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.

Cargo Commands

The commands will let you interact with Cargo using its command-line interface.

Frequently Asked Questions

Appendices:

- Glossary
- Git Authentication

Other Documentation:

- Changelog --- Detailed notes about changes in Cargo in each release.
- Rust documentation website --- Links to official Rust documentation and tools.

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

This section provides a quick sense for the `cargo` command line tool. We demonstrate its ability to generate a new ***package*** for us, its ability to compile the ***crate*** within the package, and its ability to run the resulting program.

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 would pass `--lib`, instead.

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"
edition = "2024"

[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” program for us, otherwise known as a ***binary crate***. 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

Preliminaries

In Rust, as you may know, a library or executable program is called a *crate*. Crates are compiled using the Rust compiler, `rustc`. When starting with Rust, the first source code most people encounter is that of the classic "hello world" program, which they compile by invoking `rustc` directly:

```
$ rustc hello.rs
$ ./hello
Hello, world!
```

Note that the above command required that you specify the file name explicitly. If you were to directly use `rustc` to compile a different program, a different command line invocation would be required. If you needed to specify any specific compiler flags or include external dependencies, then the needed command would be even more specific (and complex).

Furthermore, most non-trivial programs will likely have dependencies on external libraries, and will therefore also depend transitively on *their* dependencies. Obtaining the correct versions of all the necessary dependencies and keeping them up to date would be hard and error-prone if done by hand.

Rather than work only with crates and `rustc`, you can avoid the difficulties involved with performing the above tasks by introducing a higher-level *"package"* abstraction and by using a *package manager*.

Enter: Cargo

Cargo is the Rust package manager. It 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.

To a large extent, Cargo normalizes the commands needed to build a given program or library; this is one aspect to the above mentioned conventions. As we show later, the same command can be used to build different *artifacts*, regardless of their names. Rather than invoke `rustc` directly, you can instead invoke something generic such as `cargo build` and let cargo worry about constructing the correct `rustc` invocation. Furthermore, Cargo will automatically fetch any dependencies you have defined for your artifact from a *registry*, and arrange for them to be added into your build as needed.

It is only a slight exaggeration to say that once you know how to build one Cargo-based project, you know how to build *all* of them.

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"
edition = "2024"

[dependencies]
```

This is called a ***manifest***, and it contains all of the metadata that Cargo needs to compile your package. This file is written in the TOML format (pronounced /tɑməl/).

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

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

Cargo generated a “hello world” program for you, otherwise known as a *binary crate*. 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!
```

You 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 your dependencies. Since there are none 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 `regex` cloned from its repository on GitHub:

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

To build, use `cargo build`:

```
$ cargo build
   Compiling regex v1.5.0 (file:///path/to/package/regex)
```

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 on 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 you also wanted to add a dependency on the `regex` crate, you 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"
edition = "2024"

[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)
```

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

Now, if `regex` gets updated, you will still build with the same revision until you choose to run `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
├── src/
│   ├── lib.rs
│   ├── main.rs
│   └── bin/
│       ├── named-executable.rs
│       ├── another-executable.rs
│       └── multi-file-executable/
│           ├── main.rs
│           └── some_module.rs
├── benches/
│   ├── large-input.rs
│   └── multi-file-bench/
│       ├── main.rs
│       └── bench_module.rs
├── examples/
│   ├── simple.rs
│   └── multi-file-example/
│       ├── main.rs
│       └── ex_module.rs
└── tests/
    ├── some-integration-tests.rs
    └── multi-file-test/
        ├── main.rs
        └── test_module.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/`.
- Benchmarks go in the `benches` directory.
- Examples go in the `examples` directory.
- Integration tests go in the `tests` directory.

If a binary, example, bench, or integration test consists of multiple source files, place a `main.rs` file along with the extra *modules* within a subdirectory of the `src/bin`, `examples`, `benches`, or `tests` directory. The name of the executable will be the directory name.

> **Note:** By convention, binaries, examples, benches and integration tests follow `kebab-case` naming style, unless there are compatibility reasons to do otherwise (e.g. compatibility with a pre-existing binary name). Modules within those targets are `snake_case` following the Rust standard.

You can learn more about Rust's module system in the book.

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

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.

When in doubt, check `Cargo.lock` into the version control system (e.g. Git). For a better understanding of why and what the alternatives might be, see "Why have Cargo.lock in version control?" in the FAQ. We recommend pairing this with Verifying Latest Dependencies

Let's dig in a little bit more.

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

```
[package]
name = "hello_world"
version = "0.1.0"

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

This package has a single dependency, on the `regex` library. It states in this case to rely on a particular Git repository that lives on GitHub. Since you haven't specified any other information, Cargo assumes that you intend to use the latest commit on the default 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 `regex` 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.

You could fix this problem by defining a specific `rev` value in our `Cargo.toml`, so Cargo could know exactly which revision to use when building the package:

```
[dependencies]
regex = { git = "https://github.com/rust-lang/regex.git", rev
= "9f9f693" }
```

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

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

```
[package]
name = "hello_world"
version = "0.1.0"

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

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

```
[[package]]
name = "hello_world"
version = "0.1.0"
dependencies = [
       "regex       1.5.0       (git+https://github.com/rust-
lang/regex.git#9f9f693768c584971a4d53bc3c586c33ed3a6831)",
]

[[package]]
name = "regex"
version = "1.5.0"
```

```
source              =              "git+https://github.com/rust-
lang/regex.git#9f9f693768c584971a4d53bc3c586c33ed3a6831"
```

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

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

```
$ cargo update          # updates all dependencies
$ cargo update regex    # updates just "regex"
```

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 `regex` 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 documentation tests. 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 regex v1.5.0 (https://github.com/rust-
lang/regex.git#9f9f693)
                     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 your package had tests, you 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. It will compile any examples you've included to ensure they still compile. It also runs documentation tests to ensure your code samples from documentation comments compile. Please see the testing guide in the Rust documentation for a general view of writing and organizing tests. See Cargo Targets: Tests to learn more about different styles of tests in Cargo.

Continuous Integration

Getting Started

A basic CI will build and test your projects:

GitHub Actions

To test your package on GitHub Actions, here is a sample `.github/workflows/ci.yml` file:

```
name: Cargo Build & Test

on:
  push:
  pull_request:

env:
  CARGO_TERM_COLOR: always

jobs:
  build_and_test:
    name: Rust project - latest
    runs-on: ubuntu-latest
    strategy:
      matrix:
        toolchain:
          - stable
          - beta
          - nightly
    steps:
      - uses: actions/checkout@v4
       - run: rustup update ${{ matrix.toolchain }} && rustup
default ${{ matrix.toolchain }}
      - run: cargo build --verbose
      - run: cargo test --verbose
```

This will test all three release channels (note a failure in any toolchain version will fail the entire job). You can also click `"Actions" > "new workflow"` in the GitHub UI and select Rust to add the default configuration to your repo. See GitHub Actions 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 documentation 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.

CircleCI

To test your package on CircleCI, here is a sample `.circleci/config.yml` file:

```
version: 2.1
jobs:
  build:
    docker:
```

```
                                                    #        check
https://circleci.com/developer/images/image/cimg/rust#image-
tags for latest
      - image: cimg/rust:1.77.2
    steps:
      - checkout
      - run: cargo test
```

To run more complex pipelines, including flaky test detection, caching, and artifact management, please see CircleCI Configuration Reference.

Verifying Latest Dependencies

When specifying dependencies in `Cargo.toml`, they generally match a range of versions. Exhaustively testing all version combination would be unwieldy. Verifying the latest versions would at least test for users who run `cargo add` or `cargo install`.

When testing the latest versions some considerations are:

- Minimizing external factors affecting local development or CI
- Rate of new dependencies being published
- Level of risk a project is willing to accept
- CI costs, including indirect costs like if a CI service has a maximum for parallel runners, causing new jobs to be serialized when at the maximum.

Some potential solutions include:

- Not checking in the `Cargo.lock`
    - Depending on PR velocity, many versions may go untested
    - This comes at the cost of determinism
- Have a CI job verify the latest dependencies but mark it to "continue on failure"
    - Depending on the CI service, failures might not be obvious
    - Depending on PR velocity, may use more resources than necessary
- Have a scheduled CI job to verify latest dependencies
    - A hosted CI service may disable scheduled jobs for repositories that haven't been touched in a while, affecting passively maintained packages
    - Depending on the CI service, notifications might not be routed to people who can act on the failure
    - If not balanced with dependency publish rate, may not test enough versions or may do redundant testing

- Regularly update dependencies through PRs, like with Dependabot or RenovateBot
    - Can isolate dependencies to their own PR or roll them up into a single PR
    - Only uses the resources necessary
    - Can configure the frequency to balance CI resources and coverage of dependency versions

An example CI job to verify latest dependencies, using GitHub Actions:

```
jobs:
  latest_deps:
    name: Latest Dependencies
    runs-on: ubuntu-latest
    continue-on-error: true
    env:
      CARGO_RESOLVER_INCOMPATIBLE_RUST_VERSIONS: allow
    steps:
      - uses: actions/checkout@v4
      - run: rustup update stable && rustup default stable
      - run: cargo update --verbose
      - run: cargo build --verbose
      - run: cargo test --verbose
```

Notes:

- `CARGO_RESOLVER_INCOMPATIBLE_RUST_VERSIONS` is set to ensure the resolver doesn't limit selected dependencies because of your project's Rust version.

For projects with higher risks of per-platform or per-Rust version failures, more combinations may want to be tested.

Verifying `rust-version`

When publishing packages that specify `rust-version`, it is important to verify the correctness of that field.

Some third-party tools that can help with this include:

- `cargo-msrv`
- `cargo-hack`

An example of one way to do this, using GitHub Actions:

```
jobs:
  msrv:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - uses: taiki-e/install-action@cargo-hack
     - run: cargo hack check --rust-version --workspace --all-
targets --ignore-private
```

This tries to balance thoroughness with turnaround time:

- A single platform is used as most projects are platform-agnostic, trusting platform-specific dependencies to verify their behavior.
- `cargo check` is used as most issues contributors will run into are API availability and not behavior.
- Unpublished packages are skipped as this assumes only consumers of the verified project, through a registry, will care about `rust-version`.

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 things 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). You will also need to provide and verify your email address on the Account Settings page. Once that is done create an API token, make sure you copy it. Once you leave the page you will not be able to see it again.

Then run the `cargo login` command.

```
$ cargo login
```

Then at the prompt put in the token specified.

```
please paste the API Token found on https://crates.io/me below
abcdefghijklmnopqrstuvwxyz012345
```

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

Note: The `cargo logout` command can be used to remove the token from `credentials.toml`. This can be useful if you no longer need it stored on the local machine.

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:

- `license` or `license-file`
- `description`
- `homepage`
- `repository`
- `readme`

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",
]
```

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 which provide guidelines on what is a compatible change. Then run `cargo publish` as described above to upload the new version.

Recommendation: Consider the full release process and automate what you can.

Each version should include:

- A changelog entry, preferably manually curated though a generated one is better than nothing
- A git tag pointing to the published commit

Examples of third-party tools that are representative of different workflows include (in alphabetical order):

- cargo-release
- cargo-smart-release
- release-plz

For more, see crates.io.

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 --version 1.0.1
$ cargo yank --version 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.

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.

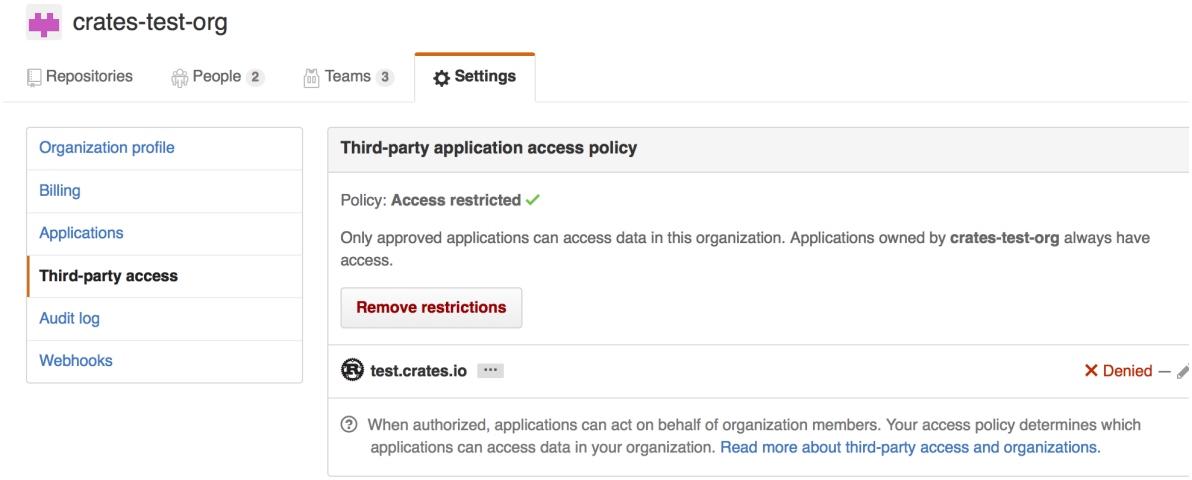
You are free to deny us this scope, and everything that worked before teams were introduced will keep working. However you will never be able to add a team as an owner, or publish a crate as a team owner. If you ever attempt to do this, you will get the error above. You may also see this error if you 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/ and re-authenticate, 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:

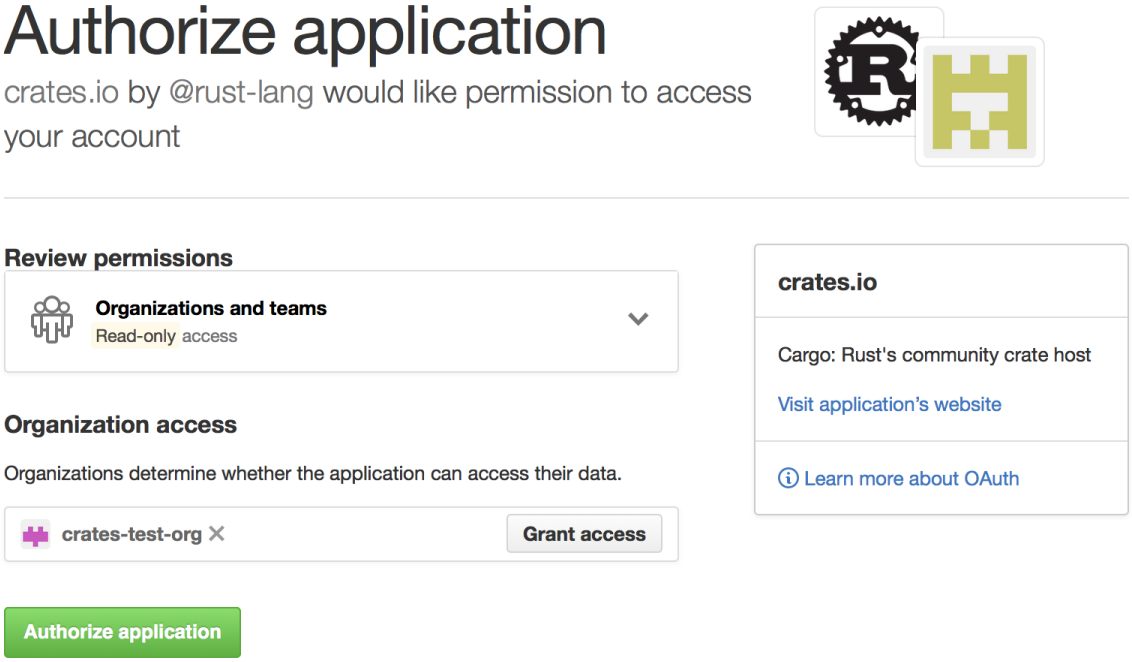
```
https://github.com/organizations/:org/settings/oauth_application_policy
```

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:



Troubleshooting GitHub team access errors

When trying to add a GitHub team as crate owner, you may see an error like:

```
error: failed to invite owners to crate <crate_name>: api errors (status 200 OK): could not find the github team org/repo
```

In that case, you should go to the GitHub Application settings page and check if crates.io is listed in the `Authorized OAuth Apps` tab. If it isn't, you should go to https://crates.io/ and authorize it. Then go back to the Application Settings page on GitHub, click on the crates.io application in the list, and make sure you or your organization is listed in the "Organization access" list with a green check mark. If there's a button labeled `Grant` or `Request`, you should grant the access or request the org owner to do so.

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.toml` Cargo's global configuration file, see the config entry in the reference.
- `credentials.toml` Private login credentials from `cargo login` in order to log in to a registry.
- `.crates.toml`, `.crates2.json` These hidden files contain 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`. That can unnecessarily slow down the build as downloading, extracting, recompressing and reuploading the cache to the CI servers can take some time.

If you wish to cache binaries installed with `cargo install`, you need to cache the `bin/` folder and the `.crates.toml` and `.crates2.json` files.

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

- `.crates.toml`
- `.crates2.json`
- `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.

Cargo Reference

The reference covers the details of various areas of Cargo.

- Future incompat report
- Reporting build timings
- Lints
- Unstable Features

The Manifest Format

The `Cargo.toml` file for each package is called its *manifest*. It is written in the TOML format. It contains metadata that is needed to compile the package. Checkout the `cargo locate-project` section for more detail on how cargo finds the manifest file.

Every manifest file consists of the following sections:

- `cargo-features` --- Unstable, nightly-only features.
- `[package]` --- Defines a package.
    - `name` --- The name of the package.
    - `version` --- The version of the package.
    - `authors` --- The authors of the package.
    - `edition` --- The Rust edition.
    - `rust-version` --- The minimal supported Rust version.
    - `description` --- A description of the package.
    - `documentation` --- URL of the package documentation.
    - `readme` --- Path to the package's README file.
    - `homepage` --- URL of the package homepage.
    - `repository` --- URL of the package source repository.
    - `license` --- The package license.
    - `license-file` --- Path to the text of the license.
    - `keywords` --- Keywords for the package.
    - `categories` --- Categories of the package.
    - `workspace` --- Path to the workspace for the package.
    - `build` --- Path to the package build script.
    - `links` --- Name of the native library the package links with.
    - `exclude` --- Files to exclude when publishing.
    - `include` --- Files to include when publishing.
    - `publish` --- Can be used to prevent publishing the package.
    - `metadata` --- Extra settings for external tools.

    - `default-run` --- The default binary to run by `cargo run`.
    - `autolib` --- Disables library auto discovery.
    - `autobins` --- Disables binary auto discovery.
    - `autoexamples` --- Disables example auto discovery.
    - `autotests` --- Disables test auto discovery.
    - `autobenches` --- Disables bench auto discovery.
    - `resolver` --- Sets the dependency resolver to use.
- Target tables: (see configuration for settings)
    - `[lib]` --- Library target settings.
    - `[[bin]]` --- Binary target settings.
    - `[[example]]` --- Example target settings.
    - `[[test]]` --- Test target settings.
    - `[[bench]]` --- Benchmark target settings.
- Dependency tables:
    - `[dependencies]` --- Package library dependencies.
    - `[dev-dependencies]` --- Dependencies for examples, tests, and benchmarks.
    - `[build-dependencies]` --- Dependencies for build scripts.
    - `[target]` --- Platform-specific dependencies.
- `[badges]` --- Badges to display on a registry.
- `[features]` --- Conditional compilation features.
- `[lints]` --- Configure linters for this package.
- `[patch]` --- Override dependencies.
- `[replace]` --- Override dependencies (deprecated).
- `[profile]` --- Compiler settings and optimizations.
- `[workspace]` --- The workspace definition.

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
```

The only field required by Cargo is `name`. If publishing to a registry, the registry may require additional fields. See the notes below and the publishing chapter for requirements for publishing to crates.io.

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 use only alphanumeric characters or `-` or `_`, and cannot be empty.

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:

- Only ASCII characters are allowed.
- Do not use reserved names.
- Do not use special Windows names such as "nul".
- Use a maximum of 64 characters of length.

The `version` field

The `version` field is formatted according to the SemVer specification:

Versions must have three numeric parts, the major version, the minor version, and the patch version.

A pre-release part can be added after a dash such as `1.0.0-alpha`. The pre-release part may be separated with periods to distinguish separate components. Numeric components will use numeric comparison while

everything else will be compared lexicographically. For example, `1.0.0-alpha.11` is higher than `1.0.0-alpha.4`.

A metadata part can be added after a plus, such as `1.0.0+21AF26D3`. This is for informational purposes only and is generally ignored by Cargo.

Cargo bakes in the concept of Semantic Versioning, so versions are considered compatible if their left-most non-zero major/minor/patch component is the same. See the Resolver chapter for more information on how Cargo uses versions to resolve dependencies.

This field is optional and defaults to `0.0.0`. The field is required for publishing packages.

> **MSRV:** Before 1.75, this field was required

The `authors` field

> **Warning**: This field is deprecated

The optional `authors` field lists in an array the people or organizations that are considered the "authors" of the package. An optional email address may be included within angled brackets at the end of each author entry.

```
[package]
# ...
authors = ["Graydon Hoare", "Fnu Lnu <no-reply@rust-lang.org>"]
```

This field is surfaced in package metadata and in the `CARGO_PKG_AUTHORS` environment variable within `build.rs` for backwards compatibility.

The `edition` field

The `edition` key is an optional key that affects which Rust Edition your package is compiled with. Setting the `edition` key in `[package]` will affect all targets/crates in the package, including test suites, benchmarks, binaries, examples, etc.

```
[package]
# ...
```

```
edition = '2024'
```

Most manifests have the `edition` field filled in automatically by `cargo new` with the latest stable edition. By default `cargo new` creates a manifest with the 2024 edition currently.

If the `edition` field is not present in `Cargo.toml`, then the 2015 edition is assumed for backwards compatibility. Note that all manifests created with `cargo new` will not use this historical fallback because they will have `edition` explicitly specified to a newer value.

The `rust-version` field

The `rust-version` field tells cargo what version of the Rust toolchain you support for your package. See the Rust version chapter for more detail.

The `description` field

The description is a short blurb about the package. crates.io will display this with your package. This should be plain text (not Markdown).

```
[package]
# ...
description = "A short description of my package"
```

> **Note**: crates.io requires the `description` to be set.

The `documentation` field

The `documentation` 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 when the documentation has been built and is available (see docs.rs queue).

```
[package]
# ...
documentation = "https://docs.rs/bitflags"
```

The `readme` field

The `readme` field should be the path to a file in the package root (relative to this `Cargo.toml`) that contains general information about the package. This file will be transferred to the registry when you publish. crates.io will interpret it as Markdown and render it on the crate's page.

```
[package]
# ...
readme = "README.md"
```

If no value is specified for this field, and a file named `README.md`, `README.txt` or `README` exists in the package root, then the name of that file will be used. You can suppress this behavior by setting this field to `false`. If the field is set to `true`, a default value of `README.md` will be assumed.

The `homepage` field

The `homepage` field should be a URL to a site that is the home page for your package.

```
[package]
# ...
homepage = "https://serde.rs"
```

A value should only be set for `homepage` if there is a dedicated website for the crate other than the source repository or API documentation. Do not make `homepage` redundant with either the `documentation` or `repository` values.

The `repository` field

The `repository` field should be a URL to the source repository for your package.

```
[package]
# ...
repository = "https://github.com/rust-lang/cargo"
```

The `license` and `license-file` fields

The `license` field contains the name of the software license that the package is released under. The `license-file` field contains the path to a file containing the text of the license (relative to this `Cargo.toml`).

crates.io interprets the `license` field as an SPDX 2.3 license expression. The name must be a known license from the SPDX license list 3.20. See the SPDX site for more information.

SPDX license expressions support AND and OR operators to combine multiple licenses.[1]

```
[package]
# ...
license = "MIT OR Apache-2.0"
```

Using `OR` indicates the user may choose either license. Using `AND` indicates the user must comply with both licenses simultaneously. The `WITH` operator indicates a license with a special exception. Some examples:

- `MIT OR Apache-2.0`
- `LGPL-2.1-only AND MIT AND BSD-2-Clause`
- `GPL-2.0-or-later WITH Bison-exception-2.2`

If a package is using a nonstandard license, then the `license-file` field may be specified in lieu of the `license` field.

```
[package]
# ...
license-file = "LICENSE.txt"
```

Note: crates.io requires either `license` or `license-file` to be set.

1

Previously multiple licenses could be separated with a `/`, but that usage is deprecated.

The `keywords` field

The `keywords` field is an array of strings that describe this package. This can help when searching for the package on a registry, and you may

choose any words that would help someone find this crate.

```
[package]
# ...
keywords = ["gamedev", "graphics"]
```

> **Note**: crates.io allows a maximum of 5 keywords. Each keyword must be ASCII text, have at most 20 characters, start with an alphanumeric character, and only contain letters, numbers, `_`, `-` or `+`.

The `categories` field

The `categories` field is an array of strings of the categories this package belongs to.

```
categories = ["command-line-utilities", "development-tools::cargo-plugins"]
```

> **Note**: crates.io has a maximum of 5 categories. Each category should match one of the strings available at https://crates.io/category_slugs, and must match exactly.

The `workspace` field

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. Setting this is useful if the member is not inside a subdirectory of the workspace root.

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

This field cannot be specified if the manifest already has a `[workspace]` table defined. 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`).

For more information, see the workspaces chapter.

The `build` field

The `build` 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

The `links` 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.

For example, a crate that links a native library called "git2" (e.g. `libgit2.a` on Linux) may specify:

```
[package]
# ...
links = "git2"
```

The `exclude` and `include` fields

The `exclude` and `include` fields can be used to explicitly specify which files are included when packaging a project to be published, and certain kinds of change tracking (described below). 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. You may run `cargo package --list` to verify which files will be included in the package.

```
[package]
# ...
exclude = ["/ci", "images/", ".*"]
```

```
[package]
# ...
include      =      ["/src",      "COPYRIGHT",      "/examples",
"!/examples/big_example"]
```

The default if neither field is specified is to include all files from the root of the package, except for the exclusions listed below.

If `include` is not specified, then the following files will be excluded:

- If the package is not in a git repository, all "hidden" files starting with a dot will be skipped.
- If the package is in a git repository, any files that are ignored by the gitignore rules of the repository and global git configuration will be skipped.

Regardless of whether `exclude` or `include` is specified, the following files are always excluded:

- Any sub-packages will be skipped (any subdirectory that contains a `Cargo.toml` file).
- A directory named `target` in the root of the package will be skipped.

The following files are always included:

- The `Cargo.toml` file of the package itself is always included, it does not need to be listed in `include`.
- A minimized `Cargo.lock` is automatically included. See `cargo package` for more information.
- If a `license-file` is specified, it is always included.

The options are mutually exclusive; setting `include` will override an `exclude`. If you need to have exclusions to a set of `include` files, use the `!` operator described below.

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`.

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

The `publish` field can be used to control which registries names the package may be published to:

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

To prevent a package from being published to a registry (like crates.io) by mistake, for instance to keep a package private in a company, you can omit the `version` field. If you'd like to be more explicit, you can disable publishing:

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

If publish array contains a single registry, `cargo publish` command will use it when `--registry` flag is not specified.

The `metadata` table

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"
```

You'll need to look in the documentation for your tool to see how to use this field. For Rust Projects that use `package.metadata` tables, see:

- docs.rs

There is a similar table at the workspace level at `workspace.metadata`. While cargo does not specify a format for the content of either of these tables, it is suggested that external tools may wish to use them in a consistent fashion, such as referring to the data in `workspace.metadata` if data is missing from `package.metadata`, if that makes sense for the tool in question.

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"
```

The `[lints]` section

Override the default level of lints from different tools by assigning them to a new level in a table, for example:

```
[lints.rust]
unsafe_code = "forbid"
```

This is short-hand for:

```
[lints.rust]
unsafe_code = { level = "forbid", priority = 0 }
```

`level` corresponds to the lint levels in `rustc`:

- `forbid`
- `deny`
- `warn`
- `allow`

`priority` is a signed integer that controls which lints or lint groups override other lint groups:

- lower (particularly negative) numbers have lower priority, being overridden by higher numbers, and show up first on the command-line to tools like `rustc`

To know which table under `[lints]` a particular lint belongs under, it is the part before `::` in the lint name. If there isn't a `::`, then the tool is `rust`. For example a warning about `unsafe_code` would be `lints.rust.unsafe_code` but a lint about `clippy::enum_glob_use` would be `lints.clippy.enum_glob_use`.

For example:

```
[lints.rust]
unsafe_code = "forbid"

[lints.clippy]
enum_glob_use = "deny"
```

Generally, these will only affect local development of the current package. Cargo only applies these to the current package and not to dependencies. As for dependents, Cargo suppresses lints from non-path dependencies with features like `--cap-lints`.

MSRV: Respected as of 1.74

The `[badges]` section

The `[badges]` section is for specifying status badges that can be displayed on a registry website when the package is published.

Note: crates.io previously displayed badges next to a crate on its website, but that functionality has been removed. Packages should place badges in its README file which will be displayed on crates.io (see the `readme` field).

```
[badges]
# The `maintenance` table indicates the status of the maintenance of
# the crate. This may be used by a registry, but is currently not
# used by crates.io. See https://github.com/rust-lang/crates.io/issues/2437
# and https://github.com/rust-lang/crates.io/issues/2438 for more details.
#
# The `status` field 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 = "..." }
```

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.

Cargo Targets

Cargo packages consist of *targets* which correspond to source files which can be compiled into a crate. Packages can have library, binary, example, test, and benchmark targets. The list of targets can be configured in the `Cargo.toml` manifest, often inferred automatically by the directory layout of the source files.

See Configuring a target below for details on configuring the settings for a target.

Library

The library target defines a "library" that can be used and linked by other libraries and executables. The filename defaults to `src/lib.rs`, and the name of the library defaults to the name of the package, with any dashes replaced with underscores. A package can have only one library. The settings for the library can be customized in the `[lib]` table in `Cargo.toml`.

```
# Example of customizing the library in Cargo.toml.
[lib]
crate-type = ["cdylib"]
bench = false
```

Binaries

Binary targets are executable programs that can be run after being compiled. A binary's source can be `src/main.rs` and/or stored in the `src/bin/` directory. For `src/main.rs`, the default binary name is the package name. The settings for each binary can be customized in the `[[bin]]` tables in `Cargo.toml`.

Binaries can use the public API of the package's library. They are also linked with the `[dependencies]` defined in `Cargo.toml`.

You can run individual binaries with the `cargo run` command with the `--bin <bin-name>` option. `cargo install` can be used to copy the executable to a common location.

```
# Example of customizing binaries in Cargo.toml.
[[bin]]
name = "cool-tool"
test = false
bench = false

[[bin]]
name = "frobnicator"
required-features = ["frobnicate"]
```

Examples

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

Examples can use the public API of the package's library. They are also linked with the `[dependencies]` and `[dev-dependencies]` defined in `Cargo.toml`.

By default, examples are executable binaries (with a `main()` function). You can specify the `crate-type` field to make an example be compiled as a library:

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

You can run individual executable examples with the `cargo run` command with the `--example <example-name>` option. Library examples can be built with `cargo build` with the `--example <example-name>` option. `cargo install` with the `--example <example-name>` option can be used to copy executable binaries to a common location. Examples are compiled by `cargo test` by default to protect them from bit-rotting. Set the `test` field to `true` if you have `#[test]` functions in the example that you want to run with `cargo test`.

Tests

There are two styles of tests within a Cargo project:

- *Unit tests* which are functions marked with the `#[test]` attribute located within your library or binaries (or any target enabled with the `test` field). These tests have access to private APIs located within the target they are defined in.
- *Integration tests* which is a separate executable binary, also containing `#[test]` functions, which is linked with the project's library and has access to its *public* API.

Tests are run with the `cargo test` command. By default, Cargo and `rustc` use the libtest harness which is responsible for collecting functions annotated with the `#[test]` attribute and executing them in parallel, reporting the success and failure of each test. See the `harness` field if you want to use a different harness or test strategy.

> **Note**: There is another special style of test in Cargo: documentation tests. They are handled by `rustdoc` and have a slightly different execution model. For more information, please see `cargo test`.

Integration tests

Files located under the `tests` directory are integration tests. When you run `cargo test`, Cargo will compile each of these files as a separate crate, and execute them.

Integration tests can use the public API of the package's library. They are also linked with the `[dependencies]` and `[dev-dependencies]` defined in `Cargo.toml`.

If you want to share code among multiple integration tests, you can place it in a separate module such as `tests/common/mod.rs` and then put `mod common;` in each test to import it.

Each integration test results in a separate executable binary, and `cargo test` will run them serially. In some cases this can be inefficient, as it can

take longer to compile, and may not make full use of multiple CPUs when running the tests. If you have a lot of integration tests, you may want to consider creating a single integration test, and split the tests into multiple modules. The libtest harness will automatically find all of the `#[test]` annotated functions and run them in parallel. You can pass module names to `cargo test` to only run the tests within that module.

Binary targets are automatically built if there is an integration test. This allows an integration test to execute the binary to exercise and test its behavior. The `CARGO_BIN_EXE_<name>` environment variable is set when the integration test is built so that it can use the `env` macro to locate the executable.

Benchmarks

Benchmarks provide a way to test the performance of your code using the `cargo bench` command. They follow the same structure as tests, with each benchmark function annotated with the `#[bench]` attribute. Similarly to tests:

- Benchmarks are placed in the `benches` directory.
- Benchmark functions defined in libraries and binaries have access to the *private* API within the target they are defined in. Benchmarks in the `benches` directory may use the *public* API.
- The `bench` field can be used to define which targets are benchmarked by default.
- The `harness` field can be used to disable the built-in harness.

Note: The `#[bench]` attribute is currently unstable and only available on the nightly channel. There are some packages available on crates.io that may help with running benchmarks on the stable channel, such as Criterion.

Configuring a target

All of the `[lib]`, `[[bin]]`, `[[example]]`, `[[test]]`, and `[[bench]]` sections in `Cargo.toml` 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. You can only specify one library, so `[lib]` is a normal TOML table.

The following is an overview of the TOML settings for each target, with each field described in detail below.

```
[lib]
name = "foo"           # The name of the target.
path = "src/lib.rs"    # The source file of the target.
test = true            # Is tested by default.
doctest = true          # Documentation examples are tested by
default.
bench = true           # Is benchmarked by default.
doc = true             # Is documented by default.
proc-macro = false       # Set to `true` for a proc-macro
library.
harness = true         # Use libtest harness.
crate-type = ["lib"]   # The crate types to generate.
required-features = [] # Features required to build this
target (N/A for lib).
```

The `name` field

The `name` field specifies the name of the target, which corresponds to the filename of the artifact that will be generated. For a library, this is the crate name that dependencies will use to reference it.

For the library target, this defaults to the name of the package , with any dashes replaced with underscores. For the default binary (`src/main.rs`), it also defaults to the name of the package, with no replacement for dashes. For auto discovered targets, it defaults to the directory or file name.

This is required for all targets except `[lib]`.

The `path` field

The `path` field specifies where the source for the crate is located, relative to the `Cargo.toml` file.

If not specified, the inferred path is used based on the target name.

The `test` field

The `test` field indicates whether or not the target is tested by default by `cargo test`. The default is `true` for lib, bins, and tests.

> **Note**: Examples are built by `cargo test` by default to ensure they continue to compile, but they are not *tested* by default. Setting `test = true` for an example will also build it as a test and run any `#[test]` functions defined in the example.

The `doctest` field

The `doctest` field indicates whether or not documentation examples are tested by default by `cargo test`. This is only relevant for libraries, it has no effect on other sections. The default is `true` for the library.

The `bench` field

The `bench` field indicates whether or not the target is benchmarked by default by `cargo bench`. The default is `true` for lib, bins, and benchmarks.

The `doc` field

The `doc` field indicates whether or not the target is included in the documentation generated by `cargo doc` by default. The default is `true` for libraries and binaries.

> **Note**: The binary will be skipped if its name is the same as the lib target.

The `plugin` field

This option is deprecated and unused.

The `proc-macro` field

The `proc-macro` field indicates that the library is a procedural macro (reference). This is only valid for the `[lib]` target.

The `harness` field

The `harness` field indicates that the `--test` flag will be passed to `rustc` which will automatically include the libtest library which is the driver for collecting and running tests marked with the `#[test]` attribute or benchmarks with the `#[bench]` attribute. The default is `true` for all targets.

If set to `false`, then you are responsible for defining a `main()` function to run tests and benchmarks.

Tests have the `cfg(test)` conditional expression enabled whether or not the harness is enabled.

The `crate-type` field

The `crate-type` field defines the crate types that will be generated by the target. It is an array of strings, allowing you to specify multiple crate types for a single target. This can only be specified for libraries and examples. Binaries, tests, and benchmarks are always the "bin" crate type. The defaults are:

Target	Crate Type
Normal library	`"lib"`
Proc-macro library	`"proc-macro"`
Example	`"bin"`

The available options are `bin`, `lib`, `rlib`, `dylib`, `cdylib`, `staticlib`, and `proc-macro`. You can read more about the different crate types in the Rust Reference Manual.

The `required-features` field

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

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

[[bin]]
name = "my-pg-tool"
required-features = ["postgres", "tools"]
```

The `edition` field

The `edition` field defines the Rust edition the target will use. If not specified, it defaults to the `edition` field for the `[package]`.

> **Note:** This field is deprecated and will be removed in a future Edition

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 `autolib`, `autobins`, `autoexamples`, `autotests`, or `autobenches` to `false` in the `[package]` section will disable auto-discovery of the corresponding target type.

```
[package]
# ...
autolib = false
autobins = false
autoexamples = false
autotests = false
autobenches = false
```

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:

```
[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`.

MSRV: Respected as of 1.27 for `autobins`, `autoexamples`, `autotests`, and `autobenches`

MSRV: Respected as of 1.83 for `autolib`

Rust Version

The `rust-version` field is an optional key that tells cargo what version of the Rust toolchain you support for your package.

```
[package]
# ...
rust-version = "1.56"
```

The Rust version must be a bare version number with at least one component; it cannot include semver operators or pre-release identifiers. Compiler pre-release identifiers such as -nightly will be ignored while checking the Rust version.

> **MSRV:** Respected as of 1.56

Uses

Diagnostics:

When your package is compiled on an unsupported toolchain, Cargo will report that as an error to the user. This makes the support expectations clear and avoids reporting a less direct diagnostic like invalid syntax or missing functionality in the standard library. This affects all Cargo targets in the package, including binaries, examples, test suites, benchmarks, etc. A user can opt-in to an unsupported build of a package with the `--ignore-rust-version` flag.

Development aid:

`cargo add` will auto-select the dependency's version requirement to be the latest version compatible with your `rust-version`. If that isn't the latest version, `cargo add` will inform users so they can make the choice on whether to keep it or update your `rust-version`.

The resolver may take Rust version into account when picking dependencies.

Other tools may also take advantage of it, like `cargo clippy`'s `incompatible_msrv` lint.

> **Note:** The `rust-version` may be ignored using the `--ignore-rust-version` option.

Support Expectations

These are general expectations; some packages may document when they do not follow these.

Complete:

All functionality, including binaries and API, are available on the supported Rust versions under every feature.

Verified:

A package's functionality is verified on its supported Rust versions, including automated testing. See also our Rust version CI guide.

Patchable:

When licenses allow it, users can override their local dependency with a fork of your package. In this situation, Cargo may load the entire workspace for the patched dependency which should work on the supported Rust versions, even if other packages in the workspace have different supported Rust versions.

Dependency Support:

In support of the above, it is expected that each dependency's version-requirement supports at least one version compatible with your `rust-version`. However, it is **not** expected that the dependency specification excludes versions incompatible with your `rust-version`. In fact, supporting both allows you to balance the needs of users that support older Rust versions with those that don't.

Setting and Updating Rust Version

What Rust versions to support is a trade off between

- Costs for the maintainer in not using newer features of the Rust toolchain or their dependencies
- Costs to users who would benefit from a package using newer features of a toolchain, e.g. reducing build times by migrating to a feature in the standard library from a polyfill
- Availability of a package to users supporting older Rust versions

Note: Changing `rust-version` is assumed to be a minor incompatibility

Recommendation: Choose a policy for what Rust versions to support and when that is changed so users can compare it with their own policy and, if it isn't compatible, decide whether the loss of general improvements or the risk of a blocking bug that won't be fixed is acceptable or not.

The simplest policy to support is to always use the latest Rust version.

Depending on your risk profile, the next simplest approach is to continue to support old major or minor versions of your package that support older Rust versions.

Selecting supported Rust versions

Users of your package are most likely to track their supported Rust versions to:

- Their Rust toolchain vendor's support policy, e.g. The Rust Project or a Linux distribution
    - Note: the Rust Project only offers bug fixes and security updates for the latest version.
- A fixed schedule for users to re-verify their packages with the new toolchain, e.g. the first release of the year, every 5 releases.

In addition, users are unlikely to be using the new Rust version immediately but need time to notice and re-verify or might not be aligned on the exact same schedule..

Example version policies:

- "N-2", meaning "latest version with a 2 release grace window for updating"
- Every even release with a 2 release grace window for updating
- Every version from this calendar year with a one year grace window for updating

Note: To find the minimum `rust-version` compatible with your project as-is, you can use third-party tools like `cargo-msrv`.

Update timeline

When your policy specifies you no longer need to support a Rust version, you can update `rust-version` immediately or when needed.

By allowing `rust-version` to drift from your policy, you offer users more of a grace window for upgrading. However, this is too unpredictable to be relied on for aligning with the Rust version users track.

The further `rust-version` drifts from your specified policy, the more likely users are to infer a policy you did not intend, leading to frustration at the unmet expectations.

When drift is allowed, there is the question of what is "justifiable enough" to drop supported Versions. Each person can come to a reasonably different justification; working through that discussion can be frustrating for the involved parties. This will disempower those who would want to avoid that type of conflict, which is particularly the case for new or casual contributors who either feel that they are not in a position to raise the question or that the conflict may hurt the chance of their change being merged.

Multiple Policies in a Workspace

Cargo allows supporting multiple policies within one workspace.

Verifying specific packages under specific Rust versions can get complicated. Tools like `cargo-hack` can help.

For any dependency shared across policies, the lowest common versions must be used as Cargo unifies SemVer-compatible versions, potentially limiting access to features of the shared dependency for the workspace member with the higher `rust-version`.

To allow users to patch a dependency on one of your workspace members, every package in the workspace would need to be loadable in the oldest Rust version supported by the workspace.

When using `incompatible-rust-versions = "fallback"`, the Rust version of one package can affect dependency versions selected for another package with a different Rust version. See the resolver chapter for more details.

One or More Policies

One way to mitigate the downsides of supporting older Rust versions is to apply your policy to older major or minor versions of your package that you continue to support. You likely still need a policy for what Rust versions the development branch support compared to the release branches for those major or minor versions.

Only updating the development branch when "needed"' can help reduce the number of supported release branches.

There is the question of what can be backported into these release branches. By backporting new functionality between minor versions, the next available version would be missing it which could be considered a breaking change, violating SemVer. Backporting changes also comes with the risk of introducing bugs.

Supporting older versions comes at a cost. This cost is dependent on the risk and impact of bugs within the package and what is acceptable for backporting. Creating the release branches on-demand and putting the backport burden on the community are ways to balance this cost.

There is not yet a way for dependency management tools to report that a non-latest version is still supported, shifting the responsibility to users to notice this in documentation.

For example, a Rust version support policy could look like:

- The development branch tracks to the latest stable release from the Rust Project, updated when needed
    - The minor version will be raised when changing `rust-version`
- The project supports every version for this calendar year, with another year grace window
    - The last minor version that supports a supported Rust version will receive community provided bug fixes
    - Fixes must be backported to all supported minor releases between the development branch and the needed supported Rust version

Workspaces

A *workspace* is a collection of one or more packages, called *workspace members*, that are managed together.

The key points of workspaces are:

- Common commands can run across all workspace members, like `cargo check --workspace`.
- All packages share a common `Cargo.lock` file which resides in the *workspace root*.
- All packages share a common output directory, which defaults to a directory named `target` in the *workspace root*.
- Sharing package metadata, like with `workspace.package`.
- The `[patch]`, `[replace]` and `[profile.*]` sections in `Cargo.toml` are only recognized in the *root* manifest, and ignored in member crates' manifests.

The root `Cargo.toml` of a workspace supports the following sections:

- `[workspace]` --- Defines a workspace.
  - `resolver` --- Sets the dependency resolver to use.
  - `members` --- Packages to include in the workspace.
  - `exclude` --- Packages to exclude from the workspace.
  - `default-members` --- Packages to operate on when a specific package wasn't selected.
  - `package` --- Keys for inheriting in packages.
  - `dependencies` --- Keys for inheriting in package dependencies.
  - `lints` --- Keys for inheriting in package lints.
  - `metadata` --- Extra settings for external tools.
- `[patch]` --- Override dependencies.
- `[replace]` --- Override dependencies (deprecated).
- `[profile]` --- Compiler settings and optimizations.

The `[workspace]` section

To create a workspace, you add the `[workspace]` table to a `Cargo.toml`:

```
[workspace]
# ...
```

At minimum, a workspace has to have a member, either with a root package or as a virtual manifest.

Root package

If the `[workspace]` section is added to a `Cargo.toml` that already defines a `[package]`, the package is the *root package* of the workspace. The *workspace root* is the directory where the workspace's `Cargo.toml` is located.

```
[workspace]

[package]
name = "hello_world" # the name of the package
version = "0.1.0"    # the current version, obeying semver
```

Virtual workspace

Alternatively, a `Cargo.toml` file can be created with a `[workspace]` section but without a `[package]` section. This is called a *virtual manifest*. This is typically useful when there isn't a "primary" package, or you want to keep all the packages organized in separate directories.

```
# [PROJECT_DIR]/Cargo.toml
[workspace]
members = ["hello_world"]
resolver = "3"

# [PROJECT_DIR]/hello_world/Cargo.toml
[package]
name = "hello_world" # the name of the package
version = "0.1.0"    # the current version, obeying semver
```

```
edition = "2024"      # the edition, will have no effect on a
resolver used in the workspace
```

By having a workspace without a root package,

- `resolver` must be set explicitly in virtual workspaces as they have no `package.edition` to infer it from resolver version.
- Commands run in the workspace root will run against all workspace members by default, see `default-members`.

The `members` and `exclude` fields

The `members` and `exclude` fields define which packages are members of the workspace:

```
[workspace]
members = ["member1", "path/to/member2", "crates/*"]
exclude = ["crates/foo", "path/to/other"]
```

All `path` dependencies residing in the workspace directory automatically become members. Additional members can be listed with the `members` key, which should be an array of strings containing directories with `Cargo.toml` files.

The `members` list also supports globs to match multiple paths, using typical filename glob patterns like `*` and `?`.

The `exclude` key can be used to prevent 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, or using a glob pattern and you want to remove a directory.

When inside a subdirectory within the workspace, Cargo will automatically search the parent directories for a `Cargo.toml` file with a `[workspace]` definition to determine which workspace to use. The `package.workspace` manifest key can be used in member crates to point at a workspace's root to override this automatic search. The manual setting can be useful if the member is not inside a subdirectory of the workspace root.

Package selection

In a workspace, package-related Cargo commands like `cargo build` can use the `-p` / `--package` or `--workspace` command-line flags to determine which packages to operate on. If neither of those flags are specified, Cargo will use the package in the current working directory. However, if the current directory is a workspace root, the `default-members` will be used.

The `default-members` field

The `default-members` field specifies paths of members to operate on when in the workspace root and the package selection flags are not used:

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

Note: when a root package is present, you can only operate on it using `--package` and `--workspace` flags.

When unspecified, the root package will be used. In the case of a virtual workspace, all members will be used (as if `--workspace` were specified on the command-line).

The package table

The `workspace.package` table is where you define keys that can be inherited by members of a workspace. These keys can be inherited by defining them in the member package with `{key}.workspace = true`.

Keys that are supported:

`authors`	`categories`
`description`	`documentation`
`edition`	`exclude`
`homepage`	`include`
`keywords`	`license`
`license-file`	`publish`
`readme`	`repository`
`rust-version`	`version`

- `license-file` and `readme` are relative to the workspace root
- `include` and `exclude` are relative to your package root

Example:

```
# [PROJECT_DIR]/Cargo.toml
[workspace]
members = ["bar"]

[workspace.package]
version = "1.2.3"
authors = ["Nice Folks"]
description = "A short description of my package"
documentation = "https://example.com/bar"
```

```
# [PROJECT_DIR]/bar/Cargo.toml
[package]
```

```
name = "bar"
version.workspace = true
authors.workspace = true
description.workspace = true
documentation.workspace = true
```

MSRV: Requires 1.64+

The `dependencies` table

The `workspace.dependencies` table is where you define dependencies to be inherited by members of a workspace.

Specifying a workspace dependency is similar to package dependencies except:

- Dependencies from this table cannot be declared as `optional`
- `features` declared in this table are additive with the `features` from `[dependencies]`

You can then inherit the workspace dependency as a package dependency

Example:

```
# [PROJECT_DIR]/Cargo.toml
[workspace]
members = ["bar"]

[workspace.dependencies]
cc = "1.0.73"
rand = "0.8.5"
regex = { version = "1.6.0", default-features = false,
features = ["std"] }
# [PROJECT_DIR]/bar/Cargo.toml
[package]
name = "bar"
version = "0.2.0"

[dependencies]
regex = { workspace = true, features = ["unicode"] }

[build-dependencies]
cc.workspace = true
```

```
[dev-dependencies]
rand.workspace = true
```

MSRV: Requires 1.64+

The `lints` table

The `workspace.lints` table is where you define lint configuration to be inherited by members of a workspace.

Specifying a workspace lint configuration is similar to package lints.

Example:

```
# [PROJECT_DIR]/Cargo.toml
[workspace]
members = ["crates/*"]

[workspace.lints.rust]
unsafe_code = "forbid"
```

```
# [PROJECT_DIR]/crates/bar/Cargo.toml
[package]
name = "bar"
version = "0.1.0"

[lints]
workspace = true
```

MSRV: Respected as of 1.74

The `metadata` table

The `workspace.metadata` table is ignored by Cargo and will not be warned about. This section can be used for tools that would like to store workspace configuration in `Cargo.toml`. For example:

```
[workspace]
members = ["member1", "member2"]

[workspace.metadata.webcontents]
root = "path/to/webproject"
tool = ["npm", "run", "build"]
# ...
```

There is a similar set of tables at the package level at `package.metadata`. While cargo does not specify a format for the content of either of these tables, it is suggested that external tools may wish to use them in a consistent fashion, such as referring to the data in `workspace.metadata` if data is missing from `package.metadata`, if that makes sense for the tool in question.

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 version string `"0.1.12"` is called a version requirement. It specifies a range of versions that can be selected from when resolving dependencies. In this case, `"0.1.12"` represents the version range `>=0.1.12, <0.2.0`. An update is allowed if it is within that range. In this case, if we ran `cargo update 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`.

Version requirement syntax

Default requirements

Default requirements specify a minimum version with the ability to update to SemVer compatible versions. Versions are considered compatible if their left-most non-zero major/minor/patch component is the same. This is different from SemVer which considers all pre-1.0.0 packages to be incompatible.

`1.2.3` is an example of a default requirement.

```
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
```

Caret requirements

Caret requirements are the default version requirement strategy. This version strategy allows SemVer compatible updates. They are specified as version requirements with a leading caret (`^`).

`^1.2.3` is an example of a caret requirement.

Leaving off the caret is a simplified equivalent syntax to using caret requirements. While caret requirements are the default, it is recommended to use the simplified syntax when possible.

`log = "^1.2.3"` is exactly equivalent to `log = "1.2.3"`.

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
```

> **Note**: crates.io does not allow bare `*` versions.

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 version requirements

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

Pre-releases

Version requirements exclude pre-release versions, such as `1.0.0-alpha`, unless specifically asked for. For example, if `1.0.0-alpha` of package `foo` is published, then a requirement of `foo = "1.0"` will *not* match, and will return an error. The pre-release must be specified, such as

`foo = "1.0.0-alpha"`. Similarly `cargo install` will avoid pre-releases unless explicitly asked to install one.

Cargo allows "newer" pre-releases to be used automatically. For example, if `1.0.0-beta` is published, then a requirement `foo = "1.0.0-alpha"` will allow updating to the `beta` version. Note that this only works on the same release version, `foo = "1.0.0-alpha"` will not allow updating to `foo = "1.0.1-alpha"` or `foo = "1.0.1-beta"`.

Cargo will also upgrade automatically to semver-compatible released versions from prereleases. The requirement `foo = "1.0.0-alpha"` will allow updating to `foo = "1.0.0"` as well as `foo = "1.2.0"`.

Beware that pre-release versions can be unstable, and as such care should be taken when using them. Some projects may choose to publish breaking changes between pre-release versions. It is recommended to not use pre-release dependencies in a library if your library is not also a pre-release. Care should also be taken when updating your `Cargo.lock`, and be prepared if a pre-release update causes issues.

Version metadata

Version metadata, such as `1.0.0+21AF26D3`, is ignored and should not be used in version requirements.

> **Recommendation:** When in doubt, use the default version requirement operator.
>
> In rare circumstances, a package with a "public dependency" (re-exports the dependency or interoperates with it in its public API) that is compatible with multiple semver-incompatible versions (e.g. only uses a simple type that hasn't changed between releases, like an `Id`) may support users choosing which version of the "public dependency" to use. In this case, a version requirement like `">=0.4, <2"` may be of interest. *However* users of the package will likely run into errors and need to manually select a version of the "public dependency" via `cargo update` if they also depend on it as Cargo might pick different versions of the "public dependency" when resolving dependency versions (see #10599).

Avoid constraining the upper bound of a version to be anything less than the next semver incompatible version (e.g. avoid `">=2.0, <2.4"`, `"2.0.*"`, or `~2.0`), as other packages in the dependency tree may require a newer version, leading to an unresolvable error (see #9029). Consider whether controlling the version in your `Cargo.lock` would be more appropriate.

In some instances this won't matter or the benefits might outweigh the cost, including:

- When no one else depends on your package; e.g. it only has a `[[bin]]`
- When depending on a pre-release package and wishing to avoid breaking changes, then a fully specified `"=1.2.3-alpha.3"` might be warranted (see #2222)
- When a library re-exports a proc-macro but the proc-macro generates code that calls into the re-exporting library, then a fully specified `=1.2.3` might be warranted to ensure the proc-macro isn't newer than the re-exporting library and generating code that uses parts of the API that don't exist within the current version

Specifying dependencies from other registries

To specify a dependency from a registry other than crates.io set the `registry` key to the name of the registry to use:

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

where `my-registry` is the registry name configured in `.cargo/config.toml` file. See the registries documentation for more information.

Note: crates.io does not allow packages to be published with dependencies on code published outside of crates.io.

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]
regex = { git = "https://github.com/rust-lang/regex.git" }
```

Cargo fetches the `git` repository at that location and traverses the file tree to find `Cargo.toml` file for the requested crate anywhere inside the `git` repository. For example, `regex-lite` and `regex-syntax` are members of `rust-lang/regex` repo and can be referred to by the repo's root URL (`https://github.com/rust-lang/regex.git`) regardless of where in the file tree they reside.

```
regex-lite = { git = "https://github.com/rust-
lang/regex.git" }
regex-syntax = { git = "https://github.com/rust-
lang/regex.git" }
```

The above rule does not apply to `path` dependencies.

Choice of commit

Cargo assumes that we intend to use the latest commit on the default branch to build our package if we only specify the repo URL, as in the examples above.

You can combine the `git` key with the `rev`, `tag`, or `branch` keys to be more specific about which commit to use. Here's an example of using the latest commit on a branch named `next`:

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

Anything that is not a branch or a tag falls under `rev` key. This can be a commit hash like `rev = "4c59b707"`, or a named reference exposed by the remote repository such as `rev = "refs/pull/493/head"`.

What references are available for the `rev` key varies by where the repo is hosted.
GitHub exposes a reference to the most recent commit of every pull request as in the example above. Other git hosts may provide something equivalent under a different naming scheme.

More `git` dependency examples:

```
# .git suffix can be omitted if the host accepts such URLs -
both examples work the same
regex = { git = "https://github.com/rust-lang/regex" }
regex = { git = "https://github.com/rust-lang/regex.git" }

# a commit with a particular tag
regex = { git = "https://github.com/rust-lang/regex.git", tag
= "1.10.3" }

# a commit by its SHA1 hash
regex = { git = "https://github.com/rust-lang/regex.git", rev
= "0c0990399270277832fbb5b91a1fa118e6f63dba" }

# HEAD commit of PR 493
regex = { git = "https://github.com/rust-lang/regex.git", rev
= "refs/pull/493/head" }

# INVALID EXAMPLES

# specifying the commit after # ignores the commit ID and
generates a warning
regex = { git = "https://github.com/rust-
lang/regex.git#4c59b70" }

# git and path cannot be used at the same time
regex = { git = "https://github.com/rust-
lang/regex.git#4c59b70", path = "../regex" }
```

Cargo locks the commits of `git` dependencies in `Cargo.lock` file at the time of their addition and checks for updates only when you run `cargo update` command.

The role of the `version` key

The `version` key always implies that the package is available in a registry, regardless of the presence of `git` or `path` keys.

The `version` key does *not* affect which commit is used when Cargo retrieves the `git` dependency, but Cargo checks the version information in the dependency's `Cargo.toml` file against the `version` key and raises an error if the check fails.

In this example, Cargo retrieves the HEAD commit of the branch called `next` from Git and checks if the crate's version is compatible with `version = "1.10.3"`:

```
[dependencies]
regex = { version = "1.10.3", git = "https://github.com/rust-lang/regex.git", branch = "next" }
```

`version`, `git`, and `path` keys are considered separate locations for resolving the dependency. See Multiple locations section below for detailed explanations.

> **Note**: crates.io does not allow packages to be published with dependencies on code published outside of crates.io itself (dev-dependencies are ignored). See the Multiple locations section for a fallback alternative for `git` and `path` dependencies.

Accessing private Git repositories

See Git Authentication for help with Git authentication for private repos.

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 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. 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` file it's written in.

The next `cargo build` will automatically build `hello_utils` and all of its dependencies.

No local path traversal

The local paths must point to the exact folder with the dependency's `Cargo.toml`. Unlike with `git` dependencies, Cargo does not traverse local paths. For example, if `regex-lite` and `regex-syntax` are members of a locally cloned `rust-lang/regex` repo, they have to be referred to by the full path:

```
# git key accepts the repo root URL and Cargo traverses the
tree to find the crate
[dependencies]
regex-lite = { git = "https://github.com/rust-
lang/regex.git" }
```

```
regex-syntax    =    {    git    =    "https://github.com/rust-
lang/regex.git" }

# path key requires the member name to be included in the
local path
[dependencies]
regex-lite   = { path = "../regex/regex-lite" }
regex-syntax = { path = "../regex/regex-syntax" }
```

Local paths in published crates

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 as a separate crate and specify its version in the dependencies line of `hello_world`:

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

The use of `path` and `version` keys together is explained in the Multiple locations section.

> **Note**: crates.io does not allow packages to be published with dependencies on code outside of crates.io, except for dev-dependencies. See the Multiple locations section for a fallback alternative for `git` and `path` dependencies.

Multiple locations

It is possible to specify both a registry version and a `git` or `path` location. The `git` or `path` dependency will be used locally (in which case the `version` is checked against the local copy), and when published to a registry like crates.io, it will use the registry version. Other combinations are not allowed. Examples:

```
[dependencies]
# Uses `my-bitflags` when used locally, and uses
# version 1.0 from crates.io when published.
bitflags = { path = "my-bitflags", version = "1.0" }

# Uses the given git repo when used locally, and uses
# version 1.0 from crates.io when published.
smallvec = { git = "https://github.com/servo/rust-
smallvec.git", version = "1.0" }

# Note: if a version doesn't match, Cargo will fail to
compile!
```

One example where this can be useful is when you have split up a library into multiple packages within the same workspace. You can then use `path` dependencies to point to the local packages within the workspace to use the local version during development, and then use the crates.io version once it is published. This is similar to specifying an override, but only applies to this one dependency declaration.

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-i686 = { path = "native/i686" }

[target.'cfg(target_arch = "x86_64")'.dependencies]
native-x86_64 = { 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 = "fancy-feature")'.dependencies]` to add dependencies based on optional features. Use the `[features]` section instead:

```
[dependencies]
foo = { version = "1.0", optional = true }
bar = { version = "1.0", optional = true }

[features]
fancy-feature = ["foo", "bar"]
```

The same applies to `cfg(debug_assertions)`, `cfg(test)` and `cfg(proc_macro)`. These values will not work as expected and will always have the default value returned by `rustc --print=cfg`. There is currently no way to add dependencies based on these configuration values.

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"
```

Custom target specifications

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" }
```

Note: Custom target specifications are not usable on the stable channel.

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"
```

> **Note**: When a package is published, only dev-dependencies that specify a `version` will be included in the published crate. For most use cases, dev-dependencies are not needed when published, though some users (like OS packagers) may want to run tests within a crate, so providing a `version` if possible can still be beneficial.

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"
```

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

```
[target.'cfg(unix)'.build-dependencies]
cc = "1.0.3"
```

In this case, the dependency will only be built when the host platform matches the specified target.

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 features chapter.

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.git",
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]
bar = { version = "0.1", package = 'foo', optional = true }
```

you're depending on the crate `foo` from crates.io, but your crate has a `bar` feature instead of a `foo` 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 = ['bar/log-debug'] # using 'foo/log-debug' would be
an error!
```

Inheriting a dependency from a workspace

Dependencies can be inherited from a workspace by specifying the dependency in the workspace's `[workspace.dependencies]` table. After that, add it to the `[dependencies]` table with `workspace = true`.

Along with the `workspace` key, dependencies can also include these keys:

- `optional`: Note that the `[workspace.dependencies]` table is not allowed to specify `optional`.
- `features`: These are additive with the features declared in the `[workspace.dependencies]`

Other than `optional` and `features`, inherited dependencies cannot use any other dependency key (such as `version` or `default-features`).

Dependencies in the `[dependencies]`, `[dev-dependencies]`, `[build-dependencies]`, and `[target."...".dependencies]` sections support the ability to reference the `[workspace.dependencies]` definition of dependencies.

```
[package]
name = "bar"
version = "0.2.0"

[dependencies]
regex = { workspace = true, features = ["unicode"] }

[build-dependencies]
cc.workspace = true

[dev-dependencies]
rand = { workspace = true, optional = true }
```

Overriding Dependencies

The desire to override a dependency 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 can be solved with the `[patch]` manifest section.

This chapter walks through a few different use cases, and includes details on the different ways to override a dependency.

- `paths` overrides

> **Note**: See also specifying a dependency with multiple locations, which can be used to override the source for a single dependency declaration in a local package.

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"

[dependencies]
uuid = "1.0"
```

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

```
$ git clone https://github.com/uuid-rs/uuid.git
```

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 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/uuid-rs/uuid.git' }
```

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"

[dependencies]
uuid = "1.0.1"

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

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"

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

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

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]`. For example, if the dependency is a git dependency, you can override it to a local path with:

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

And that's it!

Prepublishing a breaking change

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/uuid-rs/uuid.git", 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"

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

[patch.crates-io]
uuid = { git = 'https://github.com/uuid-rs/uuid.git', 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 forced to update everything all at once.

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.git' }
serde2 = { git = 'https://github.com/example/serde.git',
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 `[patch]` section

The `[patch]` 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.git' }
bar = { path = 'my/local/bar' }

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

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

Note: The `[patch]` table can also be specified as a configuration option, such as in a `.cargo/config.toml` file or a CLI option like `--config 'patch.crates-io.rand.path="rand"'`. This can be useful for local-only changes that you don't want to commit, or temporarily testing a patch.

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.

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

The `[replace]` section

Note: `[replace]` is deprecated. You should use the `[patch]` table instead.

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.git' }
"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 3-part version number is required). 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).

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

`paths` overrides

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.toml` instead of `Cargo.toml`. Inside of `.cargo/config.toml` 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.

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 `[patch]` key, and private registry support is described in the Registries chapter.

When using source replacement, running commands that need to contact the registry directly[1] requires passing the `--registry` option. This helps avoid any ambiguity about which registry to contact, and will use the authentication token for the specified registry.

1

Examples of such commands are in Publishing Commands.

Configuration

Configuration of replacement sources is done through `.cargo/config.toml` 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.toml`
# 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.
#
# The `replace-with` key can also reference an alternative registry name
# defined in the `[registries]` table.
[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 by 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.

Dependency Resolution

One of Cargo's primary tasks is to determine the versions of dependencies to use based on the version requirements specified in each package. This process is called "dependency resolution" and is performed by the "resolver". The result of the resolution is stored in the `Cargo.lock` file which "locks" the dependencies to specific versions, and keeps them fixed over time. The `cargo tree` command can be used to visualize the result of the resolver.

Constraints and Heuristics

In many cases there is no single "best" dependency resolution. The resolver operates under various constraints and heuristics to find a generally applicable resolution. To understand how these interact, it is helpful to have a coarse understanding of how dependency resolution works.

This pseudo-code approximates what Cargo's resolver does:

```
pub fn resolve(workspace: &[Package], policy: Policy) ->
Option<ResolveGraph> {
    let dep_queue = Queue::new(workspace);
    let resolved = ResolveGraph::new();
    resolve_next(pkq_queue, resolved, policy)
}

fn resolve_next(dep_queue: Queue, resolved: ResolveGraph,
policy: Policy) -> Option<ResolveGraph> {
     let Some(dep_spec) = policy.pick_next_dep(dep_queue) else
{
        // Done
        return Some(resolved);
    };

    if let Some(resolved) = policy.try_unify_version(dep_spec,
resolved.clone()) {
        return Some(resolved);
    }

    let dep_versions = dep_spec.lookup_versions()?;
      let mut dep_versions = policy.filter_versions(dep_spec,
dep_versions);
                    while     let     Some(dep_version)     =
policy.pick_next_version(&mut dep_versions) {
              if policy.needs_version_unification(dep_version,
&resolved) {
            continue;
```

```
        }

        let mut dep_queue = dep_queue.clone();
        dep_queue.enqueue(dep_version.dependencies);
        let mut resolved = resolved.clone();
        resolved.register(dep_version);
            if let Some(resolved) = resolve_next(dep_queue,
resolved) {
            return Some(resolved);
        }
    }

           // No valid solution found, backtrack and
`pick_next_version`
    None
}
```

Key steps:

- Walking dependencies (`pick_next_dep`): The order dependencies are walked can affect how related version requirements for the same dependency get resolved, see unifying versions, and how much the resolver backtracks, affecting resolver performance,
- Unifying versions (`try_unify_version`, `needs_version_unification`): Cargo reuses versions where possible to reduce build times and allow types from common dependencies to be passed between APIs. If multiple versions would have been unified if it wasn't for conflicts in their dependency specifications, Cargo will backtrack, erroring if no solution is found, rather than selecting multiple versions. A dependency specification or Cargo may decide that a version is undesirable, preferring to backtrack or error rather than use it.
- Preferring versions (`pick_next_version`): Cargo may decide that it should prefer a specific version, falling back to the next version when backtracking.

Version numbers

Generally, Cargo prefers the highest version currently available.

For example, if you had a package in the resolve graph with:

```
[dependencies]
bitflags = "*"
```

If at the time the `Cargo.lock` file is generated, the greatest version of `bitflags` is `1.2.1`, then the package will use `1.2.1`.

For an example of a possible exception, see Rust version.

Version requirements

Package specify what versions they support, rejecting all others, through version requirements.

For example, if you had a package in the resolve graph with:

```
[dependencies]
bitflags = "1.0"  # meaning `>=1.0.0,<2.0.0`
```

If at the time the `Cargo.lock` file is generated, the greatest version of `bitflags` is `1.2.1`, then the package will use `1.2.1` because it is the greatest within the compatibility range. If `2.0.0` is published, it will still use `1.2.1` because `2.0.0` is considered incompatible.

SemVer compatibility

Cargo assumes packages follow SemVer and will unify dependency versions if they are SemVer compatible according to the Caret version requirements. If two compatible versions cannot be unified because of conflicting version requirements, Cargo will error.

See the SemVer Compatibility chapter for guidance on what is considered a "compatible" change.

Examples:

The following two packages will have their dependencies on `bitflags` unified because any version picked will be compatible with each other.

```
# Package A
[dependencies]
bitflags = "1.0"  # meaning `>=1.0.0,<2.0.0`

# Package B
[dependencies]
bitflags = "1.1"  # meaning `>=1.1.0,<2.0.0`
```

The following packages will error because the version requirements conflict, selecting two distinct compatible versions.

```
# Package A
[dependencies]
log = "=0.4.11"

# Package B
[dependencies]
log = "=0.4.8"
```

The following two packages will not have their dependencies on `rand` unified because only incompatible versions are available for each. Instead, two different versions (e.g. 0.6.5 and 0.7.3) will be resolved and built. This can lead to potential problems, see the Version-incompatibility hazards section for more details.

```
# Package A
[dependencies]
rand = "0.7"  # meaning `>=0.7.0,<0.8.0`

# Package B
[dependencies]
rand = "0.6"  # meaning `>=0.6.0,<0.7.0`
```

Generally, the following two packages will not have their dependencies unified because incompatible versions are available that satisfy the version requirements: Instead, two different versions (e.g. 0.6.5 and 0.7.3) will be resolved and built. The application of other constraints or heuristics may cause these to be unified, picking one version (e.g. 0.6.5).

