Chapter 14

More about Cargo and Crates.io

So far we’ve used only the most basic features of Cargo to build, run, and test our code, but it can do a lot more. Here we’ll go over some of its other, more advanced features to show you how to:

Customize your build through release profiles

Publish libraries on crates.io

Organize larger projects with workspaces

Install binaries from crates.io

Extend Cargo with your own custom commands

Cargo can do even more than what we can cover in this chapter too, so for a full explanation, see its documentation at [*http://doc.rust-lang.org/cargo/*](http://doc.rust-lang.org/cargo/).

Customizing Builds with Release Profiles

In Rust release profiles are pre-defined, and customizable, profiles with different configurations, to allow the programmer more control over various options for compiling your code. Each profile is configured independently of the others.

Cargo has four profiles defined with good default configurations for each use case. Cargo uses the different profiles based on which command you’re running. The commands correspond to the profiles as shown in Table 14-1:

| Command | Profile |
| --- | --- |
| cargo build | dev |
| cargo build --release | release |
| cargo test | test |
| cargo doc | doc |

Table 14-1: Which profile is used when you run different Cargo commands

This may be familiar from the output of your builds, which shows the profile used in the build:

$ cargo build

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

$ cargo build --release

Finished release [optimized] target(s) in 0.0 secs

The “dev” and “release” notifications here indicate that the compiler is using different profiles.

Customizing Release Profiles

Cargo has default settings for each of the profiles that apply when there aren’t any [profile.\*] sections in the project’s Cargo.toml file. By adding [profile.\*] sections for any profile we want to customize, we can choose to override any subset of the default settings. For example, here are the default values for the opt-level setting for the dev and release profiles:

[profile.dev]

opt-level = 0

[profile.release]

opt-level = 3

The opt-level setting controls how many optimizations Rust will apply to your code, with a range of zero to three. Applying more optimizations makes compilation take longer, so if you’re in development and compiling very often, you’d want compiling to be fast at the expense of the resulting code running slower. That’s why the default opt-level for dev is 0. When you’re ready to release, it’s better to spend more time compiling. You’ll only be compiling in release mode once, and running the compiled program many times, so release mode trades longer compile time for code that runs faster. That’s why the default opt-level for the release profile is 3.

We can choose to override any default setting by adding a different value for them in Cargo.toml. If we wanted to use optimization level 1 in the development profile, for example, we can add these two lines to our project’s Cargo.toml:

Filename: Cargo.toml

[profile.dev]

opt-level = 1

This overrides the default setting of 0. Now when we run cargo build, Cargo will use the defaults for the dev profile plus our customization to opt-level. Because we set opt-level to 1, Cargo will apply more optimizations than the default, but not as many as a release build.

For the full list of configuration options and defaults for each profile, see Cargo’s documentation at [*http://doc.rust-lang.org/cargo/*](http://doc.rust-lang.org/cargo/).

Publishing a Crate to Crates.io

We’ve used packages from crates.io as dependencies of our project, but you can also share your code for other people to use by publishing your own packages. Crates.io distributes the source code of your packages, so it primarily hosts code that’s open source.

Rust and Cargo have features that help make your published package easier for people to find and use. We’ll talk about some of those features, then cover how to publish a package.

Making Useful Documentation Comments

Accurately documenting your packages will help other users know how and when to use them, so it’s worth spending some time to write documentation. In Chapter 3, we discussed how to comment Rust code with //. Rust also has particular kind of comment for documentation, known conveniently as documentation comments, that will generate HTML documentation. The HTML displays the contents of documentation comments for public API items, intended for programmers interested in knowing how to use your crate, as opposed to how your crate is implemented.

Documentation comments use /// instead of // and support Markdown notation for formatting the text if you’d like. You place documentation comments just before the item they are documenting. Listing 14-2 shows documentation comments for an add\_one function in a crate named my\_crate:

Filename: src/lib.rs

/// Adds one to the number given.

