends on the context:

- A *feature* is a named flag which allows for conditional compilation. A feature can refer to an optional dependency, or an arbitrary name defined in a `Cargo.toml` manifest that can be checked within source code.
- Cargo has *unstable feature flags* which can be used to enable experimental behavior of Cargo itself.
- The Rust compiler and Rustdoc have their own unstable feature flags (see The Unstable Book and The Rustdoc Book).
- CPU targets have *target features* which specify capabilities of a CPU.

Index

The index is the searchable list of crates in a registry.

Lock file

The `Cargo.lock` *lock file* is a file that captures the exact version of every dependency used in a workspace or package. It is automatically generated by Cargo. See Cargo.toml vs Cargo.lock.

Manifest

A *manifest* is a description of a package or a workspace in a file named `Cargo.toml`.

A *virtual manifest* is a `Cargo.toml` file that only describes a workspace, and does not include a package.

Member

A *member* is a package that belongs to a workspace.

Package

A *package* is a collection of source files and a `Cargo.toml` manifest which describes the package. A package has a name and version which is used for specifying dependencies between packages. A package contains multiple targets, which are either libraries or executable binaries.

The *package root* is the directory where the package's `Cargo.toml` manifest is located.

The *package ID specification*, or *SPEC*, is a string used to uniquely reference a specific version of a package from a specific source.

Project

Another name for a package.

Registry

A *registry* is a service that contains a collection of downloadable crates that can be installed or used as dependencies for a package. The default registry is crates.io. The registry has an *index* which contains a list of all crates, and tells Cargo how to download the crates that are needed.

Source

A *source* is a provider that contains crates that may be included as dependencies for a package. There are several kinds of sources:

- **Registry source** — See registry.
- **Local registry source** — A set of crates stored as compressed files on the filesystem. See Local Registry Sources.
- **Directory source** — A set of crates stored as uncompressed files on the filesystem. See Directory Sources.
- **Path source** — An individual package located on the filesystem (such as a path dependency) or a set of multiple packages (such as path overrides).
- **Git source** — Packages located in a git repository (such as a git dependency or git source).

See Source Replacement for more information.

Spec

See package ID specification.

Target

The meaning of the term *target* depends on the context:

- **Cargo Target** — Cargo packages consist of *targets* which correspond to artifacts that will be produced. Packages can have library, binary, example, test, and benchmark targets. The list of targets are configured in the `Cargo.toml` manifest, often inferred automatically by the directory layout of the source files.
- **Target Directory** — Cargo places all built artifacts and intermediate files in the *target* directory. By default this is a directory named `target` at the workspace root, or the package root if not using a workspace. The directory may be changed with the `--target-dir` command-line option, the `CARGO_TARGET_DIR` environment variable, or the `build.target-dir` config option.
- **Target Architecture** — The OS and machine architecture for the built artifacts are typically referred to as a *target*.
- **Target Triple** — A triple is a specific format for specifying a target architecture. Triples may be referred to as a *target triple* which is the architecture for the artifact produced, and the *host triple* which is the architecture that the compiler is running on. The target triple can be specified with the `--target` command-line option or the `build.target` config option. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>` where:
  - `arch` = The base CPU architecture, for example `x86_64`, `i686`, `arm`, `thumb`, `mips`, etc.
  - `sub` = The CPU sub-architecture, for example `arm` has `v7`, `v7s`, `v5te`, etc.
  - `vendor` = The vendor, for example `unknown`, `apple`, `pc`, `linux`, etc.
  - `sys` = The system name, for example `linux`, `windows`, etc. `none` is typically used for bare-metal without an OS.
  - `abi` = The ABI, for example `gnu`, `android`, `eabi`, etc.

  Some parameters may be omitted. Run `rustc --print target-list` for a list of supported targets.

Test Targets

Cargo *test targets* generate binaries which help verify proper operation and correctness of code. There are two types of test artifacts:

- **Unit test** — A *unit test* is an executable binary compiled directly from a library or a binary target. It contains the entire contents of the library or binary code, and runs `#[test]` annotated functions, intended to verify individual units of code.
- **Integration test target** — An *integration test target* is an executable binary compiled from a *test target* which is a distinct crate whose source is located in the `tests`

directory or specified by the `[[test]]` table in the `Cargo.toml` manifest. It is intended to only test the public API of a library, or execute a binary to verify its operation.

Workspace

A *workspace* is a collection of one or more packages that share common dependency resolution (with a shared `Cargo.lock`), output directory, and various settings such as profiles.

A *virtual workspace* is a workspace where the root `Cargo.toml` manifest does not define a package, and only lists the workspace members.

The *workspace root* is the directory where the workspace's `Cargo.toml` manifest is located.