```
# Package A
[dependencies]
rand = ">=0.6,<0.8.0"

# Package B
[dependencies]
rand = "0.6"  # meaning `>=0.6.0,<0.7.0`
```

Version-incompatibility hazards

When multiple versions of a crate appear in the resolve graph, this can cause problems when types from those crates are exposed by the crates using them. This is because the types and items are considered different by the Rust compiler, even if they have the same name. Libraries should take care when publishing a SemVer-incompatible version (for example, publishing `2.0.0` after `1.0.0` has been in use), particularly for libraries that are widely used.

The "semver trick" is a workaround for this problem of publishing a breaking change while retaining compatibility with older versions. The linked page goes into detail about what the problem is and how to address it. In short, when a library wants to publish a SemVer-breaking release, publish the new release, and also publish a point release of the previous version that reexports the types from the newer version.

These incompatibilities usually manifest as a compile-time error, but sometimes they will only appear as a runtime misbehavior. For example, let's say there is a common library named `foo` that ends up appearing with both version `1.0.0` and `2.0.0` in the resolve graph. If `downcast_ref` is used on an object created by a library using version `1.0.0`, and the code calling `downcast_ref` is downcasting to a type from version `2.0.0`, the downcast will fail at runtime.

It is important to make sure that if you have multiple versions of a library that you are properly using them, especially if it is ever possible for the types from different versions to be used together. The `cargo tree -d` command can be used to identify duplicate versions and where they come

from. Similarly, it is important to consider the impact on the ecosystem if you publish a SemVer-incompatible version of a popular library.

Rust version

To support developing software with a minimum supported Rust version, the resolver can take into account a dependency version's compatibility with your Rust version. This is controlled by the config field `resolver.incompatible-rust-versions`.

With the `fallback` setting, the resolver will prefer packages with a Rust version that is less than or equal to your own Rust version. For example, you are using Rust 1.85 to develop the following package:

```
[package]
name = "my-cli"
rust-version = "1.62"

[dependencies]
clap = "4.0"  # resolves to 4.0.32
```

The resolver would pick version 4.0.32 because it has a Rust version of 1.60.0.

- 4.0.0 is not picked because it is a lower version number despite it also having a Rust version of 1.60.0.
- 4.5.20 is not picked because it is incompatible with `my-cli`'s Rust version of 1.62 despite having a much higher version and it has a Rust version of 1.74.0 which is compatible with your 1.85 toolchain.

If a version requirement does not include a Rust version compatible dependency version, the resolver won't error but will instead pick a version, even if its potentially suboptimal. For example, you change the dependency on `clap`:

```
[package]
name = "my-cli"
rust-version = "1.62"
```

```
[dependencies]
clap = "4.2"  # resolves to 4.5.20
```

No version of `clap` matches that version requirement that is compatible with Rust version 1.62. The resolver will then pick an incompatible version, like 4.5.20 despite it having a Rust version of 1.74.

When the resolver selects a dependency version of a package, it does not know all the workspace members that will eventually have a transitive dependency on that version and so it cannot take into account only the Rust versions relevant for that dependency. The resolver has heuristics to find a "good enough" solution when workspace members have different Rust versions. This applies even for packages in a workspace without a Rust version.

When a workspace has members with different Rust versions, the resolver may pick a lower dependency version than necessary. For example, you have the following workspace members:

```
[package]
name = "a"
rust-version = "1.62"

[package]
name = "b"

[dependencies]
clap = "4.2"  # resolves to 4.5.20
```

Though package `b` does not have a Rust version and could use a higher version like 4.5.20, 4.0.32 will be selected because of package `a`'s Rust version of 1.62.

Or the resolver may pick too high of a version. For example, you have the following workspace members:

```
[package]
name = "a"
rust-version = "1.62"
```

```
[dependencies]
clap = "4.2"  # resolves to 4.5.20

[package]
name = "b"

[dependencies]
clap = "4.5"  # resolves to 4.5.20
```

Though each package has a version requirement for `clap` that would meet its own Rust version, because of version unification, the resolver will need to pick one version that works in both cases and that would be a version like 4.5.20.

Features

For the purpose of generating `Cargo.lock`, the resolver builds the dependency graph as-if all features of all workspace members are enabled. This ensures that any optional dependencies are available and properly resolved with the rest of the graph when features are added or removed with the `--features` command-line flag. The resolver runs a second time to determine the actual features used when *compiling* a crate, based on the features selected on the command-line.

Dependencies are resolved with the union of all features enabled on them. For example, if one package depends on the `im` package with the `serde` dependency enabled and another package depends on it with the `rayon` dependency enabled, then `im` will be built with both features enabled, and the `serde` and `rayon` crates will be included in the resolve graph. If no packages depend on `im` with those features, then those optional dependencies will be ignored, and they will not affect resolution.

When building multiple packages in a workspace (such as with `--workspace` or multiple `-p` flags), the features of the dependencies of all of those packages are unified. If you have a circumstance where you want to avoid that unification for different workspace members, you will need to build them via separate `cargo` invocations.

The resolver will skip over versions of packages that are missing required features. For example, if a package depends on version `^1` of `regex` with the `perf` feature, then the oldest version it can select is `1.3.0`, because versions prior to that did not contain the `perf` feature. Similarly, if a feature is removed from a new release, then packages that require that feature will be stuck on the older releases that contain that feature. It is discouraged to remove features in a SemVer-compatible release. Beware that optional dependencies also define an implicit feature, so removing an optional dependency or making it non-optional can cause problems, see removing an optional dependency.

Feature resolver version 2

When `resolver = "2"` is specified in `Cargo.toml` (see resolver versions below), a different feature resolver is used which uses a different algorithm for unifying features. The version `"1"` resolver will unify features for a package no matter where it is specified. The version `"2"` resolver will avoid unifying features in the following situations:

- Features for target-specific dependencies are not enabled if the target is not currently being built. For example:

```
[dependencies.common]
version = "1.0"
features = ["f1"]

[target.'cfg(windows)'.dependencies.common]
version = "1.0"
features = ["f2"]
```

  When building this example for a non-Windows platform, the `f2` feature will *not* be enabled.

- Features enabled on build-dependencies or proc-macros will not be unified when those same dependencies are used as a normal dependency. For example:

```
[dependencies]
log = "0.4"
```

```
[build-dependencies]
log = {version = "0.4", features=['std']}
```

When building the build script, the `log` crate will be built with the `std` feature. When building the library of your package, it will not enable the feature.

- Features enabled on dev-dependencies will not be unified when those same dependencies are used as a normal dependency, unless those dev-dependencies are currently being built. For example:

```
[dependencies]
serde = {version = "1.0", default-features = false}

[dev-dependencies]
serde = {version = "1.0", features = ["std"]}
```

In this example, the library will normally link against `serde` without the `std` feature. However, when built as a test or example, it will include the `std` feature. For example, `cargo test` or `cargo build --all-targets` will unify these features. Note that dev-dependencies in dependencies are always ignored, this is only relevant for the top-level package or workspace members.

`links`

The `links` field is used to ensure only one copy of a native library is linked into a binary. The resolver will attempt to find a graph where there is only one instance of each `links` name. If it is unable to find a graph that satisfies that constraint, it will return an error.

For example, it is an error if one package depends on `libgit2-sys` version `0.11` and another depends on `0.12`, because Cargo is unable to unify those, but they both link to the `git2` native library. Due to this requirement, it is encouraged to be very careful when making SemVer-incompatible releases with the `links` field if your library is in common use.

Yanked versions

Yanked releases are those that are marked that they should not be used. When the resolver is building the graph, it will ignore all yanked releases unless they already exist in the `Cargo.lock` file or are explicitly requested by the `--precise` flag of `cargo update` (nightly only).

Dependency updates

Dependency resolution is automatically performed by all Cargo commands that need to know about the dependency graph. For example, `cargo build` will run the resolver to discover all the dependencies to build. After the first time it runs, the result is stored in the `Cargo.lock` file. Subsequent commands will run the resolver, keeping dependencies locked to the versions in `Cargo.lock` *if it can*.

If the dependency list in `Cargo.toml` has been modified, for example changing the version of a dependency from `1.0` to `2.0`, then the resolver will select a new version for that dependency that matches the new requirements. If that new dependency introduces new requirements, those new requirements may also trigger additional updates. The `Cargo.lock` file will be updated with the new result. The `--locked` or `--frozen` flags can be used to change this behavior to prevent automatic updates when requirements change, and return an error instead.

`cargo update` can be used to update the entries in `Cargo.lock` when new versions are published. Without any options, it will attempt to update all packages in the lock file. The `-p` flag can be used to target the update for a specific package, and other flags such as `--recursive` or `--precise` can be used to control how versions are selected.

Overrides

Cargo has several mechanisms to override dependencies within the graph. The Overriding Dependencies chapter goes into detail on how to use overrides. The overrides appear as an overlay to a registry, replacing the patched version with the new entry. Otherwise, resolution is performed like normal.

Dependency kinds

There are three kinds of dependencies in a package: normal, build, and dev. For the most part these are all treated the same from the perspective of the resolver. One difference is that dev-dependencies for non-workspace members are always ignored, and do not influence resolution.

Platform-specific dependencies with the `[target]` table are resolved as-if all platforms are enabled. In other words, the resolver ignores the platform or `cfg` expression.

dev-dependency cycles

Usually the resolver does not allow cycles in the graph, but it does allow them for dev-dependencies. For example, project "foo" has a dev-dependency on "bar", which has a normal dependency on "foo" (usually as a "path" dependency). This is allowed because there isn't really a cycle from the perspective of the build artifacts. In this example, the "foo" library is built (which does not need "bar" because "bar" is only used for tests), and then "bar" can be built depending on "foo", then the "foo" tests can be built linking to "bar".

Beware that this can lead to confusing errors. In the case of building library unit tests, there are actually two copies of the library linked into the final test binary: the one that was linked with "bar", and the one built that contains the unit tests. Similar to the issues highlighted in the Version-incompatibility hazards section, the types between the two are not compatible. Be careful when exposing types of "foo" from "bar" in this situation, since the "foo" unit tests won't treat them the same as the local types.

If possible, try to split your package into multiple packages and restructure it so that it remains strictly acyclic.

Resolver versions

Different resolver behavior can be specified through the resolver version in `Cargo.toml` like this:

```
[package]
name = "my-package"
version = "1.0.0"
resolver = "2"
```

- `"1"` (default)
- `"2"` (`edition = "2021"` default): Introduces changes in feature unification. See the features chapter for more details.
- `"3"` (`edition = "2024"` default, requires Rust 1.84+): Change the default for `resolver.incompatible-rust-versions` from `allow` to `fallback`

The resolver is a global option that affects the entire workspace. The `resolver` version in dependencies is ignored, only the value in the top-level package will be used. If using a virtual workspace, the version should be specified in the `[workspace]` table, for example:

```
[workspace]
members = ["member1", "member2"]
resolver = "2"
```

MSRV: Requires 1.51+

Recommendations

The following are some recommendations for setting the version within your package, and for specifying dependency requirements. These are general guidelines that should apply to common situations, but of course some situations may require specifying unusual requirements.

- Follow the SemVer guidelines when deciding how to update your version number, and whether or not you will need to make a SemVer-incompatible version change.
- Use caret requirements for dependencies, such as `"1.2.3"`, for most situations. This ensures that the resolver can be maximally flexible in choosing a version while maintaining build compatibility.
    - Specify all three components with the version you are currently using. This helps set the minimum version that will be used, and ensures that other users won't end up with an older version of the dependency that might be missing something that your package requires.
    - Avoid `*` requirements, as they are not allowed on crates.io, and they can pull in SemVer-breaking changes during a normal `cargo update`.
    - Avoid overly broad version requirements. For example, `>=2.0.0` can pull in any SemVer-incompatible version, like version `5.0.0`, which can result in broken builds in the future.
    - Avoid overly narrow version requirements if possible. For example, if you specify a tilde requirement like `bar="~1.3"`, and another package specifies a requirement of `bar="1.4"`, this will fail to resolve, even though minor releases should be compatible.
- Try to keep the dependency versions up-to-date with the actual minimum versions that your library requires. For example, if you have a requirement of `bar="1.0.12"`, and then in a future release you start using new features added in the `1.1.0` release of "bar", update your dependency requirement to `bar="1.1.0"`.

If you fail to do this, it may not be immediately obvious because Cargo can opportunistically choose the newest version when you run a blanket `cargo update`. However, if another user depends on your library, and runs `cargo update your-library`, it will *not* automatically update "bar" if it is locked in their `Cargo.lock`. It will only update "bar" in that situation if the dependency declaration is also updated. Failure to do so can cause confusing build errors for the user using `cargo update your-library`.

- If two packages are tightly coupled, then an `=` dependency requirement may help ensure that they stay in sync. For example, a library with a companion proc-macro library will sometimes make assumptions between the two libraries that won't work well if the two are out of sync (and it is never expected to use the two libraries independently). The parent library can use an `=` requirement on the proc-macro, and re-export the macros for easy access.
- `0.0.x` versions can be used for packages that are permanently unstable.

In general, the stricter you make the dependency requirements, the more likely it will be for the resolver to fail. Conversely, if you use requirements that are too loose, it may be possible for new versions to be published that will break the build.

Troubleshooting

The following illustrates some problems you may experience, and some possible solutions.

Why was a dependency included?

Say you see dependency `rand` in the `cargo check` output but don't think it's needed and want to understand why it's being pulled in.

You can run

```
$ cargo tree --workspace --target all --all-features --invert
rand
rand v0.8.5
└── ...

rand v0.8.5
└── ...
```

Why was that feature on this dependency enabled?

You might identify that it was an activated feature that caused `rand` to show up. **To figure out which package activated the feature, you can add the `--edges features`**

```
$ cargo tree --workspace --target all --all-features --edges
features --invert rand
rand v0.8.5
└── ...

rand v0.8.5
└── ...
```

Unexpected dependency duplication

You see multiple instances of `rand` when you run

```
$ cargo tree --workspace --target all --all-features --
duplicates
rand v0.7.3
└── ...

rand v0.8.5
└── ...
```

The resolver algorithm has converged on a solution that includes two copies of a dependency when one would suffice. For example:

```
# Package A
[dependencies]
rand = "0.7"

# Package B
[dependencies]
rand = ">=0.6"  # note: open requirements such as this are
discouraged
```

In this example, Cargo may build two copies of the `rand` crate, even though a single copy at version `0.7.3` would meet all requirements. This is because the resolver's algorithm favors building the latest available version of `rand` for Package B, which is `0.8.5` at the time of this writing, and that is incompatible with Package A's specification. The resolver's algorithm does not currently attempt to "deduplicate" in this situation.

The use of open-ended version requirements like `>=0.6` is discouraged in Cargo. But, if you run into this situation, the `cargo update` command with the `--precise` flag can be used to manually remove such duplications.

Why wasn't a newer version selected?

Say you noticed that the latest version of a dependency wasn't selected when you ran:

```
$ cargo update
```

You can enable some extra logging to see why this happened:

```
$ env CARGO_LOG=cargo::core::resolver=trace cargo update
```

Note: Cargo log targets and levels may change over time.

SemVer-breaking patch release breaks the build

Sometimes a project may inadvertently publish a point release with a SemVer-breaking change. When users update with `cargo update`, they will pick up this new release, and then their build may break. In this situation, it is recommended that the project should yank the release, and either remove the SemVer-breaking change, or publish it as a new SemVer-major version increase.

If the change happened in a third-party project, if possible try to (politely!) work with the project to resolve the issue.

While waiting for the release to be yanked, some workarounds depend on the circumstances:

- If your project is the end product (such as a binary executable), just avoid updating the offending package in `Cargo.lock`. This can be done with the `--precise` flag in `cargo update`.
- If you publish a binary on crates.io, then you can temporarily add an `=` requirement to force the dependency to a specific good version.
    - Binary projects can alternatively recommend users to use the `--locked` flag with `cargo install` to use the original `Cargo.lock` that contains the known good version.
- Libraries may also consider publishing a temporary new release with stricter requirements that avoid the troublesome dependency. You may want to consider using range requirements (instead of `=`) to avoid overly-strict requirements that may conflict with other packages using the same dependency. Once the problem has been resolved, you can publish another point release that relaxes the dependency back to a caret requirement.
- If it looks like the third-party project is unable or unwilling to yank the release, then one option is to update your code to be compatible with the changes, and update the dependency requirement to set the

minimum version to the new release. You will also need to consider if this is a SemVer-breaking change of your own library, for example if it exposes types from the dependency.

Features

Cargo "features" provide a mechanism to express conditional compilation and optional dependencies. A package defines a set of named features in the `[features]` table of `Cargo.toml`, and each feature can either be enabled or disabled. Features for the package being built can be enabled on the command-line with flags such as `--features`. Features for dependencies can be enabled in the dependency declaration in `Cargo.toml`.

> **Note**: New crates or versions published on crates.io are now limited to a maximum of 300 features. Exceptions are granted on a case-by-case basis. See this blog post for details. Participation in solution discussions is encouraged via the crates.io Zulip stream.

See also the Features Examples chapter for some examples of how features can be used.

The `[features]` section

Features are defined in the `[features]` table in `Cargo.toml`. Each feature specifies an array of other features or optional dependencies that it enables. The following examples illustrate how features could be used for a 2D image processing library where support for different image formats can be optionally included:

```
[features]
# Defines a feature named `webp` that does not enable any
other features.
webp = []
```

With this feature defined, `cfg` expressions can be used to conditionally include code to support the requested feature at compile time. For example, inside `lib.rs` of the package could include this:

```
// This conditionally includes a module which implements WEBP
support.
#[cfg(feature = "webp")]
pub mod webp;
```

Cargo sets features in the package using the `rustc` `--cfg` flag, and code can test for their presence with the `cfg` attribute or the `cfg` macro.

Features can list other features to enable. For example, the ICO image format can contain BMP and PNG images, so when it is enabled, it should make sure those other features are enabled, too:

```
[features]
bmp = []
png = []
ico = ["bmp", "png"]
webp = []
```

Feature names may include characters from the Unicode XID standard (which includes most letters), and additionally allows starting with `_` or digits `0` through `9`, and after the first character may also contain `-`, `+`, or `.`.

Note: crates.io imposes additional constraints on feature name syntax that they must only be ASCII alphanumeric characters or `_`, `-`, or `+`.

The `default` feature

By default, all features are disabled unless explicitly enabled. This can be changed by specifying the `default` feature:

```
[features]
default = ["ico", "webp"]
bmp = []
png = []
ico = ["bmp", "png"]
webp = []
```

When the package is built, the `default` feature is enabled which in turn enables the listed features. This behavior can be changed by:

- The `--no-default-features` command-line flag disables the default features of the package.
- The `default-features = false` option can be specified in a dependency declaration.

Note: Be careful about choosing the default feature set. The default features are a convenience that make it easier to use a package without forcing the user to carefully select which features to enable for common use, but there are some drawbacks. Dependencies automatically enable default features unless `default-features = false` is specified. This can make it difficult to ensure that the default features are not enabled, especially for a dependency that appears multiple times in the dependency graph. Every package must ensure that `default-features = false` is specified to avoid enabling them.

Another issue is that it can be a SemVer incompatible change to remove a feature from the default set, so you should be confident that you will keep those features.

Optional dependencies

Dependencies can be marked "optional", which means they will not be compiled by default. For example, let's say that our 2D image processing library uses an external package to handle GIF images. This can be expressed like this:

```
[dependencies]
gif = { version = "0.11.1", optional = true }
```

By default, this optional dependency implicitly defines a feature that looks like this:

```
[features]
gif = ["dep:gif"]
```

This means that this dependency will only be included if the `gif` feature is enabled. The same `cfg(feature = "gif")` syntax can be used in the code, and the dependency can be enabled just like any feature such as `--features gif` (see Command-line feature options below).

In some cases, you may not want to expose a feature that has the same name as the optional dependency. For example, perhaps the optional dependency is an internal detail, or you want to group multiple optional dependencies together, or you just want to use a better name. If you specify the optional dependency with the `dep:` prefix anywhere in the `[features]` table, that disables the implicit feature.

Note: The `dep:` syntax is only available starting with Rust 1.60. Previous versions can only use the implicit feature name.

For example, let's say in order to support the AVIF image format, our library needs two other dependencies to be enabled:

```
[dependencies]
ravif = { version = "0.6.3", optional = true }
rgb = { version = "0.8.25", optional = true }

[features]
avif = ["dep:ravif", "dep:rgb"]
```

In this example, the `avif` feature will enable the two listed dependencies. This also avoids creating the implicit `ravif` and `rgb` features, since we don't want users to enable those individually as they are internal details to our crate.

> **Note**: Another way to optionally include a dependency is to use platform-specific dependencies. Instead of using features, these are conditional based on the target platform.

Dependency features

Features of dependencies can be enabled within the dependency declaration. The `features` key indicates which features to enable:

```
[dependencies]
# Enables the `derive` feature of serde.
serde = { version = "1.0.118", features = ["derive"] }
```

The `default` features can be disabled using `default-features = false`:

```
[dependencies]
flate2 = { version = "1.0.3", default-features = false,
features = ["zlib-rs"] }
```

> **Note**: This may not ensure the default features are disabled. If another dependency includes `flate2` without specifying `default-features = false`, then the default features will be enabled. See feature unification below for more details.

Features of dependencies can also be enabled in the `[features]` table. The syntax is `"package-name/feature-name"`. For example:

```
[dependencies]
jpeg-decoder = { version = "0.1.20", default-features = false
}

[features]
# Enables parallel processing support by enabling the "rayon"
feature of jpeg-decoder.
parallel = ["jpeg-decoder/rayon"]
```

The `"package-name/feature-name"` syntax will also enable `package-name` if it is an optional dependency. Often this is not what you want. You can add a `?` as in `"package-name?/feature-name"` which will only enable the given feature if something else enables the optional dependency.

> **Note**: The `?` syntax is only available starting with Rust 1.60.

For example, let's say we have added some serialization support to our library, and it requires enabling a corresponding feature in some optional dependencies. That can be done like this:

```
[dependencies]
serde = { version = "1.0.133", optional = true }
rgb = { version = "0.8.25", optional = true }

[features]
serde = ["dep:serde", "rgb?/serde"]
```

In this example, enabling the `serde` feature will enable the serde dependency. It will also enable the `serde` feature for the `rgb` dependency, but only if something else has enabled the `rgb` dependency.

Command-line feature options

The following command-line flags can be used to control which features are enabled:

- `--features` *FEATURES*: Enables the listed features. Multiple features may be separated with commas or spaces. If using spaces, be sure to use quotes around all the features if running Cargo from a shell (such as `--features "foo bar"`). If building multiple packages in a workspace, the `package-name/feature-name` syntax can be used to specify features for specific workspace members.
- `--all-features`: Activates all features of all packages selected on the command line.
- `--no-default-features`: Does not activate the `default` feature of the selected packages.

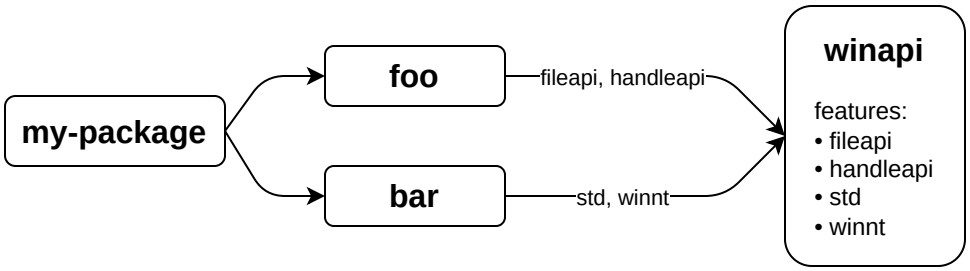
NOTE: check the individual subcommand documentation for details. Not all flags are available for all subcommands.

Feature unification

Features are unique to the package that defines them. Enabling a feature on a package does not enable a feature of the same name on other packages.

When a dependency is used by multiple packages, Cargo will use the union of all features enabled on that dependency when building it. This helps ensure that only a single copy of the dependency is used. See the features section of the resolver documentation for more details.

For example, let's look at the `winapi` package which uses a large number of features. If your package depends on a package `foo` which enables the "fileapi" and "handleapi" features of `winapi`, and another dependency `bar` which enables the "std" and "winnt" features of `winapi`, then `winapi` will be built with all four of those features enabled.



A consequence of this is that features should be *additive*. That is, enabling a feature should not disable functionality, and it should usually be safe to enable any combination of features. A feature should not introduce a SemVer-incompatible change.

For example, if you want to optionally support `no_std` environments, **do not** use a `no_std` feature. Instead, use a `std` feature that *enables* `std`. For example:

```
#![no_std]

#[cfg(feature = "std")]
extern crate std;

#[cfg(feature = "std")]
pub fn function_that_requires_std() {
```

```
    // ...
}
```

Mutually exclusive features

There are rare cases where features may be mutually incompatible with one another. This should be avoided if at all possible, because it requires coordinating all uses of the package in the dependency graph to cooperate to avoid enabling them together. If it is not possible, consider adding a compile error to detect this scenario. For example:

```
#[cfg(all(feature = "foo", feature = "bar"))]
compile_error!("feature \"foo\" and feature \"bar\" cannot be enabled at the same time");
```

Instead of using mutually exclusive features, consider some other options:

- Split the functionality into separate packages.
- When there is a conflict, choose one feature over another. The `cfg-if` package can help with writing more complex `cfg` expressions.
- Architect the code to allow the features to be enabled concurrently, and use runtime options to control which is used. For example, use a config file, command-line argument, or environment variable to choose which behavior to enable.

Inspecting resolved features

In complex dependency graphs, it can sometimes be difficult to understand how different features get enabled on various packages. The `cargo tree` command offers several options to help inspect and visualize which features are enabled. Some options to try:

- `cargo tree -e features`: This will show features in the dependency graph. Each feature will appear showing which package enabled it.
- `cargo tree -f "{p} {f}"`: This is a more compact view that shows a comma-separated list of features enabled on each package.

- `cargo tree -e features -i foo`: This will invert the tree, showing how features flow into the given package "foo". This can be useful because viewing the entire graph can be quite large and overwhelming. Use this when you are trying to figure out which features are enabled on a specific package and why. See the example at the bottom of the `cargo tree` page on how to read this.

Feature resolver version 2

A different feature resolver can be specified with the `resolver` field in `Cargo.toml`, like this:

```
[package]
name = "my-package"
version = "1.0.0"
resolver = "2"
```

See the resolver versions section for more detail on specifying resolver versions.

The version `"2"` resolver avoids unifying features in a few situations where that unification can be unwanted. The exact situations are described in the resolver chapter, but in short, it avoids unifying in these situations:

- Features enabled on platform-specific dependencies for target architectures not currently being built are ignored.
- Build-dependencies and proc-macros do not share features with normal dependencies.
- Dev-dependencies do not activate features unless building a Cargo target that needs them (like tests or examples).

Avoiding the unification is necessary for some situations. For example, if a build-dependency enables a `std` feature, and the same dependency is used as a normal dependency for a `no_std` environment, enabling `std` would break the build.

However, one drawback is that this can increase build times because the dependency is built multiple times (each with different features). When using the version `"2"` resolver, it is recommended to check for dependencies that are built multiple times to reduce overall build time. If it is not *required* to build those duplicated packages with separate features, consider adding features to the `features` list in the dependency declaration so that the duplicates end up with the same features (and thus Cargo will build it only once). You can detect these duplicate dependencies with the `cargo tree --duplicates` command. It will show which packages are

built multiple times; look for any entries listed with the same version. See Inspecting resolved features for more on fetching information on the resolved features. For build dependencies, this is not necessary if you are cross-compiling with the `--target` flag because build dependencies are always built separately from normal dependencies in that scenario.

Resolver version 2 command-line flags

The `resolver = "2"` setting also changes the behavior of the `--features` and `--no-default-features` command-line options.

With version `"1"`, you can only enable features for the package in the current working directory. For example, in a workspace with packages `foo` and `bar`, and you are in the directory for package `foo`, and ran the command `cargo build -p bar --features bar-feat`, this would fail because the `--features` flag only allowed enabling features on `foo`.

With `resolver = "2"`, the features flags allow enabling features for any of the packages selected on the command-line with `-p` and `--workspace` flags. For example:

```
# This command is allowed with resolver = "2", regardless of
which directory
# you are in.
cargo build -p foo -p bar --features foo-feat,bar-feat

# This explicit equivalent works with any resolver version:
cargo build -p foo -p bar --features foo/foo-feat,bar/bar-feat
```

Additionally, with `resolver = "1"`, the `--no-default-features` flag only disables the default feature for the package in the current directory. With version "2", it will disable the default features for all workspace members.

Build scripts

Build scripts can detect which features are enabled on the package by inspecting the `CARGO_FEATURE_<name>` environment variable, where `<name>` is the feature name converted to uppercase and `-` converted to `_`.

Required features

The `required-features` field can be used to disable specific Cargo targets if a feature is not enabled. See the linked documentation for more details.

SemVer compatibility

Enabling a feature should not introduce a SemVer-incompatible change. For example, the feature shouldn't change an existing API in a way that could break existing uses. More details about what changes are compatible can be found in the SemVer Compatibility chapter.

Care should be taken when adding and removing feature definitions and optional dependencies, as these can sometimes be backwards-incompatible changes. More details can be found in the Cargo section of the SemVer Compatibility chapter. In short, follow these rules:

- The following is usually safe to do in a minor release:
    - Add a new feature or optional dependency.
    - Change the features used on a dependency.
- The following should usually **not** be done in a minor release:
    - Remove a feature or optional dependency.
    - Moving existing public code behind a feature.
    - Remove a feature from a feature list.

See the links for caveats and examples.

Feature documentation and discovery

You are encouraged to document which features are available in your package. This can be done by adding doc comments at the top of `lib.rs`. As an example, see the regex crate source, which when rendered can be viewed on docs.rs. If you have other documentation, such as a user guide, consider adding the documentation there (for example, see serde.rs). If you have a binary project, consider documenting the features in the README or other documentation for the project (for example, see sccache).

Clearly documenting the features can set expectations about features that are considered "unstable" or otherwise shouldn't be used. For example, if there is an optional dependency, but you don't want users to explicitly list that optional dependency as a feature, exclude it from the documented list.

Documentation published on docs.rs can use metadata in `Cargo.toml` to control which features are enabled when the documentation is built. See docs.rs metadata documentation for more details.

> **Note**: Rustdoc has experimental support for annotating the documentation to indicate which features are required to use certain APIs. See the `doc cfg` documentation for more details. An example is the `syn` documentation, where you can see colored boxes which note which features are required to use it.

Discovering features

When features are documented in the library API, this can make it easier for your users to discover which features are available and what they do. If the feature documentation for a package isn't readily available, you can look at the `Cargo.toml` file, but sometimes it can be hard to track it down. The crate page on crates.io has a link to the source repository if available. Tools like `cargo vendor` or cargo-clone-crate can be used to download the source and inspect it.

Feature combinations

Because features are a form of conditional compilation, they require an exponential number of configurations and test cases to be 100% covered. By default, tests, docs, and other tooling such as Clippy will only run with the default set of features.

We encourage you to consider your strategy and tooling in regards to different feature combinations --- Every project will have different requirements in conjunction with time, resources, and the cost-benefit of covering specific scenarios. Common configurations may be with / without default features, specific combinations of features, or all combinations of features.

Features Examples

The following illustrates some real-world examples of features in action.

Minimizing build times and file sizes

Some packages use features so that if the features are not enabled, it reduces the size of the crate and reduces compile time. Some examples are:

- `syn` is a popular crate for parsing Rust code. Since it is so popular, it is helpful to reduce compile times since it affects so many projects. It has a clearly documented list of features which can be used to minimize the amount of code it contains.
- `regex` has a several features that are well documented. Cutting out Unicode support can reduce the resulting file size as it can remove some large tables.
- `winapi` has a large number of features that limit which Windows API bindings it supports.
- `web-sys` is another example similar to `winapi` that provides a huge surface area of API bindings that are limited by using features.

Extending behavior

The `serde_json` package has a `preserve_order` feature which changes the behavior of JSON maps to preserve the order that keys are inserted. Notice that it enables an optional dependency `indexmap` to implement the new behavior.

When changing behavior like this, be careful to make sure the changes are SemVer compatible. That is, enabling the feature should not break code that usually builds with the feature off.

`no_std` support

Some packages want to support both `no_std` and `std` environments. This is useful for supporting embedded and resource-constrained platforms, but still allowing extended capabilities for platforms that support the full standard library.

The `wasm-bindgen` package defines a `std` feature that is enabled by default. At the top of the library, it unconditionally enables the `no_std` attribute. This ensures that `std` and the `std` prelude are not automatically in scope. Then, in various places in the code (example1, example2), it uses `#[cfg(feature = "std")]` attributes to conditionally enable extra functionality that requires `std`.

Re-exporting dependency features

It can be convenient to re-export the features from a dependency. This allows the user depending on the crate to control those features without needing to specify those dependencies directly. For example, `regex` re-exports the features from the `regex_syntax` package. Users of `regex` don't need to know about the `regex_syntax` package, but they can still access the features it contains.

Vendoring of C libraries

Some packages provide bindings to common C libraries (sometimes referred to as "sys" crates). Sometimes these packages give you the choice to use the C library installed on the system, or to build it from source. For example, the `openssl` package has a `vendored` feature which enables the corresponding `vendored` feature of `openssl-sys`. The `openssl-sys` build script has some conditional logic which causes it to build from a local copy of the OpenSSL source code instead of using the version from the system.

The `curl-sys` package is another example where the `static-curl` feature causes it to build libcurl from source. Notice that it also has a `force-system-lib-on-osx` feature which forces it to use the system libcurl, overriding the static-curl setting.

Feature precedence

Some packages may have mutually-exclusive features. One option to handle this is to prefer one feature over another. The `log` package is an example. It has several features for choosing the maximum logging level at compile-time described here. It uses `cfg-if` to choose a precedence. If multiple features are enabled, the higher "max" levels will be preferred over the lower levels.

Proc-macro companion package

Some packages have a proc-macro that is intimately tied with it. However, not all users will need to use the proc-macro. By making the proc-macro an optional-dependency, this allows you to conveniently choose whether or not it is included. This is helpful, because sometimes the proc-macro version must stay in sync with the parent package, and you don't want to force the users to have to specify both dependencies and keep them in sync.

An example is `serde` which has a `derive` feature which enables the `serde_derive` proc-macro. The `serde_derive` crate is very tightly tied to `serde`, so it uses an equals version requirement to ensure they stay in sync.

Nightly-only features

Some packages want to experiment with APIs or language features that are only available on the Rust nightly channel. However, they may not want to require their users to also use the nightly channel. An example is `wasm-bindgen` which has a `nightly` feature which enables an extended API that uses the `Unsize` marker trait that is only available on the nightly channel at the time of this writing.

Note that at the root of the crate it uses `cfg_attr` to enable the nightly feature. Keep in mind that the `feature` attribute is unrelated to Cargo features, and is used to opt-in to experimental language features.

The `simd_support` feature of the `rand` package is another example, which relies on a dependency that only builds on the nightly channel.

Experimental features

Some packages have new functionality that they may want to experiment with, without having to commit to the stability of those APIs. The features are usually documented that they are experimental, and thus may change or break in the future, even during a minor release. An example is the `async-std` package, which has an `unstable` feature, which gates new APIs that people can opt-in to using, but may not be completely ready to be relied upon.

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`. The profile is automatically chosen based on which command is being run if a profile is not specified on the command-line. In addition to the built-in profiles, custom user-defined profiles can also be specified.

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.

Additionally, profiles can be overridden from a config definition. Specifying a profile in a config file or environment variable will override the settings from `Cargo.toml`.

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
- `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` change 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`, `false`, or `"none"`: no debug info at all, default for `release`

- `"line-directives-only"`: line info directives only. For the nvptx* targets this enables profiling. For other use cases, `line-tables-only` is the better, more compatible choice.
- `"line-tables-only"`: line tables only. Generates the minimal amount of debug info for backtraces with filename/line number info, but not anything else, i.e. no variable or function parameter info.
- `1` or `"limited"`: debug info without type or variable-level information. Generates more detailed module-level info than `line-tables-only`.
- `2`, `true`, or `"full"`: full debug info, default for `dev`

For more information on what each option does see `rustc`'s docs on debuginfo.

You may wish to also configure the `split-debuginfo` option depending on your needs as well.

MSRV: 1.71 is required for `none`, `limited`, `full`, `line-directives-only`, and `line-tables-only`

split-debuginfo

The `split-debuginfo` setting controls the `-C split-debuginfo` flag which controls whether debug information, if generated, is either placed in the executable itself or adjacent to it.

This option is a string and acceptable values are the same as those the compiler accepts. The default value for this option is `unpacked` on macOS for profiles that have debug information otherwise enabled. Otherwise the default for this option is documented with rustc and is platform-specific. Some options are only available on the nightly channel. The Cargo default may change in the future once more testing has been performed, and support for DWARF is stabilized.

Be aware that Cargo and rustc have different defaults for this option. This option exists to allow Cargo to experiment on different combinations of flags thus providing better debugging and developer experience.

strip

The `strip` option controls the `-C strip` flag, which directs rustc to strip either symbols or debuginfo from a binary. This can be enabled like so:

```
[package]
# ...

[profile.release]
strip = "debuginfo"
```

Possible string values of `strip` are `"none"`, `"debuginfo"`, and `"symbols"`. The default is `"none"`.

You can also configure this option with the boolean values `true` or `false`. `strip = true` is equivalent to `strip = "symbols"`. `strip = false` is equivalent to `strip = "none"` and disables `strip` completely.

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 `rustc`'s `-C lto`, `-C linker-plugin-lto`, and `-C embed-bitcode` options, which control 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 the linker-plugin-lto chapter if you are interested in cross-language LTO. This is not yet supported natively in Cargo, but can be performed via `RUSTFLAGS`.

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 build 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 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.

The default is 256 for incremental builds, and 16 for non-incremental builds.

rpath

The `rpath` setting controls 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`, and is used for `cargo install --debug`.

The default settings for the `dev` profile are:

```
[profile.dev]
opt-level = 0
debug = true
split-debuginfo = '...'  # Platform-specific.
strip = "none"
debug-assertions = true
overflow-checks = true
lto = false
panic = 'unwind'
incremental = true
codegen-units = 256
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
split-debuginfo = '...'  # Platform-specific.
strip = "none"
debug-assertions = false
overflow-checks = false
```

```
lto = false
panic = 'unwind'
incremental = false
codegen-units = 16
rpath = false
```

test

The `test` profile is the default profile used by `cargo test`. The `test` profile inherits the settings from the `dev` profile.

bench

The `bench` profile is the default profile used by `cargo bench`. The `bench` profile inherits the settings from the `release` profile.

Build Dependencies

To compile quickly, all profiles, by default, do not optimize build dependencies (build scripts, proc macros, and their dependencies), and avoid computing debug info when a build dependency is not used as a runtime dependency. The default settings for build overrides are:

```
[profile.dev.build-override]
opt-level = 0
codegen-units = 256
debug = false # when possible

[profile.release.build-override]
opt-level = 0
codegen-units = 256
```

However, if errors occur while running build dependencies, turning full debug info on will improve backtraces and debuggability when needed:

```
debug = true
```

Build dependencies otherwise inherit settings from the active profile in use, as described in Profile selection.

Custom profiles

In addition to the built-in profiles, additional custom profiles can be defined. These may be useful for setting up multiple workflows and build modes. When defining a custom profile, you must specify the `inherits` key to specify which profile the custom profile inherits settings from when the setting is not specified.

For example, let's say you want to compare a normal release build with a release build with LTO optimizations, you can specify something like the following in `Cargo.toml`:

```
[profile.release-lto]
inherits = "release"
lto = true
```

The `--profile` flag can then be used to choose this custom profile:

```
cargo build --profile release-lto
```

The output for each profile will be placed in a directory of the same name as the profile in the `target` directory. As in the example above, the output would go into the `target/release-lto` directory.

Profile selection

The profile used depends on the command, the command-line flags like `--release` or `--profile`, and the package (in the case of overrides). The default profile if none is specified is:

Command	Default Profile
`cargo run`, `cargo build`, `cargo check`, `cargo rustc`	`dev` profile
`cargo test`	`test` profile
`cargo bench`	`bench` profile
`cargo install`	`release` profile

You can switch to a different profile using the `--profile=NAME` option which will used the given profile. The `--release` flag is equivalent to `--profile=release`.

The selected profile applies to all Cargo targets, including library, binary, example, test, and benchmark.

The profile for specific packages can be specified with overrides, described below.

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

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 with higher precedence items being placed later in the merged array.

At present, when being invoked from a workspace, Cargo does not read config files from crates within the workspace. i.e. if a workspace has two crates in it, named `/projects/foo/bar/baz/mylib` and `/projects/foo/bar/baz/mybin`, and there are Cargo configs at `/projects/foo/bar/baz/mylib/.cargo/config.toml` and `/projects/foo/bar/baz/mybin/.cargo/config.toml`, Cargo does not read those configuration files if it is invoked from the workspace root (`/projects/foo/bar/baz/`).

Note: Cargo also reads config files without the `.toml` extension, such as `.cargo/config`. Support for the `.toml` extension was added in version 1.39 and is the preferred form. If both files exist, Cargo will use the file without the extension.

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"
recursive_example = "rr --example recursions"
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`
rustc-workspace-wrapper = "…" # run this wrapper instead of
`rustc` for workspace members
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

[credential-alias]
# Provides a way to define aliases for credential providers.
my-alias = ["/usr/bin/cargo-credential-example", "--argument",
"value", "--flag"]

[doc]
browser = "chromium"          # browser to use with `cargo doc
--open`,
                                    # overrides the `BROWSER`
environment variable

[env]
# Set ENV_VAR_NAME=value for any process run by Cargo
ENV_VAR_NAME = "value"
# Set even if already present in environment
ENV_VAR_NAME_2 = { value = "value", force = true }
# `value` is relative to the parent of `.cargo/config.toml`,
env var will be the full absolute path
ENV_VAR_NAME_3 = { value = "relative/path", relative = true }

[future-incompat-report]
frequency = 'always' # when to display a notification about a
future incompat report

[cache]
auto-clean-frequency  =  "1  day"     #  How  often  to  perform
automatic cache cleaning

[cargo-new]
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
proxy-cainfo = "cert.pem"     # path to proxy 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 = 3                   # network retries
git-fetch-with-cli = true   # use the `git` executable for git operations
offline = true              # do not access the network

[net.ssh]
known-hosts = ["..."]       # known SSH host keys

[patch.<registry>]
# Same keys as for [patch] in Cargo.toml

[profile.<name>]         # Modify profile settings via config.
inherits = "dev"               # Inherits settings from
```

```
[profile.dev].
opt-level = 0            # Optimization level.
debug = true             # Include debug info.
split-debuginfo = '...'  # Debug info splitting behavior.
strip = "none"           # Removes symbols or debuginfo.
debug-assertions = true  # Enables debug assertions.
overflow-checks = true    # Enables runtime integer overflow
checks.
lto = false              # Sets link-time optimization.
panic = 'unwind'         # The panic strategy.
incremental = true       # Incremental compilation.
codegen-units = 16       # Number of code generation units.
rpath = false            # Sets the rpath linking option.
[profile.<name>.build-override]    #  Overrides  build-script
settings.
# Same keys for a normal profile.
[profile.<name>.package.<name>]    #  Override  profile  for  a
package.
# Same keys for a normal profile (minus `panic`, `lto`, and
`rpath`).

[resolver]
incompatible-rust-versions = "allow"  # Specifies how resolver
reacts to these

[registries.<name>]  # registries other than crates.io
index = "…"          # URL of the registry index
token = "…"          # authentication token for the registry
credential-provider = "cargo:token" # The credential provider
for this registry.

[registries.crates-io]
protocol  =  "sparse"    #  The  protocol  to  use  to  access
crates.io.

[registry]
```

```
default = "…"        # name of the default registry
token = "…"          # authentication token for crates.io
credential-provider = "cargo:token"          # The credential
provider for crates.io.
global-credential-providers = ["cargo:token"] # The credential
providers to use by default.

[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`
rustdocflags = ["…", "…"] # custom flags for `rustdoc`

[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]
quiet = false                       # whether cargo output is
quiet
verbose = false                      # whether cargo provides
verbose output
color = 'auto'                      # whether cargo colorizes
output
hyperlinks = true                # whether cargo inserts links
into output
unicode = true                      # whether cargo can render
output using non-ASCII unicode characters
progress.when = 'auto'                # whether cargo shows
progress bar
progress.width = 80              # width of progress bar
progress.term-integration = true # whether cargo reports
progress to terminal emulator
```

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 and otherwise they are not supported due to technical issues.

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

Command-line overrides

Cargo also accepts arbitrary configuration overrides through the `--config` command-line option. The argument should be in TOML syntax of `KEY=VALUE` or provided as a path to an extra configuration file:

```
# With `KEY=VALUE` in TOML syntax
cargo --config net.git-fetch-with-cli=true fetch

# With a path to a configuration file
cargo --config ./path/to/my/extra-config.toml fetch
```

The `--config` option may be specified multiple times, in which case the values are merged in left-to-right order, using the same merging logic that is used when multiple configuration files apply. Configuration values specified this way take precedence over environment variables, which take precedence over configuration files.

When the `--config` option is provided as an extra configuration file, The configuration file loaded this way follow the same precedence rules as other options specified directly with `--config`.

Some examples of what it looks like using Bourne shell syntax:

```
# Most shells will require escaping.
cargo --config http.proxy=\"http://example.com\" …

# Spaces may be used.
cargo --config "net.git-fetch-with-cli = true" …

# TOML array example. Single quotes make it easier to read and
write.
cargo  --config  'build.rustdocflags  =  ["--html-in-header",
"header.html"]' …

# Example of a complex TOML key.
cargo   --config   "target.'cfg(all(target_arch   =   \"arm\",
target_os = \"none\"))'.runner = 'my-runner'" …
```

```
# Example of overriding a profile setting.
cargo --config profile.dev.package.image.opt-level=3 …
```

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.

In particular, rules are:

- For environment variables, paths are relative to the current working directory.
- For config values loaded directly from the `--config KEY=VALUE` option, paths are relative to the current working directory.
- For config files, paths are relative to the parent directory of the directory where the config files were defined, no matter those files are from either the hierarchical probing or the `--config <path>` option.

Note: To maintain consistency with existing `.cargo/config.toml` probing behavior, it is by design that a path in a config file passed via `--config <path>` is also relative to two levels up from the config file itself.

To avoid unexpected results, the rule of thumb is putting your extra config files at the same level of discovered `.cargo/config.toml` in your project. For instance, given a project `/my/project`, it is recommended to put config files under `/my/project/.cargo` or a new directory at the same level, such as `/my/project/.config`.

```
# 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.toml` is located.
# For example, `/my/project/.cargo/config.toml` would result
```

```
in `/my/project/vendor`.
directory = "vendor"
```

Executable paths with arguments

Some Cargo commands invoke external programs, which can be configured as a path and some number of arguments.

The value may be an array of strings like `['/path/to/program', 'somearg']` or a space-separated string like `'/path/to/program somearg'`. If the path to the executable contains a space, the list form must be used.

If Cargo is passing other arguments to the program such as a path to open or run, they will be passed after the last specified argument in the value of an option of this format. If the specified program does not have path separators, Cargo will search `PATH` for its executable.

Credentials

Configuration values with sensitive information are stored in the `$CARGO_HOME/credentials.toml` file. This file is automatically created and updated by `cargo login` and `cargo logout` when using the `cargo:token` credential provider.

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.

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
```

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.

Note: Cargo also reads and writes credential files without the `.toml` extension, such as `.cargo/credentials`. Support for the `.toml` extension was added in version 1.39. In version 1.68, Cargo writes to the file with the extension by default. However, for backward compatibility reason, when both files exist, Cargo will read and write the file without the extension.

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 Overriding 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"
d = "doc"
t = "test"
r = "run"
rm = "remove"
```

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

Aliases are recursive:

```
[alias]
rr = "run --release"
recursive_example = "rr --example recursions"
```

`[build]`

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

`build.jobs`

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

Sets the maximum number of compiler processes to run in parallel. If negative, it sets the maximum number of compiler processes to the number of logical CPUs plus provided value. Should not be 0. If a string `default` is provided, it sets the value back to defaults.

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 executable to use (i.e., `build.rustc`, if that is set, or `"rustc"` otherwise).

`build.rustc-workspace-wrapper`

- Type: string (program path)
- Default: none
- Environment: `CARGO_BUILD_RUSTC_WORKSPACE_WRAPPER` or `RUSTC_WORKSPACE_WRAPPER`

Sets a wrapper to execute instead of `rustc`, for workspace members only. When building a single-package project without workspaces, that package is considered to be the workspace. The first argument passed to the wrapper is the path to the actual executable to use (i.e., `build.rustc`, if that is set, or `"rustc"` otherwise). It affects the filename hash so that artifacts produced by the wrapper are cached separately.

If both `rustc-wrapper` and `rustc-workspace-wrapper` are set, then they will be nested: the final invocation is `$RUSTC_WRAPPER $RUSTC_WORKSPACE_WRAPPER $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 or array of strings
- Default: host platform
- Environment: `CARGO_BUILD_TARGET`

The default target platform triples to compile to.

This allows passing either a string or an array of strings. Each string value is a target platform triple. The selected build targets will be built for each of the selected architectures.

The string value may also be a relative path to a `.json` target spec file.

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

```
[build]
target = ["x86_64-unknown-linux-gnu", "i686-unknown-linux-gnu"]
```

`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 `CARGO_ENCODED_RUSTFLAGS` or `RUSTFLAGS`

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

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

1. `CARGO_ENCODED_RUSTFLAGS` environment variable.
2. `RUSTFLAGS` environment variable.
3. All matching `target.<triple>.rustflags` and `target.<cfg>.rustflags` config entries joined together.
4. `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.

It is not recommended to pass in flags that Cargo itself usually manages. For example, the flags driven by profiles are best handled by setting the appropriate profile setting.

> **Caution**: Due to the low-level nature of passing flags directly to the compiler, this may cause a conflict with future versions of Cargo which may issue the same or similar flags on its own which may interfere with the flags you specify. This is an area where Cargo may not always be backwards compatible.

`build.rustdocflags`

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

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

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

1. `CARGO_ENCODED_RUSTDOCFLAGS` environment variable.
2. `RUSTDOCFLAGS` environment variable.
3. All matching `target.<triple>.rustdocflags` config entries joined together.
4. `build.rustdocflags` config value.

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

> **Caution**: Due to the low-level nature of passing flags directly to the compiler, this may cause a conflict with future versions of Cargo which may issue the same or similar flags on its own which may interfere with the flags you specify. This is an area where Cargo may not always be backwards compatible.

`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. This config setting is intended to convert absolute paths to relative 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`

This option is deprecated and unused. Cargo always has pipelining enabled.

`[credential-alias]`

- Type: string or array of strings
- Default: empty
- Environment: `CARGO_CREDENTIAL_ALIAS_<name>`

The `[credential-alias]` table defines credential provider aliases. These aliases can be referenced as an element of the `registry.global-`

`credential-providers` array, or as a credential provider for a specific registry under `registries.<NAME>.credential-provider`.

If specified as a string, the value will be split on spaces into path and arguments.

For example, to define an alias called `my-alias`:

```
[credential-alias]
my-alias = ["/usr/bin/cargo-credential-example", "--argument",
"value", "--flag"]
```

See Registry Authentication for more information.

[doc]

The `[doc]` table defines options for the `cargo doc` command.

doc.browser

- Type: string or array of strings (program path with args)
- Default: `BROWSER` environment variable, or, if that is missing, opening the link in a system specific way

This option sets the browser to be used by `cargo doc`, overriding the `BROWSER` environment variable when opening documentation with the `--open` option.

[cargo-new]

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

cargo-new.name

This option is deprecated and unused.

cargo-new.email

This option is deprecated and unused.

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.

`[env]`

The `[env]` section allows you to set additional environment variables for build scripts, rustc invocations, `cargo run` and `cargo build`.

```
[env]
OPENSSL_DIR = "/opt/openssl"
```

By default, the variables specified will not override values that already exist in the environment. This behavior can be changed by setting the `force` flag.

Setting the `relative` flag evaluates the value as a config-relative path that is relative to the parent directory of the `.cargo` directory that contains the `config.toml` file. The value of the environment variable will be the full absolute path.

```
[env]
TMPDIR = { value = "/home/tmp", force = true }
OPENSSL_DIR = { value = "vendor/openssl", relative = true }
```

`[future-incompat-report]`

The `[future-incompat-report]` table controls setting for future incompat reporting

`future-incompat-report.frequency`

- Type: string
- Default: `"always"`
- Environment: `CARGO_FUTURE_INCOMPAT_REPORT_FREQUENCY`

Controls how often we display a notification to the terminal when a future incompat report is available. Possible values:

- `always` (default): Always display a notification when a command (e.g. `cargo build`) produces a future incompat report
- `never`: Never display a notification

`[cache]`

The `[cache]` table defines settings for cargo's caches.

Global caches

When running `cargo` commands, Cargo will automatically track which files you are using within the global cache. Periodically, Cargo will delete files that have not been used for some period of time. It will delete files that have to be downloaded from the network if they have not been used in 3 months. Files that can be generated without network access will be deleted if they have not been used in 1 month.

The automatic deletion of files only occurs when running commands that are already doing a significant amount of work, such as all of the build commands (`cargo build`, `cargo test`, `cargo check`, etc.), and `cargo fetch`.

Automatic deletion is disabled if cargo is offline such as with `--offline` or `--frozen` to avoid deleting artifacts that may need to be used if you are offline for a long period of time.

> **Note**: This tracking is currently only implemented for the global cache in Cargo's home directory. This includes registry indexes and source files downloaded from registries and git dependencies. Support for tracking build artifacts is not yet implemented, and tracked in cargo#13136.
>
> Additionally, there is an unstable feature to support *manually* triggering cache cleaning, and to further customize the configuration options. See the Unstable chapter for more information.

`cache.auto-clean-frequency`

- Type: string
- Default: `"1 day"`

- Environment: `CARGO_CACHE_AUTO_CLEAN_FREQUENCY`

This option defines how often Cargo will automatically delete unused files in the global cache. This does *not* define how old the files must be, those thresholds are described above.

It supports the following settings:

- `"never"` --- Never deletes old files.
- `"always"` --- Checks to delete old files every time Cargo runs.
- An integer followed by "seconds", "minutes", "hours", "days", "weeks", or "months" --- Checks to delete old files at most the given time frame.

`[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=network=debug` environment variable (or use `network=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.proxy-cainfo`

- Type: string (path)
- Default: falls back to `http.cainfo` if not set
- Environment: `CARGO_HTTP_PROXY_CAINFO`

Path to a Certificate Authority (CA) bundle file, used to verify proxy TLS 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.

To track information of installed executables, some extra files, such as `.crates.toml` and `.crates2.json`, are also created under this 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: 3
- 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. See Git Authentication for more information about setting up git authentication.

`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.

`net.ssh`

The `[net.ssh]` table contains settings for SSH connections.

`net.ssh.known-hosts`

- Type: array of strings
- Default: see description
- Environment: not supported

The `known-hosts` array contains a list of SSH host keys that should be accepted as valid when connecting to an SSH server (such as for SSH git dependencies). Each entry should be a string in a format similar to OpenSSH `known_hosts` files. Each string should start with one or more hostnames separated by commas, a space, the key type name, a space, and the base64-encoded key. For example:

```
[net.ssh]
known-hosts = [
```

```
                                    "example.com       ssh-ed25519
AAAAC3NzaC1lZDI1NTE5AAAAIFO4Q5T0UV0SQevair9PFwoxY9dl4pQl3u5pho
qJH3cF"
]
```

Cargo will attempt to load known hosts keys from common locations supported in OpenSSH, and will join those with any listed in a Cargo configuration file. If any matching entry has the correct key, the connection will be allowed.

Cargo comes with the host keys for github.com built-in. If those ever change, you can add the new keys to the config or known_hosts file.

See Git Authentication for more details.

`[patch]`

Just as you can override dependencies using `[patch]` in `Cargo.toml`, you can override them in the cargo configuration file to apply those patches to any affected build. The format is identical to the one used in `Cargo.toml`.

Since `.cargo/config.toml` files are not usually checked into source control, you should prefer patching using `Cargo.toml` where possible to ensure that other developers can compile your crate in their own environments. Patching through cargo configuration files is generally only appropriate when the patch section is automatically generated by an external build tool.

If a given dependency is patched both in a cargo configuration file and a `Cargo.toml` file, the patch in the configuration file is used. If multiple configuration files patch the same dependency, standard cargo configuration merging is used, which prefers the value defined closest to the current directory, with `$HOME/.cargo/config.toml` taking the lowest precedence.

Relative `path` dependencies in such a `[patch]` section are resolved relative to the configuration file they appear in.

`[profile]`

The `[profile]` table can be used to globally change profile settings, and override settings specified in `Cargo.toml`. It has the same syntax and options as profiles specified in `Cargo.toml`. See the Profiles chapter for details about the options.

`[profile.<name>.build-override]`

- Environment: `CARGO_PROFILE_<name>_BUILD_OVERRIDE_<key>`

The build-override table overrides settings for build scripts, proc macros, and their dependencies. It has the same keys as a normal profile. See the overrides section for more details.

`[profile.<name>.package.<name>]`

- Environment: not supported

The package table overrides settings for specific packages. It has the same keys as a normal profile, minus the `panic`, `lto`, and `rpath` settings. See the overrides section for more details.

`profile.<name>.codegen-units`

- Type: integer
- Default: See profile docs.
- Environment: `CARGO_PROFILE_<name>_CODEGEN_UNITS`

See codegen-units.

`profile.<name>.debug`

- Type: integer or boolean
- Default: See profile docs.
- Environment: `CARGO_PROFILE_<name>_DEBUG`

See debug.

`profile.<name>.split-debuginfo`

- Type: string
- Default: See profile docs.

- Environment: `CARGO_PROFILE_<name>_SPLIT_DEBUGINFO`

See split-debuginfo.

`profile.<name>.debug-assertions`

- Type: boolean
- Default: See profile docs.
- Environment: `CARGO_PROFILE_<name>_DEBUG_ASSERTIONS`

See debug-assertions.

`profile.<name>.incremental`

- Type: boolean
- Default: See profile docs.
- Environment: `CARGO_PROFILE_<name>_INCREMENTAL`

See incremental.

`profile.<name>.lto`

- Type: string or boolean
- Default: See profile docs.
- Environment: `CARGO_PROFILE_<name>_LTO`

See lto.

`profile.<name>.overflow-checks`

- Type: boolean
- Default: See profile docs.
- Environment: `CARGO_PROFILE_<name>_OVERFLOW_CHECKS`

See overflow-checks.

`profile.<name>.opt-level`

- Type: integer or string
- Default: See profile docs.
- Environment: `CARGO_PROFILE_<name>_OPT_LEVEL`

See opt-level.

`profile.<name>.panic`

- Type: string
- Default: See profile docs.
- Environment: `CARGO_PROFILE_<name>_PANIC`

See panic.

`profile.<name>.rpath`

- Type: boolean
- Default: See profile docs.
- Environment: `CARGO_PROFILE_<name>_RPATH`

See rpath.

`profile.<name>.strip`

- Type: string or boolean
- Default: See profile docs.
- Environment: `CARGO_PROFILE_<name>_STRIP`

See strip.

`[resolver]`

The `[resolver]` table overrides dependency resolution behavior for local development (e.g. excludes `cargo install`).

`resolver.incompatible-rust-versions`

- Type: string
- Default: See `resolver` docs
- Environment: `CARGO_RESOLVER_INCOMPATIBLE_RUST_VERSIONS`

When resolving which version of a dependency to use, select how versions with incompatible `package.rust-version`s are treated. Values include:

- `allow`: treat `rust-version`-incompatible versions like any other version

- `fallback`: only consider `rust-version`-incompatible versions if no other version matched

Can be overridden with

- `--ignore-rust-version` CLI option
- Setting the dependency's version requirement higher than any version with a compatible `rust-version`
- Specifying the version to `cargo update` with `--precise`

See the resolver chapter for more details.

> **MSRV:**
>
> - `allow` is supported on any version
> - `fallback` is respected as of 1.84

`[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 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.

`registries.<name>.credential-provider`

- Type: string or array of path and arguments
- Default: none
- Environment: `CARGO_REGISTRIES_<name>_CREDENTIAL_PROVIDER`

Specifies the credential provider for the given registry. If not set, the providers in `registry.global-credential-providers` will be used.

If specified as a string, path and arguments will be split on spaces. For paths or arguments that contain spaces, use an array.

If the value exists in the `[credential-alias]` table, the alias will be used.

See Registry Authentication for more information.

`registries.crates-io.protocol`

- Type: string
- Default: `"sparse"`
- Environment: `CARGO_REGISTRIES_CRATES_IO_PROTOCOL`

Specifies the protocol used to access crates.io. Allowed values are `git` or `sparse`.

`git` causes Cargo to clone the entire index of all packages ever published to crates.io from https://github.com/rust-lang/crates.io-index/. This can have performance implications due to the size of the index. `sparse` is a newer protocol which uses HTTPS to download only what is necessary from https://index.crates.io/. This can result in a significant performance improvement for resolving new dependencies in most situations.

More information about registry protocols may be found in the Registries chapter.

`[registry]`

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

`registry.index`

This value is no longer accepted 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.credential-provider`

- Type: string or array of path and arguments
- Default: none
- Environment: `CARGO_REGISTRY_CREDENTIAL_PROVIDER`

Specifies the credential provider for crates.io. If not set, the providers in `registry.global-credential-providers` will be used.

If specified as a string, path and arguments will be split on spaces. For paths or arguments that contain spaces, use an array.

If the value exists in the `[credential-alias]` table, the alias will be used.

See Registry Authentication for more information.

`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.

`registry.global-credential-providers`

- Type: array
- Default: `["cargo:token"]`
- Environment: `CARGO_REGISTRY_GLOBAL_CREDENTIAL_PROVIDERS`

Specifies the list of global credential providers. If credential provider is not set for a specific registry using `registries.<name>.credential-provider`, Cargo will use the credential providers in this list. Providers toward the end of the list have precedence.

Path and arguments are split on spaces. If the path or arguments contains spaces, the credential provider should be defined in the `[credential-alias]` table and referenced here by its alias.

See Registry Authentication for more information.

`[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 or named registry.

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

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.<cfg>.linker`

This is similar to the target linker, 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>.runner`

- Type: string or array of strings (program path with 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.

`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 an 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>.rustdocflags`

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

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

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

`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.quiet`

- Type: boolean
- Default: false

- Environment: `CARGO_TERM_QUIET`

Controls whether or not log messages are displayed by Cargo.

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

`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.

`term.hyperlinks`

- Type: bool
- Default: auto-detect
- Environment: `CARGO_TERM_HYPERLINKS`

Controls whether or not hyperlinks are used in the terminal.

`term.unicode`

- Type: bool
- Default: auto-detect
- Environment: `CARGO_TERM_UNICODE`

Control whether output can be rendered using non-ASCII unicode characters.

`term.progress.when`

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

Controls whether or not progress bar is shown in the terminal. Possible values:

- `auto` (default): Intelligently guess whether to show progress bar.
- `always`: Always show progress bar.
- `never`: Never show progress bar.

`term.progress.width`

- Type: integer
- Default: none
- Environment: `CARGO_TERM_PROGRESS_WIDTH`

Sets the width for progress bar.

`term.progress.term-integration`

- Type: bool
- Default: auto-detect
- Environment: `CARGO_TERM_PROGRESS_TERM_INTEGRATION`

Report progress to the terminal emulator for display in places like the task bar.

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_LOG` --- Cargo uses the `tracing` crate to display debug log messages. The `CARGO_LOG` environment variable can be set to enable debug logging, with a value such as `trace`, `debug`, or `warn`. Usually it is only used during debugging. For more details refer to the Debug logging.
- `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.
- `CARGO` --- If set, Cargo will forward this value instead of setting it to its own auto-detected path when it builds crates and when it executes build scripts and external subcommands. This value is not directly executed by Cargo, and should always point at a command that behaves exactly like `cargo`, as that's what users of the variable will be expecting.
- `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, passing as its command-line arguments the rustc invocation, with the first argument being the path to the actual rustc. Useful to set up a build cache tool such as `sccache`. See `build.rustc-wrapper` to set via config. Setting this to the empty string overwrites the config and resets cargo to not use a wrapper.

- `RUSTC_WORKSPACE_WRAPPER` --- Instead of simply running `rustc`, for workspace members Cargo will execute this specified wrapper, passing as its command-line arguments the rustc invocation, with the first argument being the path to the actual rustc. When building a single-package project without workspaces, that package is considered to be the workspace. It affects the filename hash so that artifacts produced by the wrapper are cached separately. See `build.rustc-workspace-wrapper` to set via config. Setting this to the empty string overwrites the config and resets cargo to not use a wrapper for workspace members. If both `RUSTC_WRAPPER` and `RUSTC_WORKSPACE_WRAPPER` are set, then they will be nested: the final invocation is `$RUSTC_WRAPPER $RUSTC_WORKSPACE_WRAPPER $RUSTC`.
- `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. This string is split by whitespace; for a more robust encoding of multiple arguments, see `CARGO_ENCODED_RUSTDOCFLAGS`.
- `CARGO_ENCODED_RUSTDOCFLAGS` --- A list of custom flags separated by `0x1f` (ASCII Unit Separator) to pass to all `rustdoc` invocations that Cargo performs.
- `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. This string is split by whitespace; for a more robust encoding of multiple arguments, see `CARGO_ENCODED_RUSTFLAGS`.
- `CARGO_ENCODED_RUSTFLAGS` --- A list of custom flags separated by `0x1f` (ASCII Unit Separator) to pass to all compiler invocations that Cargo performs.

- `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.
- `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, see `doc.browser` for more details.
- `RUSTFMT` --- Instead of running `rustfmt`, `cargo fmt` will execute this specified `rustfmt` instance instead.

Configuration environment variables

Cargo reads environment variables for some 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_RUSTC_WORKSPACE_WRAPPER` --- The `rustc` wrapper for workspace members only, see `build.rustc-workspace-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_CACHE_AUTO_CLEAN_FREQUENCY` --- Configures how often automatic cache cleaning runs, see `cache.auto-clean-frequency`.
- `CARGO_CARGO_NEW_VCS` --- The default source control system with `cargo new`, see `cargo-new.vcs`.
- `CARGO_FUTURE_INCOMPAT_REPORT_FREQUENCY` --- How often we should generate a future incompat report notification, see `future-incompat-report.frequency`.
- `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_PROXY_CAINFO` --- The proxy TLS certificate Certificate Authority file, see `http.proxy-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_PROFILE_<name>_BUILD_OVERRIDE_<key>` --- Override build script profile, see `profile.<name>.build-override`.
- `CARGO_PROFILE_<name>_CODEGEN_UNITS` --- Set code generation units, see `profile.<name>.codegen-units`.
- `CARGO_PROFILE_<name>_DEBUG` --- What kind of debug info to include, see `profile.<name>.debug`.
- `CARGO_PROFILE_<name>_DEBUG_ASSERTIONS` --- Enable/disable debug assertions, see `profile.<name>.debug-assertions`.
- `CARGO_PROFILE_<name>_INCREMENTAL` --- Enable/disable incremental compilation, see `profile.<name>.incremental`.
- `CARGO_PROFILE_<name>_LTO` --- Link-time optimization, see `profile.<name>.lto`.
- `CARGO_PROFILE_<name>_OVERFLOW_CHECKS` --- Enable/disable overflow checks, see `profile.<name>.overflow-checks`.
- `CARGO_PROFILE_<name>_OPT_LEVEL` --- Set the optimization level, see `profile.<name>.opt-level`.
- `CARGO_PROFILE_<name>_PANIC` --- The panic strategy to use, see `profile.<name>.panic`.
- `CARGO_PROFILE_<name>_RPATH` --- The rpath linking option, see `profile.<name>.rpath`.
- `CARGO_PROFILE_<name>_SPLIT_DEBUGINFO` --- Controls debug file output behavior, see `profile.<name>.split-debuginfo`.

- `CARGO_PROFILE_<name>_STRIP` --- Controls stripping of symbols and/or debuginfos, see `profile.<name>.strip`.
- `CARGO_REGISTRIES_<name>_CREDENTIAL_PROVIDER` --- Credential provider for a registry, see `registries.<name>.credential-provider`.
- `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_CREDENTIAL_PROVIDER` --- Credential provider for crates.io, see `registry.credential-provider`.
- `CARGO_REGISTRY_DEFAULT` --- Default registry for the `--registry` flag, see `registry.default`.
- `CARGO_REGISTRY_GLOBAL_CREDENTIAL_PROVIDERS` --- Credential providers for registries that do not have a specific provider defined. See `registry.global-credential-providers`.
- `CARGO_REGISTRY_TOKEN` --- Authentication token for crates.io, see `registry.token`.
- `CARGO_TARGET_<triple>_LINKER` --- The linker to use, see `target.<triple>.linker`. The triple must be converted to uppercase and underscores.
- `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_QUIET` --- Quiet mode, see `term.quiet`.
- `CARGO_TERM_VERBOSE` --- The default terminal verbosity, see `term.verbose`.
- `CARGO_TERM_COLOR` --- The default color mode, see `term.color`.
- `CARGO_TERM_PROGRESS_WHEN` --- The default progress bar showing mode, see `term.progress.when`.

- `CARGO_TERM_PROGRESS_WIDTH` --- The default progress bar width, see `term.progress.width`.

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`.

Note that if one of these values is not provided in the manifest, the corresponding environment variable is set to the empty string, `""`.

- `CARGO` --- Path to the `cargo` binary performing the build.
- `CARGO_MANIFEST_DIR` --- The directory containing the manifest of your package.
- `CARGO_MANIFEST_PATH` --- The path to 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.
- `CARGO_PKG_LICENSE` --- The license from the manifest of your package.

- `CARGO_PKG_LICENSE_FILE` --- The license file from the manifest of your package.
- `CARGO_PKG_RUST_VERSION` --- The Rust version from the manifest of your package. Note that this is the minimum Rust version supported by the package, not the current Rust version.
- `CARGO_PKG_README` --- Path to the README file of your package.
- `CARGO_CRATE_NAME` --- The name of the crate that is currently being compiled. It is the name of the Cargo target with `-` converted to `_`, such as the name of the library, binary, example, integration test, or benchmark.
- `CARGO_BIN_NAME` --- The name of the binary that is currently being compiled. Only set for binaries or binary examples. This name does not include any file extension, such as `.exe`.
- `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.)
- `CARGO_BIN_EXE_<name>` --- The absolute path to a binary target's executable. This is only set when building an integration test or benchmark. This may be used with the `env` macro to find the executable to run for testing purposes. The `<name>` is the name of the binary target, exactly as-is. For example, `CARGO_BIN_EXE_my-program` for a binary named `my-program`. Binaries are automatically built when the test is built, unless the binary has required features that are not enabled.
- `CARGO_PRIMARY_PACKAGE` --- This environment variable will be set if the package being built is primary. Primary packages are the ones the user selected on the command-line, either with `-p` flags or the defaults based on the current directory and the default workspace members. This variable will not be set when building dependencies, unless a dependency is also a workspace member that was also selected on the command-line. This is only set when compiling the package (not when running binaries or tests).
- `CARGO_TARGET_TMPDIR` --- Only set when building integration test or benchmark code. This is a path to a directory inside the target directory

where integration tests or benchmarks are free to put any data needed by the tests/benches. Cargo initially creates this directory but doesn't manage its content in any way, this is the responsibility of the test code.

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`
- AIX: `LIBPATH`

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 support of proc-macros.
- 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_PATH` --- The path to the manifest of your package.
- `CARGO_MANIFEST_LINKS` --- the manifest `links` value.
- `CARGO_MAKEFLAGS` --- Contains parameters needed for Cargo's jobserver implementation to parallelize subprocesses. Rustc or cargo invocations from build.rs can already read `CARGO_MAKEFLAGS`, but GNU Make requires the flags to be specified either directly as arguments, or through the `MAKEFLAGS` environment variable. Currently Cargo doesn't set the `MAKEFLAGS` variable, but it's free for build scripts invoking GNU Make to set it to the contents of `CARGO_MAKEFLAGS`.
- `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_FEATURE` --- Each activated feature of the package being built.
- `CARGO_CFG_UNIX` --- Set on unix-like platforms.
- `CARGO_CFG_WINDOWS` --- Set on windows-like platforms.
- `CARGO_CFG_TARGET_FAMILY=unix,wasm` --- The target family.
- `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_ABI=eabihf` --- The target ABI.
- `CARGO_CFG_TARGET_POINTER_WIDTH=64` --- The CPU pointer width.
- `CARGO_CFG_TARGET_ENDIAN=little` --- The CPU target endianness.
- `CARGO_CFG_TARGET_FEATURE=mmx,sse` --- List of CPU target features enabled.

Note that different target triples have different sets of `cfg` values, hence variables present in one target triple might not be available in the other.

Some cfg values like `debug_assertions` and `test` are not available.

- `OUT_DIR` --- the folder in which all output and intermediate artifacts 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`, and instead can set the `MAKEFLAGS` env var to the content of `CARGO_MAKEFLAGS` to activate the use of Cargo's GNU Make compatible jobserver for sub-make invocations.
- `OPT_LEVEL`, `DEBUG` --- values of the corresponding variables for the profile currently being built.
- `PROFILE` --- `release` for release builds, `debug` for other builds. This is determined based on if the profile inherits from the `dev` or `release` profile. Using this environment variable is not recommended. Using other environment variables like `OPT_LEVEL` provide a more correct view of the actual settings being used.
- `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_WRAPPER` --- the `rustc` wrapper, if any, that Cargo is using. See `build.rustc-wrapper`.
- `RUSTC_WORKSPACE_WRAPPER` --- the `rustc` wrapper, if any, that Cargo is using for workspace members. See `build.rustc-workspace-wrapper`.
- `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.toml`; see the documentation about cargo configuration for more information.

- `CARGO_ENCODED_RUSTFLAGS` --- extra flags that Cargo invokes `rustc` with, separated by a `0x1f` character (ASCII Unit Separator). See `build.rustflags`. Note that since Rust 1.55, `RUSTFLAGS` is removed from the environment; scripts should use `CARGO_ENCODED_RUSTFLAGS` instead.
- `CARGO_PKG_<var>` --- The package information variables, with the same names and values as are provided during crate building.

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.
- `CARGO_MAKEFLAGS` --- Contains parameters needed for Cargo's jobserver implementation to parallelize subprocesses. This is set only when Cargo detects the existence of a jobserver.

For extended information about your environment you may run `cargo metadata`.

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 formatted 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 or intermediate artifacts 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 order of `cargo::` instructions printed by the build script *may* affect the order of arguments that `cargo` passes to `rustc`. In turn, the order of arguments passed to `rustc` may affect the order of arguments passed to the linker. Therefore, you will want to pay attention to the order of the build script's instructions. For example, if object `foo` needs to link against library `bar`, you may want to make sure that library `bar`'s `cargo::rustc-link-lib` instruction appears *after* instructions to link object `foo`.

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-arg=FLAG` --- Passes custom flags to a linker for benchmarks, binaries, `cdylib` crates, examples, and tests.
- `cargo::rustc-link-arg-cdylib=FLAG` --- Passes custom flags to a linker for cdylib crates.
- `cargo::rustc-link-arg-bin=BIN=FLAG` --- Passes custom flags to a linker for the binary `BIN`.
- `cargo::rustc-link-arg-bins=FLAG` --- Passes custom flags to a linker for binaries.
- `cargo::rustc-link-arg-tests=FLAG` --- Passes custom flags to a linker for tests.
- `cargo::rustc-link-arg-examples=FLAG` --- Passes custom flags to a linker for examples.
- `cargo::rustc-link-arg-benches=FLAG` --- Passes custom flags to a linker for benchmarks.
- `cargo::rustc-link-lib=LIB` --- 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-check-cfg=CHECK_CFG` -- Register custom `cfg`s as expected for compile-time checking of configs.
- `cargo::rustc-env=VAR=VALUE` --- Sets an environment variable.