///

/// # Examples

///

/// ```

/// let five = 5;

///

/// assert\_eq!(6, my\_crate::add\_one(5));

/// ```

pub fn add\_one(x: i32) -> i32 {

x + 1

}

Listing 14-2: A documentation comment for a function

Here, we give a description of what the add\_one function does, then start a section with the heading “Examples”, and code that demonstrates how to use the add\_one function. We can generate the HTML documentation from this documentation comment by running cargo doc. This command runs the rustdoc tool distributed with Rust and puts the generated HTML documentation in the target/doc directory.

For convenience, running cargo doc --open will build the HTML for your current crate’s documentation (as well as the documentation for all of your crate’s dependencies) and open the result in a web browser. Navigate to the add\_one function and you’ll see how the text in the documentation comments gets rendered, shown here in Figure 14-3:

Figure 14-3: HTML documentation for the add\_one function

au to add image

Commonly Used Sections

We used the # Examples markdown heading in Listing 14-2 to create a section in the HTML with the title “Examples”. Some other sections that crate authors commonly use in their documentation include:

Panics: The scenarios in which this function could panic!. Callers of this function who don’t want their programs to panic should make sure that they don’t call this function in these situations.

Errors: If this function returns a Result, describing the kinds of errors that might occur and what conditions might cause those errors to be returned can be helpful to callers so that they can write code to handle the different kinds of errors in different ways.

Safety: If this function uses unsafe code (which we will discuss in Chapter 19), there should be a section covering the invariants that this function expects callers to uphold in order for the code in unsafe blocks to function correctly.

Most documentation comment sections don’t need all of these sections, but this is a good list to check to remind you of the kinds of things that people calling your code will be interested in knowing about.

Documentation Comments as Tests

Adding examples in code blocks in your documentation comments is a way to clearly demonstrate how to use your library, but it has an additional bonus: running cargo test will run the code examples in your documentation as tests! Nothing is better than documentation with examples. Nothing is worse than examples that don’t actually work because the code has changed since the documentation has been written. Try running cargo test with the documentation for the add\_one function like in Listing 14-2; you should see a section in the test results like this:

Doc-tests my\_crate

running 1 test

test src/lib.rs - add\_one (line 5) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

Now try changing either the function or the example so that the assert\_eq! in the example will panic. Run cargo test again, and you’ll see that the doc tests catch that the example and the code are out of sync from one another!

Commenting Contained Items

There’s another style of doc comment, //!, that adds documentation to the item that contains the comments, rather than adding documentation to the items following the comments. These are typically used inside the crate root file (src/lib.rs) or inside a module’s root (mod.rs) to document the crate or the module as a whole.

For example, if we wanted to add documentation that described the purpose of the my\_crate crate that contains the add\_one function, we can add documentation comments that start with //! to the beginning of src/lib.rs as shown in Listing 14-4:

Filename: src/lib.rs

//! # My Crate

//!

//! `my\_crate` is a collection of utilities to make performing certain

//! calculations more convenient.

/// Adds one to the number given.

// ...snip...

Listing 14-4: Documentation for the my\_crate crate as a whole

Notice there isn’t any code after the last line that begins with //!. Because we started the comments with //! instead of ///, we’re documenting the item that contains this comment rather than an item that follows this comment. In this case, the item that contains this comment is the src/lib.rs file, which is the crate root. These comments describe the entire crate.

If we run cargo doc --open, we’ll see these comments displayed on the front page of the documentation for my\_crate above the list of public items in the crate, as shown in Figure 14-5:

Figure 14-5: Rendered documentation for my\_crate including the comment describing the crate as a whole

Documentation comments within items are useful for describing crates and modules especially. Use them to talk about the purpose of the container overall to help users of your crate understand your organization.

Exporting a Convenient Public API with pub use

In Chapter 7, we covered how to organize our code into modules with the mod keyword, how to make items public with the pub keyword, and how to bring items into a scope with the use keyword. The structure that makes sense to you while you’re developing a crate may not be very convenient for your users, however. You may wish to organize your structs in a hierarchy containing multiple levels, but people that want to use a type you’ve defined deep in the hierarchy might have trouble finding out that those types exist. They might also be annoyed at having to type use my\_crate::some\_module::another\_module::UsefulType; rather than use my\_crate::UsefulType;.