- `cargo::error=MESSAGE` --- Displays an error on the terminal.

- `cargo::warning=MESSAGE` --- Displays a warning on the terminal.
- `cargo::metadata=KEY=VALUE` --- Metadata, used by `links` scripts.

MSRV: 1.77 is required for `cargo::KEY=VALUE` syntax. To support older versions, use the `cargo:KEY=VALUE` syntax.

`cargo::rustc-link-arg=FLAG` {#rustc-link-arg}

The `rustc-link-arg` instruction tells Cargo to pass the `-C link-arg=FLAG` option to the compiler, but only when building supported targets (benchmarks, binaries, `cdylib` crates, examples, and tests). Its usage is highly platform specific. It is useful to set the shared library version or linker script.

`cargo::rustc-link-arg-cdylib=FLAG` {#rustc-cdylib-link-arg}

The `rustc-link-arg-cdylib` 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.

For historical reasons, the `cargo::rustc-cdylib-link-arg` form is an alias for `cargo::rustc-link-arg-cdylib`, and has the same meaning.

`cargo::rustc-link-arg-bin=BIN=FLAG` {#rustc-link-arg-bin}

The `rustc-link-arg-bin` instruction tells Cargo to pass the `-C link-arg=FLAG` option to the compiler, but only when building the binary target with name `BIN`. Its usage is highly platform specific. It is useful to set a linker script or other linker options.

`cargo::rustc-link-arg-bins=FLAG` {#rustc-link-arg-bins}

The `rustc-link-arg-bins` instruction tells Cargo to pass the `-C link-arg=FLAG` option to the compiler, but only when building a binary target. Its usage is highly platform specific. It is useful to set a linker script or other linker options.

`cargo::rustc-link-arg-tests=FLAG` {#rustc-link-arg-tests}

The `rustc-link-arg-tests` instruction tells Cargo to pass the `-C link-arg=FLAG` option to the compiler, but only when building a tests target.

`cargo::rustc-link-arg-examples=FLAG` {#rustc-link-arg-examples}

The `rustc-link-arg-examples` instruction tells Cargo to pass the `-C link-arg=FLAG` option to the compiler, but only when building an examples target.

`cargo::rustc-link-arg-benches=FLAG` {#rustc-link-arg-benches}

The `rustc-link-arg-benches` instruction tells Cargo to pass the `-C link-arg=FLAG` option to the compiler, but only when building a benchmark target.

`cargo::rustc-link-lib=LIB` {#rustc-link-lib}

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 `LIB` string is passed directly to rustc, so it supports any syntax that `-l` does.
Currently the fully supported syntax for `LIB` is `[KIND[:MODIFIERS]=]NAME[:RENAME]`.

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` {#rustc-link-search}

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` {#rustc-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"]` {#rustc-cfg}

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. Custom cfgs must either be expected using the `cargo::rustc-check-cfg` instruction or usage will need to allow the `unexpected_cfgs` lint to avoid unexpected cfgs warnings.

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-check-cfg=CHECK_CFG` {#rustc-check-cfg}

Add to the list of expected config names and values that is used when checking the *reachable* cfg expressions with the `unexpected_cfgs` lint.

The syntax of `CHECK_CFG` mirrors the `rustc --check-cfg` flag, see Checking conditional configurations for more details.

The instruction can be used like this:

```
// build.rs
println!("cargo::rustc-check-cfg=cfg(foo, values(\"bar\"))");
if foo_bar_condition {
    println!("cargo::rustc-cfg=foo=\"bar\"");
}
```

Note that all possible cfgs should be defined, regardless of which cfgs are currently enabled. This includes all possible values of a given cfg name.

It is recommended to group the `cargo::rustc-check-cfg` and `cargo::rustc-cfg` instructions as closely as possible in order to avoid typos, missing check-cfg, stale cfgs...

See also the conditional compilation example.

> **MSRV:** Respected as of 1.80

`cargo::rustc-env=VAR=VALUE` {#rustc-env}

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.

Note: These environment variables are also set when running an executable with `cargo run` or `cargo test`. However, this usage is discouraged since it ties the executable to Cargo's execution environment. Normally, these environment variables should only be checked at compile-time with the `env!` macro.

`cargo::error=MESSAGE` {#cargo-error}

The `error` instruction tells Cargo to display an error after the build script has finished running, and then fail the build.

Note: Build script libraries should carefully consider if they want to use `cargo::error` versus returning a `Result`. It may be better to return a `Result`, and allow the caller to decide if the error is fatal or not. The caller can then decide whether or not to display the `Err` variant using `cargo::error`.

MSRV: Respected as of 1.84

`cargo::warning=MESSAGE` {#cargo-warning}

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, unless the build fails. 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 (or the list of files controlled by the `exclude` and `include` fields). 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. If Cargo is re-running the build scripts of your own crate or a dependency and you don't know why, see "Why is Cargo rebuilding my code?" in the FAQ.

`cargo::rerun-if-changed=PATH` {#rerun-if-changed}

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 will scan the entire directory for any modifications.

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 (otherwise, the default if no `rerun-if` instructions are emitted is to scan the entire package directory for changes). 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` {#rerun-if-env-changed}

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 possible to use this for environment variables like `TARGET` that Cargo sets for build scripts. The environment variables in use are those received by `cargo` invocations, not those received by the executable of the build script.

As of 1.46, using `env!` and `option_env!` in source code will automatically detect changes and trigger rebuilds. `rerun-if-env-changed` is no longer needed for variables already referenced by these macros.

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.

Build scripts can generate an arbitrary set of metadata in the form of key-value pairs. This metadata is set with the `cargo::metadata=KEY=VALUE` instruction.

The metadata is passed to the build scripts of **dependent** packages. For example, if the package `foo` depends on `bar`, which links `baz`, then if `bar` generates `key=value` as part of its build script metadata, then the build script of `foo` will have the environment variables `DEP_BAZ_KEY=value` (note that the value of the `links` key is used). 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.

> **MSRV:** 1.77 is required for `cargo::metadata=KEY=VALUE`. To support older versions, use `cargo:KEY=VALUE` (unsupported directives are assumed to be metadata keys).

`*-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 types and functions in `libfoo`, but **not** 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 `config.toml` file.

```
[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"
edition = "2024"
```

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 complex 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 = "2024"
```

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.
unsafe 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 = "2024"
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};

unsafe 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 = "2024"

[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::metadata=version_number=
{openssl_version:x}");
```

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)

println!("cargo::rustc-check-cfg=cfg(ossl101,ossl102)");
println!("cargo::rustc-check-
cfg=cfg(ossl110,ossl110g,ossl111)");

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.

Build cache

Cargo stores the output of a build into the "target" directory. By default, this is the directory named `target` in the root of your *workspace*. To change the location, you can set the `CARGO_TARGET_DIR` environment variable, the `build.target-dir` config value, or the `--target-dir` command-line flag.

The directory layout depends on whether or not you are using the `--target` flag to build for a specific platform. If `--target` is not specified, Cargo runs in a mode where it builds for the host architecture. The output goes into the root of the target directory, with each profile stored in a separate subdirectory:

Directory	Description
`target/debug/`	Contains output for the `dev` profile.
`target/release/`	Contains output for the `release` profile (with the `--release` option).
`target/foo/`	Contains build output for the `foo` profile (with the `--profile=foo` option).

For historical reasons, the `dev` and `test` profiles are stored in the `debug` directory, and the `release` and `bench` profiles are stored in the `release` directory. User-defined profiles are stored in a directory with the same name as the profile.

When building for another target with `--target`, the output is placed in a directory with the name of the target:

Directory	Example
`target/<triple>/debug/`	`target/thumbv7em-none-eabihf/debug/`

Directory	Example
`target/<triple>/release/`	`target/thumbv7em-none-eabihf/release/`

Note: When not using `--target`, this has a consequence that Cargo will share your dependencies with build scripts and proc macros. `RUSTFLAGS` will be shared with every `rustc` invocation. With the `--target` flag, build scripts and proc macros are built separately (for the host architecture), and do not share `RUSTFLAGS`.

Within the profile directory (such as `debug` or `release`), artifacts are placed into the following directories:

Directory	Description
`target/debug/`	Contains the output of the package being built (the binary executables and library targets).
`target/debug/examples/`	Contains example targets.

Some commands place their output in dedicated directories in the top level of the `target` directory:

Directory	Description
`target/doc/`	Contains rustdoc documentation (`cargo doc`).
`target/package/`	Contains the output of the `cargo package` and `cargo publish` commands.

Cargo also creates several other directories and files needed for the build process. Their layout is considered internal to Cargo, and is subject to change. Some of these directories are:

Directory	Description
`target/debug/deps/`	Dependencies and other artifacts.
`target/debug/incremental/`	`rustc` incremental output, a cache used to speed up subsequent builds.
`target/debug/build/`	Output from build scripts.

Dep-info files

Next to each compiled artifact is a file called a "dep info" file with a `.d` suffix. This file 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. See the `build.dep-info-basedir` config option to use relative paths.

```
# Example dep-info file found in target/debug/foo.d
/path/to/myproj/target/debug/foo:   /path/to/myproj/src/lib.rs
/path/to/myproj/src/main.rs
```

Shared cache

A third party tool, sccache, can be used to share built dependencies across different workspaces.

To setup `sccache`, install it with `cargo install sccache` and set `RUSTC_WRAPPER` environment 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.

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.

The specification may be fully qualified, such as `https://github.com/rust-lang/crates.io-index#regex@1.4.3` or it may be abbreviated, such as `regex`. The abbreviated form may be used as long as it uniquely identifies a single package in the dependency graph. If there is ambiguity, additional qualifiers can be added to make it unique. For example, if there are two versions of the `regex` package in the graph, then it can be qualified with a version to make it unique, such as `regex@1.4.3`.

Specification grammar

The formal grammar for a Package Id Specification is:

```
spec := pkgname |
           [ kind "+" ] proto "://" hostname-and-path [ "?"
query] [ "#" ( pkgname | semver ) ]
query = ( "branch" | "tag" | "rev" ) "=" ref
pkgname := name [ ("@" | ":" ) semver ]
semver := digits [ "." digits [ "." digits [ "-" prerelease ]
[ "+" build ]]]

kind = "registry" | "git" | "path"
proto := "http" | "git" | "file" | ...
```

Here, brackets indicate that the contents are optional.

The URL form can be used for git dependencies, or to differentiate packages that come from different sources such as different registries.

Example specifications

The following are references to the `regex` package on `crates.io`:

Spec	Name	Version
`regex`	`regex`	`*`
`regex@1.4`	`regex`	`1.4.*`
`regex@1.4.3`	`regex`	`1.4.3`
`https://github.com/rust-lang/crates.io-index#regex`	`regex`	`*`
`https://github.com/rust-lang/crates.io-index#regex@1.4.3`	`regex`	`1.4.3`
`registry+https://github.com/rust-lang/crates.io-index#regex@1.4.3`	`regex`	`1.4.3`

The following are some examples of specs for several different git dependencies:

Spec	Name	Version
`https://github.com/rust-lang/cargo#0.52.0`	`cargo`	`0.52.0`
`https://github.com/rust-lang/cargo#cargo-platform@0.1.2`	`cargo-platform`	`0.1.2`
`ssh://git@github.com/rust-lang/regex.git#regex@1.4.3`	`regex`	`1.4.3`
`git+ssh://git@github.com/rust-lang/regex.git#regex@1.4.3`	`regex`	`1.4.3`

Spec	Name	Version
`git+ssh://git@github.com/rust-lang/regex.git?branch=dev#regex@1.4.3`	`regex`	`1.4.3`

Local packages on the filesystem can use `file://` URLs to reference them:

Spec	Name	Version
`file:///path/to/my/project/foo`	`foo`	`*`
`file:///path/to/my/project/foo#1.1.8`	`foo`	`1.1.8`
`path+file:///path/to/my/project/foo#1.1.8`	`foo`	`1.1.8`

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.

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 `package_id` field is a unique identifier for referring to the package, and as the `--package` argument to many commands. The syntax grammar can be found in chapter Package ID Specifications.

> **Note:** `--message-format=json` only controls Cargo and Rustc's output. This cannot control the output of other tools, e.g. `cargo run --message-format=json`, or arbitrary output from procedural macros. A possible workaround in these situations is to only interpret a line as JSON if it starts with `{`.

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.

If you are using Rust, the cargo_metadata crate can be used to parse these messages.

> **MSRV:** 1.77 is required for `package_id` to be a Package ID Specification. Before that, it was opaque.

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": "file:///path/to/my-package#0.1.0",
    /* Absolute path to the package manifest. */
    "manifest_path": "/path/to/my-package/Cargo.toml",
      /* 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.
```

```
               For lib targets, dashes will be replaced with
underscores.
        */
        "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 the target should be documented by `cargo
doc`. */
        "doc": true,
         /* Whether or not this target has doc tests enabled,
and
           the target is compatible with doc testing.
        */
        "doctest": true
         /* Whether or not this target should be built and run
with `--test`
        */
        "test": 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": "file:///path/to/my-package#0.1.0",
    /* Absolute path to the package manifest. */
    "manifest_path": "/path/to/my-package/Cargo.toml",
      /* 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",
        "doc": true,
        "doctest": true,
        "test": 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, or a
string
             "line-directives-only" or "line-tables-only". 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": "file:///path/to/my-package#0.1.0",
      /* 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"]
    ],
    /* An absolute path which is used as a value of `OUT_DIR`
environmental
       variable when compiling current package.
    */
    "out_dir": "/some/path/in/target/dir"
}
```

Build finished

The "build-finished" message is emitted at the end of the build.

```
{
    /* The "reason" indicates the kind of message. */
    "reason": "build-finished",
    /* Whether or not the build finished successfully. */
    "success": true,
}
```

This message can be helpful for tools to know when to stop reading JSON messages. Commands such as `cargo test` or `cargo run` can produce additional output after the build has finished. This message lets a tool know that Cargo will not produce additional JSON messages, but there may be additional output that may be generated afterwards (such as the output generated by the program executed by `cargo run`).

Note: There is experimental nightly-only support for JSON output for tests, so additional test-specific JSON messages may begin arriving after the "build-finished" message if that is enabled.

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.

> **Note**: Cargo defaults to prioritizing external tools in `$CARGO_HOME/bin` over `$PATH`. Users can override this precedence by adding `$CARGO_HOME/bin` to `$PATH`.

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

Instead, it is encouraged to use the CLI interface to drive Cargo. The `cargo metadata` command can be used to obtain information about the current project (the `cargo_metadata` crate provides a Rust interface to this command).

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.

If you are implementing a registry server, see Running a Registry for more details about the protocol between Cargo and a registry.

If you're using a registry that requires authentication, see Registry Authentication. If you are implementing a credential provider, see Credential Provider Protocol for details.

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.toml` 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 or a Cargo sparse registry URL with the `sparse+` prefix.

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"
edition = "2024"

[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.toml` with the `registry.default` key. For example:

```
[registry]
default = "my-registry"
```

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.toml` file in the Cargo home directory (default `$HOME/.cargo`). It has a separate table for each registry, for example:

```
[registries.my-registry]
token = "854DvwSlUwEHtIo3kWy6x7UCPKHfzCmy"
```

Registry Protocols

Cargo supports two remote registry protocols: `git` and `sparse`. If the registry index URL starts with `sparse+`, Cargo uses the sparse protocol. Otherwise Cargo uses the `git` protocol.

The `git` protocol stores index metadata in a git repository and requires Cargo to clone the entire repo.

The `sparse` protocol fetches individual metadata files using plain HTTP requests. Since Cargo only downloads the metadata for relevant crates, the `sparse` protocol can save significant time and bandwidth.

The crates.io registry supports both protocols. The protocol for crates.io is controlled via the `registries.crates-io.protocol` config key.

Registry Authentication

Cargo authenticates to registries with credential providers. These credential providers are external executables or built-in providers that Cargo uses to store and retrieve credentials.

Using alternative registries with authentication *requires* a credential provider to be configured to avoid unknowingly storing unencrypted credentials on disk. For historical reasons, public (non-authenticated) registries do not require credential provider configuration, and the `cargo:token` provider is used if no providers are configured.

Cargo also includes platform-specific providers that use the operating system to securely store tokens. The `cargo:token` provider is also included which stores credentials in unencrypted plain text in the credentials file.

Recommended configuration

It's recommended to configure a global credential provider list in `$CARGO_HOME/config.toml` which defaults to:

- Windows: `%USERPROFILE%\.cargo\config.toml`
- Unix: `~/.cargo/config.toml`

This recommended configuration uses the operating system provider, with a fallback to `cargo:token` to look in Cargo's credentials file or environment variables:

```
# ~/.cargo/config.toml
[registry]
global-credential-providers = ["cargo:token", "cargo:libsecret", "cargo:macos-keychain", "cargo:wincred"]
```

Note that later entries have higher precedence. See `registry.global-credential-providers` for more details.

Some private registries may also recommend a registry-specific credential-provider. Check your registry's documentation to see if this is the case.

Built-in providers

Cargo includes several built-in credential providers. The available built-in providers may change in future Cargo releases (though there are currently no plans to do so).

`cargo:token`

Uses Cargo's credentials file to store tokens unencrypted in plain text. When retrieving tokens, checks the `CARGO_REGISTRIES_<NAME>_TOKEN` environment variable. If this credential provider is not listed, then the `*_TOKEN` environment variables will not work.

`cargo:wincred`

Uses the Windows Credential Manager to store tokens.

The credentials are stored as `cargo-registry:<index-url>` in the Credential Manager under "Windows Credentials".

`cargo:macos-keychain`

Uses the macOS Keychain to store tokens.

The Keychain Access app can be used to view stored tokens.

`cargo:libsecret`

Uses libsecret to store tokens.

Any password manager with libsecret support can be used to view stored tokens. The following are a few examples (non-exhaustive):

- GNOME Keyring
- KDE Wallet Manager (since KDE Frameworks 5.97.0)
- KeePassXC (since 2.5.0)

`cargo:token-from-stdout <command> <args>`

Launch a subprocess that returns a token on stdout. Newlines will be trimmed.

- The process inherits the user's stdin and stderr.

- It should exit 0 on success, and nonzero on error.
- `cargo login` and `cargo logout` are not supported and return an error if used.

The following environment variables will be provided to the executed command:

- `CARGO` --- Path to the `cargo` binary executing the command.
- `CARGO_REGISTRY_INDEX_URL` --- The URL of the registry index.
- `CARGO_REGISTRY_NAME_OPT` --- Optional name of the registry. Should not be used as a lookup key.

Arguments will be passed on to the subcommand.

Credential plugins

For credential provider plugins that follow Cargo's credential provider protocol, the configuration value should be a string with the path to the executable (or the executable name if on the `PATH`).

For example, to install cargo-credential-1password from crates.io do the following:

Install the provider with `cargo install cargo-credential-1password`

In the config, add to (or create) `registry.global-credential-providers`:

```
[registry]
global-credential-providers = ["cargo:token", "cargo-credential-1password --account my.1password.com"]
```

The values in `global-credential-providers` are split on spaces into path and command-line arguments. To define a global credential provider where the path or arguments contain spaces, use the `[credential-alias]` table.

Credential Provider Protocol

This document describes information for building a Cargo credential provider. For information on setting up or using a credential provider, see Registry Authentication.

When using an external credential provider, Cargo communicates with the credential provider using stdin/stdout messages passed as single lines of JSON.

Cargo will always execute the credential provider with the `--cargo-plugin` argument. This enables a credential provider executable to have additional functionality beyond what Cargo needs. Additional arguments are included in the JSON via the `args` field.

JSON messages

The JSON messages in this document have newlines added for readability. Actual messages must not contain newlines.

Credential hello

- Sent by: credential provider
- Purpose: used to identify the supported protocols on process startup

```
{
    "v":[1]
}
```

Requests sent by Cargo will include a `v` field set to one of the versions listed here. If Cargo does not support any of the versions offered by the credential provider, it will issue an error and shut down the credential process.

Registry information

- Sent by: Cargo Not a message by itself. Included in all messages sent by Cargo as the `registry` field.

```
{
    // Index URL of the registry
          "index-url":"https://github.com/rust-lang/crates.io-
index",
    // Name of the registry in configuration (optional)
    "name": "crates-io",
      // HTTP headers received from attempting to access an
authenticated registry (optional)
    "headers": ["WWW-Authenticate: cargo"]
}
```

Login request

- Sent by: Cargo
- Purpose: collect and store credentials

```
{
    // Protocol version
    "v":1,
    // Action to perform: login
    "kind":"login",
    // Registry information (see Registry information)
             "registry":{"index-url":"sparse+https://registry-
url/index/", "name": "my-registry"},
       //  User-specified  token  from  stdin  or  command  line
(optional)
    "token": "<the token value>",
    // URL that the user could visit to get a token (optional)
    "login-url": "http://registry-url/login",
    // Additional command-line args (optional)
    "args":[]
}
```

If the `token` field is set, then the credential provider should use the token provided. If the `token` is not set, then the credential provider should prompt the user for a token.

In addition to the arguments that may be passed to the credential provider in configuration, `cargo login` also supports passing additional command line args via `cargo login -- <additional args>`. These additional arguments will be included in the `args` field after any args from Cargo configuration.

Read request

- Sent by: Cargo
- Purpose: Get the credential for reading crate information

```
{
    // Protocol version
    "v":1,
    // Request kind: get credentials
    "kind":"get",
    // Action to perform: read crate information
```

```
    "operation":"read",
    // Registry information (see Registry information)
             "registry":{"index-url":"sparse+https://registry-
url/index/", "name": "my-registry"},
    // Additional command-line args (optional)
    "args":[]
}
```

Publish request

- Sent by: Cargo
- Purpose: Get the credential for publishing a crate

```
{
    // Protocol version
    "v":1,
    // Request kind: get credentials
    "kind":"get",
    // Action to perform: publish crate
    "operation":"publish",
    // Crate name
    "name":"sample",
    // Crate version
    "vers":"0.1.0",
    // Crate checksum
    "cksum":"...",
    // Registry information (see Registry information)
             "registry":{"index-url":"sparse+https://registry-
url/index/", "name": "my-registry"},
    // Additional command-line args (optional)
    "args":[]
}
```

Get success response

- Sent by: credential provider
- Purpose: Gives the credential to Cargo

```
{"Ok":{
    // Response kind: this was a get request
    "kind":"get",
    // Token to send to the registry
    "token":"...",
    // Cache control. Can be one of the following:
    // * "never": do not cache
    // * "session": cache for the current cargo session
    // * "expires": cache for the current cargo session until expiration
    "cache":"expires",
    // Unix timestamp (only for "cache": "expires")
    "expiration":1693942857,
    // Is the token operation independent?
    "operation_independent":true
}}
```

The `token` will be sent to the registry as the value of the `Authorization` HTTP header.

`operation_independent` indicates whether the token can be cached across different operations (such as publishing or fetching). In general, this should be `true` unless the provider wants to generate tokens that are scoped to specific operations.

Login success response

- Sent by: credential provider
- Purpose: Indicates the login was successful

```
{"Ok":{
    // Response kind: this was a login request
    "kind":"login"
}}
```

Logout success response

- Sent by: credential provider

- Purpose: Indicates the logout was successful

```
{"Ok":{
    // Response kind: this was a logout request
    "kind":"logout"
}}
```

Failure response (URL not supported)

- Sent by: credential provider
- Purpose: Gives error information to Cargo

```
{"Err":{
    "kind":"url-not-supported"
}}
```

Sent if the credential provider is designed to only handle specific registry URLs and the given URL is not supported. Cargo will attempt another provider if available.

Failure response (not found)

- Sent by: credential provider
- Purpose: Gives error information to Cargo

```
{"Err":{
      // Error: The credential could not be found in the
provider.
    "kind":"not-found"
}}
```

Sent if the credential could not be found. This is expected for `get` requests where the credential is not available, or `logout` requests where there is nothing found to erase.

Failure response (operation not supported)

- Sent by: credential provider
- Purpose: Gives error information to Cargo

```
{"Err":{
      // Error: The credential could not be found in the provider.
    "kind":"operation-not-supported"
}}
```

Sent if the credential provider does not support the requested operation. If a provider only supports `get` and a `login` is requested, the provider should respond with this error.

Failure response (other)

- Sent by: credential provider
- Purpose: Gives error information to Cargo

```
{"Err":{
    // Error: something else has failed
    "kind":"other",
    // Error message string to be displayed
    "message": "free form string error message",
    // Detailed cause chain for the error (optional)
    "caused-by": ["cause 1", "cause 2"]
}}
```

Example communication to request a token for reading:

1. Cargo spawns the credential process, capturing stdin and stdout.
2. Credential process sends the Hello message to Cargo
   ```
   { "v": [1] }
   ```
3. Cargo sends the CredentialRequest message to the credential process (newlines added for readability).
   ```
   {
       "v": 1,
       "kind": "get",
       "operation": "read",
       "registry":{"index-url":"sparse+https://registry-
   url/index/"}
   }
   ```
4. Credential process sends the CredentialResponse to Cargo (newlines added for readability).
   ```
   {
       "token": "...",
       "cache": "session",
       "operation_independent": true
   }
   ```
5. Cargo closes the stdin pipe to the credential provider and it exits.
6. Cargo uses the token for the remainder of the session (until Cargo exits) when interacting with this registry.

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. The index format is described in Registry Index.

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 described in Registry Web API.

Commercial and community projects are available for building and running a registry. See https://github.com/rust-lang/cargo/wiki/Third-party-registries for a list of what is available.

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.

Index Configuration

The root of the index contains 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 following markers which will be replaced with their corresponding value:
  - `{crate}`: The name of crate.
  - `{version}`: The crate version.
  - `{prefix}`: A directory prefix computed from the crate name. For example, a crate named `cargo` has a prefix of `ca/rg`. See below for details.
  - `{lowerprefix}`: Lowercase variant of `{prefix}`.
  - `{sha256-checksum}`: The crate's sha256 checksum.

  If none of the markers are 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.
- `auth-required`: indicates whether this is a private registry that requires all operations to be authenticated including API requests, crate downloads and sparse index updates.

Download Endpoint

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.

If `auth-required: true` is set in `config.json`, the `Authorization` header will be included with http(s) download requests.

Index files

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.

The directory name above is calculated based on the package name converted to lowercase; it is represented by the marker `{lowerprefix}`. When the original package name is used without case conversion, the resulting directory name is represented by the marker `{prefix}`. For example, the package `MyCrate` would have a `{prefix}` of `My/Cr` and a `{lowerprefix}` of `my/cr`. In general, using `{prefix}` is recommended over `{lowerprefix}`, but there are pros and cons to each choice. Using `{prefix}` on case-insensitive filesystems results in (harmless-but-inelegant) directory aliasing. For example, `crate` and `CrateTwo` have `{prefix}` values of `cr/at` and `Cr/at`; these are distinct on Unix machines but alias to the same directory on Windows. Using directories with normalized case avoids aliasing, but on case-sensitive filesystems it's harder to support older versions of Cargo that lack `{prefix}`/`{lowerprefix}`. For

example, nginx rewrite rules can easily construct `{prefix}` but can't perform case-conversion to construct `{lowerprefix}`.

Name restrictions

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.

Version uniqueness

Indexes *must* ensure that each version only appears once for each package. This includes ignoring SemVer build metadata. For example, the index must *not* contain two entries with a version `1.0.7` and `1.0.7+extra`.

JSON schema

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://doc.rust-
lang.org/cargo/reference/specifying-dependencies.html.
            "req": "^0.6",
            // Array of features (as strings) enabled for this
dependency.
            // May be omitted since Cargo 1.84.
            "features": ["i128_support"],
```

```
             // Boolean of whether or not this is an optional
dependency.
             // Since Cargo 1.84, defaults to `false` if not
specified.
            "optional": false,
             // Boolean of whether or not default features are
enabled.
              // Since Cargo 1.84, defaults to `true` if not
specified.
            "default_features": true,
            // The target platform for the dependency.
             // If not specified or `null`, it is not a target
dependency.
            // Otherwise, a string such as "cfg(windows)".
            "target": null,
            // The dependency kind.
            // "dev", "build", or "normal".
               // If not specified or `null`, it defaults to
"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 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.
    // May be omitted since Cargo 1.84.
    "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,
    // An unsigned 32-bit integer value indicating the schema version of this
    // entry.
    //
     // If this is not specified, it should be interpreted as the default of 1.
    //
    // Cargo (starting with version 1.51) will ignore versions it does not
     // recognize. This provides a method to safely introduce changes to index
     // entries and allow older versions of cargo to ignore newer entries it
     // doesn't understand. Versions older than 1.51 ignore this field, and
    // thus may misinterpret the meaning of the index entry.
    //
    // The current values are:
    //
    // * 1: The schema as documented here, not including newer additions.
```

```
    //      This is honored in Rust version 1.51 and newer.
    // * 2: The addition of the `features2` field.
    //      This is honored in Rust version 1.60 and newer.
    "v": 2,
      // This optional field contains features with new,
extended syntax.
     // Specifically, namespaced features (`dep:`) and weak
dependencies
    // (`pkg?/feat`).
    //
     // This is separated from `features` because versions
older than 1.19
    // will fail to load due to not being able to parse the
new syntax, even
    // with a `Cargo.lock` file.
    //
     // Cargo will merge any values listed here with the
"features" field.
    //
    // If this field is included, the "v" field should be set
to at least 2.
    //
     // Registries are not required to use this field for
extended feature
     // syntax, they are allowed to include those in the
"features" field.
    // Using this is only necessary if the registry wants to
support cargo
    // versions older than 1.19, which in practice is only
crates.io since
    // those older versions do not support other registries.
    "features2": {
        "serde": ["dep:serde", "chrono?/serde"]
    }
    // The minimal supported Rust version (optional)
    // This must be a valid version requirement without an
```

```
operator (e.g. no `=`)
    "rust_version": "1.60"
}
```

The JSON objects should not be modified after they are added except for the `yanked` field whose value may change at any time.

Note: The index JSON format has subtle differences from the JSON format of the Publish API and `cargo metadata`. If you are using one of those as a source to generate index entries, you are encouraged to carefully inspect the documentation differences between them.

For the Publish API, the differences are:

- `deps`
  - `name` --- When the dependency is renamed in `Cargo.toml`, the publish API puts the original package name in the `name` field and the aliased name in the `explicit_name_in_toml` field. The index places the aliased name in the `name` field, and the original package name in the `package` field.
  - `req` --- The Publish API field is called `version_req`.
- `cksum` --- The publish API does not specify the checksum, it must be computed by the registry before adding to the index.
- `features` --- Some features may be placed in the `features2` field. Note: This is only a legacy requirement for crates.io; other registries should not need to bother with modifying the features map. The `v` field indicates the presence of the `features2` field.
- The publish API includes several other fields, such as `description` and `readme`, which don't appear in the index. These are intended to make it easier for a registry to obtain the metadata about the crate to display on a website without needing to extract and parse the `.crate` file. This additional information is typically added to a database on the registry server.

- Although `rust_version` is included here, crates.io will ignore this field and instead read it from the `Cargo.toml` contained in the `.crate` file.

For `cargo metadata`, the differences are:

- `vers` --- The `cargo metadata` field is called `version`.
- `deps`
    - `name` --- When the dependency is renamed in `Cargo.toml`, `cargo metadata` puts the original package name in the `name` field and the aliased name in the `rename` field. The index places the aliased name in the `name` field, and the original package name in the `package` field.
    - `default_features` --- The `cargo metadata` field is called `uses_default_features`.
    - `registry` --- `cargo metadata` uses a value of `null` to indicate that the dependency comes from crates.io. The index uses a value of `null` to indicate that the dependency comes from the same registry as the index. When creating an index entry, a registry other than crates.io should translate a value of `null` to be `https://github.com/rust-lang/crates.io-index` and translate a URL that matches the current index to be `null`.
    - `cargo metadata` includes some extra fields, such as `source` and `path`.
- The index includes additional fields such as `yanked`, `cksum`, and `v`.

Index Protocols

Cargo supports two remote registry protocols: `git` and `sparse`. The `git` protocol stores index files in a git repository and the `sparse` protocol fetches individual files over HTTP.

Git Protocol

The git protocol has no protocol prefix in the index url. For example the git index URL for crates.io is `https://github.com/rust-lang/crates.io-index`.

Cargo caches the git repository on disk so that it can efficiently incrementally fetch updates.

Sparse Protocol

The sparse protocol uses the `sparse+` protocol prefix in the registry URL. For example, the sparse index URL for crates.io is `sparse+https://index.crates.io/`.

The sparse protocol downloads each index file using an individual HTTP request. Since this results in a large number of small HTTP requests, performance is significantly improved with a server that supports pipelining and HTTP/2.

Sparse authentication

Cargo will attempt to fetch the `config.json` file before fetching any other files. If the server responds with an HTTP 401, then Cargo will assume that the registry requires authentication and re-attempt the request for `config.json` with the authentication token included.

On authentication failure (or a missing authentication token) the server may include a `www-authenticate` header with a `Cargo login_url="<URL>"` challenge to indicate where the user can go to get a token.

Registries that require authentication must set `auth-required: true` in `config.json`.

Caching

Cargo caches the crate metadata files, and captures the `ETag` or `Last-Modified` HTTP header from the server for each entry. When refreshing crate metadata, Cargo sends the `If-None-Match` or `If-Modified-Since` header to allow the server to respond with HTTP 304 "Not Modified" if the local cache is valid, saving time and bandwidth. If both `ETag` and `Last-Modified` headers are present, Cargo uses the `ETag` only.

Cache Invalidation

If a registry is using some kind of CDN or proxy which caches access to the index files, then it is recommended that registries implement some form of cache invalidation when the files are updated. If these caches are not updated, then users may not be able to access new crates until the cache is cleared.

Nonexistent Crates

For crates that do not exist, the registry should respond with a 404 "Not Found", 410 "Gone" or 451 "Unavailable For Legal Reasons" code.

Sparse Limitations

Since the URL of the registry is stored in the lockfile, it's not recommended to offer a registry with both protocols. Discussion about a transition plan is ongoing in issue #10964. The crates.io registry is an exception, since Cargo internally substitutes the equivalent git URL when the sparse protocol is used.

If a registry does offer both protocols, it's currently recommended to choose one protocol as the canonical protocol and use source replacement for the other protocol.

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 2xx response code for success. Errors should use an appropriate response code, such as 404. Failure responses should 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"
        }
    ]
}
```

If the response has this structure Cargo will display the detailed message to the user, even if the response code is 200. If the response code indicates an error and the content does not have this structure, Cargo will display to the user a message intended to help debugging the server error. A server returning an `errors` object allows a registry to provide a more detailed or user-centric error message.

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` (for requests with a body payload)
- `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.

It is not required for the index to be updated before the successful response is sent. After a successful response, Cargo will poll the index for a short period of time to identify that the new crate has been added. If the crate does not appear in the index after a short period of time, then Cargo will display a warning letting the user know that the new crate is not yet available.

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.
    "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,
    // The minimal supported Rust version (optional)
     // This must be a valid version requirement without an
operator (e.g. no `=`)
    "rust_version": 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 yank 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 unyank 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
    // A string to be displayed to the user. Currently ignored
```

```
by cargo.
    "msg": "owners successfully removed",
}
```

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.

SemVer Compatibility

This chapter provides details on what is conventionally considered a compatible or breaking SemVer change for new releases of a package. See the SemVer compatibility section for details on what SemVer is, and how Cargo uses it to ensure compatibility of libraries.

These are only *guidelines*, and not necessarily hard-and-fast rules that all projects will obey. The Change categories section details how this guide classifies the level and severity of a change. Most of this guide focuses on changes that will cause `cargo` and `rustc` to fail to build something that previously worked. Almost every change carries some risk that it will negatively affect the runtime behavior, and for those cases it is usually a judgment call by the project maintainers whether or not it is a SemVer-incompatible change.

Change categories

All of the policies listed below are categorized by the level of change:

- **Major change**: a change that requires a major SemVer bump.
- **Minor change**: a change that requires only a minor SemVer bump.
- **Possibly-breaking change**: a change that some projects may consider major and others consider minor.

The "Possibly-breaking" category covers changes that have the *potential* to break during an update, but may not necessarily cause a breakage. The impact of these changes should be considered carefully. The exact nature will depend on the change and the principles of the project maintainers.

Some projects may choose to only bump the patch number on a minor change. It is encouraged to follow the SemVer spec, and only apply bug fixes in patch releases. However, a bug fix may require an API change that is marked as a "minor change", and shouldn't affect compatibility. This guide does not take a stance on how each individual "minor change" should be treated, as the difference between minor and patch changes are conventions that depend on the nature of the change.

Some changes are marked as "minor", even though they carry the potential risk of breaking a build. This is for situations where the potential is extremely low, and the potentially breaking code is unlikely to be written in idiomatic Rust, or is specifically discouraged from use.

This guide uses the terms "major" and "minor" assuming this relates to a "1.0.0" release or later. Initial development releases starting with "0.y.z" can treat changes in "y" as a major release, and "z" as a minor release. "0.0.z" releases are always major changes. This is because Cargo uses the convention that only changes in the left-most non-zero component are considered incompatible.

- API compatibility
    - Items
        - Major: renaming/moving/removing any public items

        - Minor: adding new public items
    - Types
        - Major: Changing the alignment, layout, or size of a well-defined type
    - Structs
        - Major: adding a private struct field when all current fields are public
        - Major: adding a public field when no private field exists
        - Minor: adding or removing private fields when at least one already exists
        - Minor: going from a tuple struct with all private fields (with at least one field) to a normal struct, or vice versa
    - Enums
        - Major: adding new enum variants (without `non_exhaustive`)
        - Major: adding new fields to an enum variant
    - Traits
        - Major: adding a non-defaulted trait item
        - Major: any change to trait item signatures
        - Possibly-breaking: adding a defaulted trait item
        - Major: adding a trait item that makes the trait non-object safe
        - Major: adding a type parameter without a default
        - Minor: adding a defaulted trait type parameter
    - Implementations
        - Possibly-breaking change: adding any inherent items
    - Generics
        - Major: tightening generic bounds
        - Minor: loosening generic bounds
        - Minor: adding defaulted type parameters

        - Minor: generalizing a type to use generics (with identical types)
        - Major: generalizing a type to use generics (with possibly different types)
        - Minor: changing a generic type to a more generic type
        - Major: capturing more generic parameters in RPIT
    - Functions
        - Major: adding/removing function parameters
        - Possibly-breaking: introducing a new function type parameter
        - Minor: generalizing a function to use generics (supporting original type)
        - Major: generalizing a function to use generics with type mismatch
        - Minor: making an `unsafe` function safe
    - Attributes
        - Major: switching from `no_std` support to requiring `std`
        - Major: adding `non_exhaustive` to an existing enum, variant, or struct with no private fields
- Tooling and environment compatibility
    - Possibly-breaking: changing the minimum version of Rust required
    - Possibly-breaking: changing the platform and environment requirements
    - Minor: introducing new lints
    - Cargo
        - Minor: adding a new Cargo feature
        - Major: removing a Cargo feature
        - Major: removing a feature from a feature list if that changes functionality or public items
        - Possibly-breaking: removing an optional dependency
        - Minor: changing dependency features

API compatibility

All of the examples below contain three parts: the original code, the code after it has been modified, and an example usage of the code that could appear in another project. In a minor change, the example usage should successfully build with both the before and after versions.

Major: renaming/moving/removing any public items {#item-remove}

The absence of a publicly exposed item will cause any uses of that item to fail to compile.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub fn foo() {}

////////////////////////////////////////////////////////
// After
// ... item has been removed

////////////////////////////////////////////////////////
// Example usage that will break.
fn main() {
    updated_crate::foo(); // Error: cannot find function `foo`
}
```

This includes adding any sort of `cfg` attribute which can change which items or behavior is available based on conditional compilation.

Mitigating strategies:

- Mark items to be removed as deprecated, and then remove them at a later date in a SemVer-breaking release.
- Mark renamed items as deprecated, and use a `pub use` item to re-export to the old name.

Minor: adding new public items {#item-new}

Adding new, public items is a minor change.

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
// ... absence of item

////////////////////////////////////////////////////////
// After
pub fn foo() {}

////////////////////////////////////////////////////////
// Example use of the library that will safely work.
// `foo` is not used since it didn't previously exist.
```

Note that in some rare cases this can be a **breaking change** due to glob imports. For example, if you add a new trait, and a project has used a glob import that brings that trait into scope, and the new trait introduces an associated item that conflicts with any types it is implemented on, this can cause a compile-time error due to the ambiguity. Example:

```
// Breaking change example

////////////////////////////////////////////////////////
// Before
// ... absence of trait

////////////////////////////////////////////////////////
// After
pub trait NewTrait {
    fn foo(&self) {}
}

impl NewTrait for i32 {}
```

```
////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::*;

pub trait LocalTrait {
    fn foo(&self) {}
}

impl LocalTrait for i32 {}

fn main() {
      123i32.foo(); // Error:   multiple applicable items in
scope
}
```

This is not considered a major change because conventionally glob imports are a known forwards-compatibility hazard. Glob imports of items from external crates should be avoided.

Major: Changing the alignment, layout, or size of a well-defined type {#type-layout}

It is a breaking change to change the alignment, layout, or size of a type that was previously well-defined.

In general, types that use the [the default representation] do not have a well-defined alignment, layout, or size. The compiler is free to alter the alignment, layout, or size, so code should not make any assumptions about it.

> **Note**: It may be possible for external crates to break if they make assumptions about the alignment, layout, or size of a type even if it is not well-defined. This is not considered a SemVer breaking change since those assumptions should not be made.

Some examples of changes that are not a breaking change are (assuming no other rules in this guide are violated):

- Adding, removing, reordering, or changing fields of a default representation struct, union, or enum in such a way that the change follows the other rules in this guide (for example, using `non_exhaustive` to allow those changes, or changes to private fields that are already private). See struct-add-private-field-when-public, struct-add-public-field-when-no-private, struct-private-fields-with-private, enum-fields-new.
- Adding variants to a default representation enum, if the enum uses `non_exhaustive`. This may change the alignment or size of the enumeration, but those are not well-defined. See enum-variant-new.
- Adding, removing, reordering, or changing private fields of a `repr(C)` struct, union, or enum, following the other rules in this guide (for example, using `non_exhaustive`, or adding private fields when other private fields already exist). See repr-c-private-change.
- Adding variants to a `repr(C)` enum, if the enum uses `non_exhaustive`. See repr-c-enum-variant-new.
- Adding `repr(C)` to a default representation struct, union, or enum. See repr-c-add.
- Adding `repr(<int>)` primitive representation to an enum. See repr-int-enum-add.
- Adding `repr(transparent)` to a default representation struct or enum. See repr-transparent-add.

Types that use the `repr` attribute can be said to have an alignment and layout that is defined in some way that code may make some assumptions about that may break as a result of changing that type.

In some cases, types with a `repr` attribute may not have an alignment, layout, or size that is well-defined. In these cases, it may be safe to make changes to the types, though care should be exercised. For example, types with private fields that do not otherwise document their alignment, layout, or size guarantees cannot be relied upon by external crates since the public API does not fully define the alignment, layout, or size of the type.

A common example where a type with *private* fields is well-defined is a type with a single private field with a generic type, using

`repr(transparent)`, and the prose of the documentation discusses that it is transparent to the generic type. For example, see `UnsafeCell`.

Some examples of breaking changes are:

- Adding `repr(packed)` to a struct or union. See repr-packed-add.
- Adding `repr(align)` to a struct, union, or enum. See repr-align-add.
- Removing `repr(packed)` from a struct or union. See repr-packed-remove.
- Changing the value N of `repr(packed(N))` if that changes the alignment or layout. See repr-packed-n-change.
- Changing the value N of `repr(align(N))` if that changes the alignment. See repr-align-n-change.
- Removing `repr(align)` from a struct, union, or enum. See repr-align-remove.
- Changing the order of public fields of a `repr(C)` type. See repr-c-shuffle.
- Removing `repr(C)` from a struct, union, or enum. See repr-c-remove.
- Removing `repr(<int>)` from an enum. See repr-int-enum-remove.
- Changing the primitive representation of a `repr(<int>)` enum. See repr-int-enum-change.
- Removing `repr(transparent)` from a struct or enum. See repr-transparent-remove.

Minor: `repr(C)` add, remove, or change a private field {#repr-c-private-change}

It is usually safe to add, remove, or change a private field of a `repr(C)` struct, union, or enum, assuming it follows the other guidelines in this guide (see struct-add-private-field-when-public, struct-add-public-field-when-no-private, struct-private-fields-with-private, enum-fields-new).

For example, adding private fields can only be done if there are already other private fields, or it is `non_exhaustive`. Public fields may be added if there are private fields, or it is `non_exhaustive`, and the addition does not alter the layout of the other fields.

However, this may change the size and alignment of the type. Care should be taken if the size or alignment changes. Code should not make assumptions about the size or alignment of types with private fields or `non_exhaustive` unless it has a documented size or alignment.

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
#[derive(Default)]
#[repr(C)]
pub struct Example {
    pub f1: i32,
    f2: i32, // a private field
}

////////////////////////////////////////////////////////
// After
#[derive(Default)]
#[repr(C)]
pub struct Example {
    pub f1: i32,
    f2: i32,
    f3: i32, // a new field
}

////////////////////////////////////////////////////////
// Example use of the library that will safely work.
fn main() {
    // NOTE: Users should not make assumptions about the size
or alignment
    // since they are not documented.
    let f = updated_crate::Example::default();
}
```

Minor: `repr(C)` add enum variant {#repr-c-enum-variant-new}

It is usually safe to add variants to a `repr(C)` enum, if the enum uses `non_exhaustive`. See enum-variant-new for more discussion.

Note that this may be a breaking change since it changes the size and alignment of the type. See repr-c-private-change for similar concerns.

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
#[repr(C)]
#[non_exhaustive]
pub enum Example {
    Variant1 { f1: i16 },
    Variant2 { f1: i32 },
}

////////////////////////////////////////////////////////
// After
#[repr(C)]
#[non_exhaustive]
pub enum Example {
    Variant1 { f1: i16 },
    Variant2 { f1: i32 },
    Variant3 { f1: i64 }, // added
}

////////////////////////////////////////////////////////
// Example use of the library that will safely work.
fn main() {
    // NOTE: Users should not make assumptions about the size
or alignment
    // since they are not specified. For example, this raised
the size from 8
    // to 16 bytes.
    let f = updated_crate::Example::Variant2 { f1: 123 };
}
```

Minor: Adding `repr(C)` to a default representation {#repr-c-add}

It is safe to add `repr(C)` to a struct, union, or enum with the default representation. This is safe because users should not make assumptions about the alignment, layout, or size of types with the default representation.

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
pub struct Example {
    pub f1: i32,
    pub f2: i16,
}

////////////////////////////////////////////////////////
// After
#[repr(C)] // added
pub struct Example {
    pub f1: i32,
    pub f2: i16,
}

////////////////////////////////////////////////////////
// Example use of the library that will safely work.
fn main() {
    let f = updated_crate::Example { f1: 123, f2: 456 };
}
```

Minor: Adding `repr(<int>)` to an enum {#repr-int-enum-add}

It is safe to add `repr(<int>)` primitive representation to an enum with the default representation. This is safe because users should not make assumptions about the alignment, layout, or size of an enum with the default representation.

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
pub enum E {
    Variant1,
    Variant2(i32),
    Variant3 { f1: f64 },
}

////////////////////////////////////////////////////////
// After
#[repr(i32)] // added
pub enum E {
    Variant1,
    Variant2(i32),
    Variant3 { f1: f64 },
}

////////////////////////////////////////////////////////
// Example use of the library that will safely work.
fn main() {
    let x = updated_crate::E::Variant3 { f1: 1.23 };
}
```

Minor: Adding `repr(transparent)` to a default representation struct or enum {#repr-transparent-add}

It is safe to add `repr(transparent)` to a struct or enum with the default representation. This is safe because users should not make assumptions about the alignment, layout, or size of a struct or enum with the default representation.

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
```

```
#[derive(Default)]
pub struct Example<T>(T);

////////////////////////////////////////////////////////
// After
#[derive(Default)]
#[repr(transparent)] // added
pub struct Example<T>(T);

////////////////////////////////////////////////////////
// Example use of the library that will safely work.
fn main() {
    let x = updated_crate::Example::<i32>::default();
}
```

Major: Adding `repr(packed)` to a struct or union {#repr-packed-add}

It is a breaking change to add `repr(packed)` to a struct or union. Making a type `repr(packed)` makes changes that can break code, such as being invalid to take a reference to a field, or causing truncation of disjoint closure captures.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub struct Example {
    pub f1: u8,
    pub f2: u16,
}

////////////////////////////////////////////////////////
// After
#[repr(packed)] // added
pub struct Example {
    pub f1: u8,
```

```
    pub f2: u16,
}

////////////////////////////////////////////////////////
// Example usage that will break.
fn main() {
    let f = updated_crate::Example { f1: 1, f2: 2 };
     let x = &f.f2; // Error: reference to packed field is
unaligned
}
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub struct Example(pub i32, pub i32);

////////////////////////////////////////////////////////
// After
#[repr(packed)]
pub struct Example(pub i32, pub i32);

////////////////////////////////////////////////////////
// Example usage that will break.
fn main() {
    let mut f = updated_crate::Example(123, 456);
    let c = || {
            // Without repr(packed), the closure precisely
captures `&f.0`.
         // With repr(packed), the closure captures `&f` to
avoid undefined behavior.
        let a = f.0;
    };
    f.1 = 789; // Error: cannot assign to `f.1` because it is
borrowed
```

```
    c();
}
```

Major: Adding `repr(align)` to a struct, union, or enum {#repr-align-add}

It is a breaking change to add `repr(align)` to a struct, union, or enum. Making a type `repr(align)` would break any use of that type in a `repr(packed)` type because that combination is not allowed.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub struct Aligned {
    pub a: i32,
}

////////////////////////////////////////////////////////
// After
#[repr(align(8))] // added
pub struct Aligned {
    pub a: i32,
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Aligned;

#[repr(packed)]
pub struct Packed { // Error: packed type cannot transitively
contain a `#[repr(align)]` type
    f1: Aligned,
}

fn main() {
    let p = Packed {
```

```
        f1: Aligned { a: 123 },
    };
}
```

Major: Removing `repr(packed)` from a struct or union {#repr-packed-remove}

It is a breaking change to remove `repr(packed)` from a struct or union. This may change the alignment or layout that extern crates are relying on.

If any fields are public, then removing `repr(packed)` may change the way disjoint closure captures work. In some cases, this can cause code to break, similar to those outlined in the edition guide.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
#[repr(C, packed)]
pub struct Packed {
    pub a: u8,
    pub b: u16,
}

////////////////////////////////////////////////////////
// After
#[repr(C)] // removed packed
pub struct Packed {
    pub a: u8,
    pub b: u16,
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Packed;

fn main() {
    let p = Packed { a: 1, b: 2 };
```

```
    // Some assumption about the size of the type.
    // Without `packed`, this fails since the size is 4.
    const _: () = assert!(std::mem::size_of::<Packed>() == 3);
// Error: evaluation of constant value failed
}
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
#[repr(C, packed)]
pub struct Packed {
    pub a: *mut i32,
    pub b: i32,
}
unsafe impl Send for Packed {}

////////////////////////////////////////////////////////
// After
#[repr(C)] // removed packed
pub struct Packed {
    pub a: *mut i32,
    pub b: i32,
}
unsafe impl Send for Packed {}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Packed;

fn main() {
    let mut x = 123;

    let p = Packed {
        a: &mut x as *mut i32,
        b: 456,
```

```
    };

    // When the structure was packed, the closure captures `p`
which is Send.
    // When `packed` is removed, this ends up capturing `p.a`
which is not Send.
    std::thread::spawn(move || unsafe {
        *(p.a) += 1; // Error: cannot be sent between threads
safely
    });
}
```

Major: Changing the value N of `repr(packed(N))` if that changes the alignment or layout {#repr-packed-n-change}

It is a breaking change to change the value of N of `repr(packed(N))` if that changes the alignment or layout. This may change the alignment or layout that external crates are relying on.

If the value `N` is lowered below the alignment of a public field, then that would break any code that attempts to take a reference of that field.

Note that some changes to `N` may not change the alignment or layout, for example increasing it when the current value is already equal to the natural alignment of the type.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
#[repr(packed(4))]
pub struct Packed {
    pub a: u8,
    pub b: u32,
}

////////////////////////////////////////////////////////
// After
#[repr(packed(2))] // changed to 2
```

```
pub struct Packed {
    pub a: u8,
    pub b: u32,
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Packed;

fn main() {
    let p = Packed { a: 1, b: 2 };
      let x = &p.b; // Error: reference to packed field is
unaligned
}
```

Major: Changing the value N of `repr(align(N))` if that changes the alignment {#repr-align-n-change}

It is a breaking change to change the value `N` of `repr(align(N))` if that changes the alignment. This may change the alignment that external crates are relying on.

This change should be safe to make if the type is not well-defined as discussed in type layout (such as having any private fields and having an undocumented alignment or layout).

Note that some changes to `N` may not change the alignment or layout, for example decreasing it when the current value is already equal to or less than the natural alignment of the type.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
#[repr(align(8))]
pub struct Packed {
    pub a: u8,
    pub b: u32,
}
```

```
////////////////////////////////////////////////////////
// After
#[repr(align(4))] // changed to 4
pub struct Packed {
    pub a: u8,
    pub b: u32,
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Packed;

fn main() {
    let p = Packed { a: 1, b: 2 };
    // Some assumption about the size of the type.
    // The alignment has changed from 8 to 4.
     const _: () = assert!(std::mem::align_of::<Packed>() ==
8); // Error: evaluation of constant value failed
}
```

Major: Removing `repr(align)` from a struct, union, or enum {#repr-align-remove}

It is a breaking change to remove `repr(align)` from a struct, union, or enum, if their layout was well-defined. This may change the alignment or layout that external crates are relying on.

This change should be safe to make if the type is not well-defined as discussed in type layout (such as having any private fields and having an undocumented alignment).

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
#[repr(C, align(8))]
pub struct Packed {
```

```
    pub a: u8,
    pub b: u32,
}

////////////////////////////////////////////////////////
// After
#[repr(C)] // removed align
pub struct Packed {
    pub a: u8,
    pub b: u32,
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Packed;

fn main() {
    let p = Packed { a: 1, b: 2 };
    // Some assumption about the size of the type.
    // The alignment has changed from 8 to 4.
     const _: () = assert!(std::mem::align_of::<Packed>() ==
8); // Error: evaluation of constant value failed
}
```

Major: Changing the order of public fields of a `repr(C)` type {#repr-c-shuffle}

It is a breaking change to change the order of public fields of a `repr(C)` type. External crates may be relying on the specific ordering of the fields.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
#[repr(C)]
pub struct SpecificLayout {
    pub a: u8,
```

```
    pub b: u32,
}

////////////////////////////////////////////////////////
// After
#[repr(C)]
pub struct SpecificLayout {
    pub b: u32, // changed order
    pub a: u8,
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::SpecificLayout;

unsafe extern "C" {
     // This C function is assuming a specific layout defined
in a C header.
    fn c_fn_get_b(x: &SpecificLayout) -> u32;
}

fn main() {
    let p = SpecificLayout { a: 1, b: 2 };
     unsafe { assert_eq!(c_fn_get_b(&p), 2) } // Error: value
not equal to 2
}

# mod cdep {
#       // This simulates what would normally be something
included from a build script.
#     // This definition would be in a C header.
#     #[repr(C)]
#     pub struct SpecificLayout {
#         pub a: u8,
#         pub b: u32,
#     }
```

```
#
#     #[no_mangle]
#     pub fn c_fn_get_b(x: &SpecificLayout) -> u32 {
#         x.b
#     }
# }
```

Major: Removing `repr(C)` from a struct, union, or enum {#repr-c-remove}

It is a breaking change to remove `repr(C)` from a struct, union, or enum. External crates may be relying on the specific layout of the type.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
#[repr(C)]
pub struct SpecificLayout {
    pub a: u8,
    pub b: u32,
}

////////////////////////////////////////////////////////
// After
// removed repr(C)
pub struct SpecificLayout {
    pub a: u8,
    pub b: u32,
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::SpecificLayout;

unsafe extern "C" {
     // This C function is assuming a specific layout defined
```

```
in a C header.
    fn c_fn_get_b(x: &SpecificLayout) -> u32; // Error: is not
FFI-safe
}

fn main() {
    let p = SpecificLayout { a: 1, b: 2 };
    unsafe { assert_eq!(c_fn_get_b(&p), 2) }
}

# mod cdep {
#       // This simulates what would normally be something
included from a build script.
#     // This definition would be in a C header.
#     #[repr(C)]
#     pub struct SpecificLayout {
#         pub a: u8,
#         pub b: u32,
#     }
#
#     #[no_mangle]
#     pub fn c_fn_get_b(x: &SpecificLayout) -> u32 {
#         x.b
#     }
# }
```

Major: Removing `repr(<int>)` from an enum {#repr-int-enum-remove}

It is a breaking change to remove `repr(<int>)` from an enum. External crates may be assuming that the discriminant is a specific size. For example, `std::mem::transmute` of an enum may fail.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
```

```
#[repr(u16)]
pub enum Example {
    Variant1,
    Variant2,
    Variant3,
}

////////////////////////////////////////////////////////
// After
// removed repr(u16)
pub enum Example {
    Variant1,
    Variant2,
    Variant3,
}

////////////////////////////////////////////////////////
// Example usage that will break.

fn main() {
    let e = updated_crate::Example::Variant2;
    let i: u16 = unsafe { std::mem::transmute(e) }; // Error:
cannot transmute between types of different sizes
}
```

Major: Changing the primitive representation of a `repr(<int>)` enum {#repr-int-enum-change}

It is a breaking change to change the primitive representation of a `repr(<int>)` enum. External crates may be assuming that the discriminant is a specific size. For example, `std::mem::transmute` of an enum may fail.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
```

```
#[repr(u16)]
pub enum Example {
    Variant1,
    Variant2,
    Variant3,
}

////////////////////////////////////////////////////////
// After
#[repr(u8)] // changed repr size
pub enum Example {
    Variant1,
    Variant2,
    Variant3,
}

////////////////////////////////////////////////////////
// Example usage that will break.

fn main() {
    let e = updated_crate::Example::Variant2;
    let i: u16 = unsafe { std::mem::transmute(e) }; // Error:
cannot transmute between types of different sizes
}
```

Major: Removing `repr(transparent)` from a struct or enum {#repr-transparent-remove}

It is a breaking change to remove `repr(transparent)` from a struct or enum. External crates may be relying on the type having the alignment, layout, or size of the transparent field.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
#[repr(transparent)]
```

```
pub struct Transparent<T>(T);

////////////////////////////////////////////////////////
// After
// removed repr
pub struct Transparent<T>(T);

////////////////////////////////////////////////////////
// Example usage that will break.
#![deny(improper_ctypes)]
use updated_crate::Transparent;

unsafe extern "C" {
    fn c_fn() -> Transparent<f64>; // Error: is not FFI-safe
}

fn main() {}
```

Major: adding a private struct field when all current fields are public {#struct-add-private-field-when-public}

When a private field is added to a struct that previously had all public fields, this will break any code that attempts to construct it with a struct literal.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub struct Foo {
    pub f1: i32,
}

////////////////////////////////////////////////////////
// After
```

```
pub struct Foo {
    pub f1: i32,
    f2: i32,
}

////////////////////////////////////////////////////////
// Example usage that will break.
fn main() {
     let x = updated_crate::Foo { f1: 123 }; // Error: cannot
construct `Foo`
}
```

Mitigation strategies:

- Do not add new fields to all-public field structs.
- Mark structs as `#[non_exhaustive]` when first introducing a struct to prevent users from using struct literal syntax, and instead provide a constructor method and/or Default implementation.

Major: adding a public field when no private field exists {#struct-add-public-field-when-no-private}

When a public field is added to a struct that has all public fields, this will break any code that attempts to construct it with a struct literal.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub struct Foo {
    pub f1: i32,
}

////////////////////////////////////////////////////////
// After
pub struct Foo {
    pub f1: i32,
    pub f2: i32,
```

```
}

////////////////////////////////////////////////////////
// Example usage that will break.
fn main() {
    let x = updated_crate::Foo { f1: 123 }; // Error: missing
field `f2`
}
```

Mitigation strategies:

- Do not add new fields to all-public field structs.
- Mark structs as `#[non_exhaustive]` when first introducing a struct to prevent users from using struct literal syntax, and instead provide a constructor method and/or Default implementation.

Minor: adding or removing private fields when at least one already exists {#struct-private-fields-with-private}

It is safe to add or remove private fields from a struct when the struct already has at least one private field.

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
#[derive(Default)]
pub struct Foo {
    f1: i32,
}

////////////////////////////////////////////////////////
// After
#[derive(Default)]
pub struct Foo {
    f2: f64,
```

```
}

////////////////////////////////////////////////////////
// Example use of the library that will safely work.
fn main() {
    // Cannot access private fields.
    let x = updated_crate::Foo::default();
}
```

This is safe because existing code cannot use a struct literal to construct it, nor exhaustively match its contents.

Note that for tuple structs, this is a **major change** if the tuple contains public fields, and the addition or removal of a private field changes the index of any public field.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
#[derive(Default)]
pub struct Foo(pub i32, i32);

////////////////////////////////////////////////////////
// After
#[derive(Default)]
pub struct Foo(f64, pub i32, i32);

////////////////////////////////////////////////////////
// Example usage that will break.
fn main() {
    let x = updated_crate::Foo::default();
    let y = x.0; // Error: is private
}
```

Minor: going from a tuple struct with all private fields (with at least one field) to a normal struct, or vice versa {#struct-tuple-normal-with-private}

Changing a tuple struct to a normal struct (or vice-versa) is safe if all fields are private.

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
#[derive(Default)]
pub struct Foo(i32);

////////////////////////////////////////////////////////
// After
#[derive(Default)]
pub struct Foo {
    f1: i32,
}

////////////////////////////////////////////////////////
// Example use of the library that will safely work.
fn main() {
    // Cannot access private fields.
    let x = updated_crate::Foo::default();
}
```

This is safe because existing code cannot use a struct literal to construct it, nor match its contents.

Major: adding new enum variants (without `non_exhaustive`) {#enum-variant-new}

It is a breaking change to add a new enum variant if the enum does not use the `#[non_exhaustive]` attribute.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub enum E {
    Variant1,
}

////////////////////////////////////////////////////////
// After
pub enum E {
    Variant1,
    Variant2,
}

////////////////////////////////////////////////////////
// Example usage that will break.
fn main() {
    use updated_crate::E;
    let x = E::Variant1;
    match x { // Error: `E::Variant2` not covered
        E::Variant1 => {}
    }
}
```

Mitigation strategies:

- When introducing the enum, mark it as `#[non_exhaustive]` to force users to use wildcard patterns to catch new variants.

Major: adding new fields to an enum variant {#enum-fields-new}

It is a breaking change to add new fields to an enum variant because all fields are public, and constructors and matching will fail to compile.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub enum E {
    Variant1 { f1: i32 },
}

////////////////////////////////////////////////////////
// After
pub enum E {
    Variant1 { f1: i32, f2: i32 },
}

////////////////////////////////////////////////////////
// Example usage that will break.
fn main() {
    use updated_crate::E;
    let x = E::Variant1 { f1: 1 }; // Error: missing f2
    match x {
        E::Variant1 { f1 } => {} // Error: missing f2
    }
}
```

Mitigation strategies:

- When introducing the enum, mark the variant as `non_exhaustive` so that it cannot be constructed or matched without wildcards.

  ```
  pub enum E {
      #[non_exhaustive]
      Variant1{f1: i32}
  }
  ```

- When introducing the enum, use an explicit struct as a value, where you can have control over the field visibility.

  ```
  pub struct Foo {
     f1: i32,
  ```

```
   f2: i32,
}
pub enum E {
    Variant1(Foo)
}
```

Major: adding a non-defaulted trait item {#trait-new-item-no-default}

It is a breaking change to add a non-defaulted item to a trait. This will break any implementors of the trait.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub trait Trait {}

////////////////////////////////////////////////////////
// After
pub trait Trait {
    fn foo(&self);
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Trait;
struct Foo;

impl Trait for Foo {}    // Error: not all trait items implemented
```

Mitigation strategies:

- Always provide a default implementation or value for new associated trait items.
- When introducing the trait, use the sealed trait technique to prevent users outside of the crate from implementing the trait.

Major: any change to trait item signatures {#trait-item-signature}

It is a breaking change to make any change to a trait item signature. This can break external implementors of the trait.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub trait Trait {
    fn f(&self, x: i32) {}
}

////////////////////////////////////////////////////////
// After
pub trait Trait {
    // For sealed traits or normal functions, this would be a
minor change
    // because generalizing with generics strictly expands the
possible uses.
     // But in this case, trait implementations must use the
same signature.
    fn f<V>(&self, x: V) {}
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Trait;
struct Foo;

impl Trait for Foo {
    fn f(&self, x: i32) {}  // Error: trait declaration has 1
type parameter
}
```

Mitigation strategies:

- Introduce new items with default implementations to cover the new functionality instead of modifying existing items.
- When introducing the trait, use the sealed trait technique to prevent users outside of the crate from implementing the trait.

Possibly-breaking: adding a defaulted trait item {#trait-new-default-item}

It is usually safe to add a defaulted trait item. However, this can sometimes cause a compile error. For example, this can introduce an ambiguity if a method of the same name exists in another trait.

```
// Breaking change example

////////////////////////////////////////////////////////
// Before
pub trait Trait {}

////////////////////////////////////////////////////////
// After
pub trait Trait {
    fn foo(&self) {}
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Trait;
struct Foo;

trait LocalTrait {
    fn foo(&self) {}
}

impl Trait for Foo {}
impl LocalTrait for Foo {}

fn main() {
```

```
    let x = Foo;
    x.foo(); // Error: multiple applicable items in scope
}
```

Note that this ambiguity does *not* exist for name collisions on inherent implementations, as they take priority over trait items.

See trait-object-safety for a special case to consider when adding trait items.

Mitigation strategies:

- Some projects may deem this acceptable breakage, particularly if the new item name is unlikely to collide with any existing code. Choose names carefully to help avoid these collisions. Additionally, it may be acceptable to require downstream users to add disambiguation syntax to select the correct function when updating the dependency.

Major: adding a trait item that makes the trait non-object safe {#trait-object-safety}

It is a breaking change to add a trait item that changes the trait to not be object safe.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub trait Trait {}

////////////////////////////////////////////////////////
// After
pub trait Trait {
    // An associated const makes the trait not object-safe.
    const CONST: i32 = 123;
}

////////////////////////////////////////////////////////
// Example usage that will break.
```

```
use updated_crate::Trait;
struct Foo;

impl Trait for Foo {}

fn main() {
     let obj: Box<dyn Trait> = Box::new(Foo); // Error: the
trait `Trait` is not dyn compatible
}
```

It is safe to do the converse (making a non-object safe trait into a safe one).

Major: adding a type parameter without a default {#trait-new-parameter-no-default}

It is a breaking change to add a type parameter without a default to a trait.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub trait Trait {}

////////////////////////////////////////////////////////
// After
pub trait Trait<T> {}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Trait;
struct Foo;

impl Trait for Foo {}  // Error: missing generics
```

Mitigating strategies:

- See adding a defaulted trait type parameter.

Minor: adding a defaulted trait type parameter {#trait-new-parameter-default}

It is safe to add a type parameter to a trait as long as it has a default. External implementors will use the default without needing to specify the parameter.

```
// MINOR CHANGE

///////////////////////////////////////////////////////////
// Before
pub trait Trait {}

///////////////////////////////////////////////////////////
// After
pub trait Trait<T = i32> {}

///////////////////////////////////////////////////////////
// Example use of the library that will safely work.
use updated_crate::Trait;
struct Foo;

impl Trait for Foo {}
```

Possibly-breaking change: adding any inherent items {#impl-item-new}

Usually adding inherent items to an implementation should be safe because inherent items take priority over trait items. However, in some cases the collision can cause problems if the name is the same as an implemented trait item with a different signature.

```
// Breaking change example

///////////////////////////////////////////////////////////
// Before
```

```
pub struct Foo;

////////////////////////////////////////////////////////////
// After
pub struct Foo;

impl Foo {
    pub fn foo(&self) {}
}

////////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Foo;

trait Trait {
    fn foo(&self, x: i32) {}
}

impl Trait for Foo {}

fn main() {
    let x = Foo;
     x.foo(1); // Error: this method takes 0 arguments but 1
argument was supplied
}
```

Note that if the signatures match, there would not be a compile-time error, but possibly a silent change in runtime behavior (because it is now executing a different function).

Mitigation strategies:

- Some projects may deem this acceptable breakage, particularly if the new item name is unlikely to collide with any existing code. Choose names carefully to help avoid these collisions. Additionally, it may be acceptable to require downstream users to add disambiguation syntax to select the correct function when updating the dependency.

Major: tightening generic bounds {#generic-bounds-tighten}

It is a breaking change to tighten generic bounds on a type since this can break users expecting the looser bounds.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub struct Foo<A> {
    pub f1: A,
}

////////////////////////////////////////////////////////
// After
pub struct Foo<A: Eq> {
    pub f1: A,
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Foo;

fn main() {
      let s = Foo { f1: 1.23 }; // Error: the trait bound
`{float}: Eq` is not satisfied
}
```

Minor: loosening generic bounds {#generic-bounds-loosen}

It is safe to loosen the generic bounds on a type, as it only expands what is allowed.

```
// MINOR CHANGE


```

```
////////////////////////////////////////////////////////
// Before
pub struct Foo<A: Clone> {
    pub f1: A,
}

////////////////////////////////////////////////////////
// After
pub struct Foo<A> {
    pub f1: A,
}

////////////////////////////////////////////////////////
// Example use of the library that will safely work.
use updated_crate::Foo;

fn main() {
    let s = Foo { f1: 123 };
}
```

Minor: adding defaulted type parameters {#generic-new-default}

It is safe to add a type parameter to a type as long as it has a default. All existing references will use the default without needing to specify the parameter.

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
#[derive(Default)]
pub struct Foo {}

////////////////////////////////////////////////////////
// After
#[derive(Default)]
```

```
pub struct Foo<A = i32> {
    f1: A,
}

////////////////////////////////////////////////////////
// Example use of the library that will safely work.
use updated_crate::Foo;

fn main() {
    let s: Foo = Default::default();
}
```

Minor: generalizing a type to use generics (with identical types) {#generic-generalize-identical}

A struct or enum field can change from a concrete type to a generic type parameter, provided that the change results in an identical type for all existing use cases. For example, the following change is permitted:

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
pub struct Foo(pub u8);

////////////////////////////////////////////////////////
// After
pub struct Foo<T = u8>(pub T);

////////////////////////////////////////////////////////
// Example use of the library that will safely work.
use updated_crate::Foo;

fn main() {
    let s: Foo = Foo(123);
}
```

because existing uses of `Foo` are shorthand for `Foo<u8>` which yields the identical field type.

Major: generalizing a type to use generics (with possibly different types) {#generic-generalize-different}

Changing a struct or enum field from a concrete type to a generic type parameter can break if the type can change.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub struct Foo<T = u8>(pub T, pub u8);

////////////////////////////////////////////////////////
// After
pub struct Foo<T = u8>(pub T, pub T);

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::Foo;

fn main() {
     let s: Foo<f32> = Foo(3.14, 123); // Error: mismatched
types
}
```

Minor: changing a generic type to a more generic type {#generic-more-generic}

It is safe to change a generic type to a more generic one. For example, the following adds a generic parameter that defaults to the original type, which is safe because all existing users will be using the same type for both fields, the defaulted parameter does not need to be specified.

```
// MINOR CHANGE

///////////////////////////////////////////////////////
// Before
pub struct Foo<T>(pub T, pub T);

///////////////////////////////////////////////////////
// After
pub struct Foo<T, U = T>(pub T, pub U);

///////////////////////////////////////////////////////
// Example use of the library that will safely work.
use updated_crate::Foo;

fn main() {
    let s: Foo<f32> = Foo(1.0, 2.0);
}
```

Major: capturing more generic parameters in RPIT {#generic-rpit-capture}

It is a breaking change to capture additional generic parameters in an RPIT (return-position impl trait).

```
// MAJOR CHANGE

///////////////////////////////////////////////////////
// Before
pub fn f<'a, 'b>(x: &'a str, y: &'b str) -> impl Iterator<Item
= char> + use<'a> {
    x.chars()
}

///////////////////////////////////////////////////////
// After
pub fn f<'a, 'b>(x: &'a str, y: &'b str) -> impl Iterator<Item
= char> + use<'a, 'b> {
```

```
    x.chars().chain(y.chars())
}

////////////////////////////////////////////////////////
// Example usage that will break.
fn main() {
    let a = String::new();
    let b = String::new();
    let iter = updated_crate::f(&a, &b);
     drop(b); // Error: cannot move out of `b` because it is
borrowed
}
```

Adding generic parameters to an RPIT places additional constraints on how the resulting type may be used.

Note that there are implicit captures when the `use<>` syntax is not specified. In Rust 2021 and earlier editions, the lifetime parameters are only captured if they appear syntactically within a bound in the RPIT type signature. Starting in Rust 2024, all lifetime parameters are unconditionally captured. This means that starting in Rust 2024, the default is maximally compatible, requiring you to be explicit when you want to capture less, which is a SemVer commitment.

See the edition guide and the reference for more information on RPIT capturing.

It is a minor change to capture fewer generic parameters in an RPIT.

Note: All in-scope type and const generic parameters must be either implicitly captured (no `+ use<…>` specified) or explicitly captured (must be listed in `+ use<…>`), and thus currently it is not allowed to change what is captured of those kinds of generics.

Major: adding/removing function parameters {#fn-change-arity}

Changing the arity of a function is a breaking change.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub fn foo() {}

////////////////////////////////////////////////////////
// After
pub fn foo(x: i32) {}

////////////////////////////////////////////////////////
// Example usage that will break.
fn main() {
      updated_crate::foo(); // Error: this function takes 1
argument
}
```

Mitigating strategies:

- Introduce a new function with the new signature and possibly deprecate the old one.
- Introduce functions that take a struct argument, where the struct is built with the builder pattern. This allows new fields to be added to the struct in the future.

Possibly-breaking: introducing a new function type parameter {#fn-generic-new}

Usually, adding a non-defaulted type parameter is safe, but in some cases it can be a breaking change:

```
// Breaking change example

////////////////////////////////////////////////////////
// Before
pub fn foo<T>() {}

////////////////////////////////////////////////////////
```

```
// After
pub fn foo<T, U>() {}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::foo;

fn main() {
    foo::<u8>(); // Error: function takes 2 generic arguments
but 1 generic argument was supplied
}
```

However, such explicit calls are rare enough (and can usually be written in other ways) that this breakage is usually acceptable. One should take into account how likely it is that the function in question is being called with explicit type arguments.

Minor: generalizing a function to use generics (supporting original type) {#fn-generalize-compatible}

The type of a parameter to a function, or its return value, can be *generalized* to use generics, including by introducing a new type parameter, as long as it can be instantiated to the original type. For example, the following changes are allowed:

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
pub fn foo(x: u8) -> u8 {
    x
}
pub fn bar<T: Iterator<Item = u8>>(t: T) {}

////////////////////////////////////////////////////////
// After
```

```
use std::ops::Add;
pub fn foo<T: Add>(x: T) -> T {
    x
}
pub fn bar<T: IntoIterator<Item = u8>>(t: T) {}

//////////////////////////////////////////////////////////
// Example use of the library that will safely work.
use updated_crate::{bar, foo};

fn main() {
    foo(1);
    bar(vec![1, 2, 3].into_iter());
}
```

because all existing uses are instantiations of the new signature.

Perhaps somewhat surprisingly, generalization applies to trait objects as well, given that every trait implements itself:

```
// MINOR CHANGE

//////////////////////////////////////////////////////////
// Before
pub trait Trait {}
pub fn foo(t: &dyn Trait) {}

//////////////////////////////////////////////////////////
// After
pub trait Trait {}
pub fn foo<T: Trait + ?Sized>(t: &T) {}

//////////////////////////////////////////////////////////
// Example use of the library that will safely work.
use updated_crate::{foo, Trait};

struct Foo;
impl Trait for Foo {}
```

```
fn main() {
    let obj = Foo;
    foo(&obj);
}
```

(The use of `?Sized` is essential; otherwise you couldn't recover the original signature.)

Introducing generics in this way can potentially create type inference failures. These are usually rare, and may be acceptable breakage for some projects, as this can be fixed with additional type annotations.

```
// Breaking change example

///////////////////////////////////////////////////////
// Before
pub fn foo() -> i32 {
    0
}

///////////////////////////////////////////////////////
// After
pub fn foo<T: Default>() -> T {
    Default::default()
}

///////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::foo;

fn main() {
    let x = foo(); // Error: type annotations needed
}
```

Major: generalizing a function to use generics with type mismatch {#fn-generalize-mismatch}

It is a breaking change to change a function parameter or return type if the generic type constrains or changes the types previously allowed. For example, the following adds a generic constraint that may not be satisfied by existing code:

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub fn foo(x: Vec<u8>) {}

////////////////////////////////////////////////////////
// After
pub fn foo<T: Copy + IntoIterator<Item = u8>>(x: T) {}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::foo;

fn main() {
     foo(vec![1, 2, 3]); // Error: `Copy` is not implemented
for `Vec<u8>`
}
```

Minor: making an `unsafe` function safe {#fn-unsafe-safe}

A previously `unsafe` function can be made safe without breaking code.

Note however that it may cause the `unused_unsafe` lint to trigger as in the example below, which will cause local crates that have specified `#![deny(warnings)]` to stop compiling. Per introducing new lints, it is allowed for updates to introduce new warnings.

Going the other way (making a safe function `unsafe`) is a breaking change.

```
// MINOR CHANGE

///////////////////////////////////////////////////////////
// Before
pub unsafe fn foo() {}

///////////////////////////////////////////////////////////
// After
pub fn foo() {}

///////////////////////////////////////////////////////////
// Example use of the library that will trigger a lint.
use updated_crate::foo;

unsafe fn bar(f: unsafe fn()) {
    f()
}

fn main() {
    unsafe { foo() }; // The `unused_unsafe` lint will trigger
here
    unsafe { bar(foo) };
}
```

Making a previously `unsafe` associated function or method on structs / enums safe is also a minor change, while the same is not true for associated function on traits (see any change to trait item signatures).

Major: switching from `no_std` support to requiring `std` {#attr-no-std-to-std}

If your library specifically supports a `no_std` environment, it is a breaking change to make a new release that requires `std`.

```
// MAJOR CHANGE

///////////////////////////////////////////////////////////
```

```
// Before
#![no_std]
pub fn foo() {}

////////////////////////////////////////////////////////
// After
pub fn foo() {
    std::time::SystemTime::now();
}

////////////////////////////////////////////////////////
// Example usage that will break.
// This will fail to link for no_std targets because they
don't have a `std` crate.
#![no_std]
use updated_crate::foo;

fn example() {
    foo();
}
```

Mitigation strategies:

- A common idiom to avoid this is to include a `std` Cargo feature that optionally enables `std` support, and when the feature is off, the library can be used in a `no_std` environment.

Major: adding `non_exhaustive` to an existing enum, variant, or struct with no private fields {#attr-adding-non-exhaustive}

Making items `#[non_exhaustive]` changes how they may be used outside the crate where they are defined:

- Non-exhaustive structs and enum variants cannot be constructed using struct literal syntax, including functional update syntax.

- Pattern matching on non-exhaustive structs requires `..` and matching on enums does not count towards exhaustiveness.
- Casting enum variants to their discriminant with `as` is not allowed.

Structs with private fields cannot be constructed using struct literal syntax regardless of whether `#[non_exhaustive]` is used. Adding `#[non_exhaustive]` to such a struct is not a breaking change.

```
// MAJOR CHANGE

////////////////////////////////////////////////////////
// Before
pub struct Foo {
    pub bar: usize,
}

pub enum Bar {
    X,
    Y(usize),
    Z { a: usize },
}

pub enum Quux {
    Var,
}

////////////////////////////////////////////////////////
// After
#[non_exhaustive]
pub struct Foo {
    pub bar: usize,
}

pub enum Bar {
    #[non_exhaustive]
    X,
```

```
    #[non_exhaustive]
    Y(usize),

    #[non_exhaustive]
    Z { a: usize },
}

#[non_exhaustive]
pub enum Quux {
    Var,
}

////////////////////////////////////////////////////////
// Example usage that will break.
use updated_crate::{Bar, Foo, Quux};

fn main() {
     let foo = Foo { bar: 0 }; // Error: cannot create non-exhaustive struct using struct expression

    let bar_x = Bar::X; // Error: unit variant `X` is private
     let bar_y = Bar::Y(0); // Error: tuple variant `Y` is private
    let bar_z = Bar::Z { a: 0 }; // Error: cannot create non-exhaustive variant using struct expression

    let q = Quux::Var;
    match q {
        Quux::Var => 0,
        // Error: non-exhaustive patterns: `_` not covered
    };
}
```

Mitigation strategies:

- Mark structs, enums, and enum variants as `#[non_exhaustive]` when first introducing them, rather than adding `#[non_exhaustive]` later on.

Tooling and environment compatibility

Possibly-breaking: changing the minimum version of Rust required {#env-new-rust}

Introducing the use of new features in a new release of Rust can break projects that are using older versions of Rust. This also includes using new features in a new release of Cargo, and requiring the use of a nightly-only feature in a crate that previously worked on stable.

It is generally recommended to treat this as a minor change, rather than as a major change, for various reasons. It is usually relatively easy to update to a newer version of Rust. Rust also has a rapid 6-week release cycle, and some projects will provide compatibility within a window of releases (such as the current stable release plus N previous releases). Just keep in mind that some large projects may not be able to update their Rust toolchain rapidly.

Mitigation strategies:

- Use Cargo features to make the new features opt-in.
- Provide a large window of support for older releases.
- Copy the source of new standard library items if possible so that you can continue to use an older version but take advantage of the new feature.
- Provide a separate branch of older minor releases that can receive backports of important bugfixes.
- Keep an eye out for the `[cfg(version(..))]` and `#[cfg(accessible(..))]` features which provide an opt-in mechanism for new features. These are currently unstable and only available in the nightly channel.

Possibly-breaking: changing the platform and environment requirements {#env-change-requirements}

There is a very wide range of assumptions a library makes about the environment that it runs in, such as the host platform, operating system version, available services, filesystem support, etc. It can be a breaking change if you make a new release that restricts what was previously supported, for example requiring a newer version of an operating system. These changes can be difficult to track, since you may not always know if a change breaks in an environment that is not automatically tested.

Some projects may deem this acceptable breakage, particularly if the breakage is unlikely for most users, or the project doesn't have the resources to support all environments. Another notable situation is when a vendor discontinues support for some hardware or OS, the project may deem it reasonable to also discontinue support.

Mitigation strategies:

- Document the platforms and environments you specifically support.
- Test your code on a wide range of environments in CI.

Minor: introducing new lints {#new-lints}

Some changes to a library may cause new lints to be triggered in users of that library. This should generally be considered a compatible change.

```
// MINOR CHANGE

////////////////////////////////////////////////////////
// Before
pub fn foo() {}

////////////////////////////////////////////////////////
// After
#[deprecated]
pub fn foo() {}

////////////////////////////////////////////////////////
// Example use of the library that will safely work.

fn main() {
```

```
        updated_crate::foo();  //  Warning:  use  of  deprecated
function
}
```

Beware that it may be possible for this to technically cause a project to fail if they have explicitly denied the warning, and the updated crate is a direct dependency. Denying warnings should be done with care and the understanding that new lints may be introduced over time. However, library authors should be cautious about introducing new warnings and may want to consider the potential impact on their users.

The following lints are examples of those that may be introduced when updating a dependency:

- `deprecated` --- Introduced when a dependency adds the `#[deprecated]` attribute to an item you are using.
- `unused_must_use` --- Introduced when a dependency adds the `#[must_use]` attribute to an item where you are not consuming the result.
- `unused_unsafe` --- Introduced when a dependency *removes* the `unsafe` qualifier from a function, and that is the only unsafe function called in an unsafe block.

Additionally, updating `rustc` to a new version may introduce new lints.

Transitive dependencies which introduce new lints should not usually cause a failure because Cargo uses `--cap-lints` to suppress all lints in dependencies.

Mitigating strategies:

- If you build with warnings denied, understand you may need to deal with resolving new warnings whenever you update your dependencies. If using RUSTFLAGS to pass `-Dwarnings`, also add the `-A` flag to allow lints that are likely to cause issues, such as `-Adeprecated`.
- Introduce deprecations behind a feature. For example `#[cfg_attr(feature = "deprecated", deprecated="use bar instead")]`. Then, when you plan to remove an item in a future

SemVer breaking change, you can communicate with your users that they should enable the `deprecated` feature *before* updating to remove the use of the deprecated items. This allows users to choose when to respond to deprecations without needing to immediately respond to them. A downside is that it can be difficult to communicate to users that they need to take these manual steps to prepare for a major update.

Cargo

Minor: adding a new Cargo feature {#cargo-feature-add}

It is usually safe to add new Cargo features. If the feature introduces new changes that cause a breaking change, this can cause difficulties for projects that have stricter backwards-compatibility needs. In that scenario, avoid adding the feature to the "default" list, and possibly document the consequences of enabling the feature.

```
# MINOR CHANGE

##############################################################
# Before
[features]
# ..empty

##############################################################
# After
[features]
std = []
```

Major: removing a Cargo feature {#cargo-feature-remove}

It is usually a breaking change to remove Cargo features. This will cause an error for any project that enabled the feature.

```
# MAJOR CHANGE

##############################################################
# Before
[features]
```

```
logging = []

################################################################
# After
[dependencies]
# ..logging removed
```

Mitigation strategies:

- Clearly document your features. If there is an internal or experimental feature, mark it as such, so that users know the status of the feature.
- Leave the old feature in `Cargo.toml`, but otherwise remove its functionality. Document that the feature is deprecated, and remove it in a future major SemVer release.

Major: removing a feature from a feature list if that changes functionality or public items {#cargo-feature-remove-another}

If removing a feature from another feature, this can break existing users if they are expecting that functionality to be available through that feature.

```
# Breaking change example

################################################################
# Before
[features]
default = ["std"]
std = []

################################################################
# After
[features]
default = []  # This may cause packages to fail if they are
expecting std to be enabled.
std = []
```

Possibly-breaking: removing an optional dependency {#cargo-remove-opt-dep}

Removing an optional dependency can break a project using your library because another project may be enabling that dependency via Cargo features.

When there is an optional dependency, cargo implicitly defines a feature of the same name to provide a mechanism to enable the dependency and to check when it is enabled. This problem can be avoided by using the `dep:` syntax in the `[features]` table, which disables this implicit feature. Using `dep:` makes it possible to hide the existence of optional dependencies under more semantically-relevant names which can be more safely modified.

```
# Breaking change example

##############################################################
# Before
[dependencies]
curl = { version = "0.4.31", optional = true }

##############################################################
# After
[dependencies]
# ..curl removed

# MINOR CHANGE
#
#  This  example  shows  how  to  avoid  breaking  changes  with
optional dependencies.

##############################################################
# Before
[dependencies]
curl = { version = "0.4.31", optional = true }

[features]
networking = ["dep:curl"]
```

```
##########################################################
# After
[dependencies]
# Here, one optional dependency was replaced with another.
hyper = { version = "0.14.27", optional = true }

[features]
networking = ["dep:hyper"]
```

Mitigation strategies:

- Use the `dep:` syntax in the `[features]` table to avoid exposing optional dependencies in the first place. See optional dependencies for more information.
- Clearly document your features. If the optional dependency is not included in the documented list of features, then you may decide to consider it safe to change undocumented entries.
- Leave the optional dependency, and just don't use it within your library.
- Replace the optional dependency with a Cargo feature that does nothing, and document that it is deprecated.
- Use high-level features which enable optional dependencies, and document those as the preferred way to enable the extended functionality. For example, if your library has optional support for something like "networking", create a generic feature name "networking" that enables the optional dependencies necessary to implement "networking". Then document the "networking" feature.

Minor: changing dependency features {#cargo-change-dep-feature}

It is usually safe to change the features on a dependency, as long as the feature does not introduce a breaking change.

```
# MINOR CHANGE

##########################################################
# Before
```

```
[dependencies]
rand = { version = "0.7.3", features = ["small_rng"] }


##############################################################
# After
[dependencies]
rand = "0.7.3"
```

Minor: adding dependencies {#cargo-dep-add}

It is usually safe to add new dependencies, as long as the new dependency does not introduce new requirements that result in a breaking change. For example, adding a new dependency that requires nightly in a project that previously worked on stable is a major change.

```
# MINOR CHANGE

##############################################################
# Before
[dependencies]
# ..empty

##############################################################
# After
[dependencies]
log = "0.4.11"
```

Application compatibility

Cargo projects may also include executable binaries which have their own interfaces (such as a CLI interface, OS-level interaction, etc.). Since these are part of the Cargo package, they often use and share the same version as the package. You will need to decide if and how you want to employ a SemVer contract with your users in the changes you make to your application. The potential breaking and compatible changes to an application are too numerous to list, so you are encouraged to use the spirit of the SemVer spec to guide your decisions on how to apply versioning to your application, or at least document what your commitments are.

Future incompat report

Cargo checks for future-incompatible warnings in all dependencies. These are warnings for changes that may become hard errors in the future, causing the dependency to stop building in a future version of rustc. If any warnings are found, a small notice is displayed indicating that the warnings were found, and provides instructions on how to display a full report.

For example, you may see something like this at the end of a build:

```
warning: the following packages contain code that will be rejected by a future
         version of Rust: rental v0.5.5
note: to see what the problems were, use the option `--future-incompat-report`,
      or run `cargo report future-incompatibilities --id 1`
```

A full report can be displayed with the `cargo report future-incompatibilities --id ID` command, or by running the build again with the `--future-incompat-report` flag. The developer should then update their dependencies to a version where the issue is fixed, or work with the developers of the dependencies to help resolve the issue.

Configuration

This feature can be configured through a `[future-incompat-report]` section in `.cargo/config.toml`. Currently, the supported options are:

```
[future-incompat-report]
frequency = "always"
```

The supported values for the frequency are `"always"` and `"never"`, which control whether or not a message is printed out at the end of `cargo build` / `cargo check`.

Reporting build timings

The `--timings` option gives some information about how long each compilation takes, and tracks concurrency information over time.

```
cargo build --timings
```

This writes an HTML report in `target/cargo-timings/cargo-timing.html`. This also writes a copy of the report to the same directory with a timestamp in the filename, if you want to look at older runs.

Reading the graphs

There are two tables and two graphs in the output.

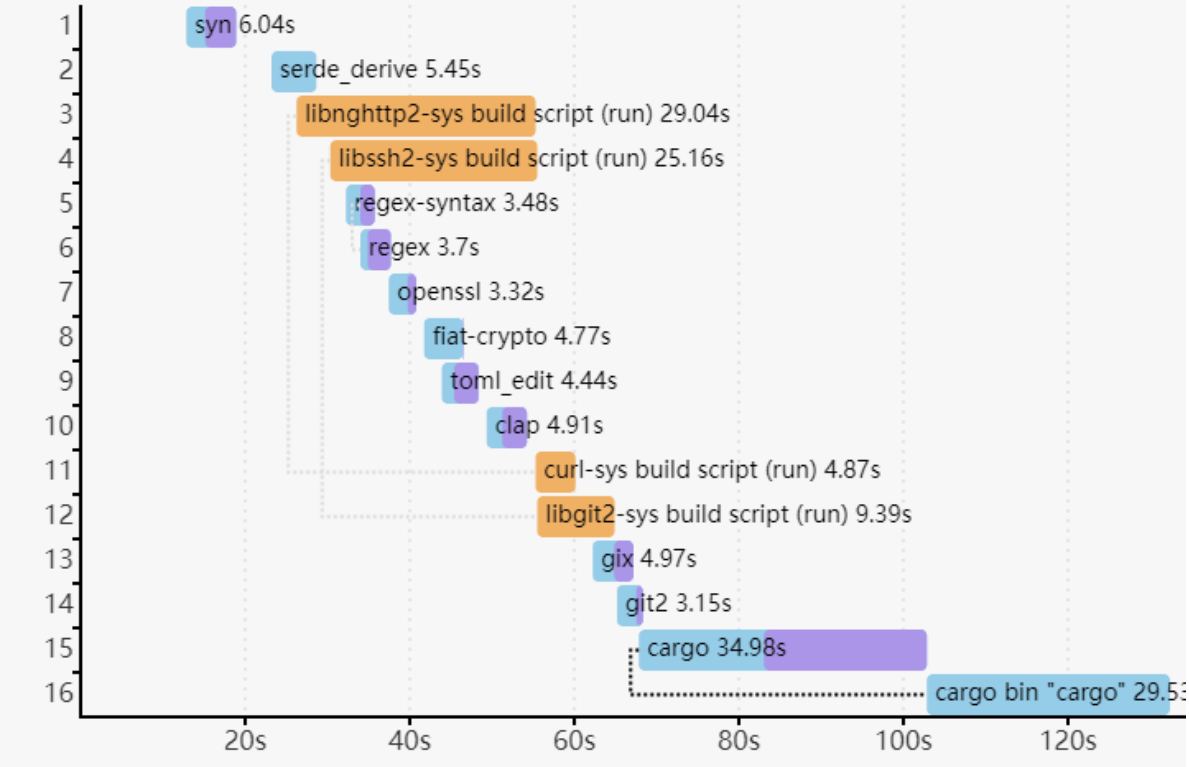
The first table displays the build information of the project, including the number of units built, the maximum number of concurrency, build time, and the version information of the currently used compiler.

Targets:	cargo 0.70.0 (lib, bin "cargo")
Profile:	dev
Fresh units:	0
Dirty units:	302
Total units:	302
Max concurrency:	6 (jobs=4 ncpu=4)
Build start:	2023-03-03T10:48:37Z
Total time:	132.5s (2m 12.5s)
rustc:	rustc 1.69.0-nightly (ef982929c 2023-01-27) Host: x86_64-unknown-linux-gnu Target: x86_64-unknown-linux-gnu

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.

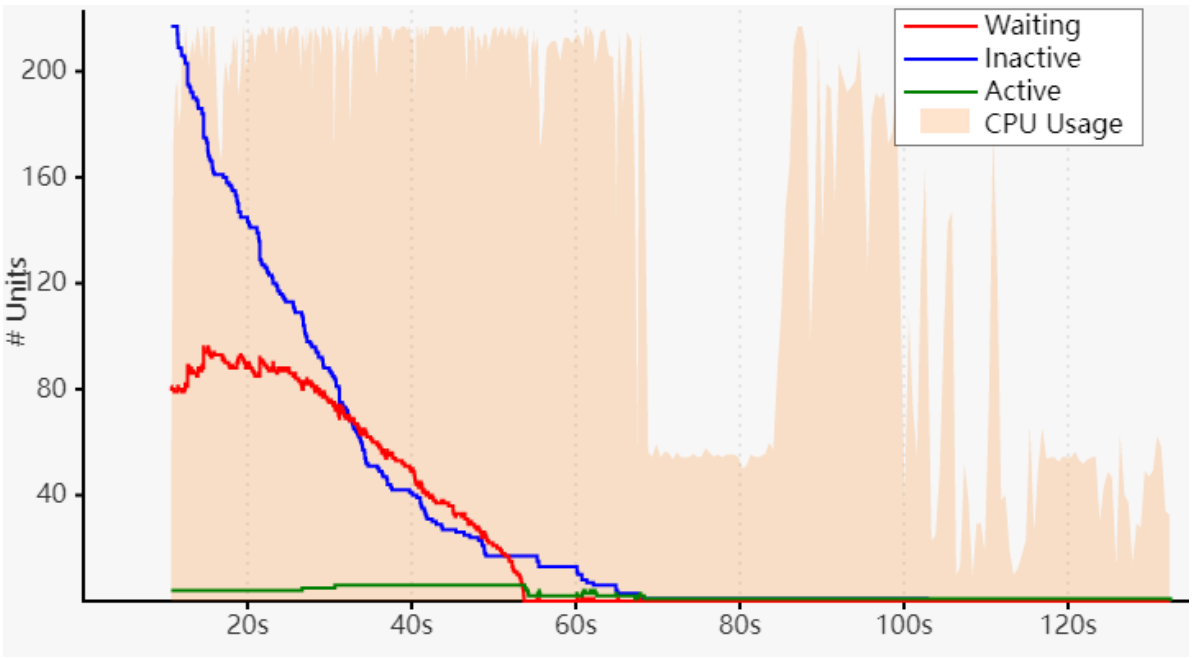
The "codegen" times are highlighted in a lavender color. In some cases, build pipelining allows units to start when their dependencies are performing code generation. This information is not always displayed (for example, binary units do not show when code generation starts).

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 background indicates CPU usage. 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.

The last table lists the total time and "codegen" time spent on each unit, as well as the features that were enabled during each unit's compilation.

Lints

Note: Cargo's linting system is unstable and can only be used on nightly toolchains

Warn-by-default

These lints are all set to the 'warn' level by default.

- `unknown_lints`

`unknown_lints`

Set to `warn` by default

What it does

Checks for unknown lints in the `[lints.cargo]` table

Why it is bad

- The lint name could be misspelled, leading to confusion as to why it is not working as expected
- The unknown lint could end up causing an error if `cargo` decides to make a lint with the same name in the future

Example

```
[lints.cargo]
this-lint-does-not-exist = "warn"
```

Unstable Features

Experimental Cargo features are only available on the nightly channel. You are encouraged to experiment with these features to see if they meet your needs, and if there are any issues or problems. Check the linked tracking issues listed below for more information on the feature, and click the GitHub subscribe button if you want future updates.

After some period of time, if the feature does not have any major concerns, it can be stabilized, which will make it available on stable once the current nightly release reaches the stable channel (anywhere from 6 to 12 weeks).

There are three different ways that unstable features can be enabled based on how the feature works:

- New syntax in `Cargo.toml` requires a `cargo-features` key at the top of `Cargo.toml`, before any tables. For example:

```
# This specifies which new Cargo.toml features are
enabled.
cargo-features = ["test-dummy-unstable"]

[package]
name = "my-package"
version = "0.1.0"
im-a-teapot = true  # This is a new option enabled by
test-dummy-unstable.
```

- New command-line flags, options, and subcommands require the `-Z unstable-options` CLI option to also be included. For example, the new `--artifact-dir` option is only available on nightly:

  `cargo +nightly build --artifact-dir=out -Z unstable-options`

- `-Z` command-line flags are used to enable new functionality that may not have an interface, or the interface has not yet been designed,

or for more complex features that affect multiple parts of Cargo. For example, the mtime-on-use feature can be enabled with:

```
cargo +nightly build -Z mtime-on-use
```

Run `cargo -Z help` to see a list of flags available.

Anything which can be configured with a `-Z` flag can also be set in the cargo config file (`.cargo/config.toml`) in the `unstable` table. For example:

```
[unstable]
mtime-on-use = true
build-std = ["core", "alloc"]
```

Each new feature described below should explain how to use it.

For the latest nightly, see the nightly version of this page.

List of unstable features

- Unstable-specific features
    - -Z allow-features --- Provides a way to restrict which unstable features are used.
- Build scripts and linking
    - Metabuild --- Provides declarative build scripts.
    - Multiple Build Scripts --- Allows use of multiple build scripts.
- Resolver and features
    - no-index-update --- Prevents cargo from updating the index cache.
    - avoid-dev-deps --- Prevents the resolver from including dev-dependencies during resolution.
    - minimal-versions --- Forces the resolver to use the lowest compatible version instead of the highest.
    - direct-minimal-versions — Forces the resolver to use the lowest compatible version instead of the highest.
    - public-dependency --- Allows dependencies to be classified as either public or private.
    - msrv-policy --- MSRV-aware resolver and version selection
    - precise-pre-release --- Allows pre-release versions to be selected with `update --precise`
    - sbom --- Generates SBOM pre-cursor files for compiled artifacts
    - update-breaking --- Allows upgrading to breaking versions with `update --breaking`
    - feature-unification --- Enable new feature unification modes in workspaces
- Output behavior
    - artifact-dir --- Adds a directory where artifacts are copied to.
    - build-dir --- Adds a directory where intermediate build artifacts are stored.

    - Different binary name --- Assign a name to the built binary that is separate from the crate name.
    - root-dir --- Controls the root directory relative to which paths are printed
- Compile behavior
    - mtime-on-use --- Updates the last-modified timestamp on every dependency every time it is used, to provide a mechanism to delete unused artifacts.
    - build-std --- Builds the standard library instead of using pre-built binaries.
    - build-std-features --- Sets features to use with the standard library.
    - binary-dep-depinfo --- Causes the dep-info file to track binary dependencies.
    - checksum-freshness --- When passed, the decision as to whether a crate needs to be rebuilt is made using file checksums instead of the file mtime.
    - panic-abort-tests --- Allows running tests with the "abort" panic strategy.
    - host-config --- Allows setting `[target]`-like configuration settings for host build targets.
    - no-embed-metadata --- Passes `-Zembed-metadata=no` to the compiler, which avoid embedding metadata into rlib and dylib artifacts, to save disk space.
    - target-applies-to-host --- Alters whether certain flags will be passed to host build targets.
    - gc --- Global cache garbage collection.
    - open-namespaces --- Allow multiple packages to participate in the same API namespace
- rustdoc
    - rustdoc-map --- Provides mappings for documentation to link to external sites like docs.rs.
    - scrape-examples --- Shows examples within documentation.
    - output-format --- Allows documentation to also be emitted in the experimental JSON format.

    - rustdoc-depinfo --- Use dep-info files in rustdoc rebuild detection.
- `Cargo.toml` extensions
    - Profile `rustflags` option --- Passed directly to rustc.
    - Profile `hint-mostly-unused` option --- Hint that a dependency is mostly unused, to optimize compilation time.
    - codegen-backend --- Select the codegen backend used by rustc.
    - per-package-target --- Sets the `--target` to use for each individual package.
    - artifact dependencies --- Allow build artifacts to be included into other build artifacts and build them for different targets.
    - Profile `trim-paths` option --- Control the sanitization of file paths in build outputs.
    - `[lints.cargo]` --- Allows configuring lints for Cargo.
    - path bases --- Named base directories for path dependencies.
    - `unstable-editions` --- Allows use of editions that are not yet stable.
- Information and metadata
    - Build-plan --- Emits JSON information on which commands will be run.
    - unit-graph --- Emits JSON for Cargo's internal graph structure.
    - `cargo rustc --print` --- Calls rustc with `--print` to display information from rustc.
- Configuration
    - config-include --- Adds the ability for config files to include other files.
    - `cargo config` --- Adds a new subcommand for viewing config files.
- Registries
    - publish-timeout --- Controls the timeout between uploading the crate and being available in the index

    - asymmetric-token --- Adds support for authentication tokens using asymmetric cryptography (`cargo:paseto` provider).
- Other
    - gitoxide --- Use `gitoxide` instead of `git2` for a set of operations.
    - script --- Enable support for single-file `.rs` packages.
    - lockfile-path --- Allows to specify a path to lockfile other than the default path `<workspace_root>/Cargo.lock`.
    - package-workspace --- Allows for packaging and publishing multiple crates in a workspace.
    - native-completions --- Move cargo shell completions to native completions.
    - warnings --- controls warning behavior; options for allowing or denying warnings.
    - Package message format --- Message format for `cargo package`.
    - `fix-edition` --- A permanently unstable edition migration helper.

allow-features

This permanently-unstable flag makes it so that only a listed set of unstable features can be used. Specifically, if you pass `-Zallow-features=foo,bar`, you'll continue to be able to pass `-Zfoo` and `-Zbar` to `cargo`, but you will be unable to pass `-Zbaz`. You can pass an empty string (`-Zallow-features=`) to disallow all unstable features.

`-Zallow-features` also restricts which unstable features can be passed to the `cargo-features` entry in `Cargo.toml`. If, for example, you want to allow

```
cargo-features = ["test-dummy-unstable"]
```

where `test-dummy-unstable` is unstable, that features would also be disallowed by `-Zallow-features=`, and allowed with `-Zallow-features=test-dummy-unstable`.

The list of features passed to cargo's `-Zallow-features` is also passed to any Rust tools that cargo ends up calling (like `rustc` or `rustdoc`). Thus, if you run `cargo -Zallow-features=`, no unstable Cargo *or* Rust features can be used.

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.toml` 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
- Tracking 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`.

direct-minimal-versions

- Original Issue: #4100
- Tracking Issue: #5657

When a `Cargo.lock` file is generated, the `-Z direct-minimal-versions` flag will resolve the dependencies to the minimum SemVer version that will satisfy the requirements (instead of the greatest version) for direct dependencies only.

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`.

Indirect dependencies are resolved as normal so as not to be blocked on their minimal version validation.

artifact-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 `--artifact-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 --artifact-dir=out -Z unstable-options
```

This can also be specified in `.cargo/config.toml` files.

```
[build]
artifact-dir = "out"
```

build-dir

- Original Issue: #14125
- Tracking Issue: #14125

The directory where intermediate build artifacts will be stored. Intermediate artifacts are produced by Rustc/Cargo during the build process.

```
[build]
build-dir = "out"
```

`build.build-dir`

- Type: string (path)
- Default: Defaults to the value of `build.target-dir`
- Environment: `CARGO_BUILD_BUILD_DIR`

The path to where internal files used as part of the build are placed.

This option supports path templating.

Available template variables:

- `{workspace-root}` resolves to root of the current workspace.
- `{cargo-cache-home}` resolves to `CARGO_HOME`
- `{workspace-path-hash}` resolves to a hash of the manifest path

root-dir

- Original Issue: #9887
- Tracking Issue: None (not currently slated for stabilization)

The `-Zroot-dir` flag sets the root directory relative to which paths are printed. This affects both diagnostics and paths emitted by the `file!()` macro.

Build-plan

- Tracking Issue: #5579

The build-plan feature is deprecated and may be removed in a future version. See https://github.com/rust-lang/cargo/issues/7614.

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.

Multiple Build Scripts

- Tracking Issue: #14903
- Original Pull Request: #15630

Multiple Build Scripts feature allows you to have multiple build scripts in your package.

Include `cargo-features` at the top of `Cargo.toml` and add `multiple-build-scripts` to enable feature. Add the paths of the build scripts as an array in `package.build`. For example:

```
cargo-features = ["multiple-build-scripts"]

[package]
name = "mypackage"
version = "0.0.1"
build = ["foo.rs", "bar.rs"]
```

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.

To enable this feature, you can either use `-Zpublic-dependency`

```
cargo +nightly run -Zpublic-dependency
```

or `[unstable]` table, for example,

```
# .cargo/config.toml
[unstable]
public-dependency = true
```

`public-dependency` could also be enabled in `cargo-features`, **though this is deprecated and will be removed soon**.

```
cargo-features = ["public-dependency"]

[dependencies]
my_dep = { version = "1.2.3", public = true }
private_dep = "2.0.0" # Will be 'private' by default
```

Documentation updates:

- For workspace's "The `dependencies` table" section, include `public` as an unsupported field for `workspace.dependencies`

msrv-policy

- RFC: MSRV-aware Resolver
- #9930 (MSRV-aware resolver)

Catch-all unstable feature for MSRV-aware cargo features under RFC 2495.

MSRV-aware cargo add

This was stabilized in 1.79 in #13608.

MSRV-aware resolver

This was stabilized in 1.84 in #14639.

Convert `incompatible_toolchain` error into a lint

Unimplemented

`--update-rust-version` flag for `cargo add`, `cargo update`

Unimplemented

`package.rust-version = "toolchain"`

Unimplemented

Update `cargo new` template to set `package.rust-version = "toolchain"`

Unimplemented

precise-pre-release

- Tracking Issue: #13290
- RFC: #3493

The `precise-pre-release` feature allows pre-release versions to be selected with `update --precise` even when a pre-release is not specified by a projects `Cargo.toml`.

Take for example this `Cargo.toml`.

```
[dependencies]
my-dependency = "0.1.1"
```

It's possible to update `my-dependency` to a pre-release with `update -Zunstable-options my-dependency --precise 0.1.2-pre.0`. This is because `0.1.2-pre.0` is considered compatible with `0.1.1`. It would not be possible to upgrade to `0.2.0-pre.0` from `0.1.1` in the same way.

sbom

- Tracking Issue: #13709
- RFC: #3553

The `sbom` build config allows to generate so-called SBOM pre-cursor files alongside each compiled artifact. A Software Bill Of Material (SBOM) tool can incorporate these generated files to collect important information from the cargo build process that are difficult or impossible to obtain in another way.

To enable this feature either set the `sbom` field in the `.cargo/config.toml`

```
[unstable]
sbom = true

[build]
sbom = true
```

or set the `CARGO_BUILD_SBOM` environment variable to `true`. The functionality is available behind the flag `-Z sbom`.

The generated output files are in JSON format and follow the naming scheme `<artifact>.cargo-sbom.json`. The JSON file contains information about dependencies, target, features and the used `rustc` compiler.

SBOM pre-cursor files are generated for all executable and linkable outputs that are uplifted into the target or artifact directories.

Environment variables Cargo sets for crates

- `CARGO_SBOM_PATH` -- a list of generated SBOM precursor files, separated by the platform PATH separator. The list can be split with `std::env::split_paths`.

SBOM pre-cursor schema

```
{
  // Schema version.
  "version": 1,
  // Index into the crates array for the root crate.
  "root": 0,
  // Array of all crates. There may be duplicates of the same crate if that
    // crate is compiled differently (different opt-level, features, etc).
  "crates": [
    {
      // Package ID specification
      "id": "path+file:///sample-package#0.1.0",
       // List of target kinds: bin, lib, rlib, dylib, cdylib, staticlib, proc-macro, example, test, bench, custom-build
      "kind": ["bin"],
      // Enabled feature flags.
      "features": [],
      // Dependencies for this crate.
      "dependencies": [
        {
          // Index in to the crates array.
          "index": 1,
          // Dependency kind:
              // Normal: A dependency linked to the artifact produced by this crate.
            // Build: A compile-time dependency used to build this crate (build-script or proc-macro).
          "kind": "normal"
        },
        {
            // A crate can depend on another crate with both normal and build edges.
          "index": 1,
          "kind": "build"
```

```
        }
      ]
    },
    {
       "id": "registry+https://github.com/rust-lang/crates.io-index#zerocopy@0.8.16",
      "kind": ["bin"],
      "features": [],
      "dependencies": []
    }
  ],
  // Information about rustc used to perform the compilation.
  "rustc": {
    // Compiler version
    "version": "1.86.0-nightly",
    // Compiler wrapper
    "wrapper": null,
    // Compiler workspace wrapper
    "workspace_wrapper": null,
    // Commit hash for rustc
    "commit_hash": "bef3c3b01f690de16738b1c9f36470fbfc6ac623",
    // Host target triple
    "host": "x86_64-pc-windows-msvc",
    // Verbose version string: `rustc -vV`
     "verbose_version": "rustc 1.86.0-nightly (bef3c3b01 2025-02-04)\nbinary:                              rustc\ncommit-hash: bef3c3b01f690de16738b1c9f36470fbfc6ac623\ncommit-date:   2025-02-04\nhost:     x86_64-pc-windows-msvc\nrelease:     1.86.0-nightly\nLLVM version: 19.1.7\n"
  }
}
```

update-breaking

- Tracking Issue: #12425

Allow upgrading dependencies version requirements in `Cargo.toml` across SemVer incompatible versions using with the `--breaking` flag.

This only applies to dependencies when

- The package is a dependency of a workspace member
- The dependency is not renamed
- A SemVer-incompatible version is available
- The "SemVer operator" is used (`^` which is the default)

Users may further restrict which packages get upgraded by specifying them on the command line.

Example:

```
$ cargo +nightly -Zunstable-options update --breaking
$ cargo +nightly -Zunstable-options update --breaking clap
```

This is meant to fill a similar role as cargo-upgrade

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
```

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 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!

build-std-features

- Tracking Repository: https://github.com/rust-lang/wg-cargo-std-aware

This flag is a sibling to the `-Zbuild-std` feature flag. This will configure the features enabled for the standard library itself when building the standard library. The default enabled features, at this time, are `backtrace` and `panic-unwind`. This flag expects a comma-separated list and, if provided, will override the default list of features enabled.

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.

checksum-freshness

- Tracking issue: #14136

The `-Z checksum-freshness` flag will replace the use of file mtimes in cargo's fingerprints with a file checksum value. This is most useful on systems with a poor mtime implementation, or in CI/CD. The checksum algorithm can change without notice between cargo versions. Fingerprints are used by cargo to determine when a crate needs to be rebuilt.

For the time being files ingested by build script will continue to use mtimes, even when `checksum-freshness` is enabled. This is not intended as a long term solution.

panic-abort-tests

- Tracking Issue: #67650
- Original Pull Request: #7460

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!

config-include

- Tracking Issue: #7723

This feature requires the `-Zconfig-include` command-line option.

The `include` key in a config file can be used to load another config file. It takes a string for a path to another file relative to the config file, or an array of config file paths. Only path ending with `.toml` is accepted.

```
# a path ending with `.toml`
include = "path/to/mordor.toml"

# or an array of paths
include = ["frodo.toml", "samwise.toml"]
```

Unlike other config values, the merge behavior of the `include` key is different. When a config file contains an `include` key:

1. The config values are first loaded from the `include` path.
    - If the value of the `include` key is an array of paths, the config values are loaded and merged from left to right for each path.
    - Recurse this step if the config values from the `include` path also contain an `include` key.
2. Then, the config file's own values are merged on top of the config from the `include` path.

target-applies-to-host

- Original Pull Request: #9322
- Tracking Issue: #9453

Historically, Cargo's behavior for whether the `linker` and `rustflags` configuration options from environment variables and `[target]` are respected for build scripts, plugins, and other artifacts that are *always* built for the host platform has been somewhat inconsistent. When `--target` is *not* passed, Cargo respects the same `linker` and `rustflags` for build scripts as for all other compile artifacts. When `--target` *is* passed, however, Cargo respects `linker` from `[target.<host triple>]`, and does not pick up any `rustflags` configuration. This dual behavior is confusing, but also makes it difficult to correctly configure builds where the host triple and the target triple happen to be the same, but artifacts intended to run on the build host should still be configured differently.

`-Ztarget-applies-to-host` enables the top-level `target-applies-to-host` setting in Cargo configuration files which allows users to opt into different (and more consistent) behavior for these properties. When `target-applies-to-host` is unset, or set to `true`, in the configuration file, the existing Cargo behavior is preserved (though see `-Zhost-config`, which changes that default). When it is set to `false`, no options from `[target.<host triple>]`, `RUSTFLAGS`, or `[build]` are respected for host artifacts regardless of whether `--target` is passed to Cargo. To customize artifacts intended to be run on the host, use `[host]` (`host-config`).

In the future, `target-applies-to-host` may end up defaulting to `false` to provide more sane and consistent default behavior.

```
# config.toml
target-applies-to-host = false
```

```
cargo +nightly -Ztarget-applies-to-host build --target x86_64-unknown-linux-gnu
```

host-config

- Original Pull Request: #9322
- Tracking Issue: #9452

The `host` key in a config file can be used to pass flags to host build targets such as build scripts that must run on the host system instead of the target system when cross compiling. It supports both generic and host arch specific tables. Matching host arch tables take precedence over generic host tables.

It requires the `-Zhost-config` and `-Ztarget-applies-to-host` command-line options to be set, and that `target-applies-to-host = false` is set in the Cargo configuration file.

```
# config.toml
[host]
linker = "/path/to/host/linker"
[host.x86_64-unknown-linux-gnu]
linker = "/path/to/host/arch/linker"
rustflags = ["-Clink-arg=--verbose"]
[target.x86_64-unknown-linux-gnu]
linker = "/path/to/target/linker"
```

The generic `host` table above will be entirely ignored when building on an `x86_64-unknown-linux-gnu` host as the `host.x86_64-unknown-linux-gnu` table takes precedence.

Setting `-Zhost-config` changes the default for `target-applies-to-host` to `false` from `true`.

```
cargo +nightly -Ztarget-applies-to-host -Zhost-config build --target x86_64-unknown-linux-gnu
```

unit-graph

- Tracking Issue: #8002

The `--unit-graph` flag can be passed to any build command (`build`, `check`, `run`, `test`, `bench`, `doc`, etc.) to emit a JSON object to stdout which represents Cargo's internal unit graph. Nothing is actually built, and the command returns immediately after printing. Each "unit" corresponds to an execution of the compiler. These objects also include which unit each unit depends on.

```
cargo +nightly build --unit-graph -Z unstable-options
```

This structure provides a more complete view of the dependency relationship as Cargo sees it. In particular, the "features" field supports the new feature resolver where a dependency can be built multiple times with different features. `cargo metadata` fundamentally cannot represent the relationship of features between different dependency kinds, and features now depend on which command is run and which packages and targets are selected. Additionally it can provide details about intra-package dependencies like build scripts or tests.

The following is a description of the JSON structure:

```
{
  /* Version of the JSON output structure. If any backwards
incompatible
     changes are made, this value will be increased.
  */
  "version": 1,
  /* Array of all build units. */
  "units": [
    {
      /* An opaque string which indicates the package.
           Information about the package can be obtained from
`cargo metadata`.
      */
        "pkg_id": "my-package 0.1.0 (path+file:///path/to/my-
```

```
package)",
          /* The Cargo target. See the `cargo metadata` documentation for more
         information about these fields.
              https://doc.rust-lang.org/cargo/commands/cargo-metadata.html
      */
      "target": {
        "kind": ["lib"],
        "crate_types": ["lib"],
        "name": "my_package",
        "src_path": "/path/to/my-package/src/lib.rs",
        "edition": "2018",
        "test": true,
        "doctest": true
      },
      /* The profile settings for this unit.
         These values may not match the profile defined in the manifest.
         Units can use modified profile settings. For example, the "panic"
          setting can be overridden for tests to force it to "unwind".
      */
      "profile": {
         /* The profile name these settings are derived from. */
        "name": "dev",
        /* The optimization level as a string. */
        "opt_level": "0",
        /* The LTO setting as a string. */
        "lto": "false",
        /* The codegen units as an integer.
           `null` if it should use the compiler's default.
        */
        "codegen_units": null,
```

```
        /* The debug information level as an integer.
           `null` if it should use the compiler's default (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 rpath is enabled. */
        "rpath": false,
        /* Whether or not incremental is enabled. */
        "incremental": true,
        /* The panic strategy, "unwind" or "abort". */
        "panic": "unwind"
      },
      /* Which platform this target is being built for.
         A value of `null` indicates it is for the host.
          Otherwise it is a string of the target triple (such
as
         "x86_64-unknown-linux-gnu").
      */
      "platform": null,
      /* The "mode" for this unit. Valid values:

         * "test" --- Build using `rustc` as a test.
         * "build" --- Build using `rustc`.
         * "check" --- Build using `rustc` in "check" mode.
         * "doc" --- Build using `rustdoc`.
         * "doctest" --- Test using `rustdoc`.
          * "run-custom-build" --- Represents the execution of
a build script.
      */
      "mode": "build",
      /* Array of features enabled on this unit as strings. */
      "features": ["somefeat"],
      /* Whether or not this is a standard-library unit,
```

```
         part of the unstable build-std feature.
         If not set, treat as `false`.
      */
      "is_std": false,
      /* Array of dependencies of this unit. */
      "dependencies": [
        {
          /* Index in the "units" array for the dependency. */
          "index": 1,
           /* The name that this dependency will be referred
as. */
          "extern_crate_name": "unicode_xid",
          /* Whether or not this dependency is "public",
             part of the unstable public-dependency feature.
              If not set, the public-dependency feature is not
enabled.
          */
          "public": false,
           /* Whether or not this dependency is injected into
the prelude,
             currently used by the build-std feature.
             If not set, treat as `false`.
          */
          "noprelude": false
        }
      ]
    },
    // ...
  ],
   /* Array of indices in the "units" array that are the
"roots" of the
     dependency graph.
  */
  "roots": [0],
}
```

Profile `rustflags` option

- Original Issue: rust-lang/cargo#7878
- Tracking Issue: rust-lang/cargo#10271

This feature provides a new option in the `[profile]` section to specify flags that are passed directly to rustc. This can be enabled like so:

```
cargo-features = ["profile-rustflags"]

[package]
# ...

[profile.release]
rustflags = [ "-C", "..." ]
```

To set this in a profile in Cargo configuration, you need to use either `-Z profile-rustflags` or `[unstable]` table to enable it. For example,

```
# .cargo/config.toml
[unstable]
profile-rustflags = true

[profile.release]
rustflags = [ "-C", "..." ]
```

Profile `hint-mostly-unused` option

- Tracking Issue: #15644

This feature provides a new option in the `[profile]` section to enable the rustc `hint-mostly-unused` option. This is primarily useful to enable for specific dependencies:

```
[profile.dev.package.huge-mostly-unused-dependency]
hint-mostly-unused = true
```

To enable this feature, pass `-Zprofile-hint-mostly-unused`. However, since this option is a hint, using it without passing `-Zprofile-hint-mostly-unused` will only warn and ignore the profile option. Versions of Cargo prior to the introduction of this feature will give an "unused manifest key" warning, but will otherwise function without erroring. This allows using the hint in a crate's `Cargo.toml` without mandating the use of a newer Cargo to build it.

rustdoc-map

- Tracking Issue: #8296

This feature adds configuration settings that are passed to `rustdoc` so that it can generate links to dependencies whose documentation is hosted elsewhere when the dependency is not documented. First, add this to `.cargo/config`:

```
[doc.extern-map.registries]
crates-io = "https://docs.rs/"
```

Then, when building documentation, use the following flags to cause links to dependencies to link to docs.rs:

```
cargo +nightly doc --no-deps -Zrustdoc-map
```

The `registries` table contains a mapping of registry name to the URL to link to. The URL may have the markers `{pkg_name}` and `{version}` which will get replaced with the corresponding values. If neither are specified, then Cargo defaults to appending `{pkg_name}/{version}/` to the end of the URL.

Another config setting is available to redirect standard library links. By default, rustdoc creates links to https://doc.rust-lang.org/nightly/. To change this behavior, use the `doc.extern-map.std` setting:

```
[doc.extern-map]
std = "local"
```

A value of `"local"` means to link to the documentation found in the `rustc` sysroot. If you are using rustup, this documentation can be installed with `rustup component add rust-docs`.

The default value is `"remote"`.

The value may also take a URL for a custom location.

per-package-target

- Tracking Issue: #9406
- Original Pull Request: #9030
- Original Issue: #7004

The `per-package-target` feature adds two keys to the manifest: `package.default-target` and `package.forced-target`. The first makes the package be compiled by default (ie. when no `--target` argument is passed) for some target. The second one makes the package always be compiled for the target.

Example:

```
[package]
forced-target = "wasm32-unknown-unknown"
```

In this example, the crate is always built for `wasm32-unknown-unknown`, for instance because it is going to be used as a plugin for a main program that runs on the host (or provided on the command line) target.

artifact-dependencies

- Tracking Issue: #9096
- Original Pull Request: #9992

Artifact dependencies allow Cargo packages to depend on `bin`, `cdylib`, and `staticlib` crates, and use the artifacts built by those crates at compile time.

Run `cargo` with `-Z bindeps` to enable this functionality.

artifact-dependencies: Dependency declarations

Artifact-dependencies adds the following keys to a dependency declaration in `Cargo.toml`:

- `artifact` --- This specifies the Cargo Target to build. Normally without this field, Cargo will only build the `[lib]` target from a dependency. This field allows specifying which target will be built, and made available as a binary at build time:
  - `"bin"` --- Compiled executable binaries, corresponding to all of the `[[bin]]` sections in the dependency's manifest.
  - `"bin:<bin-name>"` --- Compiled executable binary, corresponding to a specific binary target specified by the given `<bin-name>`.
  - `"cdylib"` --- A C-compatible dynamic library, corresponding to a `[lib]` section with `crate-type = ["cdylib"]` in the dependency's manifest.
  - `"staticlib"` --- A C-compatible static library, corresponding to a `[lib]` section with `crate-type = ["staticlib"]` in the dependency's manifest.

  The `artifact` value can be a string, or it can be an array of strings to specify multiple targets.

  Example:

```
[dependencies]
bar = { version = "1.0", artifact = "staticlib" }
zoo = { version = "1.0", artifact = ["bin:cat",
"bin:dog"]}
```

- `lib` --- This is a Boolean value which indicates whether or not to also build the dependency's library as a normal Rust `lib` dependency. This field can only be specified when `artifact` is specified.

  The default for this field is `false` when `artifact` is specified. If this is set to `true`, then the dependency's `[lib]` target will also be built for the platform target the declaring package is being built for. This allows the package to use the dependency from Rust code like a normal dependency in addition to an artifact dependency.

  Example:

```
[dependencies]
bar = { version = "1.0", artifact = "bin", lib = true }
```

- `target` --- The platform target to build the dependency for. This field can only be specified when `artifact` is specified.

  The default if this is not specified depends on the dependency kind. For build dependencies, it will be built for the host target. For all other dependencies, it will be built for the same targets the declaring package is built for.

  For a build dependency, this can also take the special value of `"target"` which means to build the dependency for the same targets that the package is being built for.

```
[build-dependencies]
bar = { version = "1.0", artifact = "cdylib", target =
"wasm32-unknown-unknown"}
same-target = { version = "1.0", artifact = "bin", target
= "target" }
```

artifact-dependencies: Environment variables

After building an artifact dependency, Cargo provides the following environment variables that you can use to access the artifact:

- `CARGO_<ARTIFACT-TYPE>_DIR_<DEP>` --- This is the directory containing all the artifacts from the dependency.

  `<ARTIFACT-TYPE>` is the `artifact` specified for the dependency (uppercased as in `CDYLIB`, `STATICLIB`, or `BIN`) and `<DEP>` is the name of the dependency. As with other Cargo environment variables, dependency names are converted to uppercase, with dashes replaced by underscores.

  If your manifest renames the dependency, `<DEP>` corresponds to the name you specify, not the original package name.
- `CARGO_<ARTIFACT-TYPE>_FILE_<DEP>_<NAME>` --- This is the full path to the artifact.

  `<ARTIFACT-TYPE>` is the `artifact` specified for the dependency (uppercased as above), `<DEP>` is the name of the dependency (transformed as above), and `<NAME>` is the name of the artifact from the dependency.

  Note that `<NAME>` is not modified in any way from the `name` specified in the crate supplying the artifact, or the crate name if not specified; for instance, it may be in lowercase, or contain dashes.

  For convenience, if the artifact name matches the original package name, cargo additionally supplies a copy of this variable with the `_<NAME>` suffix omitted. For instance, if the `cmake` crate supplies a binary named `cmake`, Cargo supplies both `CARGO_BIN_FILE_CMAKE` and `CARGO_BIN_FILE_CMAKE_cmake`.

For each kind of dependency, these variables are supplied to the same part of the build process that has access to that kind of dependency:

- For build-dependencies, these variables are supplied to the `build.rs` script, and can be accessed using `std::env::var_os`. (As with any OS file path, these may or may not be valid UTF-8.)

- For normal dependencies, these variables are supplied during the compilation of the crate, and can be accessed using the `env!` macro.
- For dev-dependencies, these variables are supplied during the compilation of examples, tests, and benchmarks, and can be accessed using the `env!` macro.

artifact-dependencies: Examples

Example: use a binary executable from a build script

In the `Cargo.toml` file, you can specify a dependency on a binary to make available for a build script:

```
[build-dependencies]
some-build-tool = { version = "1.0", artifact = "bin" }
```

Then inside the build script, the binary can be executed at build time:

```
fn main() {
                                    let        build_tool        =
std::env::var_os("CARGO_BIN_FILE_SOME_BUILD_TOOL").unwrap();
    let status = std::process::Command::new(build_tool)
        .arg("do-stuff")
        .status()
        .unwrap();
    if !status.success() {
        eprintln!("failed!");
        std::process::exit(1);
    }
}
```

Example: use *cdylib* artifact in build script

The `Cargo.toml` in the consuming package, building the `bar` library as `cdylib` for a specific build target…

```
[build-dependencies]
bar = { artifact = "cdylib", version = "1.0", target =
"wasm32-unknown-unknown" }
```

…along with the build script in `build.rs`.

```
fn main() {

wasm::run_file(std::env::var("CARGO_CDYLIB_FILE_BAR").unwrap()
);
}
```

Example: use *binary* artifact and its library in a binary

The `Cargo.toml` in the consuming package, building the `bar` binary for inclusion as artifact while making it available as library as well…

```
[dependencies]
bar = { artifact = "bin", version = "1.0", lib = true }
```

…along with the executable using `main.rs`.

```
fn main() {
    bar::init();
    command::run(env!("CARGO_BIN_FILE_BAR"));
}
```

publish-timeout

- Tracking Issue: 11222

The `publish.timeout` key in a config file can be used to control how long `cargo publish` waits between posting a package to the registry and it being available in the local index.

A timeout of `0` prevents any checks from occurring. The current default is `60` seconds.

It requires the `-Zpublish-timeout` command-line options to be set.

```
# config.toml
[publish]
timeout = 300  # in seconds
```

asymmetric-token

- Tracking Issue: 10519
- RFC: #3231

The `-Z asymmetric-token` flag enables the `cargo:paseto` credential provider which allows Cargo to authenticate to registries without sending secrets over the network.

In `config.toml` and `credentials.toml` files there is a field called `private-key`, which is a private key formatted in the secret subset of `PASERK` and is used to sign asymmetric tokens

A keypair can be generated with `cargo login --generate-keypair` which will:

- generate a public/private keypair in the currently recommended fashion.
- save the private key in `credentials.toml`.
- print the public key in PASERK public format.

It is recommended that the `private-key` be saved in `credentials.toml`. It is also supported in `config.toml`, primarily so that it can be set using the associated environment variable, which is the recommended way to provide it in CI contexts. This setup is what we have for the `token` field for setting a secret token.

There is also an optional field called `private-key-subject` which is a string chosen by the registry. This string will be included as part of an asymmetric token and should not be secret. It is intended for the rare use cases like "cryptographic proof that the central CA server authorized this action". Cargo requires it to be non-whitespace printable ASCII. Registries that need non-ASCII data should base64 encode it.

Both fields can be set with `cargo login --registry=name --private-key --private-key-subject="subject"` which will prompt you to put in the key value.

A registry can have at most one of `private-key` or `token` set.

All PASETOs will include `iat`, the current time in ISO 8601 format. Cargo will include the following where appropriate:

- `sub` an optional, non-secret string chosen by the registry that is expected to be claimed with every request. The value will be the `private-key-subject` from the `config.toml` file.
- `mutation` if present, indicates that this request is a mutating operation (or a read-only operation if not present), must be one of the strings `publish`, `yank`, or `unyank`.
    - `name` name of the crate related to this request.
    - `vers` version string of the crate related to this request.
    - `cksum` the SHA256 hash of the crate contents, as a string of 64 lowercase hexadecimal digits, must be present only when `mutation` is equal to `publish`
- `challenge` the challenge string received from a 401/403 from this server this session. Registries that issue challenges must track which challenges have been issued/used and never accept a given challenge more than once within the same validity period (avoiding the need to track every challenge ever issued).

The "footer" (which is part of the signature) will be a JSON string in UTF-8 and include:

- `url` the RFC 3986 compliant URL where cargo got the config.json file,
    - If this is a registry with an HTTP index, then this is the base URL that all index queries are relative to.
    - If this is a registry with a GIT index, it is the URL Cargo used to clone the index.
- `kid` the identifier of the private key used to sign the request, using the PASERK IDs standard.

PASETO includes the message that was signed, so the server does not have to reconstruct the exact string from the request in order to check the

signature. The server does need to check that the signature is valid for the string in the PASETO and that the contents of that string matches the request. If a claim should be expected for the request but is missing in the PASETO then the request must be rejected.

`cargo config`

- Original Issue: #2362
- Tracking Issue: #9301

The `cargo config` subcommand provides a way to display the configuration files that cargo loads. It currently includes the `get` subcommand which can take an optional config value to display.

```
cargo +nightly -Zunstable-options config get build.rustflags
```

If no config value is included, it will display all config values. See the `--help` output for more options available.

rustc `--print`

- Tracking Issue: #9357

`cargo rustc --print=VAL` forwards the `--print` flag to `rustc` in order to extract information from `rustc`. This runs `rustc` with the corresponding `--print` flag, and then immediately exits without compiling. Exposing this as a cargo flag allows cargo to inject the correct target and RUSTFLAGS based on the current configuration.

The primary use case is to run `cargo rustc --print=cfg` to get config values for the appropriate target and influenced by any other RUSTFLAGS.

Different binary name

- Tracking Issue: #9778
- PR: #9627

The `different-binary-name` feature allows setting the filename of the binary without having to obey the restrictions placed on crate names. For example, the crate name must use only `alphanumeric` characters or `-` or `_`, and cannot be empty.

The `filename` parameter should **not** include the binary extension, `cargo` will figure out the appropriate extension and use that for the binary on its own.

The `filename` parameter is only available in the `[[bin]]` section of the manifest.

```
cargo-features = ["different-binary-name"]

[package]
name =  "foo"
version = "0.0.1"

[[bin]]
name = "foo"
filename = "007bar"
path = "src/main.rs"
```

scrape-examples

- RFC: #3123
- Tracking Issue: #9910

The `-Z rustdoc-scrape-examples` flag tells Rustdoc to search crates in the current workspace for calls to functions. Those call-sites are then included as documentation. You can use the flag like this:

```
cargo doc -Z unstable-options -Z rustdoc-scrape-examples
```

By default, Cargo will scrape examples from the example targets of packages being documented. You can individually enable or disable targets from being scraped with the `doc-scrape-examples` flag, such as:

```
# Enable scraping examples from a library
[lib]
doc-scrape-examples = true

# Disable scraping examples from an example target
[[example]]
name = "my-example"
doc-scrape-examples = false
```

Note on tests: enabling `doc-scrape-examples` on test targets will not currently have any effect. Scraping examples from tests is a work-in-progress.

Note on dev-dependencies: documenting a library does not normally require the crate's dev-dependencies. However, example targets require dev-deps. For backwards compatibility, `-Z rustdoc-scrape-examples` will *not* introduce a dev-deps requirement for `cargo doc`. Therefore examples will *not* be scraped from example targets under the following conditions:

1. No target being documented requires dev-deps, AND
2. At least one crate with targets being documented has dev-deps, AND
3. The `doc-scrape-examples` parameter is unset or false for all `[[example]]` targets.

If you want examples to be scraped from example targets, then you must not satisfy one of the above conditions. For example, you can set `doc-scrape-examples` to true for one example target, and that signals to Cargo that you are ok with dev-deps being build for `cargo doc`.

output-format for rustdoc

- Tracking Issue: #13283

This flag determines the output format of `cargo rustdoc`, accepting `html` or `json`, providing tools with a way to lean on rustdoc's experimental JSON format.

You can use the flag like this:

```
cargo rustdoc -Z unstable-options --output-format json
```

codegen-backend

The `codegen-backend` feature makes it possible to select the codegen backend used by rustc using a profile.

Example:

```
[package]
name = "foo"

[dependencies]
serde = "1.0.117"

[profile.dev.package.foo]
codegen-backend = "cranelift"
```

To set this in a profile in Cargo configuration, you need to use either `-Z codegen-backend` or `[unstable]` table to enable it. For example,

```
# .cargo/config.toml
[unstable]
codegen-backend = true

[profile.dev.package.foo]
codegen-backend = "cranelift"
```

gitoxide

- Tracking Issue: #11813

With the 'gitoxide' unstable feature, all or the specified git operations will be performed by the `gitoxide` crate instead of `git2`.

While `-Zgitoxide` enables all currently implemented features, one can individually select git operations to run with `gitoxide` with the `-Zgitoxide=operation[,operationN]` syntax.

Valid operations are the following:

- `fetch` - All fetches are done with `gitoxide`, which includes git dependencies as well as the crates index.
- `checkout` *(planned)* - checkout the worktree, with support for filters and submodules.

git

- Tracking Issue: #13285

With the 'git' unstable feature, both `gitoxide` and `git2` will perform shallow fetches of the crate index and git dependencies.

While `-Zgit` enables all currently implemented features, one can individually select when to perform shallow fetches with the `-Zgit=operation[,operationN]` syntax.

Valid operations are the following:

- `shallow-index` - perform a shallow clone of the index.
- `shallow-deps` - perform a shallow clone of git dependencies.

Details on shallow clones

- To enable shallow clones, add `-Zgit=shallow-deps` for fetching git dependencies or `-Zgit=shallow-index` for fetching registry index.
- Shallow-cloned and shallow-checked-out git repositories reside at their own `-shallow` suffixed directories, i.e,
  - `~/.cargo/registry/index/*-shallow`
  - `~/.cargo/git/db/*-shallow`
  - `~/.cargo/git/checkouts/*-shallow`
- When the unstable feature is on, fetching/cloning a git repository is always a shallow fetch. This roughly equals to `git fetch --depth 1` everywhere.
- Even with the presence of `Cargo.lock` or specifying a commit `{ rev = "…" }`, gitoxide and libgit2 are still smart enough to shallow fetch without unshallowing the existing repository.

script

- Tracking Issue: #12207

Cargo can directly run `.rs` files as:

```
$ cargo +nightly -Zscript file.rs
```

where `file.rs` can be as simple as:

```
fn main() {}
```

A user may optionally specify a manifest in a `cargo` code fence in a module-level comment, like:

```
#!/usr/bin/env -S cargo +nightly -Zscript
---cargo
[dependencies]
clap = { version = "4.2", features = ["derive"] }
---

use clap::Parser;

#[derive(Parser, Debug)]
#[clap(version)]
struct Args {
    #[clap(short, long, help = "Path to config")]
    config: Option<std::path::PathBuf>,
}

fn main() {
    let args = Args::parse();
    println!("{:?}", args);
}
```

Single-file packages

In addition to today's multi-file packages (`Cargo.toml` file with other `.rs` files), we are adding the concept of single-file packages which may

contain an embedded manifest. There is no required distinguishment for a single-file `.rs` package from any other `.rs` file.

Single-file packages may be selected via `--manifest-path`, like `cargo test --manifest-path foo.rs`. Unlike `Cargo.toml`, these files cannot be auto-discovered.

A single-file package may contain an embedded manifest. An embedded manifest is stored using `TOML` in rust "frontmatter", a markdown code-fence with `cargo` at the start of the infostring at the top of the file.

Inferred / defaulted manifest fields:

- `package.name = <slugified file stem>`
- `package.edition = <current>` to avoid always having to add an embedded manifest at the cost of potentially breaking scripts on rust upgrades
    - Warn when `edition` is unspecified to raise awareness of this

Disallowed manifest fields:

- `[workspace]`, `[lib]`, `[[bin]]`, `[[example]]`, `[[test]]`, `[[bench]]`
- `package.workspace`, `package.build`, `package.links`, `package.autolib`, `package.autobins`, `package.autoexamples`, `package.autotests`, `package.autobenches`

The default `CARGO_TARGET_DIR` for single-file packages is at `$CARGO_HOME/target/<hash>`:

- Avoid conflicts from multiple single-file packages being in the same directory
- Avoid problems with the single-file package's parent directory being read-only
- Avoid cluttering the user's directory

The lockfile for single-file packages will be placed in `CARGO_TARGET_DIR`. In the future, when workspaces are supported, that will

allow a user to have a persistent lockfile.

Manifest-commands

You may pass a manifest directly to the `cargo` command, without a subcommand, like `foo/Cargo.toml` or a single-file package like `foo.rs`. This is mostly intended for being put in `#!` lines.

The precedence for how to interpret `cargo <subcommand>` is

1. Built-in xor single-file packages
2. Aliases
3. External subcommands

A parameter is identified as a manifest-command if it has one of:

- Path separators
- A `.rs` extension
- The file name is `Cargo.toml`

Differences between `cargo run --manifest-path <path>` and `cargo <path>`

- `cargo <path>` runs with the config for `<path>` and not the current dir, more like `cargo install --path <path>`
- `cargo <path>` is at a verbosity level below the normal default. Pass `-v` to get normal output.

Documentation Updates

Profile `trim-paths` option

- Tracking Issue: rust-lang/cargo#12137
- Tracking Rustc Issue: rust-lang/rust#111540

This adds a new profile setting to control how paths are sanitized in the resulting binary. This can be enabled like so:

```
cargo-features = ["trim-paths"]

[package]
# ...

[profile.release]
trim-paths = ["diagnostics", "object"]
```

To set this in a profile in Cargo configuration, you need to use either `-Z trim-paths` or `[unstable]` table to enable it. For example,

```
# .cargo/config.toml
[unstable]
trim-paths = true

[profile.release]
trim-paths = ["diagnostics", "object"]
```

Documentation updates

trim-paths

as a new "Profiles settings" entry

`trim-paths` is a profile setting which enables and controls the sanitization of file paths in build outputs. It takes the following values:

- `"none"` and `false` --- disable path sanitization
- `"macro"` --- sanitize paths in the expansion of `std::file!()` macro. This is where paths in embedded panic messages come from
- `"diagnostics"` --- sanitize paths in printed compiler diagnostics

- `"object"` --- sanitize paths in compiled executables or libraries
- `"all"` and `true` --- sanitize paths in all possible locations

It also takes an array with the combinations of `"macro"`, `"diagnostics"`, and `"object"`.

It is defaulted to `none` for the `dev` profile, and `object` for the `release` profile. You can manually override it by specifying this option in `Cargo.toml`:

```
[profile.dev]
trim-paths = "all"

[profile.release]
trim-paths = ["object", "diagnostics"]
```

The default `release` profile setting (`object`) sanitizes only the paths in emitted executable or library files. It always affects paths from macros such as panic messages, and in debug information only if they will be embedded together with the binary (the default on platforms with ELF binaries, such as Linux and windows-gnu), but will not touch them if they are in separate files (the default on Windows MSVC and macOS). But the paths to these separate files are sanitized.

If `trim-paths` is not `none` or `false`, then the following paths are sanitized if they appear in a selected scope:

1. Path to the source files of the standard and core library (sysroot) will begin with `/rustc/[rustc commit hash]`, e.g. `/home/username/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/core/src/result.rs` -> `/rustc/fe72845f7bb6a77b9e671e6a4f32fe714962cec4/library/core/src/result.rs`
2. Path to the current package will be stripped, relatively to the current workspace root, e.g. `/home/username/crate/src/lib.rs` -> `src/lib.rs`.

3. Path to dependency packages will be replaced with `[package name]-[version]`. E.g. `/home/username/deps/foo/src/lib.rs` -> `foo-0.1.0/src/lib.rs`

When a path to the source files of the standard and core library is *not* in scope for sanitization, the emitted path will depend on if `rust-src` component is present. If it is, then some paths will point to the copy of the source files on your file system; if it isn't, then they will show up as `/rustc/[rustc commit hash]/library/...` (just like when it is selected for sanitization). Paths to all other source files will not be affected.

This will not affect any hard-coded paths in the source code, such as in strings.

Environment variable

as a new entry of "Environment variables Cargo sets for build scripts"

- `CARGO_TRIM_PATHS` --- The value of `trim-paths` profile option. `false`, `"none"`, and empty arrays would be converted to `none`. `true` and `"all"` become `all`. Values in a non-empty array would be joined into a comma-separated list. If the build script introduces absolute paths to built artifacts (such as by invoking a compiler), the user may request them to be sanitized in different types of artifacts. Common paths requiring sanitization include `OUT_DIR`, `CARGO_MANIFEST_DIR` and `CARGO_MANIFEST_PATH`, plus any other introduced by the build script, such as include directories.

gc

- Tracking Issue: #12633

The `-Zgc` flag is used to enable certain features related to garbage-collection of cargo's global cache within the cargo home directory.

Automatic gc configuration

The `-Zgc` flag will enable Cargo to read extra configuration options related to garbage collection. The settings available are:

```
# Example config.toml file.

# Sub-table for defining specific settings for cleaning the global cache.
[cache.global-clean]
# Anything older than this duration will be deleted in the source cache.
max-src-age = "1 month"
# Anything older than this duration will be deleted in the compressed crate cache.
max-crate-age = "3 months"
# Any index older than this duration will be deleted from the index cache.
max-index-age = "3 months"
# Any git checkout older than this duration will be deleted from the checkout cache.
max-git-co-age = "1 month"
# Any git clone older than this duration will be deleted from the git cache.
max-git-db-age = "3 months"
```

Note that the `cache.auto-clean-frequency` option was stabilized in Rust 1.88.

Manual garbage collection with `cargo clean`

Manual deletion can be done with the `cargo clean gc -Zgc` command. Deletion of cache contents can be performed by passing one of the cache options:

- `--max-src-age=DURATION` --- Deletes source cache files that have not been used since the given age.
- `--max-crate-age=DURATION` --- Deletes crate cache files that have not been used since the given age.
- `--max-index-age=DURATION` --- Deletes registry indexes that have not been used since then given age (including their `.crate` and `src` files).
- `--max-git-co-age=DURATION` --- Deletes git dependency checkouts that have not been used since then given age.
- `--max-git-db-age=DURATION` --- Deletes git dependency clones that have not been used since then given age.
- `--max-download-age=DURATION` --- Deletes any downloaded cache data that has not been used since then given age.
- `--max-src-size=SIZE` --- Deletes the oldest source cache files until the cache is under the given size.
- `--max-crate-size=SIZE` --- Deletes the oldest crate cache files until the cache is under the given size.
- `--max-git-size=SIZE` --- Deletes the oldest git dependency caches until the cache is under the given size.
- `--max-download-size=SIZE` --- Deletes the oldest downloaded cache data until the cache is under the given size.