The structure of your public API is a major consideration when publishing a crate. People who use your crate are less familiar with the structure than you are, and might have trouble finding the pieces they want to use if the module hierarchy is large.

The good news is that, if the structure isn’t convenient for others to use from another library, you don’t have to rearrange your internal organization: you can choose to re-export items to make a public structure that’s different to your private structure, using pub use. Re-exporting takes a public item in one location and makes it public in another location as if it was defined in the other location instead.

For example, say we made a library named art for modeling artistic concepts. Within this library is a kinds module containing two enums named PrimaryColor and SecondaryColor and a utils module containing a function named mix as shown in Listing 14-6:

Filename: src/lib.rs

//! # Art

//!

//! A library for modeling artistic concepts.

pub mod kinds {

/// The primary colors according to the RYB color model.

pub enum PrimaryColor {

Red,

Yellow,

Blue,

}

/// The secondary colors according to the RYB color model.

pub enum SecondaryColor {

Orange,

Green,

Purple,

}

}

pub mod utils {

use kinds::\*;

/// Combines two primary colors in equal amounts to create

/// a secondary color.

pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {

// ...snip...

}

}

Listing 14-6: An art library with items organized into kinds and utils modules

The front page of the documentation for this crate generated by cargo doc would look like Figure 14-7:

au to add picture

Figure 14-7: Front page of the documentation for art that lists the kinds and utils modules

Note that the PrimaryColor and SecondaryColor types aren’t listed on the front page, nor is the mix function. We have to click on kinds and utils in order to see them.

Another crate depending on this library would need use statements that import the items from art including specifying the module structure that’s currently defined. Listing 14-8 shows an example of a crate that uses the PrimaryColor and mix items from the art crate:

Filename: src/main.rs

extern crate art;

use art::kinds::PrimaryColor;

use art::utils::mix;

fn main() {

let red = PrimaryColor::Red;

let yellow = PrimaryColor::Yellow;

mix(red, yellow);

}

Listing 14-8: A crate using the art crate’s items with its internal structure exported

The author of the code in Listing 14-8 that uses the art crate had to figure out that PrimaryColor is in the kinds module and mix is in the utils module. The module structure of the art crate is more relevant to developers working on the art crate than developers using the art crate. The internal structure that organizes parts of the crate into the kinds module and the utils module doesn’t add any useful information to someone trying to understand how to use the art crate. The art crate’s module structure adds confusion in having to figure out where to look and inconvenience in having to specify the module names in the use statements.

To remove the internal organization from the public API, we can take the art crate code from Listing 14-6 and add pub use statements to re-export the items at the top level, as shown in Listing 14-9:

Filename: src/lib.rs

//! # Art

//!

//! A library for modeling artistic concepts.

pub use kinds::PrimaryColor;

pub use kinds::SecondaryColor;

pub use utils::mix;

pub mod kinds {

// ...snip...

}

pub mod utils {

// ...snip...

}

Listing 14-9: Adding pub use statements to re-export items

The API documentation generated with cargo doc for this crate will now list and link re-exports on the front page as shown in Figure 14-10, which makes these types easier to find.

Au to add figure

Figure 14-10: Front page of the documentation for art that lists the re-exports

Users of the art crate can still see and choose to use the internal structure as in Listing 14-8, or they can use the more convenient structure from Listing 14-9, as shown in Listing 14-11:

Filename: src/main.rs

extern crate art;

use art::PrimaryColor;

use art::mix;

fn main() {

// ...snip...

}

Listing 14-11: A program using the re-exported items from the art crate

In cases where there are many nested modules, re-exporting the types at the top level with pub use can make a big difference in the experience of people who use the crate.

Creating a useful public API structure is more of an art than a science, and you can iterate to find the API that works best for your users. Choosing pub use gives you flexibility in how you structure your crate internally, and decouples that internal structure with what you present to your users. Take a look at some of the code of crates you’ve installed to see if their internal structure differs from their public API.

Setting up a Crates.io Account

Before you can publish any crates, you need to create an account on crates.io and get an API token. To do so, visit the home page at <https://crates.io> and log in via a GitHub account—-the GitHub account is a requirement for now, but the site may support other ways of creating an account in the future. Once you’re logged in, visit your account settings at <https://crates.io/me> and retrieve your API key. Then run the cargo login command with your API key, like this:

$ cargo login abcdefghijklmnopqrstuvwxyz012345

This command will inform Cargo of your API token and store it locally in ~/.cargo/credentials. Note that this token is a secret and should not be shared with anyone else. If it is shared with anyone for any reason, you should revoke it and generate a new token on Crates.io.

Before Publishing a New Crate

Now you have an account, and let’s say you already have a crate you want to publish. Before publishing, you’ll need to add some metadata to your crate by adding it to the [package] section of the crate’s Cargo.toml.

Your crate will first need a unique name. While you’re working on a crate locally, you may name a crate whatever you’d like. However, crate names on Crates.io are allocated on a first-come-first-serve basis. Once a crate name is taken, no one else may publish a crate with that name. Search for the name you’d like to use on the site to find out if it has been taken. If it hasn’t, edit the name in Cargo.toml under [package] to have the name you want to use for publishing like so:

[package]

name = "guessing\_game"

Even if you’ve chosen a unique name, if you try to run cargo publish to publish the crate at this point, you’ll get a warning and then an error:

$ cargo publish

Updating registry `https://github.com/rust-lang/crates.io-index`

warning: manifest has no description, license, license-file, documentation,

homepage or repository.

...snip...

error: api errors: missing or empty metadata fields: description, license.

This is because we’re missing some crucial information: a description and license are required so that people will know what your crate does and under what terms they may use it. To rectify this error, we need to include this information in Cargo.toml.

Make a description that’s just a sentence or two, as it will appear with your crate in search results and on your crate’s page. For the license field, you need to give a license identifier value. The Linux Foundation’s Software Package Data Exchange (SPDX) at [*http://spdx.org/licenses/*](http://spdx.org/licenses/) lists the identifiers you can use for this value. For example, to specify that you’ve licensed your crate using the MIT License, add the MIT identifier:

[package]

name = "guessing\_game"

license = "MIT"

If you want to use a license that doesn’t appear in the SPDX, you need to place the text of that license in a file, include the file in your project, then use license-file to specify the name of that file instead of using the license key.

Guidance on which license is right for your project is out of scope for this book. Many people in the Rust community choose to license their projects in the same way as Rust itself, with a dual license of MIT/Apache-2.0—-this demonstrates that you can also specify multiple license identifiers separated by a slash.

So, with a unique name, the version, and author details that cargo new added when you created the crate, your description, and the license you chose added, the Cargo.toml for a project that’s ready to publish might look like this:

[package]

name = "guessing\_game"

version = "0.1.0"

authors = ["Your Name <you@example.com>"]

description = "A fun game where you guess what number the computer has chosen."

license = "MIT/Apache-2.0"

[dependencies]

Cargo’s documentation at [http://doc.rust-lang.org/cargo](http://doc.rust-lang.org/cargo/) describes other metadata you can specify to ensure your crate can be discovered and used more easily!

Publishing to Crates.io

Now that you’ve created an account, saved your API token, chosen a name for your crate, and specified the required metadata, you’re ready to publish! Publishing a crate uploads a specific version to crates.io for others to use.

Take care when publishing a crate, because a publish is permanent. The version can never be overwritten, and the code cannot be deleted. One major goal of Crates.io is to act as a permanent archive of code so that builds of all projects that depend on crates from Crates.io will continue to work. Allowing deletion of versions would make fulfilling that goal impossible. However, there is no limit to the number of versions of a crate you can publish.