A DURATION is specified in the form "N seconds/minutes/days/weeks/months" where N is an integer.

A SIZE is specified in the form "N *suffix*" where *suffix* is B, kB, MB, GB, kiB, MiB, or GiB, and N is an integer or floating point number. If no suffix is specified, the number is the number of bytes.

```
cargo clean gc -Zgc
cargo clean gc -Zgc --max-download-age=1week
cargo clean gc -Zgc --max-git-size=0 --max-download-size=100MB
```

open-namespaces

- Tracking Issue: #13576

Allow multiple packages to participate in the same API namespace

This can be enabled like so:

```
cargo-features = ["open-namespaces"]

[package]
# ...
```

`[lints.cargo]`

- Tracking Issue: #12235

A new `lints` tool table for `cargo` that can be used to configure lints emitted by `cargo` itself when `-Zcargo-lints` is used

```
[lints.cargo]
implicit-features = "warn"
```

This will work with RFC 2906 `workspace-deduplicate`:

```
[workspace.lints.cargo]
implicit-features = "warn"

[lints]
workspace = true
```

Path Bases

- Tracking Issue: #14355

A `path` dependency may optionally specify a base by setting the `base` key to the name of a path base from the `[path-bases]` table in either the configuration or one of the built-in path bases. The value of that path base is prepended to the `path` value (along with a path separator if necessary) to produce the actual location where Cargo will look for the dependency.

For example, if the `Cargo.toml` contains:

```
cargo-features = ["path-bases"]

[dependencies]
foo = { base = "dev", path = "foo" }
```

Given a `[path-bases]` table in the configuration that contains:

```
[path-bases]
dev = "/home/user/dev/rust/libraries/"
```

This will produce a `path` dependency `foo` located at `/home/user/dev/rust/libraries/foo`.

Path bases can be either absolute or relative. Relative path bases are relative to the parent directory of the configuration file that declared that path base.

The name of a path base must use only alphanumeric characters or `-` or `_`, must start with an alphabetic character, and must not be empty.

If the name of path base used in a dependency is neither in the configuration nor one of the built-in path base, then Cargo will raise an error.

Built-in path bases

Cargo provides implicit path bases that can be used without the need to specify them in a `[path-bases]` table.

- `workspace` - If a project is a workspace or workspace member then this path base is defined as the parent directory of the root `Cargo.toml` of the workspace.

If a built-in path base name is also declared in the configuration, then Cargo will prefer the value in the configuration. The allows Cargo to add new built-in path bases without compatibility issues (as existing uses will shadow the built-in name).

lockfile-path

- Original Issue: #5707
- Tracking Issue: #14421

This feature allows you to specify the path of lockfile Cargo.lock. By default, lockfile is written into `<workspace_root>/Cargo.lock`. However, when sources are stored in read-only directory, most of the cargo commands would fail, trying to write a lockfile. The `--lockfile-path` flag makes it easier to work with readonly sources. Note, that currently path must end with `Cargo.lock`. Meaning, if you want to use this feature in multiple projects, lockfiles should be stored in different directories. Example:

```
cargo +nightly metadata --lockfile-path=$LOCKFILES_ROOT/my-project/Cargo.lock -Z unstable-options
```

package-workspace

- Tracking Issue: #10948

This allows cargo to package (or publish) multiple crates in a workspace, even if they have inter-dependencies. For example, consider a workspace containing packages `foo` and `dep`, where `foo` depends on `dep`. Then

```
cargo +nightly -Zpackage-workspace package -p foo -p dep
```

will package both `foo` and `dep`, while

```
cargo +nightly -Zpackage-workspace publish -p foo -p dep
```

will publish both `foo` and `dep`. If `foo` and `dep` are the only crates in the workspace, you can use the `--workspace` flag instead of specifying the crates individually:

```
cargo +nightly -Zpackage-workspace package --workspace
cargo +nightly -Zpackage-workspace publish --workspace
```

Lock-file behavior

When packaging a binary at the same time as one of its dependencies, the binary will be packaged with a lock-file pointing at the dependency's registry entry *as though the dependency were already published*, even though it has not yet been. In this case, `cargo` needs to know the registry that the dependency will eventually be published on. `cargo` will attempt to infer this registry by examining the the `publish` field, falling back to `crates.io` if no `publish` field is set. To explicitly set the registry, pass a `--registry` or `--index` flag.

```
cargo +nightly -Zpackage-workspace --registry=my-registry
package -p foo -p dep
cargo +nightly -Zpackage-workspace --index=https://example.com
package -p foo -p dep
```

native-completions

- Original Issue: #6645
- Tracking Issue: #14520

This feature moves the handwritten completion scripts to Rust native, making it easier for us to add, extend and test new completions. This feature is enabled with the nightly channel, without requiring additional `-Z` options.

Areas of particular interest for feedback

- Arguments that need escaping or quoting that aren't handled correctly
- Inaccuracies in the information
- Bugs in parsing of the command-line
- Arguments that don't report their completions
- If a known issue is being problematic

Feedback can be broken down into

- What completion candidates are reported
  - Known issues: #14520, `A-completions`
  - Report an issue or discuss the behavior
- Shell integration, command-line parsing, and completion filtering
  - Known issues: clap#3166, clap's `A-completions`
  - Report an issue or discuss the behavior

When in doubt, you can discuss this in #14520 or on zulip

How to use native-completions feature:

- bash: Add `source <(CARGO_COMPLETE=bash cargo +nightly)` to `~/.local/share/bash-completion/completions/cargo`.
- zsh: Add `source <(CARGO_COMPLETE=zsh cargo +nightly)` to your `.zshrc`.

- fish: Add `source (CARGO_COMPLETE=fish cargo +nightly | psub)` to `$XDG_CONFIG_HOME/fish/completions/cargo.fish`
- elvish: Add `eval (E:CARGO_COMPLETE=elvish cargo +nightly | slurp)` to `$XDG_CONFIG_HOME/elvish/rc.elv`
- powershell: Add `CARGO_COMPLETE=powershell cargo +nightly | Invoke-Expression` to `$PROFILE`.

warnings

- Original Issue: #8424
- Tracking Issue: #14802

The `-Z warnings` feature enables the `build.warnings` configuration option to control how Cargo handles warnings. If the `-Z warnings` unstable flag is not enabled, then the `build.warnings` config will be ignored.

This setting currently only applies to rustc warnings. It may apply to additional warnings (such as Cargo lints or Cargo warnings) in the future.

`build.warnings`

- Type: string
- Default: `warn`
- Environment: `CARGO_BUILD_WARNINGS`

Controls how Cargo handles warnings. Allowed values are:

- `warn`: warnings are emitted as warnings (default).
- `allow`: warnings are hidden.
- `deny`: if warnings are emitted, an error will be raised at the end of the operation and the process will exit with a failure exit code.

feature unification

- RFC: #3692
- Tracking Issue: #14774

The `-Z feature-unification` enables the `resolver.feature-unification` configuration option to control how features are unified across a workspace. If the `-Z feature-unification` unstable flag is not enabled, then the `resolver.feature-unification` configuration will be ignored.

`resolver.feature-unification`

- Type: string
- Default: `"selected"`
- Environment: `CARGO_RESOLVER_FEATURE_UNIFICATION`

Specify which packages participate in feature unification.

- `selected`: Merge dependency features from all packages specified for the current build.
- `workspace`: Merge dependency features across all workspace members, regardless of which packages are specified for the current build.
- `package`: Dependency features are considered on a package-by-package basis, preferring duplicate builds of dependencies when different sets of features are activated by the packages.

Package message format

- Original Issue: #11666
- Tracking Issue: #15353

The `--message-format` flag in `cargo package` controls the output message format. Currently, it only works with the `--list` flag and affects the file listing format, Requires `-Zunstable-options`. See `cargo package --message-format` for more information.

rustdoc depinfo

- Original Issue: #12266
- Tracking Issue: #15370

The `-Z rustdoc-depinfo` flag leverages rustdoc's dep-info files to determine whether documentations are required to re-generate. This can be combined with `-Z checksum-freshness` to detect checksum changes rather than file mtime.

no-embed-metadata

- Original Pull Request: #15378
- Tracking Issue: #15495

The default behavior of Rust is to embed crate metadata into `rlib` and `dylib` artifacts. Since Cargo also passes `--emit=metadata` to these intermediate artifacts to enable pipelined compilation, this means that a lot of metadata ends up being duplicated on disk, which wastes disk space in the target directory.

This feature tells Cargo to pass the `-Zembed-metadata=no` flag to the compiler, which instructs it not to embed metadata within rlib and dylib artifacts. In this case, the metadata will only be stored in `.rmeta` files.

```
cargo +nightly -Zno-embed-metadata build
```

unstable-editions

The `unstable-editions` value in the `cargo-features` list allows a `Cargo.toml` manifest to specify an edition that is not yet stable.

```
cargo-features = ["unstable-editions"]

[package]
name = "my-package"
edition = "future"
```

When new editions are introduced, the `unstable-editions` feature is required until the edition is stabilized.

The special "future" edition is a home for new features that are under development, and is permanently unstable. The "future" edition also has no new behavior by itself. Each change in the future edition requires an opt-in such as a `#![feature(...)]` attribute.

fix-edition

`-Zfix-edition` is a permanently unstable flag to assist with testing edition migrations, particularly with the use of crater. It only works with the `cargo fix` subcommand. It takes two different forms:

- `-Zfix-edition=start=$INITIAL` --- This form checks if the current edition is equal to the given number. If not, it exits with success (because we want to ignore older editions). If it is, then it runs the equivalent of `cargo check`. This is intended to be used with crater's "start" toolchain to set a baseline for the "before" toolchain.
- `-Zfix-edition=end=$INITIAL,$NEXT` --- This form checks if the current edition is equal to the given `$INITIAL` value. If not, it exits with success. If it is, then it performs an edition migration to the edition specified in `$NEXT`. Afterwards, it will modify `Cargo.toml` to add the appropriate `cargo-features = ["unstable-edition"]`, update the `edition` field, and run the equivalent of `cargo check` to verify that the migration works on the new edition.

For example:

```
cargo +nightly fix -Zfix-edition=end=2024,future
```

Stabilized and removed features

Compile progress

The compile-progress feature has been stabilized in the 1.30 release. Progress bars are now enabled by default. See `term.progress` for more information about controlling this feature.

Edition

Specifying the `edition` in `Cargo.toml` has been stabilized in the 1.31 release. See the edition field for more information about specifying this field.

rename-dependency

Specifying renamed dependencies in `Cargo.toml` has been stabilized in the 1.31 release. See renaming dependencies for more information about renaming dependencies.

Alternate Registries

Support for alternate registries has been stabilized in the 1.34 release. See the Registries chapter for more information about alternate registries.

Offline Mode

The offline feature has been stabilized in the 1.36 release. See the `--offline` flag for more information on using the offline mode.

publish-lockfile

The `publish-lockfile` feature has been removed in the 1.37 release. The `Cargo.lock` file is always included when a package is published if the package contains a binary target. `cargo install` requires the `--locked` flag to use the `Cargo.lock` file. See `cargo package` and `cargo install` for more information.

default-run

The `default-run` feature has been stabilized in the 1.37 release. See the `default-run` field for more information about specifying the default target to run.

cache-messages

Compiler message caching has been stabilized in the 1.40 release. Compiler warnings are now cached by default and will be replayed automatically when re-running Cargo.

install-upgrade

The `install-upgrade` feature has been stabilized in the 1.41 release. `cargo install` will now automatically upgrade packages if they appear to be out-of-date. See the `cargo install` documentation for more information.

Profile Overrides

Profile overrides have been stabilized in the 1.41 release. See Profile Overrides for more information on using overrides.

Config Profiles

Specifying profiles in Cargo config files and environment variables has been stabilized in the 1.43 release. See the config `[profile]` table for more information about specifying profiles in config files.

crate-versions

The `-Z crate-versions` flag has been stabilized in the 1.47 release. The crate version is now automatically included in the `cargo doc` documentation sidebar.

Features

The `-Z features` flag has been stabilized in the 1.51 release. See feature resolver version 2 for more information on using the new feature resolver.

package-features

The `-Z package-features` flag has been stabilized in the 1.51 release. See the resolver version 2 command-line flags for more information on using the features CLI options.

Resolver

The `resolver` feature in `Cargo.toml` has been stabilized in the 1.51 release. See the resolver versions for more information about specifying resolvers.

extra-link-arg

The `extra-link-arg` feature to specify additional linker arguments in build scripts has been stabilized in the 1.56 release. See the build script documentation for more information on specifying extra linker arguments.

configurable-env

The `configurable-env` feature to specify environment variables in Cargo configuration has been stabilized in the 1.56 release. See the config documentation for more information about configuring environment variables.

rust-version

The `rust-version` field in `Cargo.toml` has been stabilized in the 1.56 release. See the rust-version field for more information on using the `rust-version` field and the `--ignore-rust-version` option.

patch-in-config

The `-Z patch-in-config` flag, and the corresponding support for `[patch]` section in Cargo configuration files has been stabilized in the 1.56 release. See the patch field for more information.

edition 2021

The 2021 edition has been stabilized in the 1.56 release. See the `edition` field for more information on setting the edition. See `cargo fix --edition` and The Edition Guide for more information on migrating existing projects.

Custom named profiles

Custom named profiles have been stabilized in the 1.57 release. See the profiles chapter for more information.

Profile `strip` option

The profile `strip` option has been stabilized in the 1.59 release. See the profiles chapter for more information.

Future incompat report

Support for generating a future-incompat report has been stabilized in the 1.59 release. See the future incompat report chapter for more information.

Namespaced features

Namespaced features has been stabilized in the 1.60 release. See the Features chapter for more information.

Weak dependency features

Weak dependency features has been stabilized in the 1.60 release. See the Features chapter for more information.

timings

The `-Ztimings` option has been stabilized as `--timings` in the 1.60 release. (`--timings=html` and the machine-readable `--timings=json` output remain unstable and require `-Zunstable-options`.)

config-cli

The `--config` CLI option has been stabilized in the 1.63 release. See the config documentation for more information.

multitarget

The `-Z multitarget` option has been stabilized in the 1.64 release. See `build.target` for more information about setting the default target platform triples.

crate-type

The `--crate-type` flag for `cargo rustc` has been stabilized in the 1.64 release. See the `cargo rustc` documentation for more information.

Workspace Inheritance

Workspace Inheritance has been stabilized in the 1.64 release. See workspace.package, workspace.dependencies, and inheriting-a-dependency-from-a-workspace for more information.

terminal-width

The `-Z terminal-width` option has been stabilized in the 1.68 release. The terminal width is always passed to the compiler when running from a terminal where Cargo can automatically detect the width.

sparse-registry

Sparse registry support has been stabilized in the 1.68 release. See Registry Protocols for more information.

`cargo logout`

The `cargo logout` command has been stabilized in the 1.70 release.

doctest-in-workspace

The `-Z doctest-in-workspace` option for `cargo test` has been stabilized and enabled by default in the 1.72 release. See the `cargo test` documentation for more information about the working directory for compiling and running tests.

keep-going

The `--keep-going` option has been stabilized in the 1.74 release. See the `--keep-going` flag in `cargo build` as an example for more details.

`[lints]`

`[lints]` (enabled via `-Zlints`) has been stabilized in the 1.74 release.

credential-process

The `-Z credential-process` feature has been stabilized in the 1.74 release.

See Registry Authentication documentation for details.

registry-auth

The `-Z registry-auth` feature has been stabilized in the 1.74 release with the additional requirement that a credential-provider is configured.

See Registry Authentication documentation for details.

check-cfg

The `-Z check-cfg` feature has been stabilized in the 1.80 release by making it the default behavior.

See the build script documentation for information about specifying custom cfgs.

Edition 2024

The 2024 edition has been stabilized in the 1.85 release. See the `edition` field for more information on setting the edition. See `cargo fix --edition` and The Edition Guide for more information on migrating existing projects.

Automatic garbage collection

Support for automatically deleting old files was stabilized in Rust 1.88. More information can be found in the config chapter.

doctest-xcompile

Doctest cross-compiling is now unconditionally enabled starting in Rust 1.89. Running doctests with `cargo test` will now honor the `--target` flag.

compile-time-deps

This permanently-unstable flag to only build proc-macros and build scripts (and their required dependencies), as well as run the build scripts.

It is intended for use by tools like rust-analyzer and will never be stabilized.

Example:

```
cargo +nightly build --compile-time-deps -Z unstable-options
cargo +nightly check --compile-time-deps --all-targets -Z unstable-options
```

Cargo Commands

General Commands

- cargo
- cargo help
- cargo version

cargo(1)

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-add(1)
Add dependencies to a `Cargo.toml` manifest file.

cargo-generate-lockfile(1)
Generate `Cargo.lock` for a project.

cargo-info(1)
Display information about a package in the registry. Default registry is crates.io.

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 in machine-readable format.

cargo-pkgid(1)
Print a fully qualified package specification.

cargo-remove(1)
Remove dependencies from a `Cargo.toml` manifest file.

cargo-tree(1)
Display a tree visualization of a dependency graph.

cargo-update(1)
Update dependencies as recorded in the local lock file.

cargo-vendor(1)
Vendor all dependencies locally.

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-logout(1)
Remove 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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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.toml`

The global configuration file. See the reference for more information about configuration files.

`.cargo/config.toml`

Cargo automatically searches for a file named `.cargo/config.toml` in the current directory, and all parent directories. These configuration files will be merged with the global configuration file.

`$CARGO_HOME/credentials.toml`

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)

cargo-help(1)

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(1)

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)

Build Commands

- cargo bench
- cargo build
- cargo check
- cargo clean
- cargo clippy
- cargo doc
- cargo fetch
- cargo fix
- cargo fmt
- cargo miri
- cargo report
- cargo run
- cargo rustc
- cargo rustdoc
- cargo test

cargo-bench(1)

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 are 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` and check out the rustc book's chapter on how tests work at https://doc.rust-lang.org/rustc/tests/index.html.

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 a special executable by linking your code with libtest. The executable 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, regardless of whether the harness is libtest or a custom harness.

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.

> **Note**: The `#[bench]` attribute is currently unstable and only available on the nightly channel. There are some packages available on crates.io that may help with running benchmarks on the stable channel, such as Criterion.

By default, `cargo bench` uses the `bench` profile, which enables optimizations and disables debugging information. If you need to debug a benchmark, you can use the `--profile=dev` command-line option to switch to the dev profile. You can then run the debug-enabled benchmark within a debugger.

Working directory of benchmarks

The working directory of every benchmark is set to the root directory of the package the benchmark belongs to. Setting the working directory of benchmarks to the package's root directory makes it possible for benchmarks to reliably access the package's files using relative paths, regardless from where `cargo bench` was executed from.

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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

`--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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

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, replacing the example's `main` function with the libtest harness.

Setting targets to `bench = false` will stop them from being benchmarked by default. Target selection options that take a target by name (such as `--example foo`) ignore the `bench` flag and will always benchmark the given target.

See Configuring a target for more information on per-target settings.

Binary targets are automatically built if there is an integration test or benchmark being selected to benchmark. This allows an integration test to execute the binary to exercise and test its behavior. The `CARGO_BIN_EXE_<name>` environment variable is set when the integration test is built so that it can use the `env` macro to locate the executable.

Passing target selection flags will benchmark only the specified targets.

Note that `--bin`, `--example`, `--test` and `--bench` flags also support common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each glob pattern.

`--lib`

Benchmark the package's library.

`--bin` *name*…

Benchmark the specified binary. This flag may be specified multiple times and supports common Unix glob patterns.

`--bins`

Benchmark all binary targets.

`--example` *name*…

Benchmark the specified example. This flag may be specified multiple times and supports common Unix glob patterns.

`--examples`

Benchmark all example targets.

`--test` *name*…

Benchmark the specified integration test. This flag may be specified multiple times and supports common Unix glob patterns.

`--tests`

Benchmark all targets 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 and supports common Unix glob patterns.

`--benches`

Benchmark all targets 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

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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 flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`--profile` *name*

Benchmark with the given profile. See the reference for more details on profiles.

`--timings=`*fmts*

Output information how long each compilation takes, and track concurrency information over time. Accepts an optional comma-separated list of output formats; `--timings` without an argument will default to `--timings=html`. Specifying an output format (rather than the default) is unstable and requires `-Zunstable-options`. Valid output formats:

- `html` (unstable, requires `-Zunstable-options`): Write a human-readable file `cargo-timing.html` to the `target/cargo-timings` directory with a report of the compilation. Also write a report to the same directory with a timestamp in the filename if you want to look at older runs. HTML output is suitable for human consumption only, and does not provide machine-readable timing data.
- `json` (unstable, requires `-Zunstable-options`): Emit machine-readable JSON information about timing information.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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. Conflicts with `short` and `json`.
- `short`: Emit shorter, human-readable text messages. Conflicts with `human` and `json`.
- `json`: Emit JSON messages to stdout. See the reference for more details. Conflicts with `human` and `short`.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the “short” rendering from rustc. Cannot be used with `human` or `short`.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc’s default color scheme. Cannot be used with `human` or `short`.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics 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. Cannot be used with `human` or `short`.

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.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

While `cargo bench` involves compilation, it does not provide a `--keep-going` flag. Use `--no-fail-fast` to run as many benchmarks as possible without stopping at the first failure. To "compile" as many benchmarks as possible, use `--benches` to build benchmark binaries separately. For example:

```
cargo build --benches --release --keep-going
cargo bench --no-fail-fast
```

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(1)

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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

`--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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

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.

Binary targets are automatically built if there is an integration test or benchmark being selected to build. This allows an integration test to execute the binary to exercise and test its behavior. The `CARGO_BIN_EXE_<name>` environment variable is set when the integration test is built so that it can use the `env` macro to locate the executable.

Passing target selection flags will build only the specified targets.

Note that `--bin`, `--example`, `--test` and `--bench` flags also support common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each glob pattern.

`--lib`

Build the package's library.

`--bin` *name*…

Build the specified binary. This flag may be specified multiple times and supports common Unix glob patterns.

`--bins`

Build all binary targets.

`--example` *name*…

Build the specified example. This flag may be specified multiple times and supports common Unix glob patterns.

`--examples`

Build all example targets.

`--test` *name*…

Build the specified integration test. This flag may be specified multiple times and supports common Unix glob patterns.

`--tests`

Build all targets 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 and supports common Unix glob patterns.

`--benches`

Build all targets 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

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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 flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`-r`

`--release`

Build optimized artifacts with the `release` profile. See also the `--profile` option for choosing a specific profile by name.

`--profile` *name*

Build with the given profile. See the reference for more details on profiles.

`--timings=`*fmts*

Output information how long each compilation takes, and track concurrency information over time. Accepts an optional comma-separated list of output formats; `--timings` without an argument will default to `--timings=html`. Specifying an output format (rather than the default) is unstable and requires `-Zunstable-options`. Valid output formats:

- `html` (unstable, requires `-Zunstable-options`): Write a human-readable file `cargo-timing.html` to the `target/cargo-timings` directory with a report of the compilation. Also write a report to the same directory with a timestamp in the filename if you want to look at older runs. HTML output is suitable for human consumption only, and does not provide machine-readable timing data.
- `json` (unstable, requires `-Zunstable-options`): Emit machine-readable JSON information about timing information.

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.

`--artifact-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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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. Conflicts with `short` and `json`.
- `short`: Emit shorter, human-readable text messages. Conflicts with `human` and `json`.
- `json`: Emit JSON messages to stdout. See the reference for more details. Conflicts with `human` and `short`.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc. Cannot be used with `human` or `short`.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme. Cannot be used with `human` or `short`.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics 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. Cannot be used with `human` or `short`.

`--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.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an

error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

`--keep-going`

Build as many crates in the dependency graph as possible, rather than aborting the build on the first one that fails to build.

For example if the current package depends on dependencies `fails` and `works`, one of which fails to build, `cargo build -j1` may or may not build the one that succeeds (depending on which one of the two builds Cargo picked to run first), whereas `cargo build -j1 --keep-going` would definitely run both builds, even if the one run first fails.

`--future-incompat-report`

Displays a future-incompat report for any future-incompatible warnings produced during execution of this command

See cargo-report(1)

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(1)

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. Some diagnostics and errors are only emitted during code generation, so they inherently won't be reported with `cargo check`.

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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

`--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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

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.

Note that `--bin`, `--example`, `--test` and `--bench` flags also support common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each glob pattern.

`--lib`

Check the package's library.

`--bin` *name*…

Check the specified binary. This flag may be specified multiple times and supports common Unix glob patterns.

`--bins`

Check all binary targets.

`--example` *name*…

Check the specified example. This flag may be specified multiple times and supports common Unix glob patterns.

`--examples`

Check all example targets.

`--test` *name*…

Check the specified integration test. This flag may be specified multiple times and supports common Unix glob patterns.

`--tests`

Check all targets 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 and supports common Unix glob patterns.

`--benches`

Check all targets 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

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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 flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`-r`

`--release`

Check optimized artifacts with the `release` profile. See also the `--profile` option for choosing a specific profile by name.

`--profile` *name*

Check with the given profile.

As a special case, specifying the `test` profile will also enable checking in test mode which will enable checking tests and enable the `test` cfg option. See rustc tests for more detail.

See the reference for more details on profiles.

`--timings=`*fmts*

Output information how long each compilation takes, and track concurrency information over time. Accepts an optional comma-separated list of output formats; `--timings` without an argument will default to `--timings=html`. Specifying an output format (rather than the default) is unstable and requires `-Zunstable-options`. Valid output formats:

- `html` (unstable, requires `-Zunstable-options`): Write a human-readable file `cargo-timing.html` to the `target/cargo-timings` directory with a report of the compilation. Also write a report to the same directory with a timestamp in the filename if you want to look at older runs. HTML output is suitable for human consumption only, and does not provide machine-readable timing data.
- `json` (unstable, requires `-Zunstable-options`): Emit machine-readable JSON information about timing information.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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. Conflicts with `short` and `json`.
- `short`: Emit shorter, human-readable text messages. Conflicts with `human` and `json`.
- `json`: Emit JSON messages to stdout. See the reference for more details. Conflicts with `human` and `short`.

- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the “short” rendering from rustc. Cannot be used with `human` or `short`.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc’s default color scheme. Cannot be used with `human` or `short`.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics 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. Cannot be used with `human` or `short`.

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.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

`--keep-going`

Build as many crates in the dependency graph as possible, rather than aborting the build on the first one that fails to build.

For example if the current package depends on dependencies `fails` and `works`, one of which fails to build, `cargo check -j1` may or may not build the one that succeeds (depending on which one of the two builds Cargo picked to run first), whereas `cargo check -j1 --keep-going` would definitely run both builds, even if the one run first fails.

`--future-incompat-report`

Displays a future-incompat report for any future-incompatible warnings produced during execution of this command

See cargo-report(1)

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(1)

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

`--dry-run`

Displays a summary of what would be deleted without deleting anything. Use with `--verbose` to display the actual files that would be deleted.

`--doc`

This option will cause `cargo clean` to remove only the `doc` directory in the target directory.

`--release`

Remove all artifacts in the `release` directory.

`--profile` *name*

Remove all artifacts in the directory with the given profile name.

`--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 flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.

- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as

`+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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-clippy(1)

NAME

cargo-clippy --- Checks a package to catch common mistakes and improve your Rust code

DESCRIPTION

This is an external command distributed with the Rust toolchain as an optional component. It is not built into Cargo, and may require additional installation.

For information about usage and installation, see Clippy Documentation.

SEE ALSO

cargo(1), cargo-fix(1), cargo-fmt(1), Custom subcommands

cargo-doc(1)

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 or use the `doc.browser` configuration option.

`--no-deps`

Do not build documentation for dependencies.

`--document-private-items`

Include non-public items in the documentation. This will be enabled by default if documenting a binary target.

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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

`--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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

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 and supports common Unix glob patterns.

`--bins`

Document all binary targets.

`--example` *name*…

Document the specified example. This flag may be specified multiple times and supports common Unix glob patterns.

`--examples`

Document all example targets.

Feature Selection

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every

selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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 flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`-r`

`--release`

Document optimized artifacts with the `release` profile. See also the `--profile` option for choosing a specific profile by name.

`--profile` *name*

Document with the given profile. See the reference for more details on profiles.

`--timings=`*fmts*

Output information how long each compilation takes, and track concurrency information over time. Accepts an optional comma-separated

list of output formats; `--timings` without an argument will default to `--timings=html`. Specifying an output format (rather than the default) is unstable and requires `-Zunstable-options`. Valid output formats:

- `html` (unstable, requires `-Zunstable-options`): Write a human-readable file `cargo-timing.html` to the `target/cargo-timings` directory with a report of the compilation. Also write a report to the same directory with a timestamp in the filename if you want to look at older runs. HTML output is suitable for human consumption only, and does not provide machine-readable timing data.
- `json` (unstable, requires `-Zunstable-options`): Emit machine-readable JSON information about timing information.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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. Conflicts with `short` and `json`.
- `short`: Emit shorter, human-readable text messages. Conflicts with `human` and `json`.
- `json`: Emit JSON messages to stdout. See the reference for more details. Conflicts with `human` and `short`.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc. Cannot be used with `human` or `short`.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme. Cannot be used with `human` or `short`.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics 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. Cannot be used with `human` or `short`.

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.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

`--keep-going`

Build as many crates in the dependency graph as possible, rather than aborting the build on the first one that fails to build.

For example if the current package depends on dependencies `fails` and `works`, one of which fails to build, `cargo doc -j1` may or may not build the one that succeeds (depending on which one of the two builds Cargo picked to run first), whereas `cargo doc -j1 --keep-going` would definitely run both builds, even if the one run first fails.

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(1)

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 will be able to run offline 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 all architectures. The general format of the triple is `<arch><sub>-<vendor>-<sys>-<abi>`. Run `rustc --print target-list` for a list of supported targets. This flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

May also be specified with the `net.offline` config value.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't

exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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(1)

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!

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 apply all fixes to the current package, you can run:

```
cargo fix
```

which behaves the same as `cargo check --all-targets`.

`cargo fix` is only capable of fixing code that is normally compiled with `cargo check`. If code is conditionally enabled with optional features, you will need to enable those features for that code to be analyzed:

```
cargo fix --features foo
```

Similarly, other `cfg` expressions like platform-specific code will need to pass `--target` to fix code for the given target.

```
cargo fix --target x86_64-pc-windows-gnu
```

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.

Edition migration

The `cargo fix` subcommand can also be used to migrate a package from one edition to the next. The general procedure is:

1. Run `cargo fix --edition`. Consider also using the `--all-features` flag if your project has multiple features. You may also want to run `cargo fix --edition` multiple times with different `--target` flags if your project has platform-specific code gated by `cfg` attributes.

2. Modify `Cargo.toml` to set the edition field to the new edition.
3. Run your project tests to verify that everything still works. If new warnings are issued, you may want to consider running `cargo fix` again (without the `--edition` flag) to apply any suggestions given by the compiler.

And hopefully that's it! Just keep in mind of the caveats mentioned above that `cargo fix` cannot update code for inactive features or `cfg` expressions. Also, in some rare cases the compiler is unable to automatically migrate all code to the new edition, and this may require manual changes after building with the new edition.

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 next edition. This will not update the edition in the `Cargo.toml` manifest, which must be updated manually after `cargo fix --edition` has finished.

`--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 (including staged 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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

`--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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

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.

Note that `--bin`, `--example`, `--test` and `--bench` flags also support common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each glob pattern.

`--lib`

Fix the package's library.

`--bin` *name*…

Fix the specified binary. This flag may be specified multiple times and supports common Unix glob patterns.

`--bins`

Fix all binary targets.

`--example` *name*…

Fix the specified example. This flag may be specified multiple times and supports common Unix glob patterns.

`--examples`

Fix all example targets.

`--test` *name*…

Fix the specified integration test. This flag may be specified multiple times and supports common Unix glob patterns.

`--tests`

Fix all targets 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 and supports common Unix glob patterns.

`--benches`

Fix all targets 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

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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 flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`-r`

`--release`

Fix optimized artifacts with the `release` profile. See also the `--profile` option for choosing a specific profile by name.

`--profile` *name*

Fix with the given profile.

As a special case, specifying the `test` profile will also enable checking in test mode which will enable checking tests and enable the `test` cfg option. See rustc tests for more detail.

See the reference for more details on profiles.

`--timings=`*fmts*

Output information how long each compilation takes, and track concurrency information over time. Accepts an optional comma-separated list of output formats; `--timings` without an argument will default to `--timings=html`. Specifying an output format (rather than the default) is unstable and requires `-Zunstable-options`. Valid output formats:

- `html` (unstable, requires `-Zunstable-options`): Write a human-readable file `cargo-timing.html` to the `target/cargo-timings` directory with a report of the compilation. Also write a report to the same directory with a timestamp in the filename if you want to look at older runs. HTML output is suitable for human consumption only, and does not provide machine-readable timing data.
- `json` (unstable, requires `-Zunstable-options`): Emit machine-readable JSON information about timing information.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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. Conflicts with `short` and `json`.
- `short`: Emit shorter, human-readable text messages. Conflicts with `human` and `json`.
- `json`: Emit JSON messages to stdout. See the reference for more details. Conflicts with `human` and `short`.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc. Cannot be used with `human` or `short`.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme. Cannot be used with `human` or `short`.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics 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. Cannot be used with `human` or `short`.

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.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't

exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

`--keep-going`

Build as many crates in the dependency graph as possible, rather than aborting the build on the first one that fails to build.

For example if the current package depends on dependencies `fails` and `works`, one of which fails to build, `cargo fix -j1` may or may not build the one that succeeds (depending on which one of the two builds Cargo picked to run first), whereas `cargo fix -j1 --keep-going` would definitely run both builds, even if the one run first fails.

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. Update a package to prepare it for the next edition:

```
cargo fix --edition
```

3. Apply suggested idioms for the current edition:

```
cargo fix --edition-idioms
```

SEE ALSO

cargo(1), cargo-check(1)

cargo-fmt(1)

NAME

cargo-fmt --- Formats all bin and lib files of the current crate using rustfmt

DESCRIPTION

This is an external command distributed with the Rust toolchain as an optional component. It is not built into Cargo, and may require additional installation.

For information about usage and installation, see https://github.com/rust-lang/rustfmt.

SEE ALSO

cargo(1), cargo-fix(1), cargo-clippy(1), Custom subcommands

cargo-miri(1)

NAME

cargo-miri --- Runs binary crates and tests in Miri

DESCRIPTION

This is an external command distributed with the Rust toolchain as an optional component. It is not built into Cargo, and may require additional installation.

This command is only available on the nightly channel.

For information about usage and installation, see https://github.com/rust-lang/miri.

SEE ALSO

cargo(1), cargo-run(1), cargo-test(1), Custom subcommands

cargo-report(1)

NAME

cargo-report --- Generate and display various kinds of reports

SYNOPSIS

`cargo report` *type* [*options*]

DESCRIPTION

Displays a report of the given *type* --- currently, only `future-incompat` is supported

OPTIONS

`--id` *id*

Show the report with the specified Cargo-generated id

`-p` *spec*…

`--package` *spec*…

Only display a report for the specified package

EXAMPLES

1. Display the latest future-incompat report:

```
cargo report future-incompat
```

2. Display the latest future-incompat report for a specific package:

```
cargo report future-incompat --package my-dep:0.0.1
```

SEE ALSO

Future incompat report

cargo(1)

cargo-run(1)

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.

Unlike cargo-test(1) and cargo-bench(1), `cargo run` sets the working directory of the binary executed to the current working directory, same as if it was executed in the shell directly.

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

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`-r`

`--release`

Run optimized artifacts with the `release` profile. See also the `--profile` option for choosing a specific profile by name.

`--profile` *name*

Run with the given profile. See the reference for more details on profiles.

`--timings=`*fmts*

Output information how long each compilation takes, and track concurrency information over time. Accepts an optional comma-separated list of output formats; `--timings` without an argument will default to `--timings=html`. Specifying an output format (rather than the default) is unstable and requires `-Zunstable-options`. Valid output formats:

- `html` (unstable, requires `-Zunstable-options`): Write a human-readable file `cargo-timing.html` to the `target/cargo-timings` directory with a report of the compilation. Also write a report to the same directory with a timestamp in the filename if you want to look at older runs. HTML output is suitable for human consumption only, and does not provide machine-readable timing data.

- `json` (unstable, requires `-Zunstable-options`): Emit machine-readable JSON information about timing information.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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. Conflicts with `short` and `json`.
- `short`: Emit shorter, human-readable text messages. Conflicts with `human` and `json`.
- `json`: Emit JSON messages to stdout. See the reference for more details. Conflicts with `human` and `short`.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc. Cannot be used with `human` or `short`.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme. Cannot be used with `human` or `short`.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics 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. Cannot be used with `human` or `short`.

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.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

`--keep-going`

Build as many crates in the dependency graph as possible, rather than aborting the build on the first one that fails to build.

For example if the current package depends on dependencies `fails` and `works`, one of which fails to build, `cargo run -j1` may or may not build

the one that succeeds (depending on which one of the two builds Cargo picked to run first), whereas `cargo run -j1 --keep-going` would definitely run both builds, even if the one run first fails.

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(1)

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.

Binary targets are automatically built if there is an integration test or benchmark being selected to build. This allows an integration test to execute the binary to exercise and test its behavior. The `CARGO_BIN_EXE_<name>` environment variable is set when the integration test is built so that it can use the `env` macro to locate the executable.

Passing target selection flags will build only the specified targets.

Note that `--bin`, `--example`, `--test` and `--bench` flags also support common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each glob pattern.

`--lib`

Build the package's library.

`--bin` *name*…

Build the specified binary. This flag may be specified multiple times and supports common Unix glob patterns.

`--bins`

Build all binary targets.

`--example` *name*…

Build the specified example. This flag may be specified multiple times and supports common Unix glob patterns.

`--examples`

Build all example targets.

`--test` *name*…

Build the specified integration test. This flag may be specified multiple times and supports common Unix glob patterns.

`--tests`

Build all targets 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 and supports common Unix glob patterns.

`--benches`

Build all targets 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

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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 flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`-r`

`--release`

Build optimized artifacts with the `release` profile. See also the `--profile` option for choosing a specific profile by name.

`--profile` *name*

Build with the given profile.

The `rustc` subcommand will treat the following named profiles with special behaviors:

- `check` — Builds in the same way as the cargo-check(1) command with the `dev` profile.
- `test` — Builds in the same way as the cargo-test(1) command, enabling building in test mode which will enable tests and enable the `test` cfg option. See rustc tests for more detail.

- `bench` — Builds in the same was as the cargo-bench(1) command, similar to the `test` profile.

See the reference for more details on profiles.

`--timings=`*fmts*

Output information how long each compilation takes, and track concurrency information over time. Accepts an optional comma-separated list of output formats; `--timings` without an argument will default to `--timings=html`. Specifying an output format (rather than the default) is unstable and requires `-Zunstable-options`. Valid output formats:

- `html` (unstable, requires `-Zunstable-options`): Write a human-readable file `cargo-timing.html` to the `target/cargo-timings` directory with a report of the compilation. Also write a report to the same directory with a timestamp in the filename if you want to look at older runs. HTML output is suitable for human consumption only, and does not provide machine-readable timing data.
- `json` (unstable, requires `-Zunstable-options`): Emit machine-readable JSON information about timing information.

`--crate-type` *crate-type*

Build for the given crate type. This flag accepts a comma-separated list of 1 or more crate types, of which the allowed values are the same as `crate-type` field in the manifest for configuring a Cargo target. See `crate-type` field for possible values.

If the manifest contains a list, and `--crate-type` is provided, the command-line argument value will override what is in the manifest.

This flag only works when building a `lib` or `example` library target.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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. Conflicts with `short` and `json`.
- `short`: Emit shorter, human-readable text messages. Conflicts with `human` and `json`.
- `json`: Emit JSON messages to stdout. See the reference for more details. Conflicts with `human` and `short`.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the “short” rendering from rustc. Cannot be used with `human` or `short`.

- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme. Cannot be used with `human` or `short`.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics 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. Cannot be used with `human` or `short`.

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.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear

before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

`--keep-going`

Build as many crates in the dependency graph as possible, rather than aborting the build on the first one that fails to build.

For example if the current package depends on dependencies `fails` and `works`, one of which fails to build, `cargo rustc -j1` may or may not build the one that succeeds (depending on which one of the two builds Cargo picked to run first), whereas `cargo rustc -j1 --keep-going` would definitely run both builds, even if the one run first fails.

`--future-incompat-report`

Displays a future-incompat report for any future-incompatible warnings produced during execution of this command

See cargo-report(1)

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
```

3. Override `crate-type` field in Cargo.toml with command-line option:

```
cargo rustc --lib --crate-type lib,cdylib
```

SEE ALSO

cargo(1), cargo-build(1), rustc(1)

cargo-rustdoc(1)

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` config value.

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 or use the `doc.browser` configuration option.

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.

Note that `--bin`, `--example`, `--test` and `--bench` flags also support common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each glob pattern.

`--lib`

Document the package's library.

`--bin` *name*…

Document the specified binary. This flag may be specified multiple times and supports common Unix glob patterns.

`--bins`

Document all binary targets.

`--example` *name…*

Document the specified example. This flag may be specified multiple times and supports common Unix glob patterns.

`--examples`

Document all example targets.

`--test` *name…*

Document the specified integration test. This flag may be specified multiple times and supports common Unix glob patterns.

`--tests`

Document all targets 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 and supports common Unix glob patterns.

`--benches`

Document all targets 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

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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 flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`-r`

`--release`

Document optimized artifacts with the `release` profile. See also the `--profile` option for choosing a specific profile by name.

`--profile` *name*

Document with the given profile. See the reference for more details on profiles.

`--timings=`*fmts*

Output information how long each compilation takes, and track concurrency information over time. Accepts an optional comma-separated list of output formats; `--timings` without an argument will default to `--`

`timings=html`. Specifying an output format (rather than the default) is unstable and requires `-Zunstable-options`. Valid output formats:

- `html` (unstable, requires `-Zunstable-options`): Write a human-readable file `cargo-timing.html` to the `target/cargo-timings` directory with a report of the compilation. Also write a report to the same directory with a timestamp in the filename if you want to look at older runs. HTML output is suitable for human consumption only, and does not provide machine-readable timing data.
- `json` (unstable, requires `-Zunstable-options`): Emit machine-readable JSON information about timing information.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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. Conflicts with `short` and `json`.
- `short`: Emit shorter, human-readable text messages. Conflicts with `human` and `json`.
- `json`: Emit JSON messages to stdout. See the reference for more details. Conflicts with `human` and `short`.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc. Cannot be used with `human` or `short`.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme. Cannot be used with `human` or `short`.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics 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. Cannot be used with `human` or `short`.

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.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus

provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

`--keep-going`

Build as many crates in the dependency graph as possible, rather than aborting the build on the first one that fails to build.

For example if the current package depends on dependencies `fails` and `works`, one of which fails to build, `cargo rustdoc -j1` may or may not build the one that succeeds (depending on which one of the two builds Cargo picked to run first), whereas `cargo rustdoc -j1 --keep-going` would definitely run both builds, even if the one run first fails.

`--output-format`

The output type for the documentation emitted. Valid values:

- `html` (default): Emit the documentation in HTML format.
- `json`: Emit the documentation in the experimental JSON format.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable.

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(1)

NAME

cargo-test --- Execute unit and integration tests of a package

SYNOPSIS

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

DESCRIPTION

Compile and execute unit, integration, and documentation 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` and check out the rustc book's chapter on how tests work at https://doc.rust-lang.org/rustc/tests/index.html.

As an example, this will filter for tests with `foo` in their name and run them on 3 threads in parallel:

```
cargo test foo -- --test-threads 3
```

Tests are built with the `--test` option to `rustc` which creates a special executable by linking your code with libtest. The executable 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.

If the package contains multiple test targets, each target compiles to a special executable as aforementioned, and then is run serially.

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

Documentation tests are also run by default, which is handled by `rustdoc`. It extracts code samples from documentation comments of the library target, and then executes them.

Different from normal test targets, each code block compiles to a doctest executable on the fly with `rustc`. These executables run in parallel in separate processes. The compilation of a code block is in fact a part of test function controlled by libtest, so some options such as `--jobs` might not

take effect. Note that this execution model of doctests is not guaranteed and may change in the future; beware of depending on it.

See the rustdoc book for more information on writing doc tests.

Working directory of tests

The working directory when running each unit and integration test is set to the root directory of the package the test belongs to. Setting the working directory of tests to the package's root directory makes it possible for tests to reliably access the package's files using relative paths, regardless from where `cargo test` was executed from.

For documentation tests, the working directory when invoking `rustdoc` is set to the workspace root directory, and is also the directory `rustdoc` uses as the compilation directory of each documentation test. The working directory when running each documentation test is set to the root directory of the package the test belongs to, and is controlled via `rustdoc`'s `--test-run-directory` option.

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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

`--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 and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

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, replacing the example's `main` function with the libtest harness. If you don't want the `main` function replaced, also include `harness = false`, in which case the example will be built and executed as-is.

Setting targets to `test = false` will stop them from being tested by default. Target selection options that take a target by name (such as `--example foo`) 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.

See Configuring a target for more information on per-target settings.

Binary targets are automatically built if there is an integration test or benchmark being selected to test. This allows an integration test to execute the binary to exercise and test its behavior. The `CARGO_BIN_EXE_<name>` environment variable is set when the integration test is built so that it can use the `env` macro to locate the executable.

Passing target selection flags will test only the specified targets.

Note that `--bin`, `--example`, `--test` and `--bench` flags also support common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each glob pattern.

`--lib`

Test the package's library.

`--bin` *name*…

Test the specified binary. This flag may be specified multiple times and supports common Unix glob patterns.

`--bins`

Test all binary targets.

`--example` *name*…

Test the specified example. This flag may be specified multiple times and supports common Unix glob patterns.

`--examples`

Test all example targets.

`--test` *name*…

Test the specified integration test. This flag may be specified multiple times and supports common Unix glob patterns.

`--tests`

Test all targets 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 and supports common Unix glob patterns.

`--benches`

Test all targets 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

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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 flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`-r`

`--release`

Test optimized artifacts with the `release` profile. See also the `--profile` option for choosing a specific profile by name.

`--profile` *name*

Test with the given profile. See the reference for more details on profiles.

`--timings=`*fmts*

Output information how long each compilation takes, and track concurrency information over time. Accepts an optional comma-separated list of output formats; `--timings` without an argument will default to `--timings=html`. Specifying an output format (rather than the default) is unstable and requires `-Zunstable-options`. Valid output formats:

- `html` (unstable, requires `-Zunstable-options`): Write a human-readable file `cargo-timing.html` to the `target/cargo-timings` directory with a report of the compilation. Also write a report to the same directory with a timestamp in the filename if you want to look at older runs. HTML output is suitable for human consumption only, and does not provide machine-readable timing data.
- `json` (unstable, requires `-Zunstable-options`): Emit machine-readable JSON information about timing information.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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. Conflicts with `short` and `json`.

- `short`: Emit shorter, human-readable text messages. Conflicts with `human` and `json`.
- `json`: Emit JSON messages to stdout. See the reference for more details. Conflicts with `human` and `short`.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc. Cannot be used with `human` or `short`.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme. Cannot be used with `human` or `short`.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics 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. Cannot be used with `human` or `short`.

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.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file.

This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

`--future-incompat-report`

Displays a future-incompat report for any future-incompatible warnings produced during execution of this command

See cargo-report(1)

While `cargo test` involves compilation, it does not provide a `--keep-going` flag. Use `--no-fail-fast` to run as many tests as possible without stopping at the first failure. To "compile" as many tests as possible, use `--tests` to build test binaries separately. For example:

```
cargo build --tests --keep-going
cargo test --tests --no-fail-fast
```

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 tests whose names match against a filter string:

```
cargo test name_filter
```

3. 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), types of tests, how to write tests

Manifest Commands

cargo-add(1)

NAME

cargo-add --- Add dependencies to a Cargo.toml manifest file

SYNOPSIS

`cargo add` [*options*] *crate*...
`cargo add` [*options*] `--path` *path*
`cargo add` [*options*] `--git` *url* [*crate*...]

DESCRIPTION

This command can add or modify dependencies.

The source for the dependency can be specified with:

- *crate*`@`*version*: Fetch from a registry with a version constraint of "*version*"
- `--path` *path*: Fetch from the specified *path*
- `--git` *url*: Pull from a git repo at *url*

If no source is specified, then a best effort will be made to select one, including:

- Existing dependencies in other tables (like `dev-dependencies`)
- Workspace members
- Latest release in the registry

When you add a package that is already present, the existing entry will be updated with the flags specified.

Upon successful invocation, the enabled (`+`) and disabled (`-`) features of the specified dependency will be listed in the command's output.

OPTIONS

Source options

`--git` *url*

Git URL to add the specified crate from.

`--branch` *branch*

Branch to use when adding from git.

`--tag` *tag*

Tag to use when adding from git.

`--rev` *sha*

Specific commit to use when adding from git.

`--path` *path*

Filesystem path to local crate to add.

`--base` *base*

The path base to use when adding a local crate.

Unstable (nightly-only)

`--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`.

Section options

`--dev`

Add as a development dependency.

`--build`

Add as a build dependency.

`--target` *target*

Add as a dependency to the given target platform.

To avoid unexpected shell expansions, you may use quotes around each target, e.g., `--target 'cfg(unix)'`.

Dependency options

`--dry-run`

Don’t actually write the manifest

`--rename` *name*

Rename the dependency.

`--optional`

Mark the dependency as optional.

`--no-optional`

Mark the dependency as required.

`--public`

Mark the dependency as public.

The dependency can be referenced in your library’s public API.

Unstable (nightly-only)

`--no-public`

Mark the dependency as private.

While you can use the crate in your implementation, it cannot be referenced in your public API.

Unstable (nightly-only)

`--no-default-features`

Disable the default features.

`--default-features`

Re-enable the default features.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. When adding multiple crates, the features for a specific crate may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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.

`-p` *spec*

`--package` *spec*

Add dependencies to only the specified package.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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. Add `regex` as a dependency

```
cargo add regex
```

2. Add `trybuild` as a dev-dependency

```
cargo add --dev trybuild
```

3. Add an older version of `nom` as a dependency

```
cargo add nom@5
```

4. Add support for serializing data structures to json with `derive`s

```
cargo add serde serde_json -F serde/derive
```

5. Add `windows` as a platform specific dependency on `cfg(windows)`

```
cargo add windows --target 'cfg(windows)'
```

SEE ALSO

cargo(1), cargo-remove(1)

cargo-generate-lockfile(1)

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 with the latest available version of every package.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.

- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as

`+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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-info(1)

NAME

cargo-info --- Display information about a package.

SYNOPSIS

`cargo info` [*options*] *spec*

DESCRIPTION

This command displays information about a package. It fetches data from the package's Cargo.toml file and presents it in a human-readable format.

OPTIONS

Info Options

spec
Fetch information about the specified package. The *spec* can be a package ID, see cargo-pkgid(1) for the SPEC format. If the specified package is part of the current workspace, information from the local Cargo.toml file will be displayed. If the `Cargo.lock` file does not exist, it will be created. If no version is specified, the appropriate version will be selected based on the Minimum Supported Rust Version (MSRV).

`--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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as

`+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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. Inspect the `serde` package from crates.io:

```
cargo info serde
```

2. Inspect the `serde` package with version `1.0.0`:

```
cargo info serde@1.0.0
```

3. Inspect the `serde` package form the local registry:

```
cargo info serde --registry my-registry
```

SEE ALSO

cargo(1), cargo-search(1)

cargo-locate-project(1)

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 manifest. The manifest is found by searching upward for a file named `Cargo.toml` starting from the current working directory.

If the project happens to be a part of a workspace, the manifest of the project, rather than the workspace root, is output. This can be overridden by the `--workspace` flag. The root workspace is found by traversing further upward or by using the field `package.workspace` after locating the manifest of a workspace member.

OPTIONS

`--workspace`

Locate the `Cargo.toml` at the root of the workspace, as opposed to the current workspace member.

Display Options

`--message-format` *fmt*

The representation in which to print the project location. Valid values:

- `json` (default): JSON object with the path under the key “root”.
- `plain`: Just the path.

`-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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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(1)

NAME

cargo-metadata --- Machine-readable metadata about the current package

SYNOPSIS

`cargo metadata` [*options*]

DESCRIPTION

Output JSON to stdout containing information about the workspace members and resolved dependencies of the current package.

The output format is subject to change in future versions of Cargo. It is recommended to include the `--format-version` flag to future-proof your code and ensure the output is in the format you are expecting. For more on the expectations, see "Compatibility".

See the cargo_metadata crate for a Rust API for reading the metadata.

OUTPUT FORMAT

Compatibility

Within the same output format version, the compatibility is maintained, except some scenarios. The following is a non-exhaustive list of changes that are not considered as incompatible:

- **Adding new fields** — New fields will be added when needed. Reserving this helps Cargo evolve without bumping the format version too often.
- **Adding new values for enum-like fields** — Same as adding new fields. It keeps metadata evolving without stagnation.
- **Changing opaque representations** — The inner representations of some fields are implementation details. For example, fields related to "Source ID" are treated as opaque identifiers to differentiate packages or sources. Consumers shouldn't rely on those representations unless specified.

JSON format

The JSON 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 for referring to the
                   package within the document and as the `--
package` argument to many commands
```

```
            */
            "id": "file:///path/to/my-package#0.1.0",
             /* 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, an "opaque"
identifier representing
                      where a package is retrieved from. See
"Compatibility" above for
               the stability guarantee.

                      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#5e85ba14aaa20f813
3863373404cb0af69eeef2c"
                - "sparse+URL" for dependencies from a sparse
registry
                            Example: "sparse+https://my-sparse-
registry.org"
```

```
                  The value after the `+` is not explicitly
defined, and may change
                between versions of Cargo and may not directly
correlate to other
                  things, such as registry definitions in a
config file. New source
                 kinds may be added in the future which will
have different `+`
               prefixed identifiers.
            */
            "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)",
                     /* The file system path for a local path
dependency.
                       not present if not a path dependency.
                    */
                    "path": "/path/to/dep",
                       /* 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,
                     /* (unstable) Boolean flag of whether or
not this is a pulbic
                        dependency. This field is only present
when
                       `-Zpublic-dependency` is enabled.
                    */
                    "public": false
                }
```

```
            ],
            /* 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.
                             For lib targets, dashes will be
replaced with underscores.
                    */
                    "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 the target should be documented
by `cargo doc`. */
                    "doc": true,
                        /* Whether or not this target has doc
tests enabled, and
                             the target is compatible with doc
testing.
                    */
                    "doctest": false,
                      /* Whether or not this target should be
built and run with `--test`
                    */
                    "test": true
                }
            ],
            /* 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"
            ],
               /* Optional string that is the default binary
picked by cargo run. */
            "default_run": null,
             /* Optional string that is the minimum supported
```

```
rust version */
            "rust_version": "1.56",
            /* 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 homepage value from the manifest or null if
not specified. */
            "homepage": "https://rust-lang.org",
              /* The documentation value from the manifest or
null if not specified. */
                            "documentation":  "https://doc.rust-
lang.org/stable/std",
            /* 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": [
```

```
        "file:///path/to/my-package#0.1.0",
    ],
    /* Array of default members of the workspace.
       Each entry is the Package ID for the package.
    */
    "workspace_default_members": [
        "file:///path/to/my-package#0.1.0",
    ],
    // The resolved dependency graph for the entire workspace. The enabled
     // features are based on the enabled features for the "current" package.
    // Inactivated optional dependencies are not listed.
    //
    // 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": "file:///path/to/my-package#0.1.0",
                 /* The dependencies of this package, an array of Package IDs. */
                "dependencies": [
                      "https://github.com/rust-lang/crates.io-index#bitflags@1.0.4"
                ],
                 /* 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": "https://github.com/rust-
lang/crates.io-index#bitflags@1.0.4"
                        /* 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 package in the current working directory (if --
manifest-path is not given).
              This is null if there is a virtual workspace.
Otherwise it is
           the Package ID of the package.
        */
        "root": "file:///path/to/my-package#0.1.0",
    },
     /* The absolute path to the target directory where Cargo
places its output. */
    "target_directory": "/path/to/my-package/target",
     /* The absolute path to the build directory where Cargo
places intermediate build artifacts. (unstable) */
    "build_directory": "/path/to/my-package/build-dir",
    /* 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"
    /* Workspace metadata.
       This is null if no metadata is specified. */
    "metadata": {
        "docs": {
            "rs": {
                "all-features": true
```

```
            }
        }
    }
}
```

Notes:

- For `"id"` field syntax, see Package ID Specifications in the reference.

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

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file.

This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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(1), Package ID Specifications, JSON messages

cargo-pkgid(1)

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. This specifier is also used by other parts in Cargo, such as cargo-metadata(1) and JSON messages emitted by Cargo.

The format of a *spec* can be one of the following:

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`

The specification grammar can be found in chapter Package ID Specifications.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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
```

4. Retrieve package specification for `foo` from a local package:

```
cargo pkgid file:///path/to/local/package#foo
```

SEE ALSO

cargo(1), cargo-generate-lockfile(1), cargo-metadata(1), Package ID Specifications, JSON messages

cargo-remove(1)

NAME

cargo-remove --- Remove dependencies from a Cargo.toml manifest file

SYNOPSIS

`cargo remove` [*options*] *dependency...*

DESCRIPTION

Remove one or more dependencies from a `Cargo.toml` manifest.

OPTIONS

Section options

`--dev`

Remove as a development dependency.

`--build`

Remove as a build dependency.

`--target` *target*

Remove as a dependency to the given target platform.

To avoid unexpected shell expansions, you may use quotes around each target, e.g., `--target 'cfg(unix)'`.

Miscellaneous Options

`--dry-run`

Don’t actually write to the manifest.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-`

`lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Package Selection

`-p` *spec*…

`--package` *spec*…

Package to remove from.

Common Options

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 `regex` as a dependency

```
cargo remove regex
```

2. Remove `trybuild` as a dev-dependency

```
cargo remove --dev trybuild
```

3. Remove `nom` from the `x86_64-pc-windows-gnu` dependencies table

```
cargo remove --target x86_64-pc-windows-gnu nom
```

SEE ALSO

cargo(1), cargo-add(1)

cargo-tree(1)

NAME

cargo-tree --- Display a tree visualization of a dependency graph

SYNOPSIS

`cargo tree` [*options*]

DESCRIPTION

This command will display a tree of dependencies to the terminal. An example of a simple project that depends on the "rand" package:

```
myproject v0.1.0 (/myproject)
└── rand v0.7.3
    ├── getrandom v0.1.14
    │   ├── cfg-if v0.1.10
    │   └── libc v0.2.68
    ├── libc v0.2.68 (*)
    ├── rand_chacha v0.2.2
    │   ├── ppv-lite86 v0.2.6
    │   └── rand_core v0.5.1
    │       └── getrandom v0.1.14 (*)
    └── rand_core v0.5.1 (*)
[build-dependencies]
└── cc v1.0.50
```

Packages marked with `(*)` have been "de-duplicated". The dependencies for the package have already been shown elsewhere in the graph, and so are not repeated. Use the `--no-dedupe` option to repeat the duplicates.

The `-e` flag can be used to select the dependency kinds to display. The "features" kind changes the output to display the features enabled by each dependency. For example, `cargo tree -e features`:

```
myproject v0.1.0 (/myproject)
└── log feature "serde"
    └── log v0.4.8
        ├── serde v1.0.106
        └── cfg-if feature "default"
            └── cfg-if v0.1.10
```

In this tree, `myproject` depends on `log` with the `serde` feature. `log` in turn depends on `cfg-if` with "default" features. When using `-e features`

it can be helpful to use `-i` flag to show how the features flow into a package. See the examples below for more detail.

Feature Unification

This command shows a graph much closer to a feature-unified graph Cargo will build, rather than what you list in `Cargo.toml`. For instance, if you specify the same dependency in both `[dependencies]` and `[dev-dependencies]` but with different features on. This command may merge all features and show a `(*)` on one of the dependency to indicate the duplicate.

As a result, for a mostly equivalent overview of what `cargo build` does, `cargo tree -e normal,build` is pretty close; for a mostly equivalent overview of what `cargo test` does, `cargo tree` is pretty close. However, it doesn't guarantee the exact equivalence to what Cargo is going to build, since a compilation is complex and depends on lots of different factors.

To learn more about feature unification, check out this dedicated section.

OPTIONS

Tree Options

`-i` *spec*

`--invert` *spec*

Show the reverse dependencies for the given package. This flag will invert the tree and display the packages that depend on the given package.

Note that in a workspace, by default it will only display the package's reverse dependencies inside the tree of the workspace member in the current directory. The `--workspace` flag can be used to extend it so that it will show the package's reverse dependencies across the entire workspace. The `-p` flag can be used to display the package's reverse dependencies only with the subtree of the package given to `-p`.

`--prune` *spec*

Prune the given package from the display of the dependency tree.

`--depth` *depth*

Maximum display depth of the dependency tree. A depth of 1 displays the direct dependencies, for example.

If the given value is `workspace`, only shows the dependencies that are member of the current workspace, instead.

`--no-dedupe`

Do not de-duplicate repeated dependencies. Usually, when a package has already displayed its dependencies, further occurrences will not re-display its dependencies, and will include a `(*)` to indicate it has already been shown. This flag will cause those duplicates to be repeated.

`-d`

`--duplicates`

Show only dependencies which come in multiple versions (implies `--invert`). When used with the `-p` flag, only shows duplicates within the subtree of the given package.

It can be beneficial for build times and executable sizes to avoid building that same package multiple times. This flag can help identify the offending

packages. You can then investigate if the package that depends on the duplicate with the older version can be updated to the newer version so that only one instance is built.

`-e` *kinds*

`--edges` *kinds*

The dependency kinds to display. Takes a comma separated list of values:

- `all` — Show all edge kinds.
- `normal` — Show normal dependencies.
- `build` — Show build dependencies.
- `dev` — Show development dependencies.
- `features` — Show features enabled by each dependency. If this is the only kind given, then it will automatically include the other dependency kinds.
- `no-normal` — Do not include normal dependencies.
- `no-build` — Do not include build dependencies.
- `no-dev` — Do not include development dependencies.
- `no-proc-macro` — Do not include procedural macro dependencies.

The `normal`, `build`, `dev`, and `all` dependency kinds cannot be mixed with `no-normal`, `no-build`, or `no-dev` dependency kinds.

The default is `normal,build,dev`.

`--target` *triple*

Filter dependencies matching the given target triple. The default is the host platform. Use the value `all` to include *all* targets.

Tree Formatting Options

`--charset` *charset*

Chooses the character set to use for the tree. Valid values are “utf8” or “ascii”. When unspecified, cargo will auto-select a value.

`-f` *format*

`--format` *format*

Set the format string for each package. The default is “{p}”.

This is an arbitrary string which will be used to display each package. The following strings will be replaced with the corresponding value:

- `{p}` — The package name.
- `{l}` — The package license.
- `{r}` — The package repository URL.
- `{f}` — Comma-separated list of package features that are enabled.
- `{lib}` — The name, as used in a `use` statement, of the package's library.

`--prefix` *prefix*

Sets how each line is displayed. The *prefix* value can be one of:

- `indent` (default) — Shows each line indented as a tree.
- `depth` — Show as a list, with the numeric depth printed before each entry.
- `none` — Show as a flat list.

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*…

Display only the specified packages. See cargo-pkgid(1) for the SPEC format. This flag may be specified multiple times and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell

accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

`--workspace`

Display all members in the workspace.

`--exclude` *SPEC…*

Exclude the specified packages. Must be used in conjunction with the `--workspace` flag. This flag may be specified multiple times and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

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.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Feature Selection

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the

project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 tree for the package in the current directory:

```
cargo tree
```

2. Display all the packages that depend on the `syn` package:

```
cargo tree -i syn
```

3. Show the features enabled on each package:

```
cargo tree --format "{p} {f}"
```

4. Show all packages that are built multiple times. This can happen if multiple semver-incompatible versions appear in the tree (like 1.0.0 and 2.0.0).

```
cargo tree -d
```

5. Explain why features are enabled for the `syn` package:

```
cargo tree -e features -i syn
```

   The `-e features` flag is used to show features. The `-i` flag is used to invert the graph so that it displays the packages that depend on `syn`. An example of what this would display:

```
syn v1.0.17
├── syn feature "clone-impls"
│   └── syn feature "default"
│       └── rustversion v1.0.2
│           └── rustversion feature "default"
│               └── myproject v0.1.0 (/myproject)
│                   └── myproject feature "default"
(command-line)
├── syn feature "default" (*)
├── syn feature "derive"
│   └── syn feature "default" (*)
├── syn feature "full"
│   └── rustversion v1.0.2 (*)
├── syn feature "parsing"
```

```
│   └── syn feature "default" (*)
├── syn feature "printing"
│   └── syn feature "default" (*)
├── syn feature "proc-macro"
│   └── syn feature "default" (*)
└── syn feature "quote"
    ├── syn feature "printing" (*)
    └── syn feature "proc-macro" (*)
```

To read this graph, you can follow the chain for each feature from the root to see why it is included. For example, the "full" feature is added by the `rustversion` crate which is included from `myproject` (with the default features), and `myproject` is the package selected on the command-line. All of the other `syn` features are added by the "default" feature ("quote" is added by "printing" and "proc-macro", both of which are default features).

If you're having difficulty cross-referencing the de-duplicated `(*)` entries, try with the `--no-dedupe` flag to get the full output.

SEE ALSO

cargo(1), cargo-metadata(1)

cargo-update(1)

NAME

cargo-update --- Update dependencies as recorded in the local lock file

SYNOPSIS

`cargo update` [*options*] *spec*

DESCRIPTION

This command will update dependencies in the `Cargo.lock` file to the latest version. If the `Cargo.lock` file does not exist, it will be created with the latest available versions.

OPTIONS

Update Options

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 *spec*, 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 *spec* is not specified, all dependencies are updated.

`--recursive`
When used with *spec*, dependencies of *spec* are forced to update as well. Cannot be used with `--precise`.

`--precise` *precise*
When used with *spec*, 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).

While not recommended, you can specify a yanked version of a package. When possible, try other non-yanked SemVer-compatible versions or seek help from the maintainers of the package.

A compatible `pre-release` version can also be specified even when the version requirement in `Cargo.toml` doesn't contain any pre-release identifier (nightly only).

`--breaking` *directory*
Update *spec* to latest SemVer-breaking version.

Version requirements will be modified to allow this update.

This only applies to dependencies when

- The package is a dependency of a workspace member
- The dependency is not renamed
- A SemVer-incompatible version is available

- The “SemVer operator” is used (`^` which is the default)

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/12425 for more information.

`-w`

`--workspace`

Attempt to update only packages defined in the workspace. Other packages are updated only if they don’t already exist in the lockfile. This option is useful for updating `Cargo.lock` after you’ve changed version numbers in `Cargo.toml`.

`--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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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.

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore

existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 foo bar
```

3. Set a specific dependency to a specific version:

```
cargo update foo --precise 1.2.3
```

SEE ALSO

cargo(1), cargo-generate-lockfile(1)

cargo-vendor(1)

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 configuration necessary to use the vendored sources would be printed to stdout after `cargo vendor` completes the vendoring process. You will need to add or redirect it to your Cargo configuration file, which is usually `.cargo/config.toml` locally for the current package.

Cargo treats vendored sources as read-only as it does to registry and git sources. If you intend to modify a crate from a remote source, use `[patch]` or a `path` dependency pointing to a local copy of that crate. Cargo will then correctly handle the crate on incremental rebuilds, as it knows that it is no longer a read-only dependency.

OPTIONS

Vendor Options

`-s` *manifest*

`--sync` *manifest*

Specify an extra `Cargo.toml` manifest to workspaces which should also be vendored and synced to the output. May be specified multiple times.

`--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.toml` read it and use it when downloading crates from crates.io, for example

`--versioned-dirs`

Normally versions are only added to disambiguate multiple versions of the same package. This option causes all directories in the “vendor” directory to be versioned, which makes it easier to track the history of vendored packages over time, and can help with the performance of re-vendoring when only a subset of the packages have changed.

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.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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. 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
```

4. Vendor and redirect the necessary vendor configs to a config file.

```
cargo vendor > path/to/my/cargo/config.toml
```

SEE ALSO

cargo(1)

Package Commands

- cargo init
- cargo install
- cargo new
- cargo search
- cargo uninstall

cargo-init(1)

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).

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 2024. Possible values: 2015, 2018, 2021, 2024

`--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`
Do not print cargo log messages. May also be specified with the `term.quiet` config value.
`--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

+*toolchain*
If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.
`--config` *KEY=VALUE* or *PATH*
Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.
`-C` *PATH*
Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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(1)

NAME

cargo-install --- Build and install a Rust binary

SYNOPSIS

`cargo install` [*options*] *crate*[*@version*]...
`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. By default only binaries, not examples, are installed.

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 source 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 profile (`--profile`).
- 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.

Dealing with the Lockfile

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.

Configuration Discovery

This command operates on system or user level, not project level. This means that the local configuration discovery is ignored. Instead, the configuration discovery begins at `$CARGO_HOME/config.toml`. If the package is installed with `--path $PATH`, the local configuration will be used, beginning discovery at `$PATH/.cargo/config.toml`.

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 from.

`--list`

List all installed packages and their versions.

`-n`

`--dry-run`

(unstable) Perform all checks without installing.

`-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. This is the default behavior.

`--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`.

`--index` *index*

The URL of the registry index to use.

Feature Selection

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`
Activate all available features of all selected packages.
`--no-default-features`
Do not activate the `default` feature of the selected packages.

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.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`--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 a new temporary folder located in the temporary directory of the platform.

When using `--path`, by default it will use `target` directory in the workspace of the local crate unless `--target-dir` is specified.

`--debug`
Build with the `dev` profile instead of the `release` profile. See also the `--profile` option for choosing a specific profile by name.

`--profile` *name*
Install with the given profile. See the reference for more details on profiles.

`--timings=`*fmts*
Output information how long each compilation takes, and track concurrency information over time. Accepts an optional comma-separated list of output formats; `--timings` without an argument will default to `--timings=html`. Specifying an output format (rather than the default) is unstable and requires `-Zunstable-options`. Valid output formats:

- `html` (unstable, requires `-Zunstable-options`): Write a human-readable file `cargo-timing.html` to the `target/cargo-timings` directory with a report of the compilation. Also write a report to the same directory with a timestamp in the filename if you want to look at older runs. HTML output is suitable for human consumption only, and does not provide machine-readable timing data.
- `json` (unstable, requires `-Zunstable-options`): Emit machine-readable JSON information about timing information.

Manifest Options

`--ignore-rust-version`

Ignore `rust-version` specification in packages.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

`--keep-going`

Build as many crates in the dependency graph as possible, rather than aborting the build on the first one that fails to build.

For example if the current package depends on dependencies `fails` and `works`, one of which fails to build, `cargo install -j1` may or may not build the one that succeeds (depending on which one of the two builds Cargo picked to run first), whereas `cargo install -j1 --keep-going` would definitely run both builds, even if the one run first fails.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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. Conflicts with `short` and `json`.
- `short`: Emit shorter, human-readable text messages. Conflicts with `human` and `json`.
- `json`: Emit JSON messages to stdout. See the reference for more details. Conflicts with `human` and `short`.
- `json-diagnostic-short`: Ensure the `rendered` field of JSON messages contains the "short" rendering from rustc. Cannot be used with `human` or `short`.
- `json-diagnostic-rendered-ansi`: Ensure the `rendered` field of JSON messages contains embedded ANSI color codes for respecting rustc's default color scheme. Cannot be used with `human` or `short`.
- `json-render-diagnostics`: Instruct Cargo to not include rustc diagnostics 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. Cannot be used with `human` or `short`.

Common Options

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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(1)

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).

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 2024. Possible values: 2015, 2018, 2021, 2024

`--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`
Do not print cargo log messages. May also be specified with the `term.quiet` config value.
`--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

+*toolchain*
If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.
`--config` *KEY=VALUE* or *PATH*
Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.
`-C` *PATH*
Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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(1)

NAME

cargo-search --- Search packages in the registry. Default registry is 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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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(1)

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

Uninstall 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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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
- cargo logout
- cargo owner
- cargo package
- cargo publish
- cargo yank

cargo-login(1)

NAME

cargo-login --- Log in to a registry

SYNOPSIS

`cargo login` [*options*] [`--` *args*]

DESCRIPTION

This command will run a credential provider to save a token so that commands that require authentication, such as cargo-publish(1), will be automatically authenticated.

All the arguments following the two dashes (`--`) are passed to the credential provider.

For the default `cargo:token` credential provider, the token is saved in `$CARGO_HOME/credentials.toml`. `CARGO_HOME` defaults to `.cargo` in your home directory.

If a registry has a credential-provider specified, it will be used. Otherwise, the providers from the config value `registry.global-credential-providers` will be attempted, starting from the end of the list.

The *token* 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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as

`+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 token for the default registry:

```
cargo login
```

2. Save the token for a specific registry:

```
cargo login --registry my-registry
```

SEE ALSO

cargo(1), cargo-logout(1), cargo-publish(1)

cargo-logout(1)

NAME

cargo-logout --- Remove an API token from the registry locally

SYNOPSIS

`cargo logout` [*options*]

DESCRIPTION

This command will run a credential provider to remove a saved token.

For the default `cargo:token` credential provider, credentials are stored in `$CARGO_HOME/credentials.toml` where `$CARGO_HOME` defaults to `.cargo` in your home directory.

If a registry has a credential-provider specified, it will be used. Otherwise, the providers from the config value `registry.global-credential-providers` will be attempted, starting from the end of the list.

If `--registry` is not specified, then the credentials for the default registry will be removed (configured by `registry.default`, which defaults to https://crates.io/).

This will not revoke the token on the server. If you need to revoke the token, visit the registry website and follow its instructions (see https://crates.io/me to revoke the token for https://crates.io/).

OPTIONS

Logout 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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as

`+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 default registry token:

```
cargo logout
```

2. Remove the token for a specific registry:

```
cargo logout --registry my-registry
```

SEE ALSO

cargo(1), cargo-login(1)

cargo-owner(1)

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

+toolchain

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear

before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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(1)

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 always included. When missing, a new lock file will be generated unless the `--exclude-lockfile` flag is used. 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, as well as a flag if the worktree is dirty.
    - Symlinks are flattened to their target files.
    - Files and directories are included or excluded based on rules mentioned in the `[include]` and `[exclude]` fields.
3. Extract the `.crate` file and build it to verify it can build.
    - This will rebuild your package from scratch to ensure that it can be built from a pristine state. The `--no-verify` flag can be used to skip this step.
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.

.cargo_vcs_info.json format

Will generate a `.cargo_vcs_info.json` in the following format

```
{
 "git": {
   "sha1": "aac20b6e7e543e6dd4118b246c77225e3a3a1302",
   "dirty": true
 },
 "path_in_vcs": ""
}
```

`dirty` indicates that the Git worktree was dirty when the package was built.

`path_in_vcs` will be set to a repo-relative path for packages in subdirectories of the version control repository.

The compatibility of this file is maintained under the same policy as the JSON output of cargo-metadata(1).

Note that this file provides a best-effort snapshot of the VCS information. However, the provenance of the package is not verified. There is no guarantee that the source code in the tarball matches the VCS information.

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.

`--exclude-lockfile`

Don’t include the lock file when packaging.

This flag is not for general use. Some tools may expect a lock file to be present (e.g. `cargo install --locked`). Consider other options before using this.

`--index` *index*

The URL of the registry index to use.

`--registry` *registry*

Name of the registry to package for; see `cargo publish --help` for more details about configuration of registry names. The packages will not be published to this registry, but if we are packaging multiple inter-dependent crates, lock-files will be generated under the assumption that dependencies will be published to this registry.

`--message-format` *fmt*

Specifies the output message format. Currently, it only works with `--list` and affects the file listing format. This is unstable and requires `-Zunstable-options`. Valid output formats:

- `human` (default): Display in a file-per-line format.

- `json`: Emit machine-readable JSON information about each package. One package per JSON line (Newline delimited JSON).