Let’s run the cargo publish command again. It should succeed now:

$ cargo publish

Updating registry `https://github.com/rust-lang/crates.io-index`

Packaging guessing\_game v0.1.0 (file:///projects/guessing\_game)

Verifying guessing\_game v0.1.0 (file:///projects/guessing\_game)

Compiling guessing\_game v0.1.0

(file:///projects/guessing\_game/target/package/guessing\_game-0.1.0)

Finished dev [unoptimized + debuginfo] target(s) in 0.19 secs

Uploading guessing\_game v0.1.0 (file:///projects/guessing\_game)

Congratulations! You’ve now shared your code with the Rust community, and anyone can easily add your crate as a dependency of their project.

Publishing a New Version of an Existing Crate

When you’ve made changes to your crate and are ready to release a new version, you change the version value specified in your Cargo.toml and republish. Use the Semantic Versioning rules at http://semver.org to decide what an appropriate next version number is based on the kinds of changes you’ve made. Then run cargo publish to upload the new version.

Removing Versions from Crates.io with cargo yank

While you can’t remove previous versions of a crate, you can prevent any future projects from adding them as a new dependency. This is useful when a version of a crate ends up being broken for one reason or another. For situations such as this, Cargo supports yanking a version of a crate.

Yanking a version prevents new projects from starting to depend on that version while allowing all existing projects that depend on it to continue to download and depend on that version. Essentially, a yank means that all projects with a Cargo.lock will not break, while any future Cargo.lock files generated will not use the yanked version.

To yank a version of a crate, run cargo yank and specify which version you want to yank:

$ cargo yank --vers 1.0.1

You can also undo a yank, and allow projects to start depending on a version again, by adding --undo to the command:

$ cargo yank --vers 1.0.1 --undo

A yank does not delete any code. The yank feature is not intended for deleting accidentally uploaded secrets, for example. If that happens, you must reset those secrets immediately.

Cargo Workspaces

In Chapter 12, we built a package that included both a binary crate and a library crate. You may find, as your project develops, that the library crate continues to get bigger and you want to split your package up further into multiple library crates. In this situation, Cargo has a feature called workspaces that can help manage multiple related packages that are developed in tandem.

A workspace is a set of packages that will all share the same Cargo.lock and output directory. Let’s make a project using a workspace, using trivial code so we can concentrate on the structure of a workspace. We’ll have a binary that uses two libraries: one library that will provide an add\_one function and a second library that will provide an add\_two function. These three crates will all be part of the same workspace. We’ll start by creating a new crate for the binary:

$ cargo new --bin adder

Created binary (application) `adder` project

$ cd adder

We need to modify the binary package’s Cargo.toml and add a [workspace] section to tell Cargo the adder package is a workspace. Add this at the bottom of the file:

[workspace]

Like many Cargo features, workspaces support convention over configuration: we don’t need to add anything more than this to Cargo.toml to define our workspace as long as we follow the convention.

Specifying Workspace Dependencies

The workspace convention says any crates in any subdirectories that the top-level crate depends on are part of the workspace. Any crate, whether in a workspace or not, can specify that it has a dependency on a crate in a local directory by using the path attribute on the dependency specification in Cargo.toml. If a crate has the [workspace] key and we specify path dependencies where the paths are subdirectories of the crate’s directory, those dependent crates will be considered part of the workspace. Let’s specify in the Cargo.toml for the top-level adder crate that it will have a dependency on an add-one crate that will be in the add-one subdirectory, by changing Cargo.toml to look like this:

[dependencies]

add-one = { path = "add-one" }

If we add dependencies to Cargo.toml that don’t have a path specified, those dependencies will be normal dependencies that aren’t in this workspace and are assumed to come from Crates.io.