```
{
  /* The Package ID Spec of the package. */
  "id": "path+file:///home/foo#0.0.0",
  /* Files of this package */
  "files" {
    /* Relative path in the archive file. */
    "Cargo.toml.orig": {
      /* Where the file is from.
         - "generate" for file being generated during
packaging
         - "copy" for file being copied from another
location.
      */
      "kind": "copy",
      /* For the "copy" kind,
         it is an absolute path to the actual file
content.
         For the "generate" kind,
         it is the original file the generated one is
based on.
      */
      "path": "/home/foo/Cargo.toml"
    },
    "Cargo.toml": {
      "kind": "generate",
      "path": "/home/foo/Cargo.toml"
    },
    "src/main.rs": {
      "kind": "copy",
      "path": "/home/foo/src/main.rs"
    }
  }
}
```

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*…

Package only the specified packages. See cargo-pkgid(1) for the SPEC format. This flag may be specified multiple times and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

`--workspace`

Package all members in the workspace.

`--exclude` *SPEC*…

Exclude the specified packages. Must be used in conjunction with the `--workspace` flag. This flag may be specified multiple times and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

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 flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`--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

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an

error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

`--keep-going`

Build as many crates in the dependency graph as possible, rather than aborting the build on the first one that fails to build.

For example if the current package depends on dependencies `fails` and `works`, one of which fails to build, `cargo package -j1` may or may not build the one that succeeds (depending on which one of the two builds Cargo picked to run first), whereas `cargo package -j1 --keep-going` would definitely run both builds, even if the one run first fails.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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(1)

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. The server will perform additional checks on the crate.
4. The client will poll waiting for the package to appear in the index, and may timeout. In that case, you will need to check for completion manually. This timeout does not affect the upload.

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 publish to. Registry names are defined in Cargo config files. If not specified, and there is a `package.publish` field in `Cargo.toml` with a single registry, then it will publish to that registry. Otherwise it will use the default registry, which is defined by the `registry.default` config key which defaults to `crates-io`.

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.

Selecting more than one package is unstable and available only on the nightly channel and requires the `-Z package-workspace` flag to enable. See https://github.com/rust-lang/cargo/issues/10948 for more information.

`-p` *spec*…

`--package` *spec*…

Publish only the specified packages. See cargo-pkgid(1) for the SPEC format. This flag may be specified multiple times and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

Selecting more than one package with this option is unstable and available only on the nightly channel and requires the `-Z package-workspace` flag to enable. See https://github.com/rust-lang/cargo/issues/10948 for more information.

`--workspace`

Publish all members in the workspace.

This option is unstable and available only on the nightly channel and requires the `-Z package-workspace` flag to enable. See https://github.com/rust-lang/cargo/issues/10948 for more information.

`--exclude` *SPEC*…

Exclude the specified packages. Must be used in conjunction with the `--workspace` flag. This flag may be specified multiple times and supports common Unix glob patterns like `*`, `?` and `[]`. However, to avoid your shell accidentally expanding glob patterns before Cargo handles them, you must use single quotes or double quotes around each pattern.

This option is unstable and available only on the nightly channel and requires the `-Z package-workspace` flag to enable. See

https://github.com/rust-lang/cargo/issues/10948 for more information.

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 flag may be specified multiple times.

This may also be specified with the `build.target` config value.

Note that specifying this flag makes Cargo run in a different mode where the target artifacts are placed in a separate directory. See the build cache documentation for more details.

`--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

The feature flags allow you to control which features are enabled. When no feature options are given, the `default` feature is activated for every selected package.

See the features documentation for more details.

`-F` *features*

`--features` *features*

Space or comma separated list of features to activate. Features of workspace members may be enabled with `package-name/feature-name` syntax. This flag may be specified multiple times, which enables all specified features.

`--all-features`

Activate all available features of all selected packages.

`--no-default-features`

Do not activate the `default` feature of the selected packages.

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.

`--locked`

Asserts that the exact same dependencies and versions are used as when the existing `Cargo.lock` file was originally generated. Cargo will exit with an error when either of the following scenarios arises:

- The lock file is missing.
- Cargo attempted to change the lock file due to a different dependency resolution.

It may be used in environments where deterministic builds are desired, such as in CI pipelines.

`--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.

`--frozen`

Equivalent to specifying both `--locked` and `--offline`.

`--lockfile-path` *PATH*

Changes the path of the lockfile from the default (`<workspace_root>/Cargo.lock`) to *PATH*. *PATH* must end with `Cargo.lock` (e.g. `--lockfile-path /tmp/temporary-lockfile/Cargo.lock`). Note that providing `--lockfile-path` will ignore existing lockfile at the default path, and instead will either use the lockfile

from *PATH*, or write a new lockfile into the provided *PATH* if it doesn't exist. This flag can be used to run most commands in read-only directories, writing lockfile into the provided *PATH*.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #14421).

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 logical CPUs. If negative, it sets the maximum number of parallel jobs to the number of logical CPUs plus provided value. If a string `default` is provided, it sets the value back to defaults. Should not be 0.

`--keep-going`

Build as many crates in the dependency graph as possible, rather than aborting the build on the first one that fails to build.

For example if the current package depends on dependencies `fails` and `works`, one of which fails to build, `cargo publish -j1` may or may not build the one that succeeds (depending on which one of the two builds Cargo picked to run first), whereas `cargo publish -j1 --keep-going` would definitely run both builds, even if the one run first fails.

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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

+*toolchain*
If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.
`--config` *KEY=VALUE* or *PATH*
Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.
`-C` *PATH*
Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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(1)

NAME

cargo-yank --- Remove a pushed crate from the index

SYNOPSIS

`cargo yank` [*options*] *crate@version*
`cargo yank` [*options*] `--version` *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.

Cargo will not use a yanked version for any new project or checkout without a pre-existing lockfile, and will generate an error if there are no longer any compatible versions for your crate.

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.

How yank works

For example, the `foo` crate published version `1.5.0` and another crate `bar` declared a dependency on version `foo = "1.5"`. Now `foo` releases a new, but not semver compatible, version `2.0.0`, and finds a critical issue with `1.5.0`. If `1.5.0` is yanked, no new project or checkout without an existing lockfile will be able to use crate `bar` as it relies on `1.5`.

In this case, the maintainers of `foo` should first publish a semver compatible version such as `1.5.1` prior to yanking `1.5.0` so that `bar` and all projects that depend on `bar` will continue to work.

As another example, consider a crate `bar` with published versions `1.5.0`, `1.5.1`, `1.5.2`, `2.0.0` and `3.0.0`. The following table identifies the versions cargo could use in the absence of a lockfile for different SemVer requirements, following a given release being yanked:

Yanked Version / SemVer requirement	**`bar = "1.5.0"`**	**`bar = "=1.5.0"`**	**`bar = "2.0.0"`**
`1.5.0`	Use either `1.5.1` or `1.5.2`	**Return Error**	Use `2.0.0`

Yanked Version / SemVer requirement	`bar = "1.5.0"`	`bar = "=1.5.0"`	`bar = "2.0.0"`
`1.5.1`	Use either `1.5.0` or `1.5.2`	Use `1.5.0`	Use `2.0.0`
`2.0.0`	Use either `1.5.0`, `1.5.1` or `1.5.2`	Use `1.5.0`	**Return Error**

When to yank

Crates should only be yanked in exceptional circumstances, for example, an accidental publish, an unintentional SemVer breakages, or a significantly broken and unusable crate. In the case of security vulnerabilities, RustSec is typically a less disruptive mechanism to inform users and encourage them to upgrade, and avoids the possibility of significant downstream disruption irrespective of susceptibility to the vulnerability in question.

A common workflow is to yank a crate having already published a semver compatible version, to reduce the probability of preventing dependent crates from compiling.

When addressing copyright, licensing, or personal data issues with a published crate, simply yanking it may not suffice. In such cases, contact the maintainers of the registry you used. For crates.io, refer to their policies and contact them at help@crates.io.

If credentials have been leaked, the recommended course of action is to revoke them immediately. Once a crate has been published, it is impossible to determine if the leaked credentials have been copied. Yanking the crate only prevents new users from downloading it, but cannot stop those who have already downloaded it from keeping or even spreading the leaked credentials.

OPTIONS

Yank Options

`--vers` *version*

`--version` *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`

Do not print cargo log messages. May also be specified with the `term.quiet` config value.

`--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

+*toolchain*

If Cargo has been installed with rustup, and the first argument to `cargo` begins with `+`, it will be interpreted as a rustup toolchain name (such as `+stable` or `+nightly`). See the rustup documentation for more information about how toolchain overrides work.

`--config` *KEY=VALUE* or *PATH*

Overrides a Cargo configuration value. The argument should be in TOML syntax of `KEY=VALUE`, or provided as a path to an extra configuration file. This flag may be specified multiple times. See the command-line overrides section for more information.

`-C` *PATH*

Changes the current working directory before executing any specified operations. This affects things like where cargo looks by default for the project manifest (`Cargo.toml`), as well as the directories searched for discovering `.cargo/config.toml`, for example. This option must appear before the command name, for example `cargo -C path/to/my-project build`.

This option is only available on the nightly channel and requires the `-Z unstable-options` flag to enable (see #10098).

`-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 foo@1.0.7
```

SEE ALSO

cargo(1), cargo-login(1), cargo-publish(1)

Deprecated and Removed Commands

These commands have been deprecated or removed in early Rust releases. Deprecated commands receive only critical bug fixes, and may be removed in future versions. Removed commands are no longer functional and are unsupported.

- `read-manifest` --- deprecated since Rust 1.13
- `git-checkout` --- removed since Rust 1.44
- `verify-project` --- deprecated since Rust 1.84

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.com 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 you encounter an issue while running on Windows, we consider it a bug, so please file an issue.

Why have `Cargo.lock` in version control?

While `cargo new` defaults to tracking `Cargo.lock` in version control, whether you do is dependent on the needs of your package.

The purpose of a `Cargo.lock` lockfile is to describe the state of the world at the time of a successful build. Cargo uses the lockfile to provide deterministic builds at different times and on different systems, by ensuring that the exact same dependencies and versions are used as when the `Cargo.lock` file was originally generated.

Deterministic builds help with

- Running `git bisect` to find the root cause of a bug
- Ensuring CI only fails due to new commits and not external factors
- Reducing confusion when contributors see different behavior as compared to other contributors or CI

Having this snapshot of dependencies can also help when projects need to be verified against consistent versions of dependencies, like when

- Verifying a minimum-supported Rust version (MSRV) that is less than the latest version of a dependency supports
- Verifying human readable output which won't have compatibility guarantees (e.g. snapshot testing error messages to ensure they are "understandable", a metric too fuzzy to automate)

However, this determinism can give a false sense of security because `Cargo.lock` does not affect the consumers of your package, only `Cargo.toml` does that. For example:

- `cargo install` will select the latest dependencies unless `--locked` is passed in.
- New dependencies, like those added with `cargo add`, will be locked to the latest version

The lockfile can also be a source of merge conflicts.

For strategies to verify newer versions of dependencies via CI, see Verifying Latest Dependencies.

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?

The `--offline` or `--frozen` flags tell Cargo to not touch the network. It returns an error in case it would access the network. You can use `cargo fetch` in one project to download dependencies before going offline, and then use those same dependencies in another project. Refer to configuration value) to set via Cargo configuration.

Vendoring is also related, for more information see documentation on source replacement.

Why is Cargo rebuilding my code?

Cargo is responsible for incrementally compiling crates in your project. This means that if you type `cargo build` twice the second one shouldn't rebuild your crates.io dependencies, for example. Nevertheless bugs arise and Cargo can sometimes rebuild code when you're not expecting it!

We've long wanted to provide better diagnostics about this but unfortunately haven't been able to make progress on that issue in quite some time. In the meantime, however, you can debug a rebuild at least a little by setting the `CARGO_LOG` environment variable:

```
$ CARGO_LOG=cargo::core::compiler::fingerprint=info cargo build
```

This will cause Cargo to print out a lot of information about diagnostics and rebuilding. This can often contain clues as to why your project is getting rebuilt, although you'll often need to connect some dots yourself since this output isn't super easy to read just yet. Note that the `CARGO_LOG` needs to be set for the command that rebuilds when you think it should not. Unfortunately Cargo has no way right now of after-the-fact debugging "why was that rebuilt?"

Some issues we've seen historically which can cause crates to get rebuilt are:

- A build script prints `cargo::rerun-if-changed=foo` where `foo` is a file that doesn't exist and nothing generates it. In this case Cargo will keep running the build script thinking it will generate the file but nothing ever does. The fix is to avoid printing `rerun-if-changed` in this scenario.
- Two successive Cargo builds may differ in the set of features enabled for some dependencies. For example if the first build command builds the whole workspace and the second command builds only one crate, this may cause a dependency on crates.io to have a different set of features enabled, causing it and everything that depends on it to get rebuilt. There's unfortunately not really a great fix for this,

although if possible it's best to have the set of features enabled on a crate constant regardless of what you're building in your workspace.

- Some filesystems exhibit unusual behavior around timestamps. Cargo primarily uses timestamps on files to govern whether rebuilding needs to happen, but if you're using a nonstandard filesystem it may be affecting the timestamps somehow (e.g. truncating them, causing them to drift, etc). In this scenario, feel free to open an issue and we can see if we can accommodate the filesystem somehow.
- A concurrent build process is either deleting artifacts or modifying files. Sometimes you might have a background process that either tries to build or check your project. These background processes might surprisingly delete some build artifacts or touch files (or maybe just by accident), which can cause rebuilds to look spurious! The best fix here would be to wrangle the background process to avoid clashing with your work.

If after trying to debug your issue, however, you're still running into problems then feel free to open an issue!

What does "version conflict" mean and how to resolve it?

failed to select a version for `x` which could resolve this conflict

Have you seen the error message above?

This is one of the most annoying error messages for Cargo users. There are several situations which may lead to a version conflict. Below we'll walk through possible causes and provide diagnostic techniques to help you out there:

- The project and its dependencies use links to repeatedly link the local library. Cargo forbids linking two packages with the same native library, so even with multiple layers of dependencies it is not allowed. In this case, the error message will prompt: `Only one package in the dependency graph may specify the same links value`, you may need to manually check and delete duplicate link values. The community also have conventions in place to alleviate this.
- When depending on different crates in the project, if these crates use the same dependent library, but the version used is restricted, making it impossible to determine the correct version, it will also cause conflicts. The error message will prompt: `all possible versions conflict with previously selected packages`. You may need to modify the version requirements to make them consistent.
- If there are multiple versions of dependencies in the project, when using `direct-minimal-versions`, the minimum version requirements cannot be met, which will cause conflicts. You may need to modify version requirements of your direct dependencies to meet the minimum SemVer version accordingly.
- If the dependent crate does not have the features you choose, it will also cause conflicts. At this time, you need to check the dependent version and its features.
- Conflicts may occur when merging branches or PRs, if there are non-trivial conflicts, you can reset all "yours" changes, fix all other

conflicts in the branch, and then run some cargo command (like `cargo tree` or `cargo check`), which should re-update the lockfile with your own local changes. If you previously ran some `cargo update` commands in your branch, you can re-run them that this time. The community has been looking to resolve merge conflicts with `Cargo.lock` and `Cargo.toml` using a custom merge tool.

Changelog

Cargo 1.90 (2025-09-18)

[c24e1064...HEAD](#)

Added

Changed

Fixed

- Expanded error messages around path dependency on `cargo package` and `cargo publish` #15705

Nightly only

- feat(toml): Parse support for multiple build scripts #15630

Documentation

Internal

Cargo 1.89 (2025-08-07)

873a0649...rust-1.89.0

Added

- Add `*` and `?` pattern support for SSH known hosts matching. #15508
- Stabilize doctest-xcompile. Doctests will now automatically be tested when cross-compiling to a target that is different from the host, just like other tests. #15462

Changed

- ❗ `cargo fix` and `cargo clippy --fix` now run only on the default Cargo targets by default, matching the behavior of `cargo check`. To run on all Cargo targets, use the `--all-targets` flag. This change aligns the behavior with other commands. Edition flags like `--edition` and `--edition-idioms` remain implying `--all-targets` by default. #15192
- Respect `Retry-After` header for HTTP 429 responses when talking to registries. #15463
- Improved error message for the `CRATE[@<VER>]` argument prefixed with `v`. #15484
- Improved error message for the `CRATE[@<VER>]` argument with invalid package name characters. #15441
- cargo-add: suggest similarly named features #15438

Fixed

- Fixed potential deadlock in `CacheState::lock` #15698 #15699
- Fixed the `--manifest-path` arg being ignored in `cargo fix` #15633
- When publishing, don't tell people to ctrl-c without knowing consequences. #15632
- Added missing `--offline` in shell completions. #15623
- cargo-credential-libsecret: load libsecret only once #15295

- When failing to find the mtime of a file for rebuild detection, report an explicit reason rather than "stale; unknown reason". #15617
- Fixed cargo add overwriting symlinked Cargo.toml files #15281
- Vendor files with .rej/.orig suffix #15569
- Vendor using direct extraction for registry sources. This should ensure that vendored files now always match the originals. #15514
- In the network retry message, use singular "try" for the last retry. #15328

Nightly only

- 🔥 `-Zno-embed-metadata`: This tells Cargo to pass the `-Zembed-metadata=no` flag to the compiler, which instructs it not to embed metadata within rlib and dylib artifacts. In this case, the metadata will only be stored in `.rmeta` files. (docs) #15378
- 🔥 Plumb rustc `-Zhint-mostly-unused` flag through as a profile option (docs) #15643
- Added the "future" edition #15595
- Added `-Zfix-edition` #15596
- Added perma unstable `--compile-time-deps` option for `cargo build` #15674
- `-Zscript`: Make cargo script ignore workspaces. #15496
- `-Zpackage-workspace`: keep dev-dependencies if they have a version. #15470
- Added custom completer for `cargo remove <TAB>` #15662
- Test improvements in prep for `-Zpackage-workspace` stabilization #15628
- Allow packaging of self-cycles with `-Zpackage-workspace` #15626
- With trim-paths, remap all paths to `build.build-dir` #15614
- Enable more trim-paths tests for windows-msvc #15621
- Fix doc rebuild detection by passing `toolchain-shared-resources` to get doc styled for rustdoc-depinfo tracking #15605
- Resolve multiple bugs in frontmatter parser for `-Zscript` #15573

- Remove workaround for rustc frontmatter support for `-Zscript` #15570
- Allow configuring arbitrary codegen backends #15562
- skip `publish=false` pkg when publishing entire workspace for `-Zpackage-workspace`. #15525
- Update instructions on using native-completions #15480
- Skip registry check if its not needed with `-Zpackage-workspace`. #15629

Documentation

- Clarify what commands need and remove confusing example #15457
- Update fingerprint footnote #15478
- home: update version notice for deprecation removal #15511
- docs(contrib): change clap URL to docs.rs/clap #15682
- Update links in contrib docs #15659
- docs: clarify `--all-features` not available for all commmands #15572
- docs(README): fix the link to the changelog in the Cargo book #15597

Internal

- Refactor artifact deps in FeatureResolver::deps #15492
- Added tracing spans for rustc invocations #15464
- ci: migrate renovate config #15501
- ci: Require schema job to pass #15504
- test: Remove unused nightly requirements #15498
- Update dependencies. #15456
- refactor: replace InternedString with Cow in IndexPackage #15559
- Use `Not::not` rather than a custom `is_false` function #15645
- fix: Make UI tests handle hyperlinks consistently #15640
- Update dependencies #15635 #15557
- refactor: clean up `clippy::perf` lint warnings #15631
- chore(deps): update alpine docker tag to v3.22 #15616

- chore: remove HTML comments in PR template and inline guide #15613
- Added .git-blame-ignore-revs #15612
- refactor: cleanup for `CompileMode` #15608
- refactor: separate "global" mode from CompileMode #15601
- chore: Upgrade schemars #15602
- Update gix & socket2 #15600
- chore(toml): disable `toml`'s default features, unless necessary, to reduce cargo-util-schemas build time #15598
- chore(gh): Add new-lint issue template #15575
- Fix comment for cargo/core/compiler/fingerprint/mod.rs #15565

Cargo 1.88 (2025-06-26)

a6c604d1...rust-1.88.0

Added

- 🎉 Stabilize automatic garbage collection for global caches.

  When building, Cargo downloads and caches crates needed as dependencies. Historically, these downloaded files would never be cleaned up, leading to an unbounded amount of disk usage in Cargo's home directory. In this version, Cargo introduces a garbage collection mechanism to automatically clean up old files (e.g. .crate files). Cargo will remove files downloaded from the network if not accessed in 3 months, and files obtained from the local system if not accessed in 1 month. Note that this automatic garbage collection will not take place if running offline (using `--offline` or `--frozen`).

  Cargo 1.78 and newer track the access information needed for this garbage collection. If you regularly use versions of Cargo older than 1.78, in addition to running current versions of Cargo, and you expect to have some crates accessed exclusively by the older versions of Cargo and don't want to re-download those crates every ~3 months, you may wish to set `cache.auto-clean-frequency = "never"` in the Cargo configuration. (docs) #14287
- Allow boolean literals as cfg predicates in Cargo.toml and configurations. For example, `[target.'cfg(not(false))'.dependencies]` is a valid cfg predicate. (RFC 3695) #14649

Changed

- Don't canonicalize executable path for the `CARGO` environment variable. #15355
- Print target and package names formatted as file hyperlinks. #15405
- Make sure library search paths inside `OUT_DIR` precede external paths. #15221

- Suggest similar looking feature names when feature is missing. #15454
- Use `zlib-rs` for gzip (de)compression for `.crate` tarballs. #15417

Fixed

- build-rs: Correct name of `CARGO_CFG_FEATURE` #15420
- cargo-tree: Make output more deterministic #15369
- cargo-package: dont fail the entire command when the dirtiness check failed, as git status check is mostly informational. #15416 #15419
- Fixed `cargo rustc --bin` panicking on unknown bin names #15515 #15497

Nightly only

- 🔥 `-Zrustdoc-depinfo`: A new unstable flag leveraging rustdoc's dep-info files to determine whether documentations are required to re-generate. (docs) #15359 #15371
- `build-dir`: Added validation for unmatched brackets in build-dir template #15414
- `build-dir`: Improved error message when build-dir template var is invalid #15418
- `build-dir`: Added `build_directory` field to cargo metadata output #15377
- `build-dir`: Added symlink resolution for `workspace-path-hash` #15400
- `build-dir`: Added build_directory to cargo metadata documentation #15410
- `unit-graph`: switch to Package ID Spec. #15447
- `-Zgc`: Rename the `gc` config table to `[cache]`. Low-level settings is now under `[cache.global-clean]`. #15367
- `-Zdoctest-xcompile`: Update doctest xcompile flags. #15455

Documentation

- Mention the convention of kebab-case for Cargo targets naming. #14439
- Use better example value in `CARGO_CFG_TARGET_ABI` #15404

Internal

- Fix formatting of CliUnstable parsing #15434
- ci: restore semver-checks for cargo-util #15389
- ci: add aarch64 linux runner #15077
- rustfix: Use `snapbox` for snapshot testing #15429
- test:Prevent undeclared public network access #15368
- Update dependencies. #15373 #15381 #15391 #15394 #15403 #15415 #15421 #15446

Cargo 1.87 (2025-05-15)

ce948f46...rust-1.87.0

Added

- Add terminal integration via ANSI OSC 9;4 sequences via the `term.progress.term-integration` configuration field. This reports progress to the terminal emulator for display in places like the task bar. (docs) #14615
- Forward bash completions of third party subcommands #15247
- cargo-tree: Color the output. #15242
- cargo-package: add `--exclude-lockfile` flag, which will stop verifying the lock file if present. #15234

Changed

- ❗ Cargo now depends on OpenSSL v3. This implies that Cargo in the official Rust distribution will have a hard dependency on libatomic on 32-bit platforms. #15232
- Report `<target>.edition` deprecation to users. #15321
- Leverage clap for providing default values for `--vcs`, `--color`, and `--message-format` flags. #15322
- Mention "3" as a valid value for "resolver" field in error message #15215
- Uplift windows Cygwin DLL import libraries #15193
- Include the package name also in the target hint message. #15199
- cargo-add: collapse large feature lists #15200
- cargo-vendor: Add context which workspace failed to resolve #15297

Fixed

- Do not pass cdylib link args from `cargo::rustc-link-arg-cdylib` to tests. #15317 #15326
- Don't use `$CARGO_BUILD_TARGET` in `cargo metadata`. #15271

- Allow `term.progress.when` to have default values. `CARGO_TERM_PROGRESS_WIDTH` can now be correctly set even when other settings are missing. #15287
- Fix the `CARGO` environment variable setting for external subcommands pointing to the wrong Cargo binary path . Note that the environment variable is never designed as a general Cargo wrapper. #15208
- Fix some issues with future-incompat report generation. #15345
- Respect `--frozen` everywhere `--offline` or `--locked` is accepted. #15263
- cargo-package: report also the VCS status of the workspace manifest if dirty. #15276 #15341
- cargo-publish: Fix man page with malformed `{{#options}}` block #15191
- cargo-run: Disambiguate bins from different packages that share a name. #15298
- cargo-rustc: de-duplicate crate types. #15314
- cargo-vendor: dont remove non-cached sources. #15260

Nightly only

- 🔥 cargo-package: add unstable `--message-format` flag. The flag is providing an alternative JSON output format for file listing of the `--list` flag. (docs) #15311 #15354
- 🔥 `build-dir`: the `build.build-dir` config option to set the directory where intermediate build artifacts will be stored. Intermediate artifacts are produced by Rustc/Cargo during the build process. (docs) #15104 #15236 #15334
- 🔥 `-Zsbom`: The `build.sbom` config allows to generate so-called SBOM pre-cursor files alongside each compiled artifact. (RFC 3553) (docs) #13709
- 🔥 `-Zpublic-dependency`: new `--depth public` value for `cargo tree` to display public dependencies. #15243
- `-Zscript`: Handle more frontmatter parsing corner cases #15187

- `-Zpackage-workspace`: Fix lookups to capitalized workspace member's index entry #15216
- `-Zpackage-workspace`: Register workspace member renames in overlay #15228
- `-Zpackage-workspace`: Ensure we can package directories ending with '.rs' #15240
- `native-completions`: add completions for `--profile` #15308
- `native-completions`: add completions for aliases #15319
- `native-completions`: add completions for `cargo add --path` #15288
- `native-completions`: add completions for `--manifest-path` #15225
- `native-completions`: add completions for `--lockfile-path` #15238
- `native-completions`: add completions for `cargo install --path` #15266
- `native-completions`: add completions fro `+<toolchain>` #15301

Documentation

- Note that target-edition is deprecated #15292
- Mention wrong URLs as a cause of git authentication errors #15304
- Shift focus to resolver v3 #15213
- Lockfile is always included since 1.84 #15257
- Remove `Cargo.toml` from `package.include` in example #15253
- Make it clearer that `rust_version` is enforced during compile #15303
- Fix `[env]` `relative` description in reference #15332
- Add `unsafe` to `extern` while using build scripts in Cargo Book #15294
- Mention `x.y.*` as a kind of version requirement to avoid. #15310
- contrib: Expand the description of team meetings #15349

Internal

- Show extra build description from bootstrap via the `CFG_VER_DESCRIPTION` env var. #15269

- Control byte display precision with `std::fmt` options. #15246
- Replace humantime crate with jiff. #15290
- Dont check cargo-util semver until 1.86 is released #15222
- Redox OS is part of the unix family #15307
- cargo-tree: Abstract the concept of a NodeId #15237
- cargo-tree: Abstract the concept of an edge #15233
- ci: Auto-update cargo-semver-checks #15212
- ci: Visually group output in Github #15218
- manifest: Centralize Cargo target descriptions #15291
- Update dependencies. #15250 #15249 #15245 #15224 #15282 #15211 #15217 #15268

Cargo 1.86 (2025-04-03)

d73d2caf...rust-1.86.0

Added

Changed

- ❗ When merging, replace rather than combine configuration keys that refer to a program path and its arguments. #15066
  These keys include:
  - `registry.credential-provider`
  - `registries.*.credential-provider`
  - `target.*.runner`
  - `host.runner`
  - `credential-alias.*`
  - `doc.browser`
- ❗ Error if both `--package` and `--workspace` are passed but the requested package is missing. This was previously silently ignored, which was considered a bug since missing packages should be reported. #15071
- Added warning when failing to update index cache. #15014
- Don't use "did you mean" in errors. Be upfront about what the suggestion is. #15138
- Provide a better error message for invalid SSH URLs in dependency sources. #15185
- Suggest similar feature names when the package doesn't have given features. #15133
- Print globs when workspace members can't be found. #15093
- cargo-fix: Make `--allow-dirty` imply `--allow-staged` #15013
- cargo-login: hide the `token` argument from CLI help for the preparation of deprecation. #15057
- cargo-login: Don't suggest `cargo login` when using incompatible credential providers. #15124

- cargo-package: improve the performance of VCS status check by matching certain path prefixes with pathspec. #14997

Fixed

- The `rerun-if-env-changed` build script instruction can now correctly detect changes in the `[env]` configuration table. #14756
- Force emitting warnings as warnings when learning Rust target info for an unsupported crate type. #15036
- cargo-package: Verify the VCS status of symlinks when they point to paths outside the current package root. #14981

Nightly only

- 🔥 `-Z feature-unification`: This new unstable flag enables the `resolver.feature-unification` configuration option to control how features are unified across a workspace. (RFC 3529) (docs) #15157
- cargo-util-schemas: Correct and update the JSON Schema #15000
- cargo-util-schemas: Fix the `[lints]` JSON Schema #15035
- cargo-util-schemas: Fix 'metadata' JSON Schema #15033
- `cargo rustc --print`: Setup cargo environment for `cargo rustc --print`. #15026
- `-Zbuild-std`: parse value as comma-separated list, also extends the behavior to `build-std-features`. #15065
- `-Zgc`: Make cache tracking resilient to unexpected files. #15147
- `-Zscript`: Consolidate creation of SourceId from manifest path #15172
- `-Zscript`: Integrate cargo-script logic into main parser #15168
- `-Zscript`: add `cargo pkgid` support for cargo-script #14961
- `-Zpackage-workspace`: Report all unpublishable packages #15070

Documentation

- Document that Cargo automatically registers variables used in the `env!` macro to trigger rebuilds since 1.46. #15062

- Move the changelog to The Cargo Book. #15119 #15123 #15142
- Note `package.authors` is deprecated. #15068
- Fix the wrong grammar of a Package Id Specification. #15049
- Fix the inverted logic about MSRV #15044
- cargo-metadata: Fix description of the `"root"` field. #15182
- cargo-package: note the lock file is always included. #15067
- contrib: Start guidelines for schema design. #15037

Internal

- Don't use `libc::LOCK_*` on Solaris. #15143
- Clean up field -> env var handling. #15008
- Simplify SourceID Ord/Eq. #14980 #15103
- Add manual Hash impl for SourceKind and document the reason. #15029
- ci: allow Windows reserved names in CI #15135
- cargo-test-macro: Remove condition on `RUSTUP_WINDOWS_PATH_ADD_BIN` #15017
- resolver: Simplify backtrack #15150
- resolver: Small cleanups #15040
- test: Clean up shallow fetch tests #15002
- test: Fix `https::self_signed_should_fail` for macOS #15016
- test: Fix benchsuite issue with newer versions of git #15069
- test: Fix shared_std_dependency_rebuild running on Windows #15111
- test: Update tests to fix nightly errors #15110
- test: Remove unused `-C link-arg=-fuse-ld=lld` #15097
- test: Remove `unsafe` by using `LazyLock` #15096
- test: Remove unnecessary into conversions #15042
- test: Fix race condition in panic_abort_tests #15169
- Update deny.toml #15164
- Update dependencies. #14995 #14996 #14998 #15012 #15018 #15041 #15050 #15121 #15128 #15129 #15162 #15163 #15165 #15166

Cargo 1.85 (2025-02-20)

66221abd...rust-1.85.0

Added

- 🎉 Cargo now supports the 2024 edition. More information is available in the edition guide. #14828
- cargo-tree: The `--depth` flag now accepts `workspace`, which shows only dependencies that are members of the current workspace. #14928
- Build scripts now receive a new environment variable, `CARGO_CFG_FEATURE`, which contains each activated feature of the package being built. #14902
- perf: Dependency resolution is now faster due to a more efficient hash for `ActivationsKey` #14915

Changed

- ❗ cargo-rustc: Trailing flags now have higher precedence. This behavior was nightly-only since 1.83 and is now stabilized. #14900
- ❗ Cargo now uses a cross-platform hash algorithm from `rustc-stable-hash`. As a result, the hash part of paths to dependency caches (e.g., `$CARGO_HOME/registry/index/index.crates.io-<hash>`) will change. This will trigger re-downloads of registry indices and `.crate` tarballs, as well as re-cloning of Git dependencies. #14917
- Added a future-incompatibility warning for keywords in `cfg`s in Cargo.toml and Cargo configuration. `cfg`s with keywords like `cfg(true)` and `cfg(false)` were incorrectly accepted. For backward compatibility, support for raw identifiers has been introduced; for example, use `cfg(r#true)` instead. #14671
- Dependency resolution now provides richer error messages explaining why some versions were rejected, unmatched, or invalid. #14897 #14921 #14923 #14927
- cargo-doc: improve the error message when `--open`ing a doc while no doc generated. #14969

- cargo-package: warn if symlinks checked out as plain text files #14994
- cargo-package: Shows dirty file paths relative to the Git working directory. #14968 #14970

Fixed

- Set `GIT_DIR` to ensure compatibility with bare repositories for `net.git-fetch-with-cli=true`. #14860
- Fixed workspace Cargo.toml modification didn't invalidate build cache. #14973
- Prevented build caches from being discarded after changes to `RUSTFLAGS`. #14830 #14898
- cargo-add: Don't select yanked versions when normalizing names. #14895
- cargo-fix: Migrate workspace dependencies to the 2024 edition also for virtual manifests. #14890
- cargo-package: Verify the VCS status of `package.readme` and `package.license-file` when they point to paths outside the current package root. #14966
- cargo-package: assure possibly blocking non-files (like FIFOs) won't be picked up for publishing. #14977

Nightly only

- `path-bases`: Support bases in `[patch]`es in virtual manifests #14931
- `unit-graph`: Use the configured shell to print output. #14926
- `-Zbuild-std`: Check if the build target supports `std` by probing the `metadata.std` field in the target spec JSON. #14183 #14938 #14899
- `-Zbuild-std`: always link to std when testing proc-macros. #14850 #14861
- `-Zbuild-std`: clean up build-std tests #14943 #14933 #14896
- `-Zbuild-std`: Hash relative paths to std workspace instead of absolute paths. #14951
- `-Zpackage-workspace`: Allow dry-run of a non-bumped workspace. #14847

- `-Zscript`: Allow adding/removing dependencies from cargo scripts #14857
- `-Zscript`: Migrate cargo script manifests across editions #14864
- `-Zscript`: Don't override the release profile. #14925
- `-Ztrim-paths`: Use `Path::push` to construct the `remap-path-prefix` flag. #14908

Documentation

- Clarify how `cargo::metadata` env var is selected. #14842
- cargo-info: Remove references to the default registry in `cargo-info` docs #14880
- contrib: add missing argument to Rustup Cargo workaround #14954
- SemVer: Add section on RPIT capturing #14849

Internal

- Add the `test` cfg as a well known cfg before of compiler change. #14963
- Enable triagebot merge conflict notifications #14972
- Limit release trigger to `0.*` tags #14940
- Simplify `SourceID` Hash. #14800
- build-rs: Automatically emits `rerun-if-env-changed` when accessing environment variables Cargo sets for build script executions. #14911
- build-rs: Correctly refer to the item in assert #14913
- build-rs: Add the 'error' directive #14910
- build-rs: Remove meaningless 'cargo_cfg_debug_assertions' #14901
- cargo-package: split `cargo_package` to modules #14959 #14982
- cargo-test-support: `requires` attribute accepts string literals for cmds #14875
- cargo-test-support: Switch from 'exec_with_output' to 'run' #14848
- cargo-test-support: track caller for `.crate` file publish verification #14992

- test: Verify `-Cmetadata` directly, not through `-Cextra-filename` #14846
- test: ensure PGO works #14859 #14874 #14887
- Update dependencies. #14867 #14871 #14878 #14879 #14975

Cargo 1.84 (2025-01-09)

15fbd2f6...rust-1.84.0

Added

- 🎉 Stabilize resolver v3, a.k.a the MSRV-aware dependency resolver. The stabilization includes `package.resolver = "3"` in Cargo.toml, and the `[resolver]` table in Cargo configuration. (RFC 3537) (manifest docs) (config docs) #14639 #14662 #14711 #14725 #14748 #14753 #14754
- Added a new build script invocation `cargo::error=MESSAGE` to report error messages. (docs) #14743

Changed

- ❗ cargo-publish: Always include Cargo.lock in published crates. Originally it was only included for packages that have executables or examples for use with `cargo install`. #14815
- Dependency resolver performance improvements, including shared caching, reduced iteration overhead, and removing redundant fetches and clones. #14663 #14690 #14692 #14694
- Deprecate `cargo verify-project`. #14736
- Add source replacement info when no matching package found during dependency resolving. #14715
- Hint for using `crates-io` when `[patch.crates.io]` found. #14700
- Normalize source paths of Cargo targets for better diagnostics. #14497 #14750
- Allow registries to omit empty/default fields in index metadata JSON. Due to backward compatibility, crates.io continues to emit them. #14838 #14839
- cargo-doc: display env vars in extra verbose mode. #14812
- cargo-fix: replace special-case handling of duplicate insert-only replacement. #14765 #14782

- cargo-remove: when a dependency is not found, try suggesting other dependencies with similar names. #14818
- git: skip unnecessary submodule validations for fresh checkouts on Git dependencies. #14605
- git: Enhanced the error message for fetching Git dependencies when refspec not found. #14806
- git: Pass `--no-tags` by default to git CLI when `net.git-fetch-with-cli = true`. #14688

Fixed

- Fixed old Cargos failing to read the newer format of dep-info in build caches. #14751 #14745
- Fixed rebuild detection not respecting changes in the `[env]` table. #14701 #14730
- cargo-fix: Added transactional semantics to `rustfix` to keep code fix in a valid state when multiple suggestions contain overlapping spans. #14747

Nightly only

- The unstable environment variable `CARGO_RUSTC_CURRENT_DIR` has been removed. #14799
- 🔥 Cargo now includes an experimental JSON Schema file for `Cargo.toml` in the source code. It helps external tools validate or auto-complete the schema of the manifest. (manifest.schema.json) #14683
- 🔥 `Zroot-dir`: A new unstable `-Zroot-dir` flag to configure the path from which rustc should be invoked. (docs) #14752
- 🔥 `-Zwarnings`: A new unstable feature to control how Cargo handles warnings via the `build.warnings` configuration field. (docs) #14388 #14827 #14836
- `edition2024`: Verify 2024 edition / resolver=3 doesn't affect resolution #14724
- `native-completions`: Include descriptions in zsh #14726

- `-Zbindeps`: Fix panic when running cargo tree on a package with a cross compiled bindep #14593
- `-Zbindeps`: download targeted transitive deps of with artifact deps' target platform #14723
- `-Zbuild-std`: Remove the requirement for `--target`. #14317
- `-Zpackage-workspace`: Support package selection options, such as `--exclude`, in `cargo publish` #14659
- `-Zscript`: Remove support for accepting `Cargo.toml`. #14670
- `-Zscript`: Change config paths to only check `CARGO_HOME` #14749
- `-Zscript`: Update the frontmatter parser for RFC 3503. #14792

Documentation

- Clarify the meaning of `--tests` and `--benches` flags. #14675
- Clarify tools should only interpret messages with a line starting with `{` as JSON. #14677
- Clarify what is and isn't included by `cargo package` #14684
- Document official external commands: `cargo-clippy`, `cargo-fmt`, and `cargo-miri`. #14669 #14805
- Enhance documentation on environment variables #14676
- Simplify English used in documentations. #14825 #14829
- A new doc page for deprecated and removed commands. #14739
- cargo-test-support: Document `Execs` assertions based on port effort #14793

Internal

- 🎉 Migrate `build-rs` crate to the `rust-lang/cargo` repository as an intentional artifact of the Cargo team. #14786 #14817
- Enable transfer feature in triagebot #14777
- clone-on-write when needed for InternedString #14808
- ci: Switch CI from bors to merge queue #14718
- ci: make the `lint-docs` job required #14797
- ci: Check for clippy `correctness` #14796

- ci: Switch matchPackageNames to matchDepNames for renovate #14704
- fingerprint: Track the intent for each use of `UnitHash` #14826
- fingerprint: Add more metadata to `rustc_fingerprint`. #14761
- test: Migrate remaining snapshotting to snapbox #14642 #14760 #14781 #14785 #14790
- Update dependencies. #14668 #14705 #14762 #14766 #14772

Cargo 1.83 (2024-11-28)

8f40fc59...rust-1.83.0

Added

- `--timings` HTML output can now auto-switch between light and dark color schemes based on browser preference. #14588
- Introduced a new `CARGO_MANIFEST_PATH` environment variable, similar to `CARGO_MANIFEST_DIR` but pointing directly to the manifest file. #14404
- manifest: Added `package.autolib`, allowing `[lib]` auto-discovery to be disabled. #14591

Changed

- ❗ Lockfile format v4 is now the default for creating/updating a lockfile. Rust toolchains 1.78+ support lockfile v4. For compatibility with earlier MSRV, consider setting the `package.rust-version` to 1.82 or earlier. #14595
- ❗ cargo-package: When using the `--package` flag, only the specified packages are packaged. Previously, the package in the current working directory was automatically selected for packaging. #14488
- cargo-publish: Now fails fast if the package version is already published. #14448
- Improved error messages for missing features. #14436
- Log details of `rustc` invocation failure if no errors are seen #14453
- Uplifted `windows-gnullvm` import libraries, aligning them with `windows-gnu`. #14451
- Suggest `cargo info` command in the `cargo search` result #14537
- Enhanced dependency update status messages, now displaying updates (compatible, incompatible, direct-dep) in different colors, along with messages and MSRVs. #14440 #14457 #14459 #14461 #14471 #14568

- The `Locking` status message no longer displays workspace members. #14445

Fixed

- Prevented duplicate library search environment variables when calling `cargo` recursively. #14464
- Don't double-warn about `$CARGO_HOME/config` not having `.toml` extension. #14579
- Correct diagnostic count message when using `--message-format json`. #14598
- cargo-add: Perform fuzzy searches when translating package names #13765
- cargo-new: only auto-add new packages to the workspace relative to the manifest, rather than the current directory. #14505
- cargo-rustc: Fixed parsing of comma-separated values in the `--crate-type` flag. #14499
- cargo-vendor: trusts the crate version only when it originates from registries. This causes git dependencies to be re-vendored even if they haven't changed. #14530
- cargo-publish: Downgrade version-exists error to warning on dry-run #14742 #14744

Nightly only

- ❗ cargo-rustc: give trailing flags higher precedence on nightly. The nightly gate will be removed after a few releases. Please give feedback if it breaks any workflow. A temporary environment variable `__CARGO_RUSTC_ORIG_ARGS_PRIO=1` is provided to opt-out of the behavior. #14587
- 🔥 cargo-install: a new `--dry-run` flag without actually installing binaries. #14280
- 🔥 `native-completions`: moves the handwritten shell completion scripts to Rust native, making it easier for us to add, extend, and test new completions. (docs) #14493 #14531 #14532 #14533 #14534

#14535 #14536 #14546 #14547 #14548 #14552 #14557 #14558 #14563 #14564 #14573 #14590 #14592 #14653 #14656

- 🔥 `-Zchecksum-freshness`: replace the use of file mtimes in cargo's rebuild detection with a file checksum algorithm. This is most useful on systems with a poor mtime implementation, or in CI/CD. (docs) #14137
- cargo-update: Add `matches_prerelease` semantic #14305
- `build-plan`: document it as being deprecated. #14657
- `edition2024`: Remove implicit feature removal from 2024 edition. #14630
- `lockfile-path`: implies `--locked` on `cargo install`. #14556
- `open-namespaces`: Allow open namespaces in `PackageIdSpec`s #14467
- `path-bases`: `cargo [add|remove|update]` support #14427
- `-Zmsrv-policy`: determine the workspace's MSRV by the most number of MSRVs within it. #14569
- `-Zpackage-workspace`: allows to publish multiple crates in a workspace, even if they have inter-dependencies. (docs) #14433 #14496
- `-Zpublic-dependency`: Include public/private dependency status in `cargo metadata` #14504
- `-Zpublic-dependency`: Don't require MSRV bump #14507

Documentation

- 🎉 New chapter about the uses, support expectations, and management of `package.rust-version` a.k.a MSRV. (docs) #14619 #14636
- Clarify `target.'cfg(...)'` doesn't respect cfg from build script #14312
- Clarify `[[bin]]` target auto-discovery can be `src/main.rs` and/or in `src/bin/` #14515
- Disambiguate the use of 'target' in the feature resolver v2 doc. #14540
- Make `--config <PATH>` more prominent #14631

- Minor re-grouping of pages. #14620
- contrib: Update docs for how cargo is published #14539
- contrib: Declare support level for each crate in Cargo's Charter / crate docs #14600
- contrib: Declare new Intentional Artifacts as 'small' changes #14599

Internal

- Cleanup duplicated check-cfg lint logic #14567
- Fix elided lifetime due to nightly rustc changes #14487
- Improved error reporting when a feature is not found in `activated_features`. #14647
- cargo-info: Use the `shell.note` to print the note #14554
- ci: bump CI tools #14503 #14628
- perf: zero-copy deserialization for compiler messages when possible #14608
- resolver: Add more SAT resolver tests #14583 #14614
- test: Migrated more tests to snapbox #14576 #14577
- Update dependencies. #14475 #14478 #14489 #14607 #14624 #14632

Cargo 1.82 (2024-10-17)

a2b58c3d...rust-1.82.0

Added

- 🎉 Added `cargo info` command for displaying information about a package. docs #14141 #14418 #14430

Changed

- ❗ Doctest respects Cargo's color options by passing `--color` to rustdoc invocations. #14425
- Improved error message for missing both `[package]` and `[workspace]` in Cargo.toml. #14261
- Enumerate all possible values of `profile.*.debug` for the error message. #14413

Fixed

- Use longhand gitoxide path-spec patterns. Previously the implementation used shorthand pathspecs, which could produce invalid syntax, for example, if the path to the manifest file contained a leading `_` underscore #14380
- cargo-package: fix failures on bare commit git repo. #14359
- cargo-publish: Don't strip non-dev features for renamed dependencies from the HTTP JSON body sent to the registry. The bug only affected third-party registries. #14325 #14327
- cargo-vendor: don't copy source files of excluded Cargo targets when vendoring. #14367

Nightly only

- 🔥 `lockfile-path`: Added `--lockfile-path` flag that allows specifying a path to the lockfile other than the default path

`<workspace_root>/Cargo.lock`. (docs) #14326 #14417 #14423 #14424

- 🔥 `path-bases`: Introduced a table of path "bases" in Cargo configuration files that can be used to prefix the paths of path dependencies and patch entries. (RFC 3529) (docs) #14360
- 🔥 `-Zpackage-workspace`: Enhanced the experience of `cargo package --workspace` when there are dependencies between crates in the workspace. Crates in a workspace are no longer required to publish to actual registries. This is a step toward supporting `cargo publish --workspace`. #13947 #14408 #14340
- cargo-update: Limit pre-release match semantics to use only on `OptVersionReq::Req` #14412
- `edition2024`: Revert "fix: Ensure dep/feature activates the dependency on 2024". #14295
- `update-breaking`: Improved error message when `update --breaking` has an invalid spec #14279
- `update-breaking`: Don’t downgrade on prerelease `VersionReq` when updating with `--breaking` #14250
- `-Zbuild-std`: remove hack on creating virtual std workspace #14358 #14370
- `-Zmsrv-policy`: Adjust MSRV resolve config field name / values. The previous placeholder `resolver.something-like-precedence` is now renamed to `resolver.incompatible-rust-versions`. #14296
- `-Zmsrv-policy`: : Report when incompatible-rust-version packages are selected #14401
- `-Ztarget-applies-to-host`: Fixed passing of links-overrides with target-applies-to-host and an implicit target #14205
- `-Ztarget-applies-to-host`: `-Cmetadata` includes whether extra rustflags is same as host #14432
- `-Ztrim-paths`: rustdoc supports trim-paths for diagnostics #14389

Documentation

- Convert comments to doc comments for `Workspace`. #14397
- Fix MSRV indicator for `workspace.package` and `workspace.dependencies`. #14400
- FAQ: remove outdated Cargo offline usage section. #14336

Internal

- Enhanced `cargo-test-support` usability and documentation. #14266 #14268 #14269 #14270 #14272
- Made summary sync by using Arc instead of Rc #14260
- Used `Rc` instead of `Arc` for storing rustflags #14273
- Removed rustc probe for `--check-cfg` support #14302
- Renamed 'resolved' to 'normalized' for all manifest normalization related items. #14342
- cargo-util-schemas: Added `TomlPackage::new`, `Default` for `TomlWorkspace` #14271
- ci: Switch macos aarch64 to nightly #14382
- mdman: Normalize newlines when rendering options #14428
- perf: dont call wrap in a no-op `source_id::with*` #14318
- test: Migrated more tests to snapbox #14242 #14244 #14293 #14297 #14319 #14402 #14410
- test: don't rely on absence of `RUST_BACKTRACE` #14441
- test: Use gmake on AIX #14323
- Updated to `gix` 0.64.0 #14332
- Updated to `rusqlite` 0.32.0 #14334
- Updated to `windows-sys` 0.59 #14335
- Update dependencies. #14299 #14303 #14324 #14329 #14331 #14391

Cargo 1.81 (2024-09-05)

34a6a87d...rust-1.81.0

Added

Changed

- ❗ cargo-package: Disallow `package.license-file` and `package.readme` pointing to non-existent files during packaging. #13921
- ❗ cargo-package: generated `.cargo_vcs_info.json` is always included, even when `--allow-dirty` is passed. #13960
- ❗ Disallow passing `--release`/`--debug` flag along with the `--profile` flag. #13971
- ❗ Remove `lib.plugin` key support in Cargo.toml. Rust plugin support has been deprecated for four years and was removed in 1.75.0. #13902 #14038
- Make the calculation of `-Cmetadata` for rustc consistent across platforms. #14107
- Emit a warning when `edition` is unset, even when MSRV is unset. #14110

Fixed

- Fix a proc-macro example from a dependency affecting feature resolution. #13892
- Don't warn on duplicate packages from using '..'. #14234
- Don't `du` on every git source load. #14252
- Don't warn about unreferenced duplicate packages #14239
- cargo-publish: Don't strip non-dev features for renamed dependencies from the HTTP JSON body sent to the registry. The bug only affected third-party registries. #14328
- cargo-vendor: don't copy source files of excluded Cargo targets when vendoring. #14368

Nightly only

- 🔥 `update-breaking`: Add `--breaking` to `cargo update`, allowing upgrading dependencies to breaking versions. docs #13979 #14047 #14049
- `--artifact-dir`: Rename `--out-dir` to `--artifact-dir`. The `--out-dir` flag is kept for compatibility and may be removed when the feature gets stabilized. #13809
- `edition2024`: Ensure unused optional dependencies fire for shadowed dependencies. #14028
- `edition2024`: Address problems with implicit -> explicit feature migration #14018
- `-Zcargo-lints`: Add `unknown_lints` to lints list. #14024
- `-Zcargo-lints`: Add tooling to document lints. #14025
- `-Zcargo-lints`: Keep lints updated and sorted. #14030
- `-Zconfig-include`: Allow enabling `config-include` feature in config. #14196
- `-Zpublic-dependency`: remove some legacy public dependency code from the resolver #14090
- `-Ztarget-applies-to-host`: Pass rustflags to artifacts built with implicit targets when using target-applies-to-host #13900 #14201
- cargo-update: Track the behavior of `--precise <prerelease>`. #14013

Documentation

- Clarify `CARGO_CFG_TARGET_FAMILY` is multi-valued. #14165
- Document `CARGO_CFG_TARGET_ABI` #14164
- Document MSRV for each manifest field and build script invocations. #14224
- Remove duplicate `strip` section. #14146
- Update summary of Cargo configuration to include missing keys. #14145
- Update index of Cargo documentation. #14228

- Don't mention non-existent `workspace.badges` field. #14042
- contrib: Suggest atomic commits with separate test commits. #14014
- contrib: Document how to write an RFC for Cargo. #14222
- contrib: Improve triage instructions #14052

Internal

- cargo-package: Change verification order during packaging. #14074
- ci: Add workflow to publish Cargo automatically #14202
- ci: bump CI tools #14062 #14257
- registry: Add local registry overlays. #13926
- registry: move `get_source_id` out of registry #14218
- resolver: Simplify checking for dependency cycles #14089
- rustfix: Add `CodeFix::apply_solution` and impl `Clone` #14092
- source: Clean up after `PathSource`/`RecursivePathSource` split #14169 #14231
- Remove the temporary `__CARGO_GITOXIDE_DISABLE_LIST_FILES` environment variable. #14036
- Simplify checking feature syntax #14106
- Dont make new constant `InternedString` in hot path #14211
- Use `std::fs::absolute` instead of reimplementing it #14075
- Remove unnecessary feature activations from cargo. #14122 #14160
- Revert #13630 as rustc ignores `-C strip` on MSVC. #14061
- test: Allow `unexpected_builtin_cfgs` lint in `user_specific_cfgs` test #14153
- test: Add cargo_test to test-support prelude #14243
- test: migrate Cargo testsuite to `snapbox`. For the complete list of migration pull requests, see #14039
- Updated to `gix` 0.64.0 #14431
- Update dependencies. #13995 #13998 #14037 #14063 #14067 #14174 #14186 #14254

Cargo 1.80 (2024-07-25)

b60a1555...rust-1.80.0

Added

- 🎉 Stabilize `-Zcheck-cfg`! This by default enables rustc's checking of conditional compilation at compile time, which verifies that the crate is correctly handling conditional compilation for different target platforms or features. Internally, cargo will be passing a new command line option `--check-cfg` to all rustc and rustdoc invocations.

  A new build script invocation `cargo::rustc-check-cfg=CHECK_CFG` is added along with this stabilization, as a way to add custom cfgs to the list of expected cfg names and values.

  If a build script is not an option for your package, Cargo provides a config `[lints.rust.unexpected_cfgs.check-cfg]` to add known custom cfgs statically.

  (RFC 3013) (docs) #13571 #13865 #13869 #13884 #13913 #13937 #13958
- 🎉 cargo-update: Allows `--precise` to specify a yanked version of a package, and will update the lockfile accordingly. #13974

Changed

- ❗ manifest: Disallow `[badges]` to inherit from `[workspace.package.badges]`. This was considered a bug. Keep in mind that `[badges]` is effectively deprecated. #13788
- build-script: Suggest old syntax based on MSRV. #13874
- cargo-add: Avoid escaping double quotes by using string literals. #14006
- cargo-clean: Performance improvements for cleaning specific packages via `-p` flag. #13818

- cargo-new: Use `i32` rather than `usize` as the "default integer" in library template. #13939
- cargo-package: Warn, rather than fail, if a Cargo target is excluded during packaging. #13713
- manifest: Warn, not error, on unsupported lint tool in the `[lints]` table. #13833
- perf: Avoid inferring when Cargo targets are known. #13849
- Populate git information when building Cargo from Rust's source tarball. #13832
- Improve the error message when deserializing Cargo configuration from partial environment variables. #13956

Fixed

- resolver: Make path dependencies with the same name stay locked. #13572
- cargo-add: Preserve file permissions on Unix during `write_atomic`. #13898
- cargo-clean: Remove symlink directory on Windows. #13910
- cargo-fix: Don't fix into the standard library. #13792
- cargo-fix: Support IPv6-only networks. #13907
- cargo-new: Don't say we're adding to a workspace when a regular package is in the root. #13987
- cargo-vendor: Silence the warning about forgetting the vendoring. #13886
- cargo-publish/cargo-vendor: Ensure targets in generated Cargo.toml are in a deterministic order. #13989 #14004
- cargo-credential-libsecret: Load `libsecret` by its `SONAME`, `libsecret-1.so.0`. #13927
- Don't panic when an alias doesn't include a subcommand. #13819
- Workaround copying file returning EAGAIN on ZFS on macOS. #13845
- Fetch specific commits even if the GitHub fast path fails. #13946 #13969

- Distinguish Cargo config from different environment variables that share the same prefix. #14000

Nightly only

- `-Zcargo-lints`: Don't always inherit workspace lints. #13812
- `-Zcargo-lints`: Add a test to ensure cap-lints works. #13829
- `-Zcargo-lints`: Error when unstable lints are specified but not enabled. #13805
- `-Zcargo-lints`: Add cargo-lints to unstable docs. #13881
- `-Zcargo-lints`: Refactor cargo lint tests. #13880
- `-Zcargo-lints`: Remove ability to specify `-` in lint name. #13837
- `-Zscript`: Remove unstable rejected frontmatter syntax for cargo script. The only allowed frontmatter syntax now is `---`. #13861 #13893
- `-Zbindeps`: Build only the specified artifact library when multiple types are available. #13842
- `-Zmsrv-policy`: Treat unset MSRV as compatible. #13791
- `-Zgit`/`-Zgitoxide`: Default configuration to be obtained from both environment variables and Cargo configuration. #13687
- `-Zpublic-dependency`: Don't lose 'public' when inheriting a dependency. #13836
- `edition2024`: Disallow ignored `default-features` when inheriting. #13839
- `edition2024`: Validate crate-types/proc-macro for bin like other Cargo targets. #13841

Documentation

- cargo-package: Clarify no guarantee of VCS provenance. #13984
- cargo-metadata: Clarify dash replacement rule in Cargo target names. #13887
- config: Fix wrong type of `rustc-flags` in build script overrides. #13957

- resolver: Add README for `resolver-tests`. #13977
- contrib: Update UI example code in contributor guide. #13864
- Fix libcurl proxy documentation link. #13990
- Add missing `CARGO_MAKEFLAGS` env for plugins. #13872
- Include CircleCI reference in the Continuous Integration chapter. #13850

Internal

- ci: Don't check `cargo` against beta channel. #13827
- test: Set safe.directory for git repo in apache container. #13920
- test: Silence warnings running embedded unittests. #13929
- test: Update test formatting due to nightly rustc changes. #13890 #13901 #13964
- test: Make `git::use_the_cli` test truly locale independent. #13935
- cargo-test-support: Transition direct assertions from cargo-test-support to snapbox. #13980
- cargo-test-support: Auto-redact elapsed time. #13973
- cargo-test-support: Clean up unnecessary uses of `match_exact`. #13879
- Split `RecursivePathSource` out of `PathSource`. #13993
- Adjust custom errors from cert-check due to libgit2 1.8 change. #13970
- Move diagnostic printing to Shell. #13813
- Update dependencies. #13834 #13840 #13948 #13963 #13976

Cargo 1.79 (2024-06-13)

2fe739fc...rust-1.79.0

Added

- 🎉 `cargo add` respects `package.rust-version` a.k.a. MSRV when adding new dependencies. The behavior can be overridden by specifying a version requirement, or passing the `--ignore-rust-version` flag. (RFC 3537) #13608
- A new `Locking` status message shows dependency changes on any command. For `cargo update`, it also tells you if any dependency version is outdated. #13561 #13647 #13651 #13657 #13759 #13764

Changed

- ❗ `RUSTC_WRAPPER`, `RUSTC_WORKSPACE_WRAPPER`, and variables from the `[env]` table now also apply to the initial `rustc -vV` invocation Cargo uses for probing rustc information. #13659
- ❗ Turns dependencies like `foo = { optional = true }` from `version="*"` dependencies with a warning into errors. This behavior has been considered a bug from the beginning. #13775
- ❗ Replace dashes with underscores also if `lib.name` is inferred from `package.name`. This change aligns to the documented behavior. One caveat is that JSON messages emitted by Cargo, like via `cargo metadata` or `--message-format=json`, will start reporting underscore lib names. #12783
- Switch to `gitoxide` for listing files. This improves the performance of build script and `cargo doc` for computing cache freshness, as well as fixes some subtle bugs for `cargo publish`. #13592 #13696 #13704 #13777
- Warn on `-Zlints` being passed and no longer necessary. #13632
- Warn on unused `workspace.dependencies` keys on virtual workspaces. #13664

- Emit 1.77 build script syntax error only when msrv is incompatible. #13808
- Don't warn on `lints.rust.unexpected_cfgs.check-cfg`. #13925
- cargo-init: don't assign `target.name` in Cargo.toml if the value can be inferred. #13606
- cargo-package: normalize paths in `Cargo.toml`, including replacing `\` with `/`. #13729
- cargo-test: recategorize cargo test's `--doc` flag under "Target Selection". #13756

Fixed

- Ensure `--config net.git-fetch-with-cli=true` is respected. #13992 #13997
- Dont panic when resolving an empty alias. #13613
- When using `--target`, the default debuginfo strip rule also applies. Note that on Windows MSVC Cargo no longer strips by default. #13618
- Don't crash on Cargo.toml parse errors that point to multi-byte character #13780
- Don't emit deprecation warning if one of `.cargo/{config,config.toml}` is a symlink to the other. #13793
- Follow HTTP redirections when checking if a repo on GitHub is up-to-date. #13718
- Bash completion fallback in `nounset` mode. #13686
- Rerun build script when rustflags changed and `--target` was passed. #13560
- Fix doc collision for lib/bin with a dash in the inferred name. #13640
- cargo-add: Maintain sorting of dependency features. #13682
- cargo-add: Preserve comments when updating simple deps #13655
- cargo-fix: dont apply same suggestion twice. #13728
- cargo-package: error when the package specified via `--package` cannot be found #13735
- credential-provider: trim newlines in tokens from stdin. #13770

Nightly only

- 🔥 cargo-update: allows `--precise` to specify a pre-release version of a package (RFC 3493) (docs) #13626
- RFC 3491: Unused dependencies cleanup #13778
- `-Zcargo-lints`: Add a basic linting system for Cargo. This is still under development and not available for general use. #13621 #13635 #13797 #13740 #13801 #13852 #13853
- 🔥 `edition2024`: Add default Edition2024 to resolver v3 (MSRV-aware resolver). #13785
- `edition2024`: Remove underscore field support in 2024. #13783 #13798 #13800 #13804
- `edition2024`: Error on `[project]` in Edition 2024 #13747
- `-Zmsrv-policy`: Respect '--ignore-rust-version' #13738
- `-Zmsrv-policy`: Add `--ignore-rust-version` to update/generate-lockfile #13741 #13742
- `-Zmsrv-policy`: Put MSRV-aware resolver behind a config #13769
- `-Zmsrv-policy`: Error, rather than panic, on rust-version 'x' #13771
- `-Zmsrv-policy`: Fallback to 'rustc -V' for MSRV resolving. #13743
- `-Zmsrv-policy`: Add v3 resolver for MSRV-aware resolving #13776
- `-Zmsrv-policy`: Don't respect MSRV for non-local installs #13790
- `-Zmsrv-policy`: Track when MSRV is explicitly set, either way #13732
- test: don't compress test registry crates. #13744

Documentation

- Clarify `--locked` ensuring that Cargo uses dependency versions in lockfile #13665
- Clarify the precedence of `RUSTC_WORKSPACE_WRAPPER` and `RUSTC_WRAPPER`. #13648
- Clarify only in the root Cargo.toml the `[workspace]` section is allowed. #13753
- Clarify the differences between virtual and real manifests. #13794

Internal

- 🎉 New member crates `cargo-test-support` and `cargo-test-macro`! They are designed for testing Cargo itself, so no guarantee on any stability across versions. The crates.io publish of this crate is the same as other members crates. They follow Rust's 6-week release process. #13418
- Fix publish script due to crates.io CDN change #13614
- Push diagnostic complexity on annotate-snippets #13619
- cargo-package: Simplify getting of published Manifest #13666
- ci: update macos images to macos-13 #13685
- manifest: Split out an explicit step to resolve `Cargo.toml` #13693
- manifest: Decouple target discovery from Target creation #13701
- manifest: Expose surce/spans for VirtualManifests #13603
- Update dependencies #13609 #13674 #13675 #13679 #13680 #13692 #13731 #13760 #13950

Cargo 1.78 (2024-05-02)

7bb7b539...rust-1.78.0

Added

- Stabilize global cache data tracking. The `-Zgc` flag is still unstable. This is only for Cargo to start data collection, so that when automatic gc is stabilized, it's less likely to see cache misses. #13492 #13467
- Stabilize lockfile format v4. Lockfile v3 is still the default version. #12852
- Auto-detecting whether output can be rendered using non-ASCII Unicode characters. A configuration value `term.unicode` is added to control the behavior manually. docs #13337
- Support `target.<triple>.rustdocflags` in Cargo configuration. docs #13197

Changed

- cargo-add: Print a status when a dep feature is being created #13434
- cargo-add: improve the error message when adding a package from a replaced source. #13281
- cargo-doc: Collapse down `Generated` statuses without `--verbose`. #13557
- cargo-new: Print a 'Creating', rather than 'Created' status #13367
- cargo-new: Print a note, rather than a comment, for more information #13371
- cargo-new: Print a hint when adding members to workspace #13411
- cargo-test: Suggest `--` for libtest arguments #13448
- cargo-update: Tell users when some dependencies are still behind latest. #13372
- Deprecate non-extension `.cargo/config` files. #13349
- Don't print rustdoc command lines on failure by default #13387
- Respect `package.rust-version` when generating new lockfiles. #12861

- Send `User-Agent: cargo/1.2.3` header when communicating with remote registries. Previously it was `cargo 1.2.3`, which didn't follow the HTTP specifications. #13548
- Emit a warning when `package.edition` field is missing in Cargo.toml. #13499 #13504 #13505 #13533
- Emit warnings from parsing virtual manifests. #13589
- Mention the workspace root location in the error message when collecting workspace members. #13480
- Clarify the profile in use in `Finished` status message. #13422
- Switched more notes/warnings to lowercase. #13410
- Report all packages incompatible with `package.rust-version.`, not just a random one. #13514

Fixed

- cargo-add: don't add the new package to `workspace.members` if there is no existing workspace in Cargo.toml. #13391
- cargo-add: Fix markdown line break in cargo-add #13400
- cargo-run: use Package ID Spec match packages #13335
- cargo-doc: doctest searches native libs in build script outputs. #13490
- cargo-publish: strip also features from dev-dependencies from Cargo.toml to publish. #13518
- Don't duplicate comments when editing TOML via `cargo add/rm/init/new`. #13402
- Fix confusing error messages for sparse index replaced source. #13433
- Respect `CARGO_TERM_COLOR` in '--list' and '-Zhelp'. #13479
- Control colors of errors and help texts from clap through `CARGO_TERM_COLOR`. #13463
- Don't panic on empty spans in Cargo.toml. #13375 #13376

Nightly only

- 🔥 cargo-update: allows `--precise` to specify a yanked version of a package #13333

- `-Zcheck-cfg`: Add `docsrs` cfg as a well known `--check-cfg` #13383
- `-Zcheck-cfg`: Silently ignore `cargo::rustc-check-cfg` to avoid MSRV annoyance when stabilizing `-Zcheck-cfg`. #13438
- `-Zmsrv-policy`: Fallback to `rustc -v` when no MSRV is set #13516
- `-Zscript`: Improve errors related to cargo script #13346
- `-Zpanic-abort-tests`: applies to doctests too #13388
- `-Zpublic-dependency`: supports enabling via the `-Zpublic-dependency` flag. #13340 #13556 #13547
- `-Zpublic-dependency`: test for packaging a public dependency #13536
- `-Zrustdoc-map`: Add all unit's children recursively for `doc.extern-map` option #13481 #13544
- `edition2024`: Enable edition migration for 2024. #13429
- `open-namespaces`: basic support for open namespaces (RFC 3243) (docs) #13591

Documentation

- cargo-fetch: hide `cargo-fetch` recursive link in `--offline` man page. #13364
- cargo-install: `--list` option description starting with uppercase #13344
- cargo-vendor: clarify vendored sources as read-only and ways to modify them #13512
- build-script: clarification of build script metadata set via `cargo::metadata=KEY=VALUE`. #13436
- Clarify the `version` field in `[package]` is optional in Cargo.toml #13390
- Improve "Registry Authentication" docs #13351
- Improve "Specifying Dependencies" docs #13341
- Remove `package.documentation` from the “before publishing” list. #13398

Internal

- 🎉 Integrated tracing-chrome as a basic profiler for Cargo itself. docs #13399 #13551
- Updated to `gix` 0.58.0 #13380
- Updated to `git2` 0.18.2 #13412
- Updated to `jobserver` 0.1.28 #13419
- Updated to `supports-hyperlinks` 3.0.0 #13511
- Updated to `rusqlite` 0.31.0 #13510
- bump-check: use symmetric difference when comparing source code #13581
- bump-check: include rustfix and cargo-util-schemas #13421
- ci: enable m1 runner #13377
- ci: Ensure lockfile is respected during MSRV testing via `cargo-hack`. #13523
- cargo-util-schemas: Consistently compare MSRVs via `RustVersion::is_compatible_with`. #13537
- console: Use new fancy `anstyle` API #13368 #13562
- fingerprint: remove unnecessary Option in `Freshness::Dirty` #13361
- fingerprint: abstract `std::fs` away from on-disk index cache #13515
- mdman: Updated to `pulldown-cmark` 0.10.0 #13517
- refactor: Renamed `Config` to `GlobalContext` #13409 #13486 #13506
- refactor: Removed unused `sysroot_host_libdir`. #13468
- refactor: Expose source/spans to Manifest for emitting lints #13593
- refactor: Flatten manifest parsing #13589
- refactor: Make lockfile diffing/printing more reusable #13564
- test: Updated to `snapbox` 0.5.0 #13441
- test: Verify terminal styling via snapbox's `term-svg` feature. #13461 #13465 #13520
- test: Ensure `nonzero_exit_code` test isn't affected by developers `RUST_BACKTRACE` setting #13385
- test: Add tests for using worktrees. #13567
- test: Fix old_cargos tests #13435

- test: Fixed tests due to changes in rust-lang/rust. #13362 #13382 #13415 #13424 #13444 #13455 #13464 #13466 #13469
- test: disable lldb test as it requires privileges to run on macOS #13416

Cargo 1.77.1 (2024-03-28)

Fixed

- Debuginfo is no longer stripped by default for Windows MSVC targets. This caused an unexpected regression in 1.77.0 that broke backtraces. #13654

Cargo 1.77 (2024-03-21)

1a2666dd...rust-1.77.0

Added

- 🎉 Stabilize the package identifier format as Package ID Spec. This format can be used across most of the commands in Cargo, including the `--package`/`-p` flag, `cargo pkgid`, `cargo metadata`, and JSON messages from `--message-format=json`. #12914 #13202 #13311 #13298 #13322
- Add colors to `-Zhelp` console output #13269
- build script: Extend the build directive syntax with `cargo::`. #12201 #13212

Changed

- 🎉 Disabling debuginfo now implies `strip = "debuginfo"` (when `strip` is not set) to strip pre-existing debuginfo coming from the standard library, reducing the default size of release binaries considerably (from ~4.5 MiB down to ~450 KiB for helloworld on Linux x64). #13257
- Add `rustc` style errors for manifest parsing. #13172
- Deprecate rustc plugin support in cargo #13248
- cargo-vendor: Hold the mutate exclusive lock when vendoring. #12509
- crates-io: Set `Content-Type: application/json` only for requests with a body payload #13264

Fixed

- jobserver: inherit jobserver from env for all kinds of runner #12776
- build script: Set `OUT_DIR` for all units with build scripts #13204
- cargo-add: find the correct package with given features from Git repositories with multiple packages. #13213
- cargo-fix: always inherit the jobserver #13225

- cargo-fix: Call rustc fewer times to improve the performance. #13243
- cargo-new: only inherit workspace package table if the new package is a member #13261
- cargo-update: `--precise` accepts arbitrary git revisions #13250
- manifest: Provide unused key warnings for lints table #13262
- rustfix: Support inserting new lines. #13226

Nightly only

- 🔥 `-Zgit`: Implementation of shallow libgit2 fetches behind an unstable flag docs #13252
- 🔥 Add unstable `--output-format` option to `cargo rustdoc`, providing tools with a way to lean on rustdoc's experimental JSON format. docs #12252 #13284 #13325
- `-Zcheck-cfg`: Rework `--check-cfg` generation comment #13195
- `-Zcheck-cfg`: Go back to passing an empty `values()` when no features are declared #13316
- `-Zprecise-pre-release`: the flag is added but not implemented yet. #13296 #13320
- `-Zpublic-dependency`: support publish package with a `public` field. #13245
- `-Zpublic-dependency`: help text of `--public`/`--no-public` flags for `cargo add` #13272
- `-Zscript`: Add prefix-char frontmatter syntax support #13247
- `-Zscript`: Add multiple experimental manifest syntaxes #13241
- `-Ztrim-paths`: remap common prefix only #13210

Documentation

- Added guidance on setting homepage in manifest #13293
- Clarified how custom subcommands are looked up. #13203
- Clarified why `du` function uses mutex #13273
- Highlighted "How to find features enabled on dependencies" #13305
- Delete sentence about parentheses being unsupported in license #13292

- resolver: clarify how pre-release version is handled in dependency resolution. #13286
- cargo-test: clarify the target selection of the test options. #13236
- cargo-install: clarify `--path` is the installation source not destination #13205
- contrib: Fix team HackMD links #13237
- contrib: Highlight the non-blocking feature gating technique #13307

Internal

- 🎉 New member crate `cargo-util-schemas`! This contains low-level Cargo schema types, focusing on `serde` and `FromStr` for use in reading files and parsing command-lines. Any logic for getting final semantics from these will likely need other tools to process, like `cargo metadata`. The crates.io publish of this crate is the same as other members crates. It follows Rust's 6-week release process. #13178 #13185 #13186 #13209 #13267
- Updated to `gix` 0.57.1. #13230
- cargo-fix: Remove error-format special-case in `cargo fix` #13224
- cargo-credential: bump to 0.4.3 #13221
- mdman: updated to `handlebars` 5.0.0. #13168 #13249
- rustfix: remove useless clippy rules and fix a typo #13182
- ci: fix Dependabot's MSRV auto-update #13265 #13324 #13268
- ci: Add dependency dashboard. #13255
- ci: update alpine docker tag to v3.19 #13228
- ci: Improve GitHub Actions CI config #13317
- resolver: do not panic when sorting empty summaries #13287

Cargo 1.76 (2024-02-08)

6790a512...rust-1.76.0

Added

- Added a Windows application manifest file to the built `cargo.exe` for windows msvc. #13131
  Notable changes:
  - States the compatibility with Windows versions 7, 8, 8.1, 10 and 11.
  - Sets the code page to UTF-8.
  - Enables long path awareness.
- Added color output for `cargo --list`. #12992
- cargo-add: `--optional <dep>` would create a `<dep> = "dep:<dep>"` feature. #13071
- Extends Package ID spec for unambiguous specs. docs #12933
  Specifically,
  - Supports `git+` and `path+` schemes.
  - Supports Git ref query strings, such as `?branch=dev` or `?tag=1.69.0`.

Changed

- ❗ Disallow `[lints]` in virtual workspaces as they are ignored and users likely meant `[workspace.lints]`. This was an oversight in the initial implementation (e.g. a `[dependencies]` produces the same error). #13155
- Disallow empty name in several places like package ID spec and `cargo new`. #13152
- Respect `rust-lang/rust`'s `omit-git-hash` option. #12968
- Displays error count with a number, even when there is only one error. #12484

- `all-static` feature now includes `vendored-libgit2`. #13134
- crates-io: Add support for other 2xx HTTP status codes when interacting with registries. #13158 #13160
- home: Replace SHGetFolderPathW with SHGetKnownFolderPath. #13173

Fixed

- Print rustc messages colored on wincon. #13140
- Fixed bash completion in directory with spaces. #13126
- Fixed uninstall a running binary failed on Windows. #13053 #13099
- Fixed the error message for duplicate links. #12973
- Fixed `--quiet` being used with nested subcommands. #12959
- Fixed panic when there is a cycle in dev-dependencies. #12977
- Don't panic when failed to parse rustc commit-hash. #12963 #12965
- Don't do git fetches when updating workspace members. #12975
- Avoid writing CACHEDIR.TAG if it already exists. #13132
- Accept `?` in the `--package` flag if it's a valid pkgid spec. #13315 #13318
- cargo-package: Only filter out `target` directory if it's in the package root. #12944
- cargo-package: errors out when a build script doesn't exist or is outside the package root. #12995
- cargo-credential-1password: Add missing `--account` argument to `op signin` command. #12985 #12986

Nightly only

- 🔥 The `-Zgc` flag enables garbage collection for deleting old, unused files in cargo's cache. That is, downloaded source files and registry index under the `CARGO_HOME` directory. docs #12634 #12958 #12981 #13055
- 🔥 Added a new environment variable `CARGO_RUSTC_CURRENT_DIR`. This is a path that rustc is invoked from. docs #12996

- `-Zcheck-cfg`: Include declared list of features in fingerprint for `-Zcheck-cfg`. #13012
- `-Zcheck-cfg`: Fix `--check-cfg` invocations with zero features. #13011
- `-Ztrim-paths`: reorder `--remap-path-prefix` flags for `-Zbuild-std`. #13065
- `-Ztrim-paths`: explicitly remap current dir by using `.`. #13114
- `-Ztrim-paths`: exercise with real world debugger. #13091 #13118
- `-Zpublic-dependency`: Limit `exported-private-dependencies` lints to libraries. #13135
- `-Zpublic-dependency`: Disallow workspace-inheriting of dependency public status. #13125
- `-Zpublic-dependency`: Add `--public` for `cargo add`. #13046
- `-Zpublic-dependency`: Remove unused public-deps error handling #13036
- `-Zmsrv-policy`: Prefer MSRV, rather than ignore incompatible. #12950
- `-Zmsrv-policy`: De-prioritize no-rust-version in MSRV resolver. #13066
- `-Zrustdoc-scrape-examples`: Don't filter on workspace members when scraping doc examples. #13077

Documentation

- Recommends a wider selection of libsecret-compatible password managers. #12993
- Clarified different targets has different sets of `CARGO_CFG_*` values. #13069
- Clarified `[lints]` table only affects local development of the current package. #12976
- Clarified `cargo search` can search in alternative registries. #12962
- Added common CI practices for verifying `rust-version` (MSRV) field. #13056
- Added a link to rustc lint levels doc. #12990

- Added a link to the packages lint table from the related workspace table #13057
- contrib: Add more resources to the contrib docs. #13008
- contrib: Update how that credential crates are published. #13006
- contrib: remove review capacity notice. #13070

Internal

- 🎉 Migrate `rustfix` crate to the `rust-lang/cargo` repository. #13005 #13042 #13047 #13048 #13050
- Updated to `curl-sys` 0.4.70, which corresponds to curl 8.4.0. #13147
- Updated to `gix-index` 0.27.1. #13148
- Updated to `itertools` 0.12.0. #13086
- Updated to `rusqlite` 0.30.0. #13087
- Updated to `toml_edit` 0.21.0. #13088
- Updated to `windows-sys` 0.52.0. #13089
- Updated to `tracing` 0.1.37 for being be compatible with rustc_log. #13239 #13242
- Re-enable flaky gitoxide auth tests thanks to update to `gix-config`. #13117 #13129 #13130
- Dogfood Cargo `-Zlints` table feature. #12178
- Refactored `Cargo.toml` parsing code in preparation of extracting an official schema API. #12954 #12960 #12961 #12971 #13000 #13021 #13080 #13097 #13123 #13128 #13154 #13166
- Use `IndexSummary` in `query{_vec}` functions. #12970
- ci: migrate renovate config #13106
- ci: Always update gix packages together #13093
- ci: Catch naive use of AtomicU64 early #12988
- xtask-bump-check: dont check `home` against beta/stable branches #13167
- cargo-test-support: Handle $message_type in JSON diagnostics #13016
- cargo-test-support: Add more options to registry test support. #13085
- cargo-test-support: Add features to the default Cargo.toml file #12997

- cargo-test-support: Fix clippy-wrapper test race condition. #12999
- test: Don't rely on mtime to test changes #13143
- test: remove unnecessary packages and versions for `optionals` tests #13108
- test: Remove the deleted feature `test_2018_feature` from the test. #13156
- test: remove jobserver env var in some tests. #13072
- test: Fix a rustflags test using a wrong buildfile name #12987
- test: Fix some test output validation. #12982
- test: Ignore changing_spec_relearns_crate_types on windows-gnu #12972

Cargo 1.75 (2023-12-28)

59596f0f...rust-1.75.0

Added

- `package.version` field in `Cargo.toml` is now optional and defaults to `0.0.0`. Packages without the `package.version` field cannot be published. #12786
- Links in `--timings` and `cargo doc` outputs are clickable on supported terminals, controllable through `term.hyperlinks` config value. #12889
- Print environment variables for build script executions with `-vv`. #12829
- cargo-new: add new packages to [workspace.members] automatically. #12779
- cargo-doc: print a new `Generated` status displaying the full path. #12859

Changed

- cargo-new: warn if crate name doesn't follow snake_case or kebab-case. #12766
- cargo-install: clarify the arg `<crate>` to install is positional. #12841
- cargo-install: Suggest an alternative version on MSRV failure. #12798
- cargo-install: reports more detailed SemVer errors. #12924
- cargo-install: install only once if there are crates duplicated. #12868
- cargo-remove: Clarify flag behavior of different dependency kinds. #12823
- cargo-remove: suggest the dependency to remove exists only in the other section. #12865
- cargo-update: Do not call it "Downgrading" when difference is only build metadata. #12796
- Enhanced help text to clarify `--test` flag is for Cargo targets, not test functions. #12915

- Included package name/version in build script warnings. #12799
- Provide next steps for bad -Z flag. #12857
- Suggest `cargo search` when `cargo-<command>` cannot be found. #12840
- Do not allow empty feature name. #12928
- Added unsupported short flag suggestion for `--target` and `--exclude` flags. #12805
- Added unsupported short flag suggestion for `--out-dir` flag. #12755
- Added unsupported lowercase `-z` flag suggestion for `-Z` flag. #12788
- Added better suggestion for unsupported `--path` flag. #12811
- Added detailed message when target directory path is invalid. #12820

Fixed

- Fixed corruption when cargo was killed while writing to files. #12744
- cargo-add: Preserve more comments #12838
- cargo-fix: preserve jobserver file descriptors on rustc invocation. #12951
- cargo-remove: Preserve feature comments #12837
- Removed unnecessary backslash in timings HTML report when error happens. #12934
- Fixed error message that invalid a feature name can contain `-`. #12939
- When there's a version of a dependency in the lockfile, Cargo would use that "exact" version, including the build metadata. #12772

Nightly only

- Added `Edition2024` unstable feature. docs #12771
- 🔥 The `-Ztrim-paths` feature adds a new profile setting to control how paths are sanitized in the resulting binary. (RFC 3127) (docs) #12625 #12900 #12908
- `-Zcheck-cfg`: Adjusted for new rustc syntax and behavior. #12845
- `-Zcheck-cfg`: Remove outdated option to `-Zcheck-cfg` warnings. #12884

- `public-dependency`: Support `public` dependency configuration with workspace deps. #12817

Documentation

- profile: add missing `strip` info. #12754
- features: a note about the new limit on number of features. #12913
- crates-io: Add doc comment for `NewCrate` struct. #12782
- resolver: Highlight commands to answer dep resolution questions. #12903
- cargo-bench: `--bench` is passed in unconditionally to bench harnesses. #12850
- cargo-login: mention args after `--` in manpage. #12832
- cargo-vendor: clarify config to use vendored source is printed to stdout #12893
- manifest: update to SPDX 2.3 license expression and 3.20 license list. #12827
- contrib: Policy on manifest editing #12836
- contrib: use `AND` search terms in mdbook search and fixed broken links. #12812 #12813 #12814
- contrib: Describe how to add a new package #12878
- contrib: Removed review capacity notice. #12842

Internal

- Updated to `itertools` 0.11.0. #12759
- Updated to `cargo_metadata` 0.18.0. #12758
- Updated to `curl-sys` 0.4.68, which corresponds to curl 8.4.0. #12808
- Updated to `toml` 0.8.2. #12760
- Updated to `toml_edit` 0.20.2. #12761
- Updated to `gix` to 0.55.2 #12906
- Disabled the `custom_target::custom_bin_target` test on windows-gnu. #12763
- Refactored `Cargo.toml` parsing code in preparation of extracting an official schema API. #12768 #12881 #12902 #12911 #12948

- Split out SemVer logic to its own module. #12926 #12940
- source: Prepare for new `PackageIDSpec` syntax #12938
- resolver: Consolidate logic in `VersionPreferences` #12930
- Make the `SourceId::precise` field an Enum. #12849
- shell: Write at once rather than in fragments. #12880
- Move up looking at index summary enum #12749 #12923
- Generate redirection HTML pages in CI for Cargo Contributor Guide. #12846
- Add new package cache lock modes. #12706
- Add regression test for issue 6915: features and transitive dev deps. #12907
- Auto-labeling when PR review state changes. #12856
- credential: include license files in all published crates. #12953
- credential: Filter `cargo-credential-*` dependencies by OS. #12949
- ci: bump cargo-semver-checks to 0.24.0 #12795
- ci: set and verify all MSRVs for Cargo's crates automatically. #12767 #12654
- ci: use separate concurrency group for publishing Cargo Contributor Book. #12834 #12835
- ci: update `actions/checkout` action to v4 #12762
- cargo-search: improved the margin calculation for the output. #12890

Cargo 1.74 (2023-11-16)

80eca0e5...rust-1.74.0

Added

- 🎉 The `[lints]` table has been stabilized, allowing you to configure reporting levels for rustc and other tool lints in `Cargo.toml`. (RFC 3389) (docs) #12584 #12648
- 🎉 The unstable features `credential-process` and `registry-auth` have been stabilized. These features consolidate the way to authenticate with private registries. (RFC 2730) (RFC 3139) (docs) #12590 #12622 #12623 #12626 #12641 #12644 #12649 #12671 #12709
  Notable changes:
  - Introducing a new protocol for both external and built-in providers to store and retrieve credentials for registry authentication.
  - Adding the `auth-required` field in the registry index's `config.json`, enabling authenticated sparse index, crate downloads, and search API.
  - For using alternative registries with authentication, a credential provider must be configured to avoid unknowingly storing unencrypted credentials on disk.
  - These settings can be configured in `[registry]` and `[registries]` tables.
- 🎉 `--keep-going` flag has been stabilized and is now available in each build command (except `bench` and `test`, which have `--no-fail-fast` instead). (docs) #12568
- Added `--dry-run` flag and summary line at the end for `cargo clean`. #12638
- Added a short alias `-n` for cli option `--dry-run`. #12660
- Added support for `target.'cfg(..)'.linker`. #12535

- Allowed incomplete versions when they are unambiguous for flags like `--package`. #12591 #12614 #12806

Changed

- ❗ Changed how arrays in configuration are merged. The order was unspecified and now follows how other configuration types work for consistency. summary #12515
- ❗ cargo-clean: error out if `--doc` is mixed with `-p`. #12637
- ❗ cargo-new / cargo-init no longer exclude `Cargo.lock` in VCS ignore files for libraries. #12382
- cargo-update: silently deprecate `--aggressive` in favor of the new `--recursive`. #12544
- cargo-update: `-p/--package` can be used as a positional argument. #12545 #12586
- cargo-install: suggest `--git` when the package name looks like a URL. #12575
- cargo-add: summarize the feature list when it's too long. #12662 #12702
- Shell completion for `--target` uses rustup but falls back to rustc. #12606
- Help users know possible `--target` values. #12607
- Enhanced "registry index not found" error message. #12732
- Enhanced CLI help message of `--explain`. #12592
- Enhanced deserialization errors of untagged enums with `serde-untagged`. #12574 #12581
- Enhanced the error when mismatching prerelease version candidates. #12659
- Enhanced the suggestion on ambiguous Package ID spec. #12685
- Enhanced TOML parse errors to show the context. #12556
- Enhanced filesystem error by adding wrappers around `std::fs::metadata`. #12636
- Enhanced resolver version mismatch warning. #12573

- Use clap to suggest alternative argument for unsupported arguments. #12529 #12693 #12723
- Removed redundant information from cargo new/init `--help` output. #12594
- Console output and styling tweaks. #12578 #12655 #12593

Fixed

- Use full target spec for `cargo rustc --print --target`. #12743
- Copy PDBs also for EFI targets. #12688
- Fixed resolver behavior being independent of package order. #12602
- Fixed unnecessary clean up of `profile.release.package."*"` for `cargo remove`. #12624

Nightly only

- `-Zasymmetric-token`: Created dedicated unstable flag for asymmetric-token support. #12551
- `-Zasymmetric-token`: Improved logout message for asymmetric tokens. #12587
- `-Zmsrv-policy`: **Very** preliminary MSRV resolver support. #12560
- `-Zscript`: Hack in code fence support. #12681
- `-Zbindeps`: Support dependencies from registries. #12421

Documentation

- ❗ Policy change: Checking `Cargo.lock` into version control is now the default choice, even for libraries. Lockfile and CI integration documentations are also expanded. Policy docs, Lockfile docs, CI docs, #12382 #12630
- SemVer: Update documentation about removing optional dependencies. #12687
- Contrib: Add process for security responses. #12487
- cargo-publish: warn about upload timeout. #12733
- mdbook: use *AND* search when having multiple terms. #12548
- Established publish best practices #12745

- Clarify caret requirements. #12679
- Clarify how `version` works for `git` dependencies. #12270
- Clarify and differentiate defaults for split-debuginfo. #12680
- Added missing `strip` entries in `dev` and `release` profiles. #12748

Internal

- Updated to `curl-sys` 0.4.66, which corresponds to curl 8.3.0. #12718
- Updated to `gitoxide` 0.54.1. #12731
- Updated to `git2` 0.18.0, which corresponds to libgit2 1.7.1. #12580
- Updated to `cargo_metadata` 0.17.0. #12758
- Updated target-arch-aware crates to support mips r6 targets #12720
- publish.py: Remove obsolete `sleep()` calls. #12686
- Define `{{command}}` for use in src/doc/man/includes #12570
- Set tracing target `network` for networking messages. #12582
- cargo-test-support: Add `with_stdout_unordered`. #12635
- dep: Switch from `termcolor` to `anstream`. #12751
- Put `Source` trait under `cargo::sources`. #12527
- SourceId: merge `name` and `alt_registry_key` into one enum. #12675
- TomlManifest: fail when package_root is not a directory. #12722
- util: enhanced doc of `network::retry` doc. #12583
- refactor: Pull out cargo-add MSRV code for reuse #12553
- refactor(install): Move value parsing to clap #12547
- Fixed spurious errors with networking tests. #12726
- Use a more compact relative-time format for `CARGO_LOG` internal logging. #12542
- Use newer std API for cleaner code. #12559 #12604 #12615 #12631
- Buffer console status messages. #12727
- Use enum to describe index summaries to provide a richer information when summaries are not available for resolution. #12643
- Use shortest path for resolving the path from the given dependency up to the root. #12678
- Read/write the encoded `cargo update --precise` in the same place #12629

- Set MSRV for internal packages. #12381
- ci: Update Renovate schema #12741
- ci: Ignore patch version in MSRV #12716

Cargo 1.73 (2023-10-05)

45782b6b...rust-1.73.0

Added

- Print environment variables for `cargo run/bench/test` in extra verbose mode `-vv`. #12498
- Display package versions on Cargo timings graph. #12420

Changed

- ❗ Cargo now bails out when using `cargo::` in custom build scripts. This is a preparation for an upcoming change in build script invocations. #12332
- ❗ `cargo login` no longer accept any token after the `--` syntax. Arguments after `--` are now reserved in the preparation of the new credential provider feature. This introduces a regression that overlooks the `cargo login -- <token>` support in previous versions. #12499
- Make Cargo `--help` easier to browse. #11905
- Prompt the use of `--nocapture` flag if `cargo test` process is terminated via a signal. #12463
- Preserve jobserver file descriptors on the rustc invocation for getting target information. #12447
- Clarify in `--help` that `cargo test --all-targets` excludes doctests. #12422
- Normalize `cargo.toml` to `Cargo.toml` on publish, and warn on other cases of `Cargo.toml`. #12399

Fixed

- Only skip mtime check on `~/.cargo/{git,registry}`. #12369
- Fixed `cargo doc --open` crash on WSL2. #12373
- Fixed panic when enabling `http.debug` for certain strings. #12468
- Fixed `cargo remove` incorrectly removing used patches. #12454

- Fixed crate checksum lookup query should match on semver build metadata. #11447
- Fixed printing multiple warning messages for unused fields in `[registries]` table. #12439

Nightly only

- 🔥 The `-Zcredential-process` has been reimplemented with a clearer way to communicate with different credential providers. Several built-in providers are also added to Cargo. docs #12334 #12396 #12424 #12440 #12461 #12469 #12483 #12499 #12507 #12512 #12518 #12521 #12526
  Some notable changes:
  - Renamed `credential-process` to `credential-provider` in Cargo configurations.
  - New JSON protocol for communicating with external credential providers via stdin/stdout.
  - The GNOME Secert provider now dynamically loads `libsecert`.
  - The 1password provider is no longer built-in.
  - Changed the unstable key for asymmetric tokens from `registry-auth` to `credential-process`.
- ❗ Removed `--keep-going` flag support from `cargo test` and `cargo bench`. #12478 #12492
- Fixed invalid package names generated by `-Zscript`. #12349
- `-Zscript` now errors out on unsupported commands — `publish` and `package`. #12350
- Encode URL params correctly for source ID in Cargo.lock. #12280
- Replaced invalid `panic_unwind` std feature with `panic-unwind`. #12364
- `-Zlints`: doctest extraction should respect `[lints]`. #12501

Documentation

- SemVer: Adding a section for changing the alignment, layout, or size of a well-defined type. #12169
- Use heading attributes to control the fragment. #12339
- Use "number" instead of "digit" when explaining Cargo's use of semver. #12340
- contrib: Add some more detail about how publishing works. #12344
- Clarify "Package ID" and "Source ID" in `cargo metadata` are opaque strings. #12313
- Clarify that `rerun-if-env-changed` doesn't monitor the environment variables it set for crates and build script. #12482
- Clarify that multiple versions that differ only in the metadata tag are disallowed on crates.io. #12335
- Clarify `lto` setting passing `-Clinker-plugin-lto`. #12407
- Added `profile.strip` to configuration and environment variable docs. #12337 #12408
- Added docs for artifact JSON debuginfo levels. #12376
- Added a notice for the backward compatible `.cargo/credential` file existence. #12479
- Raised the awareness of `resolver = 2` used inside workspaces. #12388
- Replaced `master` branch by default branch in documentation. #12435

Internal

- Updated to `criterion` 0.5.1. #12338
- Updated to `curl-sys` 0.4.65, which corresponds to curl 8.2.1. #12406
- Updated to `indexmap` v2. #12368
- Updated to `miow` 0.6.0, which drops old versions of `windows-sys`. #12453
- ci: automatically test new packages by using `--workspace`. #12342
- ci: automatically update dependencies monthly with Renovate. #12341 #12466
- ci: rewrote `xtask-bump-check` for respecting semver by adopting `cargo-semver-checks`. #12395 #12513 #12508

- Rearranged and renamed test directories #12397 #12398
- Migrated from `log` to `tracing`. #12458 #12488
- Track `--help` output in tests. #11912
- Cleaned up and shared package metadata within workspace. #12352
- `crates-io`: expose HTTP headers and `Error` type. #12310
- For `cargo update`, caught CLI flags conflict between `--aggressive` and `--precise` in clap. #12428
- Several fixes for either making Cargo testsuite pass on nightly or in `rust-lang/rust`. #12413 #12416 #12429 #12450 #12491 #12500

Cargo 1.72 (2023-08-24)

64fb38c9...rust-1.72.0

Added

- ❗ Enable `-Zdoctest-in-workspace` by default. When running each documentation test, the working directory is set to the root directory of the package the test belongs to. docs #12221 #12288
- Add support of the "default" keyword to reset previously set `build.jobs` parallelism back to the default. #12222

Changed

- 🚨 CVE-2023-40030: Malicious dependencies can inject arbitrary JavaScript into cargo-generated timing reports. To mitigate this, feature name validation check is now turned into a hard error. The warning was added in Rust 1.49. These extended characters aren't allowed on crates.io, so this should only impact users of other registries, or people who don't publish to a registry. #12291
- Cargo now warns when an edition 2021 package is in a virtual workspace and `workspace.resolver` is not set. It is recommended to set the resolver version for workspaces explicitly. #10910
- Set IBM AIX shared libraries search path to `LIBPATH`. #11968
- Don't pass `-C debuginfo=0` to rustc as it is the default value. #12022 #12205
- Added a message on reusing previous temporary path on `cargo install` failures. #12231
- Added a message when `rustup` override shorthand is put in a wrong position. #12226
- Respect scp-like URL as much as possible when fetching nested submodules. #12359 #12411

Fixed

- `cargo clean` uses `remove_dir_all` as a fallback to resolve race conditions. #11442
- Reduced the chance Cargo re-formats the user's `[features]` table. #12191
- Fixed nested Git submodules not able to fetch. #12244

Nightly only

- 🔥 The `-Zscript` is an experimental feature to add unstable support for single-file packages in Cargo, so we can explore the design and resolve questions with an implementation to collect feedback on. (eRFC 3424) docs #12245 #12255 #12258 #12262 #12268 #12269 #12281 #12282 #12283 #12284 #12287 #12289 #12303 #12305 #12308
- Automatically inherit workspace lints when running `cargo new`/`cargo init`. #12174
- Removed `-Zjobserver-per-rustc` again. #12285
- Added `.toml` file extension restriction for `-Zconfig-include`. #12298
- Added `-Znext-lockfile-bump` to prepare for the next lockfile bump. #12279 #12302

Documentation

- Added a description of `Cargo.lock` conflicts in the Cargo FAQ. #12185
- Added a small note about indexes ignoring SemVer build metadata. #12206
- Added doc comments for types and friends in `cargo::sources` module. #12192 #12239 #12247
- Added more documentation for `Source` download functions. #12319
- Added READMEs for the credential helpers. #12322
- Fixed version requirement example in Dependency Resolution. #12267
- Clarify the default behavior of cargo-install. #12276

- Clarify the use of "default" branch instead of `main` by default. #12251
- Provide guidance on version requirements. #12323

Internal

- Updated to `gix` 0.45 for multi-round pack negotiations. #12236
- Updated to `curl-sys` 0.4.63, which corresponds to curl 8.1.2. #12218
- Updated to `openssl` 0.10.55. #12300
- Updated several dependencies. #12261
- Removed unused features from `windows-sys` dependency. #12176
- Refactored compiler invocations. #12211
- Refactored git and registry sources, and registry data. #12203 #12197 #12240 #12248
- Lexicographically order `-Z` flags. #12182 #12223 #12224
- Several Cargo's own test infra improvements and speed-ups. #12184 #12188 #12189 #12194 #12199
- Migrated print-ban from test to clippy #12246
- Switched to `OnceLock` for interning uses. #12217
- Removed a unnecessary `.clone`. #12213
- Don't try to compile `cargo-credential-gnome-secret` on non-Linux platforms. #12321
- Use macro to remove duplication of workspace inheritable fields getters. #12317
- Extracted and rearranged registry API items to their own modules. #12290
- Show a better error when container tests fail. #12264

Cargo 1.71.1 (2023-08-03)

Fixed

- 🚨 CVE-2023-38497: Cargo 1.71.1 or later respects umask when extracting crate archives. It also purges the caches it tries to access if they were generated by older Cargo versions.

Cargo 1.71 (2023-07-13)

84b7041f...rust-1.71.0

Added

- Allowed named debuginfo options in Cargo.toml. docs #11958
- Added `workspace_default_members` to the output of `cargo metadata`. #11978
- Automatically inherit workspace fields when running `cargo new`/`cargo init`. #12069

Changed

- ❗ Optimized the usage under `rustup`. When Cargo detects it will run `rustc` pointing a rustup proxy, it'll try bypassing the proxy and use the underlying binary directly. There are assumptions around the interaction with rustup and `RUSTUP_TOOLCHAIN`. However, it's not expected to affect normal users. #11917
- ❗ When querying a package, Cargo tries only the original name, all hyphens, and all underscores to handle misspellings. Previously, Cargo tried each combination of hyphens and underscores, causing excessive requests to crates.io. #12083
- ❗ Disallow `RUSTUP_HOME` and `RUSTUP_TOOLCHAIN` in the `[env]` configuration table. This is considered to be not a use case Cargo would like to support, since it will likely cause problems or lead to confusion. #12101 #12107
- Better error message when getting an empty dependency table in Cargo.toml. #11997
- Better error message when empty dependency was specified in Cargo.toml. #12001
- `--help` text is now wrapping for readability on narrow screens. #12013
- Tweaked the order of arguments in `--help` text to clarify role of `--bin`. #12157

- `rust-version` is included in `cargo publish` requests to registries. #12041

Fixed

- Corrected the bug report URL for `cargo clippy --fix`. #11882
- Cargo now applies `[env]` to rust invocations for target info discovery. #12029
- Fixed tokens not redacted in http debug when using HTTP/2. #12095
- Fixed `-C debuginfo` not passed in some situation, leading to build cache miss. #12165
- Fixed the ambiguity when `cargo install` found packages with the same name. The ambiguity happened in a situation like a package depending on old versions of itself. #12015
- Fixed a false positive that `cargo package` checks for conflict files. #12135
- Fixed `dep/feat` syntax not working when co-exist with `dep:` syntax, and trying to enable features of an optional dependency. #12130
- Fixed `cargo tree` not handling the output with `-e no-proc-macro` correctly. #12044
- Warn instead of error in `cargo package` on empty `readme` or `license-file` in Cargo.toml. #12036
- Fixed when an HTTP proxy is in use and the Cargo executable links to a certain version of system libcurl, CURL connections might fail. Affected libcurl versions: 7.87.0, 7.88.0, 7.88.1. #12234 #12242

Nightly only

- 🔥 The `-Zgitoxide` feature now supports shallow clones and fetches for dependencies and registry indexes. docs #11840
- 🔥 The `-Zlints` feature enables configuring lints rules in Cargo.toml docs #12148 #12168
- The `-Zbuild-std` breakage of missing features in `nightly-2023-05-04` has been fixed in `nightly-2023-05-05`. #12088
- Recompile on profile rustflags changes. #11981

- Added `-Zmsrv-policy` feature flag placeholder. #12043
- `cargo add` now considers `rust-version` when selecting packages with `-Zmsrv-policy`. #12078

Documentation

- Added Cargo team charter. docs #12010
- SemVer: Adding `#[non_exhaustive]` on existing items is a breaking change. #10877
- SemVer: It is not a breaking change to make an unsafe function safe. #12116
- SemVer: changing MSRV is generally a minor change. #12122
- Clarify when and how to `cargo yank`. #11862
- Clarify that crates.io doesn't link to docs.rs right away. #12146
- Clarify documentation around test target setting. #12032
- Specify `rust_version` in Index format. #12040
- Specify `msg` in owner-remove registry API response. #12068
- Added more documentation for artifact-dependencies. #12110
- Added doc comments for `Source` and build script for cargo-the-library. #12133 #12153 #12159
- Several typo and broken link fixes. #12018 #12020 #12049 #12067 #12073 #12143
- home: clarify the behavior on each platform #12047

Internal

- Updated to `linux-raw-sys` 0.3.2 #11998
- Updated to `git2` 0.17.1, which corresponds to libgit2 1.6.4. #12096
- Updated to `windows-sys` 0.48.0 #12021
- Updated to `libc` 0.2.144 #12014 #12098
- Updated to `openssl-src` 111.25.3+1.1.1t #12005
- Updated to `home` 0.5.5 #12037
- Enabled feature `Win32_System_Console` feature since it is used. #12016

- Cargo is now a Cargo workspace. We dogfood ourselves finally! #11851 #11994 #11996 #12024 #12025 #12057
- 🔥 A new, straightforward issue labels system for Cargo contributors. docs #11995 #12002 #12003
- Allow win/mac credential managers to build on all platforms. #11993 #12027
- Use `openssl` only on non-Windows platforms. #11979
- Use restricted Damerau-Levenshtein algorithm to provide typo suggestions. #11963
- Added a new xtask `cargo build-man`. #12048
- Added a new xtask `cargo stale-label`. #12051
- Added a new xtask `cargo unpublished`. #12039 #12045 #12085
- CI: check if any version bump needed for member crates. #12126
- Fixed some test infra issues. #11976 #12026 #12055 #12117

Cargo 1.70 (2023-06-01)

9880b408...rust-1.70.0

Added

- 🎉 Added `cargo logout` command for removing an API token from the registry locally. docs #11919 #11950
- Added `--ignore-rust-version` flag to `cargo install`. #11859
- The `CARGO_PKG_README` environment variable is now set to the path to the README file when compiling a crate. #11645
- Cargo now displays richer information of Cargo target failed to compile. #11636

Changed

- 🎉 The `sparse` protocol is now the default protocol for crates.io! (RFC 2789) (docs) #11791 #11783
- ❗ `cargo login` and `cargo logout` now uses the registry specified in `registry.default`. This was an unintentional regression. #11949
- `cargo update` accurately shows `Downgrading` status when downgrading dependencies. #11839
- Added more information to HTTP errors to help with debugging. #11878
- Added delays to network retries in Cargo. #11881
- Refined `cargo publish` message when waiting for a publish complete. #11713
- Better error message when `cargo install` from a git repository but found multiple packages. #11835

Fixed

- Removed duplicates of possible values in `--charset` option of `cargo tree`. #11785

- Fixed `CARGO_CFG_` vars for configs defined both with and without value. #11790
- Broke endless loop on cyclic features in added dependency in `cargo add`. #11805
- Don't panic when `[patch]` involved in dependency resolution results in a conflict. #11770
- Fixed credential token format validation. #11951
- Added the missing token format validation on publish. #11952
- Fixed case mismatches when looking up env vars in the Config snapshot. #11824
- `cargo new` generates the correct `.hgignore` aligning semantics with other VCS ignore files. #11855
- Stopped doing unnecessary fuzzy registry index queries. This significantly reduces the amount of HTTP requests to remote registries for crates containing `-` or `_` in their names. #11936 #11937

Nightly only

- Added `-Zdirect-minimal-versions`. This behaves like `-Zminimal-versions` but only for direct dependencies. (docs) #11688
- Added `-Zgitoxide` which switches all `git fetch` operation in Cargo to use `gitoxide` crate. This is still an MVP but could improve the performance up to 2 times. (docs) #11448 #11800 #11822 #11830
- Removed `-Zjobserver-per-rustc`. Its rustc counterpart never got landed. #11764

Documentation

- Cleaned-up unstable documentation. #11793
- Enhanced the documentation of timing report with graphs. #11798
- Clarified requirements about the state of the registry index after publish. #11926
- Clarified docs on `-C` that it appears before the command. #11947
- Clarified working directory behaviour for `cargo test`, `cargo bench` and `cargo run`. #11901

- Fixed the doc of `registries.name.index` configuration. #11880
- Notice for potential unexpected shell expansions in help text of `cargo-add`. #11826
- Updated external-tools JSON docs. #11918
- Call out the differences between the index JSON and the API or metadata. #11927
- Consistently use `@` when mentioning pkgid format. #11956
- Enhanced Cargo Contributor Guide. #11825 #11842 #11869 #11876
- Moved a part of Cargo Contributor Guide to Cargo API documentation. docs #11809 #11841 #11850 #11870
- Cargo team now arranges office hours! #11903

Internal

- Switched to `sha2` crate for SHA256 calculation. #11795 #11807
- Switched benchsuite to the index archive. #11933
- Updated to `base64` 0.21.0. #11796
- Updated to `curl-sys` 0.4.61, which corresponds to curl 8.0.1. #11871
- Updated to `proptest` 1.1.0. #11886
- Updated to `git2` 0.17.0, which corresponds to libgit2 1.6.3. #11928
- Updated to `clap` 4.2. #11904
- Integrated `cargo-deny` in Cargo its own CI pipeline. #11761
- Made non-blocking IO calls more robust. #11624
- Dropped `derive` feature from `serde` in `cargo-platform`. #11915
- Replaced `std::fs::canonicalize` with a more robust `try_canonicalize`. #11866
- Enabled clippy warning on `disallowed_methods` for `std::env::var` and friends. #11828

Cargo 1.69 (2023-04-20)

985d561f...rust-1.69.0

Added

- Cargo now suggests `cargo fix` or `cargo clippy --fix` when compilation warnings are auto-fixable. #11558
- Cargo now suggests `cargo add` if you try to install a library crate. #11410
- Cargo now sets the `CARGO_BIN_NAME` environment variable also for binary examples. #11705

Changed

- ❗ When `default-features` is set to false of a workspace dependency, and an inherited dependency of a member has `default-features = true`, Cargo will enable default features of that dependency. #11409
- ❗ Deny `CARGO_HOME` in `[env]` configuration table. Cargo itself doesn't pick up this value, but recursive calls to cargo would, which was not intended. #11644
- ❗ Debuginfo for build dependencies is now off if not explicitly set. This is expected to improve the overall build time. #11252
- Cargo now emits errors on invalid alphanumeric characters in a registry token. #11600
- `cargo add` now checks only the order of `[dependencies]` without considering `[dependencies.*]`. #11612
- Cargo now respects the new jobserver IPC style in GNU Make 4.4, by updating its dependency `jobserver`. #11767
- `cargo install` now reports required features when no binary meets its requirements. #11647

Fixed

- Uplifted `.dwp` DWARF package file next to the executable for debuggers to locate them. #11572
- Fixed build scripts triggering recompiles when a `rerun-if-changed` points to a directory whose mtime is not preserved by the filesystem. #11613
- Fixed panics when using dependencies from `[workspace.dependencies]` for `[patch]`. This usage is not supposed to be supported. #11565 #11630
- Fixed `cargo report` saving the same future-incompat reports multiple times. #11648
- Fixed the incorrect inference of a directory ending with `.rs` as a file. #11678
- Fixed `.cargo-ok` file being truncated wrongly, preventing from using a dependency. #11665 #11724

Nightly only

- `-Zrustdoc-scrape-example` must fail with bad build script. #11694
- Updated 1password credential manager integration to the version 2 CLI. #11692
- Emit an error message for transitive artifact dependencies with targets the package doesn't directly interact with. #11643
- Added `-C` flag for changing current dir before build starts. #10952

Documentation

- Clarified the difference between `CARGO_CRATE_NAME` and `CARGO_PKG_NAME`. #11576
- Added links to the Target section of the glossary for occurrences of target triple. #11603
- Described how the current resolver sometimes duplicates dependencies. #11604
- Added a note about verifying your email address on crates.io. #11620
- Mention current default value in `publish.timeout` docs. #11652

- More doc comments for `cargo::core::compiler` modules. #11669 #11703 #11711 #11758
- Added more guidance on how to implement unstable features. #11675
- Fixed unstable chapter layout for `codegen-backend`. #11676
- Add a link to LTO doc. #11701
- Added documentation for the configuration discovery of `cargo install` to the man pages #11763
- Documented `-F` flag as an alias for `--features` in `cargo add`. #11774

Internal

- Disable network SSH tests on Windows. #11610
- Made some blocking tests non-blocking. #11650
- Deny warnings in CI, not locally. #11699
- Re-export `cargo_new::NewProjectKind` as public. #11700
- Made dependencies in alphabetical order. #11719
- Switched some tests from `build` to `check`. #11725
- Consolidated how Cargo reads environments variables internally. #11727 #11754
- Fixed tests with nondeterministic ordering #11766
- Added a test to verify the intermediate artifacts persist in the temp directory. #11771
- Updated cross test instructions for aarch64-apple-darwin. #11663
- Updated to `toml` v0.6 and `toml_edit` v0.18 for TOML manipulations. #11618
- Updated to `clap` v4.1.3. #11619
- Replaced `winapi` with `windows-sys` crate for Windows bindings. #11656
- Reused `url` crate for percent encoding instead of `percent-encoding`. #11750
- Cargo contributors can benefit from smart punctuations when writing documentations, e.g., `---` is auto-converted into an em dash. (docs) #11646 #11715

- Cargo's CI pipeline now covers macOS on nightly. #11712
- Re-enabled some clippy lints in Cargo itself. #11722
- Enabled sparse protocol in Cargo's CI. #11632
- Pull requests in Cargo now get autolabelled for label `A-*` and `Command-*`. #11664 #11679

Cargo 1.68.2 (2023-03-28)

115f3455...rust-1.68.0

- Updated the GitHub RSA SSH host key bundled within cargo. The key was rotated by GitHub on 2023-03-24 after the old one leaked. #11883
- Added support for SSH known hosts marker `@revoked`. #11635
- Marked the old GitHub RSA host key as revoked. This will prevent Cargo from accepting the leaked key even when trusted by the system. #11889

Cargo 1.68 (2023-03-09)

f6e737b1...rust-1.68.0

Added

- 🎉 The new "sparse" protocol has been stabilized. It should provide a significant performance improvement when accessing crates.io. (RFC 2789) (docs) #11224 #11480 #11733 #11756
- 🎉 `home` crate is now a subcrate in `rust-lang/cargo` repository. Welcome! #11359 #11481
- Long diagnostic messages now can be truncated to be more readable. #11494
- Shows the progress of crates.io index update even when `net.git-fetch-with-cli` enabled. #11579
- `cargo build --verbose` tells you more about why it recompiles. #11407
- Cargo's file locking mechanism now supports Solaris by using `fcntl`. #11439 #11474
- Added a new SemVer compatibility rule explaining the expectations around diagnostic lints #11596
- `cargo vendor` generates a different source replacement entry for each revision from the same git repository. #10690
- Cargo contributors can relabel issues via triagebot. doc #11498
- Cargo contributors can write tests in containers. #11583

Changed

- Cargo now by default saves credentials to `.cargo/credentials.toml`. If `.cargo/credentials` exists, writes to it for backward compatibility reasons. #11533
- To prevent sensitive data from being logged, Cargo introduces a new wrapper type internally. #11545
- Several documentation improvements. #11475 #11504 #11516 #11517 #11568 #11586 #11592

Fixed

- ❗ `cargo package` and `cargo publish` now respects workspace's `Cargo.lock`. This is an expected behavior but previously got overlooked. #11477
- Fixed `cargo vendor` failing on resolving git dependencies inherited from a workspace. #11414
- `cargo install` can now correctly install root package when `workspace.default-members` is specified. #11067
- Fixed panic on target specific dependency errors. #11541
- Shows `--help` if there is no man page for a subcommand. #11473
- Setting `target.cfg(…).rustflags` shouldn't erase `build.rustdocflags`. #11323
- Unsupported `profile.split-debuginfo` options are now ignored, which previously made Cargo fail to compile on certain platforms. #11347 #11633
- Don't panic in Windows headless session with really long file names. #11759

Nightly only

- Implemented initial support of asymmetric token authentication for registries. (RFC 3231) (docs) #10771
- Do not error for `auth-required: true` without `-Z sparse-registry` #11661
- Supports `codegen-backend` and `rustflags` in profiles in config file. #11562
- Suggests `cargo clippy --fix` when warnings/errors could be fixed with clippy. #11399
- Fixed artifact deps not working when target field specified coexists with `optional = true`. #11434
- Make Cargo distinguish `Unit`s with and without artifact targets. #11478
- `cargo metadata` supports artifact dependencies. #11550

- Allows builds of some crate to fail during optional doc-scraping. #11450
- Add warning if potentially-scrapable examples are skipped due to dev-dependencies. #11503
- Don't scrape examples from library targets by default. #11499
- Fixed examples of proc-macro crates being scraped for examples. #11497

Cargo 1.67 (2023-01-26)

7e484fc1...rust-1.67.0

Added

- `cargo remove` now cleans up the referenced dependency of the root workspace manifest, `profile`, `patch`, and `replace` sections after a successful removal of a dependency. #11194 #11242 #11351
- `cargo package` and `cargo publish` now report total and compressed crate size after packaging. #11270

Changed

- ❗ Cargo now reuses the value of `$CARGO` if it's already set in the environment, and forwards the value when executing external subcommands and build scripts. #11285
- ❗ Cargo now emits an error when running `cargo update --precise` without a `-p` flag. #11349
- ❗ Cargo now emits an error if there are multiple registries in the configuration with the same index URL. #10592
- Cargo now is aware of compression ratio when extracting crate files. This relaxes the hard size limit introduced in 1.64.0 to mitigate zip bomb attack. #11337
- Cargo now errors out when `cargo fix` on a git repo with uncommitted changes. #11400
- Cargo now warns when `cargo tree -i <spec>` cannot find any package. #11377
- Cargo now warns when running `cargo new/init` and `PATH` env separator is in the project path. #11318
- Better error messages when multiple packages were found and `cargo add/remove` gets confused. #11186 #11375
- A better error message when `cargo init` but existing ignore files aren't UTF-8. #11321
- A better error message for `cargo install .`. #11401

- A better warning when the same file path found in multiple build targets. #11299
- Updated the internal HTTP library libcurl with various fixes and updates. #11307 #11326

Fixed

- Fixed `cargo clean` for removing fingerprints and build script artifacts of only the requested package #10621
- Fixed `cargo install --index` not working when config `registry.default` is set. #11302
- Fixed git2 safe-directory accidentally disabled when no network configuration was found. #11366
- Migrate from crate `atty` to resolve potential soundness issue. #11420
- Cleans stale git temp files left when libgit2 indexing is interrupted. #11308

Nightly only

- Suggests `cargo fix` when some compilation warnings/errors can be auto-fixed. #10989 #11368
- Changed `rustdoc-scrape-examples` to be a target-level configuration. #10343 #11425 #11430 #11445
- Propagates change of artifact bin dependency to its parent fingerprint. #11353
- Fixed `wait-for-publish` to work with sparse registry. #11356 #11327 #11388
- Stores the `sparse+` prefix in the `SourceId` for sparse registries #11387 #11403
- Implemented alternative registry authentication support. (RFC 3139) (docs) #10592
- Added documentation of config option `registries.crates-io.protocol`. #11350

Cargo 1.66.1 (2023-01-10)

Fixed

- 🚨 CVE-2022-46176: Added validation of SSH host keys for git URLs. See the docs for more information on how to configure the known host keys.

Cargo 1.66 (2022-12-15)

08250398...rust-1.66.0

Added

- 🎉 Added `cargo remove` command for removing dependencies from `Cargo.toml`. docs #11059 #11099 #11193 #11204 #11227
- Added support for git dependencies having git submodules with relative paths. #11106
- Cargo now sends requests with an `Accept-Encoding` header to registries. #11292
- Cargo now forwards non-UTF8 arguments to external subcommands. #11118

Changed

- ❗ Disambiguate source replacements from various angles. RFC-3289 #10907
    - When the crates-io source is replaced, the user is required to specify which registry to use with `--registry <NAME>` when performing an API operation.
    - Publishing to source-replaced crates.io is no longer permitted using the crates.io token (`registry.token`).
    - In source replacement, the `replace-with` key can reference the name of an alternative registry in the `[registries]` table.
- ❗ `cargo publish` now blocks until it sees the published package in the index. #11062 #11210 #11216 #11255
- Cargo now uses the clap v4 library for command-line argument parsing. #11116 #11119 #11159 #11190 #11239 #11280
- Cargo now only warns on a user-defined alias shadowing an external command. #11170
- Several documentation improvements. #10770 #10938 #11082 #11093 #11157 #11185 #11207 #11219 #11240 #11241 #11282

Fixed

- ❗ Config file loaded via `cargo --config <file>` now takes priority over environment variables. This is a documented behaviour but the old implementation accidentally got it wrong. #11077
- ❗ Cargo collects rustflags in `target.cfg(…).rustflags` more correctly and warns if that's not enough for convergence. #11114
- Final artifacts not removed by linker should be removed before a compilation gets started. #11122
- `cargo add` now reports unknown features in a more discoverable manner. #11098
- Cargo now reports command aliasing failure with more error contexts. #11087
- A better error message when `cargo login` prompt receives empty input. #11145
- A better error message for fields with wrong types where workspace inheritance is supported. #11113
- A better error message when mixing feature syntax `dep:` with `/`. #11172
- A better error message when publishing but `package.publish` is `false` in the manifest. #11280

Nightly only

- Added new config option `publish.timeout` behind `-Zpublish-timeout`. docs #11230
- Added retry support to sparse registries. #11069
- Fixed sparse registry lockfile urls containing `registry+sparse+`. #11177
- Add new config option `registries.crates-io.protocol` for controlling crates.io protocol. #11215
- Removed `sparse+` prefix for index.crates.io. #11247
- Fixed publishing with a dependency on a sparse registry. #11268

- Fixed confusing error messages when using `-Zsparse-registry`. #11283
- Fixed 410 gone response handling for sparse registries. #11286

Cargo 1.65 (2022-11-03)

4fd148c4...rust-1.65.0

Added

- External subcommands can now inherit jobserver file descriptors from Cargo. #10511
- Added an API documentation for private items in cargo-the-library. See https://doc.rust-lang.org/nightly/nightly-rustc/cargo. #11019

Changed

- Cargo now stops adding its bin path to `PATH` if it's already there. #11023
- Improved the performance of Cargo build scheduling by sorting the queue of pending jobs. #11032
- Improved the performance fetching git dependencies from GitHub even when using a partial hash in the `rev` field. #10807
- Cargo now uses git2 v0.15 and libgit2-sys v0.14, which bring several compatibility fixes with git's new behaviors. #11004
- Registry index files are cached in a more granular way based on content hash. #11044
- Cargo now uses the standard library's `std::thread::scope` instead of the `crossbeam` crate for spawning scoped threads. #10977
- Cargo now uses the standard library's `available_parallelism` instead of the `num_cpus` crate for determining the default parallelism. #10969
- Cargo now guides you how to solve it when seeing an error message of `rust-version` requirement not satisfied. #10891
- Cargo now tells you more about possible causes and how to fix it when a subcommand cannot be found. #10924
- Cargo now lists available target names when a given Cargo target cannot be found. #10999

- `cargo update` now warns if `--precise` is given without `--package` flag. This will become a hard error after a transition period. #10988 #11011
- `cargo bench` and `cargo test` now report a more precise test execution error right after a test fails. #11028
- `cargo add` now tells you for which version the features are added. #11075
- Call out that non-ASCII crate names are not supported by Rust anymore. #11017
- Enhanced the error message when in the manifest a field is expected to be an array but a string is used. #10944

Fixed

- Removed the restriction on file locking supports on platforms other than Linux. #10975
- Fixed incorrect OS detection by bumping os_info to 3.5.0. #10943
- Scanning the package directory now ignores errors from broken but excluded symlink files. #11008
- Fixed deadlock when build scripts are waiting for input on stdin. #11257

Nightly

- Progress indicator for sparse registries becomes more straightforward. #11068

Cargo 1.64 (2022-09-22)

a5e08c47...rust-1.64.0

Added

- 🎉 Packages can now inherit settings from the workspace so that the settings can be centralized in one place. See `workspace.package` and `workspace.dependencies` for more details on how to define these common settings. #10859
- Added the `--crate-type` flag to `cargo rustc` to override the crate type. #10838
- Cargo commands can now accept multiple `--target` flags to build for multiple targets at once, and the `build.target` config option may now take an array of multiple targets. #10766
- The `--jobs` argument can now take a negative number to count backwards from the max CPUs. #10844

Changed

- Bash completion of `cargo install --path` now supports path completion. #10798
- Significantly improved the performance fetching git dependencies from GitHub when using a hash in the `rev` field. #10079
- Published packages will now include the resolver setting from the workspace to ensure that they use the same resolver when used in isolation. #10911 #10961 #10970
- `cargo add` will now update `Cargo.lock`. #10902
- The path in the config output of `cargo vendor` now translates backslashes to forward slashes so that the settings should work across platforms. #10668
- The `workspace.default-members` setting now allows a value of `"."` in a non-virtual workspace to refer to the root package. #10784

Fixed

- 🚨 CVE-2022-36113: Extracting malicious crates can corrupt arbitrary files. #11089 #11088
- 🚨 CVE-2022-36114: Extracting malicious crates can fill the file system. #11089 #11088
- The `os` output in `cargo --version --verbose` now supports more platforms. #10802
- Cached git checkouts will now be rebuilt if they are corrupted. This may happen when using `net.git-fetch-with-cli` and interrupting the clone process. #10829
- Fixed panic in `cargo add --offline`. #10817

Nightly only

- Fixed deserialization of unstable `check-cfg` in `config.toml`. #10799

Cargo 1.63 (2022-08-11)

3f052d8e...rust-1.63.0

Added

- 🎉 Added the `--config` CLI option to pass config options directly on the CLI. #10755
- The `CARGO_PKG_RUST_VERSION` environment variable is now set when compiling a crate if the manifest has the `rust-version` field set. #10713

Changed

- A warning is emitted when encountering multiple packages with the same name in a git dependency. This will ignore packages with `publish=false`. #10701 #10767
- Change tracking now uses the contents of a `.json` target spec file instead of its path. This should help avoid rebuilds if the path changes. #10746
- Git dependencies with a submodule configured with the `update=none` strategy in `.gitmodules` is now honored, and the submodule will not be fetched. #10717
- Crate files now use a more recent date (Jul 23, 2006 instead of Nov 29, 1973) for deterministic behavior. #10720
- The initial template used for `cargo new` now includes a slightly more realistic test structure that has `use super::*;` in the test module. #10706
- Updated the internal HTTP library libcurl with various small fixes and updates. #10696

Fixed

- Fix zsh completions for `cargo add` and `cargo locate-project` #10810 #10811

- Fixed `-p` being ignored with `cargo publish` in the root of a virtual workspace. Some additional checks were also added to generate an error if multiple packages were selected (previously it would pick the first one). #10677
- The human-readable executable name is no longer displayed for `cargo test` when using JSON output. #10691

Nightly only

- Added `-Zcheck-cfg=output` to support build-scripts declaring their supported set of `cfg` values with `cargo:rustc-check-cfg`. #10539
- `-Z sparse-registry` now uses https://index.crates.io/ when accessing crates-io. #10725
- Fixed formatting of `.workspace` key in `cargo add` for workspace inheritance. #10705
- Sparse HTTP registry URLs must now end with a `/`. #10698
- Fixed issue with `cargo add` and workspace inheritance of the `default-features` key. #10685

Cargo 1.62 (2022-06-30)

1ef1e0a1...rust-1.62.0

Added

- 🎉 Added the `cargo add` command for adding dependencies to `Cargo.toml` from the command-line. docs #10472 #10577 #10578
- Package ID specs now support `name@version` syntax in addition to the previous `name:version` to align with the behavior in `cargo add` and other tools. `cargo install` and `cargo yank` also now support this syntax so the version does not need to passed as a separate flag. #10582 #10650 #10597
- Added the CLI option `-F` as an alias of `--features`. #10576
- The `git` and `registry` directories in Cargo's home directory (usually `~/.cargo`) are now marked as cache directories so that they are not included in backups or content indexing (on Windows). #10553
- Added the `--version` flag to `cargo yank` to replace the `--vers` flag to be consistent with `cargo install`. #10575
- Added automatic `@` argfile support, which will use "response files" if the command-line to `rustc` exceeds the operating system's limit. #10546
- `cargo clean` now has a progress bar (if it takes longer than half a second). #10236

Changed

- `cargo install` no longer generates an error if no binaries were found to install (such as missing required features). #10508
- `cargo test` now passes `--target` to `rustdoc` if the specified target is the same as the host target. #10594
- `cargo doc` now automatically passes `-Arustdoc::private-intra-doc-links` when documenting a binary (which automatically includes `--document-private-items`). The `private-intra-doc-links` lint is

only relevant when *not* documenting private items, which doesn't apply to binaries. #10142
- The length of the short git hash in the `cargo --version` output is now fixed to 9 characters. Previously the length was inconsistent between different platforms. #10579
- Attempting to publish a package with a `Cargo.toml.orig` file will now result in an error. The filename would otherwise conflict with the automatically-generated file. #10551

Fixed

- The `build.dep-info-basedir` configuration setting now properly supports the use of `..` in the path to refer to a parent directory. #10281
- Fixed regression in automatic detection of the default number of CPUs to use on systems using cgroups v1. #10737 #10739

Nightly only

- `cargo fetch` now works with `-Zbuild-std` to fetch the standard library's dependencies. #10129
- Added support for workspace inheritance. docs #10584 #10568 #10565 #10564 #10563 #10606 #10548 #10538
- Added `-Zcheck-cfg` which adds various forms of validating `cfg` expressions for unknown names and values. docs #10486 #10566
- The `--config` CLI option no longer allows setting a registry token. #10580
- Fixed issues with proc-macros and `-Z rustdoc-scrape-examples`. #10549 #10533

Cargo 1.61 (2022-05-19)

ea2a21c9...rust-1.61.0

Added

Changed

- `cargo test --no-run` will now display the path to the test executables. #10346
- `cargo tree --duplicates` no longer reports dependencies that are shared between the host and the target as duplicates. #10466
- Updated to the 1.4.2 release of libgit2 which brings in several fixes #10442 #10479
- `cargo vendor` no longer allows multiple values for `--sync`, you must pass multiple `--sync` flags instead. #10448
- Warnings are now issued for manifest keys that have mixed both underscore and dash variants (such as specifying both `proc_macro` and `proc-macro`) #10316
- Cargo now uses the standard library's `available_parallelism` instead of the `num_cpus` crate for determining the default parallelism. #10427
- `cargo search` terms are now highlighted. #10425

Fixed

- Paths passed to VCS tools like `hg` are now added after `--` to avoid conflict with VCS flags. #10483
- Fixed the `http.timeout` configuration value to actually work. #10456
- Fixed issues with `cargo rustc --crate-type` not working in some situations. #10388

Nightly only

- Added `-Z check-cfg-features` to enable compile-time checking of features #10408
- Added `-Z bindeps` to support binary artifact dependencies (RFC-3028) #9992
- `-Z multitarget` is now supported in the `build.target` config value with an array. #10473
- Added `--keep-going` flag which will continue compilation even if one crate fails to compile. #10383
- Start work on inheriting manifest values in a workspace. #10497 #10517
- Added support for sparse HTTP registries. #10470 #10064
- Fixed panic when artifact target is used for `[target.'cfg(<target>)'.dependencies]` #10433
- Fixed host flags to pass to build scripts (`-Z target-applies-to-host`) #10395
- Added `-Z check-cfg-features` support for rustdoc #10428

Cargo 1.60 (2022-04-07)

358e79fe...rust-1.60.0

Added

- 🎉 Added the `dep:` prefix in the `[features]` table to refer to an optional dependency. This allows creating feature names with the same name as a dependency, and allows for "hiding" optional dependencies so that they do not implicitly expose a feature name. docs #10269
- 🎉 Added the `dep-name?/feature-name` syntax to the `[features]` table to only enable the feature `feature-name` if the optional dependency `dep-name` is already enabled by some other feature. docs #10269
- 🎉 Added `--timings` option to generate an HTML report about build timing, concurrency, and CPU use. docs #10245
- Added the `"v"` and `"features2"` fields to the registry index. The `"v"` field provides a method for compatibility with future changes to the index. docs #10269
- Added bash completion for `cargo clippy` #10347
- Added bash completion for `cargo report` #10295
- Added support to build scripts for `rustc-link-arg-tests`, `rustc-link-arg-examples`, and `rustc-link-arg-benches`. docs #10274

Changed

- Cargo now uses the clap 3 library for command-line argument parsing. #10265
- The `build.pipelining` config option is now deprecated, pipelining will now always be enabled. #10258
- `cargo new` will now generate a `.gitignore` which only ignores `Cargo.lock` in the root of the repo, instead of any directory. #10379
- Improved startup time of bash completion. #10365

- The `--features` flag is now honored when used with the `--all-features` flag, which allows enabling features from other packages. #10337
- Cargo now uses a different TOML parser. This should not introduce any user-visible changes. This paves the way to support format-preserving programmatic modification of TOML files for supporting `cargo add` and other future enhancements. #10086
- Setting a library to emit both a `dylib` and `cdylib` is now an error, as this combination is not supported. #10243
- `cargo --list` now includes the `help` command. #10300

Fixed

- Fixed running `cargo doc` on examples with dev-dependencies. #10341
- Fixed `cargo install --path` for a path that is relative to a directory outside of the workspace in the current directory. #10335
- `cargo test TEST_FILTER` should no longer build binaries that are explicitly disabled with `test = false`. #10305
- Fixed regression with `term.verbose` without `term.quiet`, and vice versa. #10429 #10436

Nightly only

- Added `rustflags` option to a profile definition. #10217
- Changed `--config` to only support dotted keys. #10176
- Fixed profile `rustflags` not being gated in profile overrides. #10411 #10413

Cargo 1.59 (2022-02-24)

7f08ace4...rust-1.59.0

Added

- 🎉 The `strip` option can now be specified in a profile to specify the behavior for removing symbols and debug information from binaries. docs #10088 #10376
- 🎉 Added future incompatible reporting. This provides reporting for when a future change in `rustc` may cause a package or any of its dependencies to stop building. docs #10165
- SSH authentication on Windows now supports ssh-agent. docs #10248
- Added `term.quiet` configuration option to enable the `--quiet` behavior from a config file. docs #10152
- Added `-r` CLI option as an alias for `--release`. #10133

Changed

- Scanning the package directory should now be resilient to errors, such as filesystem loops or access issues. #10188 #10214 #10286
- `cargo help <alias>` will now show the target of the alias. #10193
- Removed the deprecated `--host` CLI option. #10145 #10327
- Cargo should now report its version to always be in sync with `rustc`. #10178
- Added EOPNOTSUPP to ignored file locking errors, which is relevant to BSD operating systems. #10157

Fixed

- macOS: Fixed an issue where running an executable would sporadically be killed by the kernel (likely starting in macOS 12). #10196
- Fixed so that the `doc=false` setting is honored in the `[lib]` definition of a dependency. #10201 #10324

- The `"executable"` field in the JSON option was incorrectly including the path to `index.html` when documenting a binary. It is now null. #10171
- Documenting a binary now waits for the package library to finish documenting before starting. This fixes some race conditions if the binary has intra-doc links to the library. #10172
- Fixed panic when displaying help text to a closed pipe. #10164

Nightly only

- Added the `--crate-type` flag to `cargo rustc`. #10093

Cargo 1.58 (2022-01-13)

b2e52d7c...rust-1.58.0

Added

- Added `rust_version` field to package data in `cargo metadata`. #9967
- Added `--message-format` option to `cargo install`. #10107

Changed

- A warning is now shown when an alias shadows an external command. #10082
- Updated curl to 7.80.0. #10040 #10106

Fixed

- Doctests now include rustc-link-args from build scripts. #9916
- Fixed `cargo tree` entering an infinite loop with cyclical dev-dependencies. Fixed an edge case where the resolver would fail to handle a cyclical dev-dependency with a feature. #10103
- Fixed `cargo clean -p` when the directory path contains glob characters. #10072
- Fixed debug builds of `cargo` which could panic when downloading a crate when the server has a redirect with a non-empty body. #10048

Nightly only

- Make future-incompat-report output more user-friendly. #9953
- Added support to scrape code examples from the `examples` directory to be included in the documentation. docs #9525 #10037 #10017
- Fixed `cargo report future-incompatibilities` to check stdout if it supports color. #10024

Cargo 1.57 (2021-12-02)

18751dd3...rust-1.57.0

Added

- 🎉 Added custom named profiles. This also changes the `test` and `bench` profiles to inherit their settings from `dev` and `release`, and Cargo will now only use a single profile during a given command instead of using different profiles for dependencies and cargo-targets. docs #9943
- The `rev` option for a git dependency now supports git references that start with `refs/`. An example where this can be used is to depend on a pull request from a service like GitHub before it is merged. #9859
- Added `path_in_vcs` field to the `.cargo_vcs_info.json` file. docs #9866

Changed

- ❗ `RUSTFLAGS` is no longer set for build scripts. This change was made in 1.55, but the release notes did not highlight this change. Build scripts should use `CARGO_ENCODED_RUSTFLAGS` instead. See the documentation for more details.
- The `cargo version` command now includes some extra information. #9968
- Updated libgit2 to 1.3 which brings in a number of fixes and changes to git handling. #9963 #9988
- Shell completions now include shorthand b/r/c/d subcommands. #9951
- `cargo update --precise` now allows specifying a version without semver metadata (stuff after `+` in the version number). #9945
- zsh completions now complete `--example` names. #9939
- The progress bar now differentiates when building unittests. #9934
- Some backwards-compatibility support for invalid TOML syntax has been removed. #9932

- Reverted the change from 1.55 that triggered an error for dependency specifications that did not include any fields. #9911

Fixed

- Removed a log message (from `CARGO_LOG`) that may leak tokens. #9873
- `cargo fix` will now avoid writing fixes to the global registry cache. #9938
- Fixed `-Z help` CLI option when used with a shorthand alias (b/c/r/d). #9933

Nightly only

Cargo 1.56 (2021-10-21)

cebef295...rust-1.56.0

Added

- 🎉 Cargo now supports the 2021 edition. More information may be found in the edition guide. #9800
- 🎉 Added the `rust-version` field to `Cargo.toml` to specify the minimum supported Rust version, and the `--ignore-rust-version` command line option to override it. #9732
- Added the `[env]` table to config files to specify environment variables to set. docs #9411
- `[patch]` tables may now be specified in config files. docs #9839
- `cargo doc` now supports the `--example` and `--examples` flags. #9808
- 🎉 Build scripts can now pass additional linker arguments for binaries or all linkable targets. docs #9557
- Added support for the `-p` flag for `cargo publish` to publish a specific package in a workspace. `cargo package` also now supports `-p` and `--workspace`. #9559
- Added documentation about third-party registries. #9830
- Added the `{sha256-checksum}` placeholder for URLs in a registry `config.json`. docs #9801
- Added a warning when a dependency does not have a library. #9771

Changed

- Doc tests now support the `-q` flag to show terse test output. #9730
- `features` used in a `[replace]` table now issues a warning, as they are ignored. #9681
- Changed so that only `wasm32-unknown-emscripten` executables are built without a hash in the filename. Previously it was all `wasm32` targets. Additionally, all `apple` binaries are now built with a hash in

the filename. This allows multiple copies to be cached at once, and matches the behavior on other platforms (except `msvc`). #9653
- `cargo new` now generates an example that doesn't generate a warning with clippy. #9796
- `cargo fix --edition` now only applies edition-specific lints. #9846
- Improve resolver message to include dependency requirements. #9827
- `cargo fix` now has more debug logging available with the `CARGO_LOG` environment variable. #9831
- Changed `cargo fix --edition` to emit a warning when on the latest stable edition when running on stable instead of generating an error. #9792
- `cargo install` will now determine all of the packages to install before starting the installation, which should help with reporting errors without partially installing. #9793
- The resolver report for `cargo fix --edition` now includes differences for dev-dependencies. #9803
- `cargo fix` will now show better diagnostics for abnormal errors from `rustc`. #9799
- Entries in `cargo --list` are now deduplicated. #9773
- Aliases are now included in `cargo --list`. #9764

Fixed

- Fixed panic with build-std of a proc-macro. #9834
- Fixed running `cargo` recursively from proc-macros while running `cargo fix`. #9818
- Return an error instead of a stack overflow for command alias loops. #9791
- Updated to curl 7.79.1, which will hopefully fix intermittent http2 errors. #9937

Nightly only

- Added `[future-incompat-report]` config section. #9774

- Fixed value-after-table error with custom named profiles. #9789
- Added the `different-binary-name` feature to support specifying a non-rust-identifier for a binary name. docs #9627
- Added a profile option to select the codegen backend. docs #9118

Cargo 1.55 (2021-09-09)

aa8b0929...rust-1.55.0

Added

- The package definition in `cargo metadata` now includes the `"default_run"` field from the manifest. #9550
- ❗ Build scripts now have access to the following environment variables: `RUSTC_WRAPPER`, `RUSTC_WORKSPACE_WRAPPER`, `CARGO_ENCODED_RUSTFLAGS`. `RUSTFLAGS` is no longer set for build scripts; they should use `CARGO_ENCODED_RUSTFLAGS` instead. docs #9601
- Added `cargo d` as an alias for `cargo doc`. #9680
- Added `{lib}` to the `cargo tree --format` option to display the library name of a package. #9663
- Added `members_mut` method to the `Workspace` API. #9547

Changed

- If a build command does not match any targets when using the `--all-targets`, `--bins`, `--tests`, `--examples`, or `--benches` flags, a warning is now displayed to inform you that there were no matching targets. #9549
- The way `cargo init` detects whether or not existing source files represent a binary or library has been changed to respect the command-line flags instead of trying to guess which type it is. #9522
- Registry names are now displayed instead of registry URLs when possible. #9632
- Duplicate compiler diagnostics are no longer shown. This can often happen with `cargo test` which builds multiple copies of the same code in parallel. This also updates the warning summary to provide more context. #9675
- The output for warnings or errors is now improved to be leaner, cleaner, and show more context. #9655