Creating the Second Crate in the Workspace

Next, while in the adder directory, generate an add-one crate:

$ cargo new add-one

Created library `add-one` project

Your adder directory should now have these directories and files:

├── Cargo.toml

├── add-one

│ ├── Cargo.toml

│ └── src

│ └── lib.rs

└── src

└── main.rs

In add-one/src/lib.rs, let’s add an add\_one function:

Filename: add-one/src/lib.rs

pub fn add\_one(x: i32) -> i32 {

x + 1

}

Open up src/main.rs for adder and add an extern crate line at the top of the file to bring the new add-one library crate into scope. Then change the main function to call the add\_one function, as in Listing 14-12:

extern crate add\_one;

fn main() {

let num = 10;

println!("Hello, world! {} plus one is {}!", num, add\_one::add\_one(num));

}

Listing 14-12: Using the add-one library crate from the adder crate

Let’s build the adder crate by running cargo build in the adder directory!

$ cargo build

Compiling add-one v0.1.0 (file:///projects/adder/add-one)

Compiling adder v0.1.0 (file:///projects/adder)

Finished dev [unoptimized + debuginfo] target(s) in 0.68 secs

Note that this builds both the adder crate and the add-one crate in adder/add-one. Now your adder directory should have these files:

├── Cargo.lock

├── Cargo.toml

├── add-one

│ ├── Cargo.toml

│ └── src

│ └── lib.rs

├── src

│ └── main.rs

└── target

The workspace has one target directory at the top level; add-one doesn’t have its own target directory. Even if we go into the add-one directory and run cargo build, the compiled artifacts end up in adder/target rather than adder/add-one/target. The crates in a workspace depend on each other. If each crate had its own target directory, each crate in the workspace would have to recompile each other crate in the workspace in order to have the artifacts in its own target directory. By sharing one target directory, the crates in the workspace can avoid rebuilding the other crates in the workspace more than necessary.

Depending on an External Crate in a Workspace

Also notice the workspace only has one Cargo.lock, rather than having a top-level Cargo.lock and add-one/Cargo.lock. This ensures that all crates are using the same version of all dependencies. If we add the rand crate to both Cargo.toml and add-one/Cargo.toml, Cargo will resolve both of those to one version of rand and record that in the one Cargo.lock. Making all crates in the workspace use the same dependencies means the crates in the workspace will always be compatible with each other. Let’s try this out now.

Let’s add the rand crate to the [dependencies] section in add-one/Cargo.toml in order to be able to use the rand crate in the add-one crate:

Filename: add-one/Cargo.toml

[dependencies]

rand = "0.3.14"

We can now add extern crate rand; to add-one/src/lib.rs, and building the whole workspace by running cargo build in the adder directory will bring in and compile the rand crate:

$ cargo build

Updating registry `https://github.com/rust-lang/crates.io-index`

Downloading rand v0.3.14

...snip...

Compiling rand v0.3.14

Compiling add-one v0.1.0 (file:///projects/adder/add-one)

Compiling adder v0.1.0 (file:///projects/adder)

Finished dev [unoptimized + debuginfo] target(s) in 10.18 secs

The top level Cargo.lock now contains information about add-one’s dependency on rand. However, even though rand is used somewhere in the workspace, we can’t use it in other crates in the workspace unless we add rand to their Cargo.toml as well. If we add extern crate rand; to src/main.rs for the top level adder crate, for example, we’ll get an error:

$ cargo build

Compiling adder v0.1.0 (file:///projects/adder)

error[E0463]: can't find crate for `rand`

--> src/main.rs:1:1

|

1 | extern crate rand;

| ^^^^^^^^^^^^^^^^^^^ can't find crate

To fix this, edit Cargo.toml for the top level adder crate and indicate that rand is a dependency for that crate as well. Building the adder crate will add rand to the list of dependencies for adder in Cargo.lock, but no additional copies of rand will be downloaded. Cargo has ensured for us that any crate in the workspace using the rand crate will be using the same version. Using the same version of rand across the workspace saves space since we won’t have multiple copies and ensures that the crates in the workspace will be compatible with each other.