- Network send errors are now treated as "spurious" which means they will be retried. #9695
- Git keys (`branch`, `tag`, `rev`) on a non-git dependency are now an error. Additionally, specifying both `git` and `path` is now an error. #9689
- Specifying a dependency without any keys is now an error. #9686
- The resolver now prefers to use `[patch]` table entries of dependencies when possible. #9639
- Package name typo errors in dependencies are now displayed aligned with the original to help make it easier to see the difference. #9665
- Windows platforms may now warn on environment variables that have the wrong case. #9654
- `features` used in a `[patch]` table now issues a warning, as they are ignored. #9666
- The `target` directory is now excluded from content indexing on Windows. #9635
- When `Cargo.toml` is not found, the error message now detects if it was misnamed with a lowercase `c` to suggest the correct form. #9607
- Building `diesel` with the new resolver displays a compatibility notice. #9602
- Updated the `opener` dependency, which handles opening a web browser, which includes several changes, such as new behavior when run on WSL, and using the system `xdg-open` on Linux. #9583
- Updated to libcurl 7.78. #9809 #9810

Fixed

- Fixed dep-info files including non-local build script paths. #9596
- Handle "jobs = 0" case in cargo config files #9584
- Implement warning for ignored trailing arguments after `--` #9561
- Fixed rustc/rustdoc config values to be config-relative. #9566
- `cargo fix` now supports rustc's suggestions with multiple spans. #9567

- `cargo fix` now fixes each target serially instead of in parallel to avoid problems with fixing the same file concurrently. #9677
- Changes to the target `linker` config value now trigger a rebuild. #9647
- Git unstaged deleted files are now ignored when using the `--allow-dirty` flag with `cargo publish` or `cargo package`. #9645

Nightly only

- Enabled support for `cargo fix --edition` for 2021. #9588
- Several changes to named profiles. #9685
- Extended instructions on what to do when running `cargo fix --edition` on the 2021 edition. #9694
- Multiple updates to error messages using nightly features to help better explain the situation. #9657
- Adjusted the edition 2021 resolver diff report. #9649
- Fixed error using `cargo doc --open` with `doc.extern-map`. #9531
- Unified weak and namespaced features. #9574
- Various updates to future-incompatible reporting. #9606
- `[env]` environment variables are not allowed to set vars set by Cargo. #9579

Cargo 1.54 (2021-07-29)

4369396c...rust-1.54.0

Added

- Fetching from a git repository (such as the crates.io index) now displays the network transfer rate. #9395
- Added `--prune` option for `cargo tree` to limit what is displayed. #9520
- Added `--depth` option for `cargo tree` to limit what is displayed. #9499
- Added `cargo tree -e no-proc-macro` to hide procedural macro dependencies. #9488
- Added `doc.browser` config option to set which browser to open with `cargo doc --open`. #9473
- Added `CARGO_TARGET_TMPDIR` environment variable set for integration tests & benches. This provides a temporary or "scratch" directory in the `target` directory for tests and benches to use. #9375

Changed

- `--features` CLI flags now provide typo suggestions with the new feature resolver. #9420
- Cargo now uses a new parser for SemVer versions. This should behave mostly the same as before with some minor exceptions where invalid syntax for version requirements is now rejected. #9508
- Mtime handling of `.crate` published packages has changed slightly to avoid mtime values of 0. This was causing problems with lldb which refused to read those files. #9517
- Improved performance of git status check in `cargo package`. #9478
- `cargo new` with fossil now places the ignore settings in the new repository instead of using `fossil settings` to set them globally. This also includes several other cleanups to make it more consistent with other VCS configurations. #9469

- `rustc-cdylib-link-arg` applying transitively displays a warning that this was not intended, and may be an error in the future. #9563

Fixed

- Fixed `package.exclude` in `Cargo.toml` using inverted exclusions (`!somefile`) when not in a git repository or when vendoring a dependency. #9186
- Dep-info files now adjust build script `rerun-if-changed` paths to be absolute paths. #9421
- Fixed a bug when with resolver = "1" non-virtual package was allowing unknown features. #9437
- Fixed an issue with the index cache mishandling versions that only differed in build metadata (such as `110.0.0` and `110.0.0+1.1.0f`). #9476
- Fixed `cargo install` with a semver metadata version. #9467

Nightly only

- Added `report` subcommand, and changed `cargo describe-future-incompatibilitie` to `cargo report future-incompatibilities`. #9438
- Added a `[host]` table to the config files to be able to set build flags for host target. Also added `target-applies-to-host` to control how the `[target]` tables behave. #9322
- Added some validation to build script `rustc-link-arg-*` instructions to return an error if the target doesn't exist. #9523
- Added `cargo:rustc-link-arg-bin` instruction for build scripts. #9486

Cargo 1.53 (2021-06-17)

90691f2b...rust-1.53.0

Added

Changed

- 🔥 Cargo now supports git repositories where the default `HEAD` branch is not "master". This also includes a switch to the version 3 `Cargo.lock` format which can handle default branches correctly. #9133 #9397 #9384 #9392
- 🔥 macOS targets now default to `unpacked` split-debuginfo. #9298
- ❗ The `authors` field is no longer included in `Cargo.toml` for new projects. #9282
- `cargo update` may now work with the `--offline` flag. #9279
- `cargo doc` will now erase the `doc` directory when switching between different toolchain versions. There are shared, unversioned files (such as the search index) that can become broken when using different versions. #8640 #9404
- Improved error messages when path dependency/workspace member is missing. #9368

Fixed

- Fixed `cargo doc` detecting if the documentation needs to be rebuilt when changing some settings such as features. #9419
- `cargo doc` now deletes the output directory for the package before running rustdoc to clear out any stale files. #9419
- Fixed the `-C metadata` value to always include all information for all builds. Previously, in some situations, the hash only included the package name and version. This fixes some issues, such as incremental builds with split-debuginfo on macOS corrupting the incremental cache in some cases. #9418
- Fixed man pages not working on Windows if `man` is in `PATH`. #9378

- The `rustc` cache is now aware of `RUSTC_WRAPPER` and `RUSTC_WORKSPACE_WRAPPER`. #9348
- Track the `CARGO` environment variable in the rebuild fingerprint if the code uses `env!("CARGO")`. #9363

Nightly only

- Fixed config includes not working. #9299
- Emit note when `--future-incompat-report` had nothing to report. #9263
- Error messages for nightly features flags (like `-Z` and `cargo-features`) now provides more information. #9290
- Added the ability to set the target for an individual package in `Cargo.toml`. docs #9030
- Fixed build-std updating the index on every build. #9393
- `-Z help` now displays all the `-Z` options. #9369
- Added `-Zallow-features` to specify which nightly features are allowed to be used. #9283
- Added `cargo config` subcommand. #9302

Cargo 1.52 (2021-05-06)

34170fcd...rust-1.52.0

Added

- Added the `"manifest_path"` field to JSON messages for a package. #9022 #9247

Changed

- Build scripts are now forbidden from setting `RUSTC_BOOTSTRAP` on stable. #9181 #9385
- crates.io now supports SPDX 3.11 licenses. #9209
- An error is now reported if `CARGO_TARGET_DIR` is an empty string. #8939
- Doc tests now pass the `--message-format` flag into the test so that the "short" format can now be used for doc tests. #9128
- `cargo test` now prints a clearer indicator of which target is currently running. #9195
- The `CARGO_TARGET_<TRIPLE>` environment variable will now issue a warning if it is using lowercase letters. #9169

Fixed

- Fixed publication of packages with metadata and resolver fields in `Cargo.toml`. #9300 #9304
- Fixed logic for determining prefer-dynamic for a dylib which differed in a workspace vs a single package. #9252
- Fixed an issue where exclusive target-specific dependencies that overlapped across dependency kinds (like regular and build-dependencies) would incorrectly include the dependencies in both. #9255
- Fixed panic with certain styles of Package IDs when passed to the `-p` flag. #9188

- When running cargo with output not going to a TTY, and with the progress bar and color force-enabled, the output will now correctly clear the progress line. #9231
- Error instead of panic when JSON may contain non-utf8 paths. #9226
- Fixed a hang that can happen on broken stderr. #9201
- Fixed thin-local LTO not being disabled correctly when `lto=off` is set. #9182

Nightly only

- The `strip` profile option now supports `true` and `false` values. #9153
- `cargo fix --edition` now displays a report when switching to 2021 if the new resolver changes features. #9268
- Added `[patch]` table support in `.cargo/config` files. #9204
- Added `cargo describe-future-incompatibilities` for generating a report on dependencies that contain future-incompatible warnings. #8825
- Added easier support for testing the 2021 edition. #9184
- Switch the default resolver to "2" in the 2021 edition. #9184
- `cargo fix --edition` now supports 2021. #9184
- Added `--print` flag to `cargo rustc` to pass along to `rustc` to display information from rustc. #9002
- Added `-Zdoctest-in-workspace` for changing the directory where doctests are *run* versus where they are *compiled*. #9105
- Added support for an `[env]` section in `.cargo/config.toml` to set environment variables when running cargo. #9175
- Added a schema field and `features2` field to the index. #9161
- Changes to JSON spec targets will now trigger a rebuild. #9223

Cargo 1.51 (2021-03-25)

75d5d8cf...rust-1.51.0

Added

- 🔥 Added the `split-debuginfo` profile option. docs #9112
- Added the `path` field to `cargo metadata` for the package dependencies list to show the path for "path" dependencies. #8994
- 🔥 Added a new feature resolver, and new CLI feature flag behavior. See the new features and resolver documentation for the `resolver = "2"` option. See the CLI and resolver 2 CLI options for the new CLI behavior. And, finally, see RFC 2957 for a detailed look at what has changed. #8997

Changed

- `cargo install --locked` now emits a warning if `Cargo.lock` is not found. #9108
- Unknown or ambiguous package IDs passed on the command-line now display suggestions for the correct package ID. #9095
- Slightly optimize `cargo vendor` #8937 #9131 #9132

Fixed

- Fixed environment variables and cfg settings emitted by a build script that are set for `cargo test` and `cargo run` when the build script runs multiple times during the same build session. #9122
- Fixed a panic with `cargo doc` and the new feature resolver. This also introduces some heuristics to try to avoid path collisions with `rustdoc` by only documenting one variant of a package if there are multiple (such as multiple versions, or the same package shared for host and target platforms). #9077
- Fixed a bug in Cargo's cyclic dep graph detection that caused a stack overflow. #9075

- Fixed build script `links` environment variables (`DEP_*`) not showing up for testing packages in some cases. #9065
- Fixed features being selected in a nondeterministic way for a specific scenario when building an entire workspace with all targets with a proc-macro in the workspace with `resolver="2"`. #9059
- Fixed to use `http.proxy` setting in `~/.gitconfig`. #8986
- Fixed --feature pkg/feat for V1 resolver for non-member. #9275 #9277
- Fixed panic in `cargo doc` when there are colliding output filenames in a workspace. #9276 #9277
- Fixed `cargo install` from exiting with success if one of several packages did not install successfully. #9185 #9196
- Fix panic with doc collision orphan. #9142 #9196

Nightly only

- Removed the `publish-lockfile` unstable feature, it was stabilized without the need for an explicit flag 1.5 years ago. #9092
- Added better diagnostics, help messages, and documentation for nightly features (such as those passed with the `-Z` flag, or specified with `cargo-features` in `Cargo.toml`). #9092
- Added support for Rust edition 2021. #8922
- Added support for the `rust-version` field in project metadata. #8037
- Added a schema field to the index. #9161 #9196

Cargo 1.50 (2021-02-11)

8662ab42...rust-1.50.0

Added

- Added the `doc` field to `cargo metadata`, which indicates if a target is documented. #8869
- Added `RUSTC_WORKSPACE_WRAPPER`, an alternate RUSTC wrapper that only runs for the local workspace packages, and caches its artifacts independently of non-wrapped builds. #8976
- Added `--workspace` to `cargo update` to update only the workspace members, and not their dependencies. This is particularly useful if you update the version in `Cargo.toml` and want to update `Cargo.lock` without running any other commands. #8725

Changed

- `.crate` files uploaded to a registry are now built with reproducible settings, so that the same `.crate` file created on different machines should be identical. #8864
- Git dependencies that specify more than one of `branch`, `tag`, or `rev` are now rejected. #8984
- The `rerun-if-changed` build script directive can now point to a directory, in which case Cargo will check if any file in that directory changes. #8973
- If Cargo cannot determine the username or email address, `cargo new` will no longer fail, and instead create an empty authors list. #8912
- The progress bar width has been reduced to provide more room to display the crates currently being built. #8892
- `cargo new` will now support `includeIf` directives in `.gitconfig` to match the correct directory when determining the username and email address. #8886

Fixed

- Fixed `cargo metadata` and `cargo tree` to only download packages for the requested target. #8987
- Updated libgit2, which brings in many fixes, particularly fixing a zlib error that occasionally appeared on 32-bit systems. #8998
- Fixed stack overflow with a circular dev-dependency that uses the `links` field. #8969
- Fixed `cargo publish` failing on some filesystems, particularly 9p on WSL2. #8950

Nightly only

- Allow `resolver="1"` to specify the original feature resolution behavior. #8857
- Added `-Z extra-link-arg` which adds the `cargo:rustc-link-arg-bins` and `cargo:rustc-link-arg` build script options. docs #8441
- Implemented external credential process support, and added `cargo logout`. (RFC 2730) (docs) #8934
- Fix panic with `-Zbuild-std` and no roots. #8942
- Set docs.rs as the default extern-map for crates.io #8877

Cargo 1.49 (2020-12-31)

75615f8e...rust-1.49.0

Added

- Added `homepage` and `documentation` fields to `cargo metadata`. #8744
- Added the `CARGO_PRIMARY_PACKAGE` environment variable which is set when running `rustc` if the package is one of the "root" packages selected on the command line. #8758
- Added support for Unix-style glob patterns for package and target selection flags on the command-line (such as `-p 'serde*'` or `--test '*'`). #8752

Changed

- Computed LTO flags are now included in the filename metadata hash so that changes in LTO settings will independently cache build artifacts instead of overwriting previous ones. This prevents rebuilds in some situations such as switching between `cargo build` and `cargo test` in some circumstances. #8755
- `cargo tree` now displays `(proc-macro)` next to proc-macro packages. #8765
- Added a warning that the allowed characters for a feature name have been restricted to letters, digits, `_`, `-`, and `+` to accommodate future syntax changes. This is still a superset of the allowed syntax on crates.io, which requires ASCII. This is intended to be changed to an error in the future. #8814
- `-p` without a value will now print a list of workspace package names. #8808
- Add period to allowed feature name characters. #8932 #8943

Fixed

- Fixed building a library with both "dylib" and "rlib" crate types with LTO enabled. #8754
- Fixed paths in Cargo's dep-info files. #8819
- Fixed inconsistent source IDs in `cargo metadata` for git dependencies that explicitly specify `branch="master"`. #8824
- Fixed re-extracting dependencies which contained a `.cargo-ok` file. #8835

Nightly only

- Fixed a panic with `cargo doc -Zfeatures=itarget` in some situations. #8777
- New implementation for namespaced features, using the syntax `dep:serde`. docs #8799
- Added support for "weak" dependency features, using the syntax `dep_name?/feat_name`, which will enable a feature for a dependency without also enabling the dependency. #8818
- Fixed the new feature resolver downloading extra dependencies that weren't strictly necessary. #8823

Cargo 1.48 (2020-11-19)

51b66125...rust-1.48.0

Added

- Added `term.progress` configuration option to control when and how the progress bar is displayed. docs #8165
- Added `--message-format plain` option to `cargo locate-project` to display the project location without JSON to make it easier to use in a script. #8707
- Added `--workspace` option to `cargo locate-project` to display the path to the workspace manifest. #8712
- A new contributor guide has been added for contributing to Cargo itself. This is published at https://rust-lang.github.io/cargo/contrib/. #8715
- Zsh `--target` completion will now complete with the built-in rustc targets. #8740

Changed

Fixed

- Fixed `cargo new` creating a fossil repository to properly ignore the `target` directory. #8671
- Don't show warnings about the workspace in the current directory when using `cargo install` of a remote package. #8681
- Automatically reinitialize the index when an "Object not found" error is encountered in the git repository. #8735
- Updated libgit2, which brings in several fixes for git repository handling. #8778 #8780

Nightly only

- Fixed `cargo install` so that it will ignore the `[unstable]` table in local config files. #8656

- Fixed nondeterministic behavior of the new feature resolver. #8701
- Fixed running `cargo test` on a proc-macro with the new feature resolver under a specific combination of circumstances. #8742

Cargo 1.47 (2020-10-08)

4f74d9b2...rust-1.47.0

Added

- `cargo doc` will now include the package's version in the left sidebar. #8509
- Added the `test` field to `cargo metadata` targets. #8478
- Cargo's man pages are now displayed via the `cargo help` command (such as `cargo help build`). #8456 #8577
- Added new documentation chapters on how dependency resolution works and SemVer compatibility, along with suggestions on how to version your project and work with dependencies. #8609

Changed

- The comments added to `.gitignore` when it is modified have been tweaked to add some spacing. #8476
- `cargo metadata` output should now be sorted to be deterministic. #8489
- By default, build scripts and proc-macros are now built with `opt-level=0` and the default codegen units, even in release mode. #8500
- `workspace.default-members` is now filtered by `workspace.exclude`. #8485
- `workspace.members` globs now ignore non-directory paths. #8511
- git zlib errors now trigger a retry. #8520
- "http" class git errors now trigger a retry. #8553
- git dependencies now override the `core.autocrlf` git configuration value to ensure they behave consistently across platforms, particularly when vendoring git dependencies on Windows. #8523
- If `Cargo.lock` needs to be updated, then it will be automatically transitioned to the new V2 format. This format removes the `[metadata]` table, and should be easier to merge changes in source

control systems. This format was introduced in 1.38, and made the default for new projects in 1.41. #8554
- Added preparation for support of git repositories with a non-"master" default branch. Actual support will arrive in a future version. This introduces some warnings:
    - Warn if a git dependency does not specify a branch, and the default branch on the repository is not "master". In the future, Cargo will fetch the default branch. In this scenario, the branch should be explicitly specified.
    - Warn if a workspace has multiple dependencies to the same git repository, one without a `branch` and one with `branch="master"`. Dependencies should all use one form or the other. #8522
- Warnings are now issued if a `required-features` entry lists a feature that does not exist. #7950
- Built-in aliases are now included in `cargo --list`. #8542
- `cargo install` with a specific version that has been yanked will now display an error message that it has been yanked, instead of "could not find". #8565
- `cargo publish` with a package that has the `publish` field set to a single registry, and no `--registry` flag has been given, will now publish to that registry instead of generating an error. #8571

Fixed

- Fixed issue where if a project directory was moved, and one of the build scripts did not use the `rerun-if-changed` directive, then that build script was being rebuilt when it shouldn't. #8497
- Console colors should now work on Windows 7 and 8. #8540
- The `CARGO_TARGET_{triplet}_RUNNER` environment variable will now correctly override the config file instead of trying to merge the commands. #8629
- Fixed LTO with doctests. #8657 #8658

Nightly only

- Added support for `-Z terminal-width` which tells `rustc` the width of the terminal so that it can format diagnostics better. docs #8427
- Added ability to configure `-Z` unstable flags in config files via the `[unstable]` table. docs #8393
- Added `-Z build-std-features` flag to set features for the standard library. docs #8490

Cargo 1.46 (2020-08-27)

9fcb8c1d...rust-1.46.0

Added

- The `dl` key in `config.json` of a registry index now supports the replacement markers `{prefix}` and `{lowerprefix}` to allow spreading crates across directories similar to how the index itself is structured. docs #8267
- Added new environment variables that are set during compilation:
  - `CARGO_CRATE_NAME`: The name of the crate being built.
  - `CARGO_BIN_NAME`: The name of the executable binary (if this is a binary crate).
  - `CARGO_PKG_LICENSE`: The `license` field from the manifest.
  - `CARGO_PKG_LICENSE_FILE`: The `license-file` field from the manifest. #8270 #8325 #8387
- If the value for `readme` is not specified in `Cargo.toml`, it is now automatically inferred from the existence of a file named `README`, `README.md`, or `README.txt`. This can be suppressed by setting `readme = false`. #8277
- `cargo install` now supports the `--index` flag to install directly from an index. #8344
- Added the `metadata` table to the `workspace` definition in `Cargo.toml`. This can be used for arbitrary data similar to the `package.metadata` table. #8323
- Added the `--target-dir` flag to `cargo install` to set the target directory. #8391
- Changes to environment variables used by the `env!` or `option_env!` macros are now automatically detected to trigger a rebuild. #8421
- The `target` directory now includes the `CACHEDIR.TAG` file which is used by some tools to exclude the directory from backups. #8378
- Added docs about rustup's `+toolchain` syntax. #8455

Changed

- A warning is now displayed if a git dependency includes a `#` fragment in the URL. This was potentially confusing because Cargo itself displays git URLs with this syntax, but it does not have any meaning outside of the `Cargo.lock` file, and would not work properly. #8297
- Various optimizations and fixes for bitcode embedding and LTO. #8349
- Reduced the amount of data fetched for git dependencies. If Cargo knows the branch or tag to fetch, it will now only fetch that branch or tag instead of all branches and tags. #8363
- Enhanced git fetch error messages. #8409
- `.crate` files are now generated with GNU tar format instead of UStar, which supports longer file names. #8453

Fixed

- Fixed a rare situation where an update to `Cargo.lock` failed once, but then subsequent runs allowed it proceed. #8274
- Removed assertion that Windows dylibs must have a `.dll` extension. Some custom JSON spec targets may change the extension. #8310
- Updated libgit2, which brings in a fix for zlib errors for some remote git servers like googlesource.com. #8320
- Fixed the GitHub fast-path check for up-to-date git dependencies on non-master branches. #8363
- Fixed issue when enabling a feature with `pkg/feature` syntax, and `pkg` is an optional dependency, but also a dev-dependency, and the dev-dependency appears before the optional normal dependency in the registry summary, then the optional dependency would not get activated. #8395
- Fixed `clean -p` deleting the build directory if there is a test named `build`. #8398
- Fixed indentation of multi-line Cargo error messages. #8409

- Fixed issue where the automatic inclusion of the `--document-private-items` flag for rustdoc would override any flags passed to the `cargo rustdoc` command. #8449
- Cargo will now include a version in the hash of the fingerprint directories to support backwards-incompatible changes to the fingerprint structure. #8473 #8488

Nightly only

- Added `-Zrustdoc-map` feature which provides external mappings for rustdoc (such as https://docs.rs/ links). docs #8287
- Fixed feature calculation when a proc-macro is declared in `Cargo.toml` with an underscore (like `proc_macro = true`). #8319
- Added support for setting `-Clinker` with `-Zdoctest-xcompile`. #8359
- Fixed setting the `strip` profile field in config files. #8454

Cargo 1.45 (2020-07-16)

ebda5065e...rust-1.45.0

Added

Changed

- Changed official documentation to recommend `.cargo/config.toml` filenames (with the `.toml` extension). `.toml` extension support was added in 1.39. #8121
- The `registry.index` config value is no longer allowed (it has been deprecated for 4 years). #7973
- An error is generated if both `--index` and `--registry` are passed (previously `--index` was silently ignored). #7973
- The `registry.token` config value is no longer used with the `--index` flag. This is intended to avoid potentially leaking the crates.io token to another registry. #7973
- Added a warning if `registry.token` is used with source replacement. It is intended this will be an error in future versions. #7973
- Windows GNU targets now copy `.dll.a` import library files for DLL crate types to the output directory. #8141
- Dylibs for all dependencies are now unconditionally copied to the output directory. Some obscure scenarios can cause an old dylib to be referenced between builds, and this ensures that all the latest copies are used. #8139
- `package.exclude` can now match directory names. If a directory is specified, the entire directory will be excluded, and Cargo will not attempt to inspect it further. Previously Cargo would try to check every file in the directory which could cause problems if the directory contained unreadable files. #8095

- When packaging with `cargo publish` or `cargo package`, Cargo can use git to guide its decision on which files to include. Previously this git-based logic required a `Cargo.toml` file to exist at the root of the repository. This is no longer required, so Cargo will now use git-based guidance even if there is not a `Cargo.toml` in the root of the repository. #8095
- While unpacking a crate on Windows, if it fails to write a file because the file is a reserved Windows filename (like "aux.rs"), Cargo will display an extra message to explain why it failed. #8136
- Failures to set mtime on files are now ignored. Some filesystems did not support this. #8185
- Certain classes of git errors will now recommend enabling `net.git-fetch-with-cli`. #8166
- When doing an LTO build, Cargo will now instruct rustc not to perform codegen when possible. This may result in a faster build and use less disk space. Additionally, for non-LTO builds, Cargo will instruct rustc to not embed LLVM bitcode in libraries, which should decrease their size. #8192 #8226 #8254
- The implementation for `cargo clean -p` has been rewritten so that it can more accurately remove the files for a specific package. #8210
- The way Cargo computes the outputs from a build has been rewritten to be more complete and accurate. Newly tracked files will be displayed in JSON messages, and may be uplifted to the output directory in some cases. Some of the changes from this are:

  - `.exp` export files on Windows MSVC dynamic libraries are now tracked.
  - Proc-macros on Windows track import/export files.
  - All targets (like tests, etc.) that generate separate debug files (pdb/dSYM) are tracked.
  - Added .map files for wasm32-unknown-emscripten.
  - macOS dSYM directories are tracked for all dynamic libraries (dylib/cdylib/proc-macro) and for build scripts.

  There are a variety of other changes as a consequence of this:

    - Binary examples on Windows MSVC with a hyphen will now show up twice in the examples directory (`foo_bar.exe` and `foo-bar.exe`). Previously Cargo just renamed the file instead of hard-linking it.
    - Example libraries now follow the same rules for hyphen/underscore translation as normal libs (they will now use underscores).

  #8210
- Cargo attempts to scrub any secrets from the debug log for HTTP debugging. #8222
- Context has been added to many of Cargo's filesystem operations, so that error messages now provide more information, such as the path that caused the problem. #8232
- Several commands now ignore the error if stdout or stderr is closed while it is running. For example `cargo install --list | grep -q cargo-fuzz` would previously sometimes panic because `grep -q` may close stdout before the command finishes. Regular builds continue to fail if stdout or stderr is closed, matching the behavior of many other build systems. #8236
- If `cargo install` is given an exact version, like `--version=1.2.3`, it will now avoid updating the index if that version is already installed, and exit quickly indicating it is already installed. #8022
- Changes to the `[patch]` section will now attempt to automatically update `Cargo.lock` to the new version. It should now also provide better error messages for the rare cases where it is unable to automatically update. #8248

Fixed

- Fixed copying Windows `.pdb` files to the output directory when the filename contained dashes. #8123
- Fixed error where Cargo would fail when scanning if a package is inside a git repository when any of its ancestor paths is a symlink.

#8186
- Fixed `cargo update` with an unused `[patch]` so that it does not get stuck and refuse to update. #8243
- Fixed a situation where Cargo would hang if stderr is closed, and the compiler generated a large number of messages. #8247
- Fixed backtraces on macOS not showing filenames or line numbers. As a consequence of this, binary executables on apple targets do not include a hash in the filename in Cargo's cache. This means Cargo can only track one copy, so if you switch features or rustc versions, Cargo will need to rebuild the executable. #8329 #8335
- Fixed fingerprinting when using lld on Windows with a dylib. Cargo was erroneously thinking the dylib was never fresh. #8290 #8335

Nightly only

- Fixed passing the full path for `--target` to `rustdoc` when using JSON spec targets. #8094
- `-Cembed-bitcode=no` renamed to `-Cbitcode-in-rlib=no` #8134
- Added new `resolver` field to `Cargo.toml` to opt-in to the new feature resolver. #8129
- `-Zbuild-std` no longer treats std dependencies as "local". This means that it won't use incremental compilation for those dependencies, removes them from dep-info files, and caps lints at "allow". #8177
- Added `-Zmultitarget` which allows multiple `--target` flags to build the same thing for multiple targets at once. docs #8167
- Added `strip` option to the profile to remove symbols and debug information. docs #8246
- Fixed panic with `cargo tree --target=all -Zfeatures=all`. #8269

Cargo 1.44 (2020-06-04)

bda50510...rust-1.44.0

Added

- 🔥 Added the `cargo tree` command. docs #8062
- Added warnings if a package has Windows-restricted filenames (like `nul`, `con`, `aux`, `prn`, etc.). #7959
- Added a `"build-finished"` JSON message when compilation is complete so that tools can detect when they can stop listening for JSON messages with commands like `cargo run` or `cargo test`. #8069

Changed

- Valid package names are now restricted to Unicode XID identifiers. This is mostly the same as before, except package names cannot start with a number or `-`. #7959
- `cargo new` and `init` will warn or reject additional package names (reserved Windows names, reserved Cargo directories, non-ASCII names, conflicting std names like `core`, etc.). #7959
- Tests are no longer hard-linked into the output directory (`target/debug/`). This ensures tools will have access to debug symbols and execute tests in the same way as Cargo. Tools should use JSON messages to discover the path to the executable. #7965
- Updating git submodules now displays an "Updating" message for each submodule. #7989
- File modification times are now preserved when extracting a `.crate` file. This reverses the change made in 1.40 where the mtime was not preserved. #7935
- Build script warnings are now displayed separately when the build script fails. #8017
- Removed the `git-checkout` subcommand. #8040

- The progress bar is now enabled for all unix platforms. Previously it was only Linux, macOS, and FreeBSD. #8054
- Artifacts generated by pre-release versions of `rustc` now share the same filenames. This means that changing nightly versions will not leave stale files in the build directory. #8073
- Invalid package names are rejected when using renamed dependencies. #8090
- Added a certain class of HTTP2 errors as "spurious" that will get retried. #8102
- Allow `cargo package --list` to succeed, even if there are other validation errors (such as `Cargo.lock` generation problem, or missing dependencies). #8175 #8215

Fixed

- Cargo no longer buffers excessive amounts of compiler output in memory. #7838
- Symbolic links in git repositories now work on Windows. #7996
- Fixed an issue where `profile.dev` was not loaded from a config file with `cargo test` when the `dev` profile was not defined in `Cargo.toml`. #8012
- When a binary is built as an implicit dependency of an integration test, it now checks `dep_name/feature_name` syntax in `required-features` correctly. #8020
- Fixed an issue where Cargo would not detect that an executable (such as an integration test) needs to be rebuilt when the previous build was interrupted with Ctrl-C. #8087
- Protect against some (unknown) situations where Cargo could panic when the system monotonic clock doesn't appear to be monotonic. #8114
- Fixed panic with `cargo clean -p` if the package has a build script. #8216

Nightly only

- Fixed panic with new feature resolver and required-features. #7962
- Added `RUSTC_WORKSPACE_WRAPPER` environment variable, which provides a way to wrap `rustc` for workspace members only, and affects the filename hash so that artifacts produced by the wrapper are cached separately. This usage can be seen on nightly clippy with `cargo clippy -Zunstable-options`. #7533
- Added `--unit-graph` CLI option to display Cargo's internal dependency graph as JSON. #7977
- Changed `-Zbuild_dep` to `-Zhost_dep`, and added proc-macros to the feature decoupling logic. #8003 #8028
- Fixed so that `--crate-version` is not automatically passed when the flag is found in `RUSTDOCFLAGS`. #8014
- Fixed panic with `-Zfeatures=dev_dep` and `check --profile=test`. #8027
- Fixed panic with `-Zfeatures=itarget` with certain host dependencies. #8048
- Added support for `-Cembed-bitcode=no`, which provides a performance boost and disk-space usage reduction for non-LTO builds. #8066
- `-Zpackage-features` has been extended with several changes intended to make it easier to select features on the command-line in a workspace. #8074

Cargo 1.43 (2020-04-23)

9d32b7b0...rust-1.43.0

Added

- 🔥 Profiles may now be specified in config files (and environment variables). docs #7823
- ❗ Added `CARGO_BIN_EXE_<name>` environment variable when building integration tests. This variable contains the path to any `[[bin]]` targets in the package. Integration tests should use the `env!` macro to determine the path to a binary to execute. docs #7697

Changed

- `cargo install --git` now honors workspaces in a git repository. This allows workspace settings, like `[patch]`, `[replace]`, or `[profile]` to be used. #7768
- `cargo new` will now run `rustfmt` on the new files to pick up rustfmt settings like `tab_spaces` so that the new file matches the user's preferred indentation settings. #7827
- Environment variables printed with "very verbose" output (`-vv`) are now consistently sorted. #7877
- Debug logging for fingerprint rebuild-detection now includes more information. #7888 #7890 #7952
- Added warning during publish if the license-file doesn't exist. #7905
- The `license-file` file is automatically included during publish, even if it is not explicitly listed in the `include` list or is in a location outside of the root of the package. #7905
- `CARGO_CFG_DEBUG_ASSERTIONS` and `CARGO_CFG_PROC_MACRO` are no longer set when running a build script. These were inadvertently set in the past, but had no meaning as they were always true. Additionally, `cfg(proc-macro)` is no longer supported in a `target` expression. #7943 #7970

Fixed

- Global command-line flags now work with aliases (like `cargo -v b`). #7837
- Required-features using dependency syntax (like `renamed_dep/feat_name`) now handle renamed dependencies correctly. #7855
- Fixed a rare situation where if a build script is run multiple times during the same build, Cargo will now keep the results separate instead of losing the output of the first execution. #7857
- Fixed incorrect interpretation of environment variable `CARGO_TARGET_*_RUNNER=true` as a boolean. Also improved related env var error messages. #7891
- Updated internal libgit2 library, bringing various fixes to git support. #7939
- `cargo package` / `cargo publish` should no longer buffer the entire contents of each file in memory. #7946
- Ignore more invalid `Cargo.toml` files in a git dependency. Cargo currently walks the entire repo to find the requested package. Certain invalid manifests were already skipped, and now it should skip all of them. #7947

Nightly only

- Added `build.out-dir` config variable to set the output directory. #7810
- Added `-Zjobserver-per-rustc` feature to support improved performance for parallel rustc. #7731
- Fixed filename collision with `build-std` and crates like `cc`. #7860
- `-Ztimings` will now save its report even if there is an error. #7872
- Updated `--config` command-line flag to support taking a path to a config file to load. #7901
- Added new feature resolver. #7820

- Rustdoc docs now automatically include the version of the package in the side bar (requires `-Z crate-versions` flag). #7903

Cargo 1.42 (2020-03-12)

0bf7aafe...rust-1.42.0

Added

- Added documentation on git authentication. #7658
- Bitbucket Pipeline badges are now supported on crates.io. #7663
- `cargo vendor` now accepts the `--versioned-dirs` option to force it to always include the version number in each package's directory name. #7631
- The `proc_macro` crate is now automatically added to the extern prelude for proc-macro packages. This means that `extern crate proc_macro;` is no longer necessary for proc-macros. #7700

Changed

- Emit a warning if `debug_assertions`, `test`, `proc_macro`, or `feature=` is used in a `cfg()` expression. #7660
- Large update to the Cargo documentation, adding new chapters on Cargo targets, workspaces, and features. #7733
- Windows: `.lib` DLL import libraries are now copied next to the dll for all Windows MSVC targets. Previously it was only supported for `pc-windows-msvc`. This adds DLL support for `uwp-windows-msvc` targets. #7758
- The `ar` field in the `[target]` configuration is no longer read. It has been ignored for over 4 years. #7763
- Bash completion file simplified and updated for latest changes. #7789
- Credentials are only loaded when needed, instead of every Cargo command. #7774

Fixed

- Removed `--offline` empty index check, which was a false positive in some cases. #7655

- Files and directories starting with a `.` can now be included in a package by adding it to the `include` list. #7680
- Fixed `cargo login` removing alternative registry tokens when previous entries existed in the credentials file. #7708
- Fixed `cargo vendor` from panicking when used with alternative registries. #7718
- Fixed incorrect explanation in the fingerprint debug log message. #7749
- A `[source]` that is defined multiple times will now result in an error. Previously it was randomly picking a source, which could cause non-deterministic behavior. #7751
- `dep_kinds` in `cargo metadata` are now de-duplicated. #7756
- Fixed packaging where `Cargo.lock` was listed in `.gitignore` in a subdirectory inside a git repository. Previously it was assuming `Cargo.lock` was at the root of the repo. #7779
- Partial file transfer errors will now cause an automatic retry. #7788
- Linux: Fixed panic if CPU iowait stat decreases. #7803
- Fixed using the wrong sysroot for detecting host compiler settings when `--sysroot` is passed in via `RUSTFLAGS`. #7798

Nightly only

- `build-std` now uses `--extern` instead of `--sysroot` to find sysroot packages. #7699
- Added `--config` command-line option to set config settings. #7649
- Added `include` config setting which allows including another config file. #7649
- Profiles in config files now support any named profile. Previously it was limited to dev/release. #7750

Cargo 1.41 (2020-01-30)

5da4b4d4...rust-1.41.0

Added

- 🔥 Cargo now uses a new `Cargo.lock` file format. This new format should support easier merges in source control systems. Projects using the old format will continue to use the old format, only new `Cargo.lock` files will use the new format. #7579
- 🔥 `cargo install` will now upgrade already installed packages instead of failing. #7560
- 🔥 Profile overrides have been added. This allows overriding profiles for individual dependencies or build scripts. See the documentation for more. #7591
- Added new documentation for build scripts. #7565
- Added documentation for Cargo's JSON output. #7595
- Significant expansion of config and environment variable documentation. #7650
- Add back support for `BROWSER` environment variable for `cargo doc --open`. #7576
- Added `kind` and `platform` for dependencies in `cargo metadata`. #7132
- The `OUT_DIR` value is now included in the `build-script-executed` JSON message. #7622

Changed

- `cargo doc` will now document private items in binaries by default. #7593
- Subcommand typo suggestions now include aliases. #7486
- Tweak how the "already existing..." comment is added to `.gitignore`. #7570
- Ignore `cargo login` text from copy/paste in token. #7588

- Windows: Ignore errors for locking files when not supported by the filesystem. #7602
- Remove `**/*.rs.bk` from `.gitignore`. #7647

Fixed

- Fix unused warnings for some keys in the `build` config section. #7575
- Linux: Don't panic when parsing `/proc/stat`. #7580
- Don't show canonical path in `cargo vendor`. #7629

Nightly only

Cargo 1.40 (2019-12-19)

1c6ec66d...5da4b4d4

Added

- Added `http.ssl-version` config option to control the version of TLS, along with min/max versions. #7308
- 🔥 Compiler warnings are now cached on disk. If a build generates warnings, re-running the build will now re-display the warnings. #7450
- Added `--filter-platform` option to `cargo metadata` to narrow the nodes shown in the resolver graph to only packages included for the given target triple. #7376

Changed

- Cargo's "platform" `cfg` parsing has been extracted into a separate crate named `cargo-platform`. #7375
- Dependencies extracted into Cargo's cache no longer preserve mtimes to reduce syscall overhead. #7465
- Windows: EXE files no longer include a metadata hash in the filename. This helps with debuggers correlating the filename with the PDB file. #7400
- Wasm32: `.wasm` files are no longer treated as an "executable", allowing `cargo test` and `cargo run` to work properly with the generated `.js` file. #7476
- crates.io now supports SPDX 3.6 licenses. #7481
- Improved cyclic dependency error message. #7470
- Bare `cargo clean` no longer locks the package cache. #7502
- `cargo publish` now allows dev-dependencies without a version key to be published. A git or path-only dev-dependency will be removed from the package manifest before uploading. #7333
- `--features` and `--no-default-features` in the root of a virtual workspace will now generate an error instead of being ignored. #7507

- Generated files (like `Cargo.toml` and `Cargo.lock`) in a package archive now have their timestamp set to the current time instead of the epoch. #7523
- The `-Z` flag parser is now more strict, rejecting more invalid syntax. #7531

Fixed

- Fixed an issue where if a package had an `include` field, and `Cargo.lock` in `.gitignore`, and a binary or example target, and the `Cargo.lock` exists in the current project, it would fail to publish complaining the `Cargo.lock` was dirty. #7448
- Fixed a panic in a particular combination of `[patch]` entries. #7452
- Windows: Better error message when `cargo test` or `rustc` crashes in an abnormal way, such as a signal or seg fault. #7535

Nightly only

- The `mtime-on-use` feature may now be enabled via the `unstable.mtime_on_use` config option. #7411
- Added support for named profiles. #6989
- Added `-Zpanic-abort-tests` to allow building and running tests with the "abort" panic strategy. #7460
- Changed `build-std` to use `--sysroot`. #7421
- Various fixes and enhancements to `-Ztimings`. #7395 #7398 #7397 #7403 #7428 #7429
- Profile overrides have renamed the syntax to be `[profile.dev.package.NAME]`. #7504
- Fixed warnings for unused profile overrides in a workspace. #7536

Cargo 1.39 (2019-11-07)

e853aa97...1c6ec66d

Added

- Config files may now use the `.toml` filename extension. #7295
- The `--workspace` flag has been added as an alias for `--all` to help avoid confusion about the meaning of "all". #7241
- The `publish` field has been added to `cargo metadata`. #7354

Changed

- Display more information if parsing the output from `rustc` fails. #7236
- TOML errors now show the column number. #7248
- `cargo vendor` no longer deletes files in the `vendor` directory that starts with a `.`. #7242
- `cargo fetch` will now show manifest warnings. #7243
- `cargo publish` will now check git submodules if they contain any uncommitted changes. #7245
- In a build script, `cargo:rustc-flags` now allows `-l` and `-L` flags without spaces. #7257
- When `cargo install` replaces an older version of a package it will now delete any installed binaries that are no longer present in the newly installed version. #7246
- A git dependency may now also specify a `version` key when published. The `git` value will be stripped from the uploaded crate, matching the behavior of `path` dependencies. #7237
- The behavior of workspace default-members has changed. The default-members now only applies when running Cargo in the root of the workspace. Previously it would always apply regardless of which directory Cargo is running in. #7270
- libgit2 updated pulling in all upstream changes. #7275

- Bump `home` dependency for locating home directories. #7277
- zsh completions have been updated. #7296
- SSL connect errors are now retried. #7318
- The jobserver has been changed to acquire N tokens (instead of N-1), and then immediately acquires the extra token. This was changed to accommodate the `cc` crate on Windows to allow it to release its implicit token. #7344
- The scheduling algorithm for choosing which crate to build next has been changed. It now chooses the crate with the greatest number of transitive crates waiting on it. Previously it used a maximum topological depth. #7390
- RUSTFLAGS are no longer incorporated in the metadata and filename hash, reversing the change from 1.33 that added it. This means that any change to RUSTFLAGS will cause a recompile, and will not affect symbol munging. #7459

Fixed

- Git dependencies with submodules with shorthand SSH URLs (like `git@github.com/user/repo.git`) should now work. #7238
- Handle broken symlinks when creating `.dSYM` symlinks on macOS. #7268
- Fixed issues with multiple versions of the same crate in a `[patch]` table. #7303
- Fixed issue with custom target `.json` files where a substring of the name matches an unsupported crate type (like "bin"). #7363
- Fixed issues with generating documentation for proc-macro crate types. #7159
- Fixed hang if Cargo panics within a build thread. #7366
- Fixed rebuild detection if a `build.rs` script issues different `rerun-if` directives between builds. Cargo was erroneously causing a rebuild after the change. #7373
- Properly handle canonical URLs for `[patch]` table entries, preventing the patch from working after the first time it is used. #7368

- Fixed an issue where integration tests were waiting for the package binary to finish building before starting their own build. They now may build concurrently. #7394
- Fixed accidental change in the previous release on how `--features a b` flag is interpreted, restoring the original behavior where this is interpreted as `--features a` along with the argument `b` passed to the command. To pass multiple features, use quotes around the features to pass multiple features like `--features "a b"`, or use commas, or use multiple `--features` flags. #7419

Nightly only

- Basic support for building the standard library directly from Cargo has been added. (docs) #7216
- Added `-Ztimings` feature to generate an HTML report on the time spent on individual compilation steps. This also may output completion steps on the console and JSON data. (docs) #7311
- Added ability to cross-compile doctests. (docs) #6892

Cargo 1.38 (2019-09-26)

4c1fa54d...23ef9a4e

Added

- 🔥 Cargo build pipelining has been enabled by default to leverage more idle CPU parallelism during builds. #7143
- The `--message-format` option to Cargo can now be specified multiple times and accepts a comma-separated list of values. In addition to the previous values it also now accepts `json-diagnostic-short` and `json-diagnostic-rendered-ansi` which configures the output coming from rustc in `json` message mode. #7214
- Cirrus CI badges are now supported on crates.io. #7119
- A new format for `Cargo.lock` has been introduced. This new format is intended to avoid source-control merge conflicts more often, and to generally make it safer to merge changes. This new format is *not* enabled at this time, though Cargo will use it if it sees it. At some point in the future, it is intended that this will become the default. #7070
- Progress bar support added for FreeBSD. #7222

Changed

- The `-q` flag will no longer suppress the root error message for an error from Cargo itself. #7116
- The Cargo Book is now published with mdbook 0.3 providing a number of formatting fixes and improvements. #7140
- The `--features` command-line flag can now be specified multiple times. The list of features from all the flags are joined together. #7084
- Package include/exclude glob-vs-gitignore warnings have been removed. Packages may now use gitignore-style matching without producing any warnings. #7170
- Cargo now shows the command and output when parsing `rustc` output fails when querying `rustc` for information like `cfg` values. #7185

- `cargo package`/`cargo publish` now allows a symbolic link to a git submodule to include that submodule. #6817
- Improved the error message when a version requirement does not match any versions, but there are pre-release versions available. #7191

Fixed

- Fixed using the wrong directory when updating git repositories when using the `git-fetch-with-cli` config option, and the `GIT_DIR` environment variable is set. This may happen when running cargo from git callbacks. #7082
- Fixed dep-info files being overwritten for targets that have separate debug outputs. For example, binaries on `-apple-` targets with `.dSYM` directories would overwrite the `.d` file. #7057
- Fix `[patch]` table not preserving "one major version per source" rule. #7118
- Ignore `--remap-path-prefix` flags for the metadata hash in the `cargo rustc` command. This was causing the remap settings to inadvertently affect symbol names. #7134
- Fixed cycle detection in `[patch]` dependencies. #7174
- Fixed `cargo new` leaving behind a symlink on Windows when `core.symlinks` git config is true. Also adds a number of fixes and updates from upstream libgit2. #7176
- macOS: Fixed setting the flag to mark the `target` directory to be excluded from backups. #7192
- Fixed `cargo fix` panicking under some situations involving multi-byte characters. #7221

Nightly only

- Added `cargo fix --clippy` which will apply machine-applicable fixes from Clippy. #7069
- Added `-Z binary-dep-depinfo` flag to add change tracking for binary dependencies like the standard library. #7137 #7219

- `cargo clippy-preview` will always run, even if no changes have been made. #7157
- Fixed exponential blowup when using `CARGO_BUILD_PIPELINING`. #7062
- Fixed passing args to clippy in `cargo clippy-preview`. #7162

Cargo 1.37 (2019-08-15)

c4fcfb72...9edd0891

Added

- Added `doctest` field to `cargo metadata` to determine if a target's documentation is tested. #6953 #6965
- 🔥 The `cargo vendor` command is now built-in to Cargo. This command may be used to create a local copy of the sources of all dependencies. #6869
- 🔥 The "publish lockfile" feature is now stable. This feature will automatically include the `Cargo.lock` file when a package is published if it contains a binary executable target. By default, Cargo will ignore `Cargo.lock` when installing a package. To force Cargo to use the `Cargo.lock` file included in the published package, use `cargo install --locked`. This may be useful to ensure that `cargo install` consistently reproduces the same result. It may also be useful when a semver-incompatible change is accidentally published to a dependency, providing a way to fall back to a version that is known to work. #7026
- 🔥 The `default-run` feature has been stabilized. This feature allows you to specify which binary executable to run by default with `cargo run` when a package includes multiple binaries. Set the `default-run` key in the `[package]` table in `Cargo.toml` to the name of the binary to use by default. #7056

Changed

- `cargo package` now verifies that build scripts do not create empty directories. #6973
- A warning is now issued if `cargo doc` generates duplicate outputs, which causes files to be randomly stomped on. This may happen for a variety of reasons (renamed dependencies, multiple versions of the

same package, packages with renamed libraries, etc.). This is a known bug, which needs more work to handle correctly. #6998
- Enabling a dependency's feature with `--features foo/bar` will no longer compile the current crate with the `foo` feature if `foo` is not an optional dependency. #7010
- If `--remap-path-prefix` is passed via RUSTFLAGS, it will no longer affect the filename metadata hash. #6966
- libgit2 has been updated to 0.28.2, which Cargo uses to access git repositories. This brings in hundreds of changes and fixes since it was last updated in November. #7018
- Cargo now supports absolute paths in the dep-info files generated by rustc. This is laying the groundwork for tracking binaries, such as libstd, for rebuild detection. (Note: this contains a known bug.) #7030

Fixed

- Fixed how zsh completions fetch the list of commands. #6956
- "+ debuginfo" is no longer printed in the build summary when `debug` is set to 0. #6971
- Fixed `cargo doc` with an example configured with `doc = true` to document correctly. #7023
- Don't fail if a read-only lock cannot be acquired in CARGO_HOME. This helps when CARGO_HOME doesn't exist, but `--locked` is used which means CARGO_HOME is not needed. #7149
- Reverted a change in 1.35 which released jobserver tokens when Cargo blocked on a lock file. It caused a deadlock in some situations. #7204

Nightly only

- Added compiler message caching. The `-Z cache-messages` flag makes cargo cache the compiler output so that future runs can redisplay previous warnings. #6933
- `-Z mtime-on-use` no longer touches intermediate artifacts. #7050

Cargo 1.36 (2019-07-04)

6f3e9c36...c4fcfb72

Added

- Added more detailed documentation on target auto-discovery. #6898
- 🔥 Stabilize the `--offline` flag which allows using cargo without a network connection. #6934 #6871

Changed

- `publish = ["crates-io"]` may be added to the manifest to restrict publishing to crates.io only. #6838
- macOS: Only include the default paths if `DYLD_FALLBACK_LIBRARY_PATH` is not set. Also, remove `/lib` from the default set. #6856
- `cargo publish` will now exit early if the login token is not available. #6854
- HTTP/2 stream errors are now considered "spurious" and will cause a retry. #6861
- Setting a feature on a dependency where that feature points to a *required* dependency is now an error. Previously it was a warning. #6860
- The `registry.index` config value now supports relative `file:` URLs. #6873
- macOS: The `.dSYM` directory is now symbolically linked next to example binaries without the metadata hash so that debuggers can find it. #6891
- The default `Cargo.toml` template for now projects now includes a comment providing a link to the documentation. #6881
- Some improvements to the wording of the crate download summary. #6916 #6920
- ✨ Changed `RUST_LOG` environment variable to `CARGO_LOG` so that user code that uses the `log` crate will not display cargo's debug output.

#6918
- `Cargo.toml` is now always included when packaging, even if it is not listed in `package.include`. #6925
- Package include/exclude values now use gitignore patterns instead of glob patterns. #6924
- Provide a better error message when crates.io times out. Also improve error messages with other HTTP response codes. #6936

Performance

- Resolver performance improvements for some cases. #6853
- Optimized how cargo reads the index JSON files by caching the results. #6880 #6912 #6940
- Various performance improvements. #6867

Fixed

- More carefully track the on-disk fingerprint information for dependencies. This can help in some rare cases where the build is interrupted and restarted. #6832
- `cargo run` now correctly passes non-UTF8 arguments to the child process. #6849
- Fixed bash completion to run on bash 3.2, the stock version in macOS. #6905
- Various fixes and improvements to zsh completion. #6926 #6929
- Fix `cargo update` ignoring `-p` arguments if the `Cargo.lock` file was missing. #6904

Nightly only

- Added `-Z install-upgrade` feature to track details about installed crates and to update them if they are out-of-date. #6798
- Added the `public-dependency` feature which allows tracking public versus private dependencies. #6772
- Added build pipelining via the `build.pipelining` config option (`CARGO_BUILD_PIPELINING` env var). #6883

- The `publish-lockfile` feature has had some significant changes. The default is now `true`, the `Cargo.lock` will always be published for binary crates. The `Cargo.lock` is now regenerated during publishing. `cargo install` now ignores the `Cargo.lock` file by default, and requires `--locked` to use the lock file. Warnings have been added if yanked dependencies are detected. #6840

Cargo 1.35 (2019-05-23)

6789d8a0...6f3e9c36

Added

- Added the `rustc-cdylib-link-arg` key for build scripts to specify linker arguments for cdylib crates. #6298

Changed

- When passing a test filter, such as `cargo test foo`, don't build examples (unless they set `test = true`). #6683
- Forward the `--quiet` flag from `cargo test` to the libtest harness so that tests are actually quiet. #6358
- The verification step in `cargo package` that checks if any files are modified is now stricter. It uses a hash of the contents instead of checking filesystem mtimes. It also checks *all* files in the package. #6740
- Jobserver tokens are now released whenever Cargo blocks on a file lock. #6748
- Issue a warning for a previous bug in the TOML parser that allowed multiple table headers with the same name. #6761
- Removed the `CARGO_PKG_*` environment variables from the metadata hash and added them to the fingerprint instead. This means that when these values change, stale artifacts are not left behind. Also added the "repository" value to the fingerprint. #6785
- `cargo metadata` no longer shows a `null` field for a dependency without a library in `resolve.nodes.deps`. The dependency is no longer shown. #6534
- `cargo new` will no longer include an email address in the `authors` field if it is set to the empty string. #6802
- `cargo doc --open` now works when documenting multiple packages. #6803

- `cargo install --path P` now loads the `.cargo/config` file from the directory P. #6805
- Using semver metadata in a version requirement (such as `1.0.0+1234`) now issues a warning that it is ignored. #6806
- `cargo install` now rejects certain combinations of flags where some flags would have been ignored. #6801
- Resolver performance improvements for some cases. #6776

Fixed

- Fixed running separate commands (such as `cargo build` then `cargo test`) where the second command could use stale results from a build script. #6720
- Fixed `cargo fix` not working properly if a `.gitignore` file that matched the root package directory. #6767
- Fixed accidentally compiling a lib multiple times if `panic=unwind` was set in a profile. #6781
- Paths to JSON files in `build.target` config value are now canonicalized to fix building dependencies. #6778
- Fixed re-running a build script if its compilation was interrupted (such as if it is killed). #6782
- Fixed `cargo new` initializing a fossil repo. #6792
- Fixed supporting updating a git repo that has a force push when using the `git-fetch-with-cli` feature. `git-fetch-with-cli` also shows more error information now when it fails. #6800
- `--example` binaries built for the WASM target are fixed to no longer include a metadata hash in the filename, and are correctly emitted in the `compiler-artifact` JSON message. #6812

Nightly only

- `cargo clippy-preview` is now a built-in cargo command. #6759
- The `build-override` profile setting now includes proc-macros and their dependencies. #6811

- Optional and target dependencies now work better with `-Z offline`. #6814

Cargo 1.34 (2019-04-11)

f099fe94...6789d8a0

Added

- 🔥 Stabilized support for alternate registries. #6654
- Added documentation on using builds.sr.ht Continuous Integration with Cargo. #6565
- `Cargo.lock` now includes a comment at the top that it is `@generated`. #6548
- Azure DevOps badges are now supported. #6264
- Added a warning if `--exclude` flag specifies an unknown package. #6679

Changed

- `cargo test --doc --no-run` doesn't do anything, so it now displays an error to that effect. #6628
- Various updates to bash completion: add missing options and commands, support libtest completions, use rustup for `--target` completion, fallback to filename completion, fix editing the command line. #6644
- Publishing a crate with a `[patch]` section no longer generates an error. The `[patch]` section is removed from the manifest before publishing. #6535
- `build.incremental = true` config value is now treated the same as `CARGO_INCREMENTAL=1`, previously it was ignored. #6688
- Errors from a registry are now always displayed regardless of the HTTP response code. #6771

Fixed

- Fixed bash completion for `cargo run --example`. #6578

- Fixed a race condition when using a *local* registry and running multiple cargo commands at the same time that build the same crate. #6591
- Fixed some flickering and excessive updates of the progress bar. #6615
- Fixed a hang when using a git credential helper that returns incorrect credentials. #6681
- Fixed resolving yanked crates with a local registry. #6750

Nightly only

- Added `-Z mtime-on-use` flag to cause the mtime to be updated on the filesystem when a crate is used. This is intended to be able to track stale artifacts in the future for cleaning up unused files. #6477 #6573
- Added experimental `-Z dual-proc-macros` to build proc macros for both the host and the target. #6547

Cargo 1.33 (2019-02-28)

8610973a...f099fe94

Added

- `compiler-artifact` JSON messages now include an `"executable"` key which includes the path to the executable that was built. #6363
- The man pages have been rewritten, and are now published with the web documentation. #6405
- `cargo login` now displays a confirmation after saving the token. #6466
- A warning is now emitted if a `[patch]` entry does not match any package. #6470
- `cargo metadata` now includes the `links` key for a package. #6480
- "Very verbose" output with `-vv` now displays the environment variables that cargo sets when it runs a process. #6492
- `--example`, `--bin`, `--bench`, or `--test` without an argument now lists the available targets for those options. #6505
- Windows: If a process fails with an extended status exit code, a human-readable name for the code is now displayed. #6532
- Added `--features`, `--no-default-features`, and `--all-features` flags to the `cargo package` and `cargo publish` commands to use the given features when verifying the package. #6453

Changed

- If `cargo fix` fails to compile the fixed code, the rustc errors are now displayed on the console. #6419
- Hide the `--host` flag from `cargo login`, it is unused. #6466
- Build script fingerprints now include the rustc version. #6473
- macOS: Switched to setting `DYLD_FALLBACK_LIBRARY_PATH` instead of `DYLD_LIBRARY_PATH`. #6355
- `RUSTFLAGS` is now included in the metadata hash, meaning that changing the flags will not overwrite previously built files. #6503

- When updating the crate graph, unrelated yanked crates were erroneously removed. They are now kept at their original version if possible. This was causing unrelated packages to be downgraded during `cargo update -p somecrate`. #5702
- TOML files now support the 0.5 TOML syntax.

Fixed

- `cargo fix` will now ignore suggestions that modify multiple files. #6402
- `cargo fix` will now only fix one target at a time, to deal with targets which share the same source files. #6434
- Fixed bash completion showing the list of cargo commands. #6461
- `cargo init` will now avoid creating duplicate entries in `.gitignore` files. #6521
- Builds now attempt to detect if a file is modified in the middle of a compilation, allowing you to build again and pick up the new changes. This is done by keeping track of when the compilation *starts* not when it finishes. Also, #5919 was reverted, meaning that cargo does *not* treat equal filesystem mtimes as requiring a rebuild. #6484

Nightly only

- Allow using registry *names* in `[patch]` tables instead of just URLs. #6456
- `cargo metadata` added the `registry` key for dependencies. #6500
- Registry names are now restricted to the same style as package names (alphanumeric, `-` and `_` characters). #6469
- `cargo login` now displays the `/me` URL from the registry config. #6466
- `cargo login --registry=NAME` now supports interactive input for the token. #6466
- Registries may now elide the `api` key from `config.json` to indicate they do not support API access. #6466

- Fixed panic when using `--message-format=json` with metabuild. #6432
- Fixed detection of publishing to crates.io when using alternate registries. #6525

Cargo 1.32 (2019-01-17)

339d9f9c...8610973a

Added

- Registries may now display warnings after a successful publish. #6303
- Added a glossary to the documentation. #6321
- Added the alias `c` for `cargo check`. #6218

Changed

- 🔥 HTTP/2 multiplexing is now enabled by default. The `http.multiplexing` config value may be used to disable it. #6271
- Use ANSI escape sequences to clear lines instead of spaces. #6233
- Disable git templates when checking out git dependencies, which can cause problems. #6252
- Include the `--update-head-ok` git flag when using the `net.git-fetch-with-cli` option. This can help prevent failures when fetching some repositories. #6250
- When extracting a crate during the verification step of `cargo package`, the filesystem mtimes are no longer set, which was failing on some rare filesystems. #6257
- `crate-type = ["proc-macro"]` is now treated the same as `proc-macro = true` in `Cargo.toml`. #6256
- An error is raised if `dependencies`, `features`, `target`, or `badges` is set in a virtual workspace. Warnings are displayed if `replace` or `patch` is used in a workspace member. #6276
- Improved performance of the resolver in some cases. #6283 #6366
- `.rmeta` files are no longer hard-linked into the base target directory (`target/debug`). #6292
- A warning is issued if multiple targets are built with the same output filenames. #6308

- When using `cargo build` (without `--release`) benchmarks are now built using the "test" profile instead of "bench". This makes it easier to debug benchmarks, and avoids confusing behavior. #6309
- User aliases may now override built-in aliases (`b`, `r`, `t`, and `c`). #6259
- Setting `autobins=false` now disables auto-discovery of inferred targets. #6329
- `cargo verify-project` will now fail on stable if the project uses unstable features. #6326
- Platform targets with an internal `.` within the name are now allowed. #6255
- `cargo clean --release` now only deletes the release directory. #6349

Fixed

- Avoid adding extra angle brackets in email address for `cargo new`. #6243
- The progress bar is disabled if the CI environment variable is set. #6281
- Avoid retaining all rustc output in memory. #6289
- If JSON parsing fails, and rustc exits nonzero, don't lose the parse failure message. #6290
- Fixed renaming a project directory with build scripts. #6328
- Fixed `cargo run --example NAME` to work correctly if the example sets `crate_type = ["bin"]`. #6330
- Fixed issue with `cargo package` git discovery being too aggressive. The `--allow-dirty` now completely disables the git repo checks. #6280
- Fixed build change tracking for `[patch]` deps which resulted in `cargo build` rebuilding when it shouldn't. #6493

Nightly only

- Allow usernames in registry URLs. #6242

- Added `"compile_mode"` key to the build-plan JSON structure to be able to distinguish running a custom build script versus compiling the build script. #6331
- `--out-dir` no longer copies over build scripts. #6300

Cargo 1.31 (2018-12-06)

36d96825...339d9f9c

Added

- 🔥 Stabilized support for the 2018 edition. #5984 #5989
- 🔥 Added the ability to rename dependencies in Cargo.toml. #6319
- 🔥 Added support for HTTP/2 pipelining and multiplexing. Set the `http.multiplexing` config value to enable. #6005
- Added `http.debug` configuration value to debug HTTP connections. Use `CARGO_HTTP_DEBUG=true RUST_LOG=cargo::ops::registry cargo build` to display the debug information. #6166
- `CARGO_PKG_REPOSITORY` environment variable is set with the repository value from `Cargo.toml` when building . #6096

Changed

- `cargo test --doc` now rejects other flags instead of ignoring them. #6037
- `cargo install` ignores `~/.cargo/config`. #6026
- `cargo version --verbose` is now the same as `cargo -vV`. #6076
- Comments at the top of `Cargo.lock` are now preserved. #6181
- When building in "very verbose" mode (`cargo build -vv`), build script output is prefixed with the package name and version, such as `[foo 0.0.1]`. #6164
- If `cargo fix --broken-code` fails to compile after fixes have been applied, the files are no longer reverted and are left in their broken state. #6316

Fixed

- Windows: Pass Ctrl-C to the process with `cargo run`. #6004
- macOS: Fix bash completion. #6038

- Support arbitrary toolchain names when completing `+toolchain` in bash completion. #6038
- Fixed edge cases in the resolver, when backtracking on failed dependencies. #5988
- Fixed `cargo test --all-targets` running lib tests three times. #6039
- Fixed publishing renamed dependencies to crates.io. #5993
- Fixed `cargo install` on a git repo with multiple binaries. #6060
- Fixed deeply nested JSON emitted by rustc being lost. #6081
- Windows: Fix locking msys terminals to 60 characters. #6122
- Fixed renamed dependencies with dashes. #6140
- Fixed linking against the wrong dylib when the dylib existed in both `target/debug` and `target/debug/deps`. #6167
- Fixed some unnecessary recompiles when `panic=abort` is used. #6170

Nightly only

- Added `--registry` flag to `cargo install`. #6128
- Added `registry.default` configuration value to specify the default registry to use if `--registry` flag is not passed. #6135
- Added `--registry` flag to `cargo new` and `cargo init`. #6135

Cargo 1.30 (2018-10-25)

524a578d...36d96825

Added

- 🔥 Added an animated progress bar shows progress during building. #5995
- Added `resolve.nodes.deps` key to `cargo metadata`, which includes more information about resolved dependencies, and properly handles renamed dependencies. #5871
- When creating a package, provide more detail with `-v` when failing to discover if files are dirty in a git repository. Also fix a problem with discovery on Windows. #5858
- Filters like `--bin`, `--test`, `--example`, `--bench`, or `--lib` can be used in a workspace without selecting a specific package. #5873
- `cargo run` can be used in a workspace without selecting a specific package. #5877
- `cargo doc --message-format=json` now outputs JSON messages from rustdoc. #5878
- Added `--message-format=short` to show one-line messages. #5879
- Added `.cargo_vcs_info.json` file to `.crate` packages that captures the current git hash. #5886
- Added `net.git-fetch-with-cli` configuration option to use the `git` executable to fetch repositories instead of using the built-in libgit2 library. #5914
- Added `required-features` to `cargo metadata`. #5902
- `cargo uninstall` within a package will now uninstall that package. #5927
- Added `--allow-staged` flag to `cargo fix` to allow it to run if files are staged in git. #5943
- Added `net.low-speed-limit` config value, and also honor `net.timeout` for http operations. #5957
- Added `--edition` flag to `cargo new`. #5984

- Temporarily stabilized 2018 edition support for the duration of the beta. #5984 #5989
- Added support for `target.'cfg(…)'.runner` config value to specify the run/test/bench runner for targets that use config expressions. #5959

Changed

- Windows: `cargo run` will not kill child processes when the main process exits. #5887
- Switched to the `opener` crate to open a web browser with `cargo doc --open`. This should more reliably select the system-preferred browser on all platforms. #5888
- Equal file mtimes now cause a target to be rebuilt. Previously only if files were strictly *newer* than the last build would it cause a rebuild. #5919
- Ignore `build.target` config value when running `cargo install`. #5874
- Ignore `RUSTC_WRAPPER` for `cargo fix`. #5983
- Ignore empty `RUSTC_WRAPPER`. #5985

Fixed

- Fixed error when creating a package with an edition field in `Cargo.toml`. #5908
- More consistently use relative paths for path dependencies in a workspace. #5935
- `cargo fix` now always runs, even if it was run previously. #5944
- Windows: Attempt to more reliably detect terminal width. msys-based terminals are forced to 60 characters wide. #6010
- Allow multiple target flags with `cargo doc --document-private-items`. 6022

Nightly only

- Added metabuild. #5628

Glossary

Artifact

An *artifact* is the file or set of files created as a result of the compilation process. This includes linkable libraries, executable binaries, and generated documentation.

Cargo

Cargo is the Rust *package manager*, and the primary topic of this book.

Cargo.lock

See *lock file*.

Cargo.toml

See *manifest*.

Crate

A Rust *crate* is either a library or an executable program, referred to as either a *library crate* or a *binary crate*, respectively.

Every target defined for a Cargo package is a *crate*.

Loosely, the term *crate* may refer to either the source code of the target or to the compiled artifact that the target produces. It may also refer to a compressed package fetched from a registry.

The source code for a given crate may be subdivided into *modules*.

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*.

Module

Rust's module system is used to organize code into logical units called *modules*, which provide isolated namespaces within the code.

The source code for a given crate may be subdivided into one or more separate modules. This is usually done to organize the code into areas of related functionality or to control the visible scope (public/private) of symbols within the source (structs, functions, and so on).

A `Cargo.toml` file is primarily concerned with the package it defines, its crates, and the packages of the crates on which they depend. Nevertheless, you will see the term "module" often when working with Rust, so you should understand its relationship to a given crate.

Package

A *package* is a collection of source files and a `Cargo.toml` *manifest* file which describes the package. A package has a name and version which is used for specifying dependencies between packages.

A package contains multiple *targets*, each of which is a *crate*. The `Cargo.toml` file describes the type of the crates (binary or library) within the package, along with some metadata about each one --- how each is to be built, what their direct dependencies are, etc., as described throughout this book.

The *package root* is the directory where the package's `Cargo.toml` manifest is located. (Compare with *workspace root*.)

The *package ID specification*, or *SPEC*, is a string used to uniquely reference a specific version of a package from a specific source.

Small to medium sized Rust projects will only need a single package, though it is common for them to have multiple crates.

Larger projects may involve multiple packages, in which case Cargo *workspaces* can be used to manage common dependencies and other related metadata between the packages.

Package manager

Broadly speaking, a *package manager* is a program (or collection of related programs) in a software ecosystem that automates the process of obtaining, installing, and upgrading artifacts. Within a programming language ecosystem, a package manager is a developer-focused tool whose primary functionality is to download library artifacts and their dependencies from some central repository; this capability is often combined with the ability to perform software builds (by invoking the language-specific compiler).

Cargo is the package manager within the Rust ecosystem. Cargo downloads your Rust package's dependencies (*artifacts* known as *crates*), compiles your packages, makes distributable packages, and (optionally) uploads them to crates.io, the Rust community's *package registry*.

Package registry

See *registry*.

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 in the Rust ecosystem 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`, `nvidia`, etc.
  - `sys` = The system name, for example `linux`, `windows`, `darwin`, 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` *lock file*), 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. (Compare with *package root*.)

Git Authentication

Cargo supports some forms of authentication when using git dependencies and registries. This appendix contains some information for setting up git authentication in a way that works with Cargo.

If you need other authentication methods, the `net.git-fetch-with-cli` config value can be set to cause Cargo to execute the `git` executable to handle fetching remote repositories instead of using the built-in support. This can be enabled with the `CARGO_NET_GIT_FETCH_WITH_CLI=true` environment variable.

> **Note:** Cargo does not require authentication for public git dependencies so if you see an authentication failure in that context, ensure that the URL is correct.

HTTPS authentication

HTTPS authentication requires the `credential.helper` mechanism. There are multiple credential helpers, and you specify the one you want to use in your global git configuration file.

```
# ~/.gitconfig

[credential]
helper = store
```

Cargo does not ask for passwords, so for most helpers you will need to give the helper the initial username/password before running Cargo. One way to do this is to run `git clone` of the private git repo and enter the username/password.

Tip:
macOS users may want to consider using the osxkeychain helper.
Windows users may want to consider using the GCM helper.

Note: Windows users will need to make sure that the `sh` shell is available in your `PATH`. This typically is available with the Git for Windows installation.

SSH authentication

SSH authentication requires `ssh-agent` to be running to acquire the SSH key. Make sure the appropriate environment variables are set up (`SSH_AUTH_SOCK` on most Unix-like systems), and that the correct keys are added (with `ssh-add`).

Windows can use Pageant (part of PuTTY) or `ssh-agent`. To use `ssh-agent`, Cargo needs to use the OpenSSH that is distributed as part of Windows, as Cargo does not support the simulated Unix-domain sockets used by MinGW or Cygwin. More information about installing with Windows can be found at the Microsoft installation documentation and the page on key management has instructions on how to start `ssh-agent` and to add keys.

> **Note:** Cargo does not support git's shorthand SSH URLs like `git@example.com:user/repo.git`. Use a full SSH URL like `ssh://git@example.com/user/repo.git`.

> **Note:** SSH configuration files (like OpenSSH's `~/.ssh/config`) are not used by Cargo's built-in SSH library. More advanced requirements should use `net.git-fetch-with-cli`.

SSH Known Hosts

When connecting to an SSH host, Cargo must verify the identity of the host using "known hosts", which are a list of host keys. Cargo can look for these known hosts in OpenSSH-style `known_hosts` files located in their standard locations (`.ssh/known_hosts` in your home directory, or `/etc/ssh/ssh_known_hosts` on Unix-like platforms or `%PROGRAMDATA%\ssh\ssh_known_hosts` on Windows). More information about these files can be found in the sshd man page. Alternatively, keys may be configured in a Cargo configuration file with `net.ssh.known-hosts`.

When connecting to an SSH host before the known hosts has been configured, Cargo will display an error message instructing you how to add

the host key. This also includes a "fingerprint", which is a smaller hash of the host key, which should be easier to visually verify. The server administrator can get the fingerprint by running `ssh-keygen` against the public key (for example, `ssh-keygen -l -f /etc/ssh/ssh_host_ecdsa_key.pub`). Well-known sites may publish their fingerprints on the web; for example GitHub posts theirs at https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/githubs-ssh-key-fingerprints.

Cargo comes with the host keys for github.com built-in. If those ever change, you can add the new keys to the config or known_hosts file.

Note: Cargo doesn't support the `@cert-authority` or `@revoked` markers in `known_hosts` files. To make use of this functionality, use `net.git-fetch-with-cli`. This is also a good tip if Cargo's SSH client isn't behaving the way you expect it to.