Adding a Test to a Workspace

For another enhancement, let’s add a test of the add\_one::add\_one function within the add\_one crate:

Filename: add-one/src/lib.rs

pub fn add\_one(x: i32) -> i32 {

x + 1

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn it\_works() {

assert\_eq!(3, add\_one(2));

}

}

Now run cargo test in the top-level adder directory:

$ cargo test

Compiling adder v0.1.0 (file:///projects/adder)

Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs

Running target/debug/adder-f0253159197f7841

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

Wait a second, zero tests? We just added one! If we look at the output, we can see that cargo test in a workspace only runs tests for the top level crate. To run tests for all of the crates in the workspace, we need to pass the --all flag:

$ cargo test --all

Finished dev [unoptimized + debuginfo] target(s) in 0.37 secs

Running target/debug/deps/add\_one-abcabcabc

running 1 test

test tests::it\_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Running target/debug/deps/adder-abcabcabc

running 0 tests

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

Doc-tests add-one

running 0 tests

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

When passing --all, cargo test will run the tests for all of the crates in the workspace. We can also choose to run tests for one particular crate in a workspace from the top level directory by using the -p flag and specifying the name of the crate we want to test:

$ cargo test -p add-one

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running target/debug/deps/add\_one-b3235fea9a156f74

running 1 test

test tests::it\_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests add-one

running 0 tests

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

This output shows cargo test only ran the tests for the add-one crate and didn’t run the adder crate tests.

If you choose to publish the crates in the workspace to crates.io, each crate in the workspace will get published separately. The cargo publish command does not have an --all flag or a -p flag, so it is necessary to change to each crate’s directory and run cargo publish on each crate in the workspace in order to publish them.

Now try adding an add-two crate to this workspace in a similar way as the add-one crate for some more practice!

As your project grows, consider using a workspace: smaller components are easier to understand individually than one big blob of code. Keeping the crates in a workspace can make coordination among them easier if they work together and are often changed at the same time.

Installing Binaries from Crates.io with cargo install

The cargo install command allows you to install and use binary crates locally. This isn’t intended to replace system packages; it’s meant to be a convenient way for Rust developers to install tools that others have shared on crates.io. Only packages that have binary targets can be installed. A binary target is the runnable program that gets created if the crate has a src/main.rs or another file specified as a binary, as opposed to a library target that isn’t runnable on its own but is suitable for including within other programs. Usually, crates have information in the README file about whether a crate is a library, has a binary target, or both.

All binaries from cargo install are put into the installation root’s bin folder. If you installed Rust using rustup.rs and don’t have any custom configurations, this will be $HOME/.cargo/bin. Add that directory to your $PATH to be able to run programs you’ve gotten through cargo install.

For example, we mentioned in Chapter 12 that there’s a Rust implementation of the grep tool for searching files called ripgrep. If we want to install ripgrep, we can run:

$ cargo install ripgrep

Updating registry `https://github.com/rust-lang/crates.io-index`

Downloading ripgrep v0.3.2

...snip...

Compiling ripgrep v0.3.2

Finished release [optimized + debuginfo] target(s) in 97.91 secs

Installing ~/.cargo/bin/rg

The last line of the output shows the location and the name of the installed binary, which in the case of ripgrep is rg. As long as the installation directory is in your $PATH as mentioned above, you can then run rg --help and start using a faster, rustier tool for searching files!

Extending Cargo with Custom Commands

Cargo is designed so you can extend it with new subcommands without having to modify Cargo itself. If a binary in your $PATH is named cargo-something, you can run it as if it were a Cargo subcommand by running cargo something. Custom commands like this are also listed when you run cargo --list. Being able to cargo install extensions and then run them just like the built-in Cargo tools is a super convenient benefit of Cargo’s design!

Summary

Sharing code with Cargo and crates.io is part of what makes the Rust ecosystem useful for many different tasks. Rust’s standard library is small and stable, but crates are easy to share, use, and improve on a timeline different from the language itself. Don’t be shy about sharing code that’s useful to you on Crates.io; it’s likely that it will be useful to someone else as well!