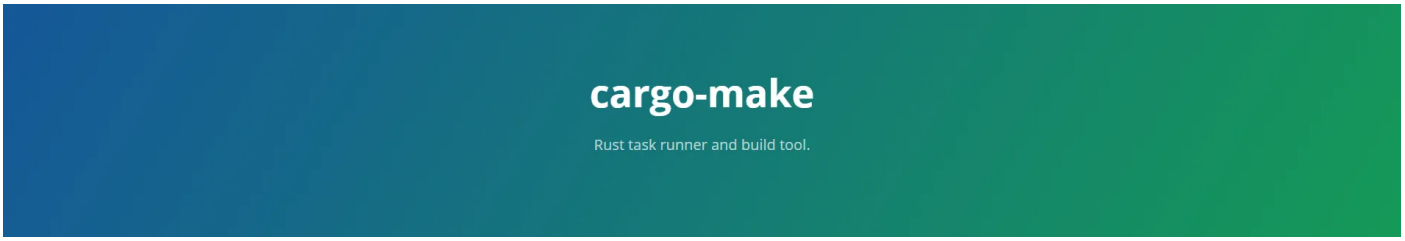
Automating your Rust workflows with cargo-make - Part 1 of 5 - Introduction and Basics

7 min read · Aug 27, 2017

 Sagie Gur-Ari Follow



cargo-make is a task runner written in Rust for Rust.

For Rust developers, cargo-make gives much added value compared to other task runners and build automation tools (such as make) or simple shell scripts as it is Rust aware, gives Rust specific features and of course, cross platform.

There are too many features to cover in a single article, so I have split it to five articles which I'll publish each day.

In this specific article I'll focus more on installation and basic features and concepts of cargo-make.

What can I do with cargo-make?

What can't you do with cargo make!!! 😉

But seriously, cargo-make supports the following features:

- Running commands
- Running shell scripts
- Creating and running flows composed of multiple tasks
- Defining dependencies for actions
- Installing missing crates when needed
- Running actions in workspace level or for each workspace member
- Define conditions on tasks which should run based on OS, Rust compiler version, environment variables or even based on custom script output
- Define different actions based on the OS you are running on
- Comes with many predefined tasks out of the box which should get most projects up and running with no coding or configuration needed, including building, testing, coverage, CI integration, publishing and much much more...
- Import shared task definitions from external sources
- Works great with online CI services such as travis and appveyor
- Use any of the many environment variables which cargo-make defines for you automatically as part of the task execution

Installation

cargo-make as the name implies is a cargo plugin and can be installed like any cargo plugin in the following way:

```
cargo install cargo-make
```

If you already have cargo-make installed and wish to upgrade to latest version,

simply run same command with the force argument as follows:

```
cargo install --force cargo-make
```

And of course, like any cargo plugin, in order to run it from the command line, the cargo/bin directory needs to be defined on your path.

For linux/mac, simply add the ~/.cargo/bin to your path.

In order to test your installation, you can run the following command:

```
cargo make --version
```

Example run

Before we dive into how cargo-make works, lets see a small example of using cargo-make

```
> cargo make
[cargo-make] info - cargo-make 0.3.67
[cargo-make] info - Using Build File: Makefile.toml
[cargo-make] info - Task: default
[cargo-make] info - External file not found, skipping.
[cargo-make] info - Setting Up Env.
[cargo-make] info - Running Task: init
[cargo-make] info - Running Task: pre-format
[cargo-make] info - Running Task: format
[cargo-make] info - Execute Command: "cargo" "fmt" "--" "--write-
mode=overwrite"
[cargo-make] info - Running Task: post-format
[cargo-make] info - Running Task: format-flow
[cargo-make] info - Running Task: pre-build
[cargo-make] info - Running Task: build
[cargo-make] info - Execute Command: "cargo" "build"
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
[cargo-make] info - Running Task: post-build
[cargo-make] info - Running Task: pre-test
[cargo-make] info - Running Task: test
[cargo-make] info - Execute Command: "cargo" "test"
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running /home/ubuntu/workspace/target/debug/deps/
member1-7cd079930eac4612

running 1 test
test test::distance_test ... ok

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

[cargo-make] info - Running Task: post-test
[cargo-make] info - Running Task: dev-test-flow
[cargo-make] info - Running Task: default
[cargo-make] info - Running Task: end
[cargo-make] info - Build Done in 0 seconds.
```

So what happened here? We ran *cargo make* in our terminal which executed the *default* flow of formatting our code, running build and finally running tests.

Tasks and the Makefile.toml

Let's start talking about how cargo-make works and how to make it work for you.

The most basic concept in cargo-make is the task.

Tasks are defined in the optional Makefile.toml. The Makefile.toml is optional as cargo-make comes with a very large set of predefined tasks so you might not have to define some on your own and just use what you get out of the box. The reason the makefiles use the toml format is because cargo uses it for the Cargo.toml file, so Rust developers use and know toml and cargo-make builds on that knowledge.

Tasks can do one of three different things: run another task, run a command or run a script. We will focus on the latter two for now.

Tasks are defined as follows:

```
[tasks.my_task]
script = [
 "echo hello"
]
```

Each task has a name, (in this case *my_task*) and optional attributes (in this case a script to execute).

In order to run the task, we invoke cargo-make with the task name as follows:

```
cargo make my_task
```

If we leave the task name out, it will invoke the default task (the task name is *default*). In later articles I'll explain how to extend and modify tasks to do other things than their predefined behavior and the default task is no different.

Running cargo plugins

Running other cargo plugins is one of the key features of cargo make and you can run and even install if missing other cargo plugins using tasks.

For example, if we wish to run code formatting using rustfmt, a task can be defined as follows:

```
[tasks.format]
description = "Runs the cargo rustfmt plugin."
install_crate = "rustfmt"
command = "cargo"
args = ["fmt", "--", "--write-mode=overwrite"]
```

So we defined a task name *format* and it will run the command 'cargo' with the following arguments: 'fmt -- --write-mode=overwrite'

However, before running the command, since it is a cargo plugin, cargo-make will first ensure that the rustfmt cargo plugin is installed and if not, it will automatically install it first.

Automatically installing missing crates is a powerful feature and leaves the machine setup to cargo-make. So instead of setting up your machine or some build server with all the cargo plugins needed as part of your build flow, cargo-make installs them for you, if needed.

This also opens up another key feature of cargo-make, unlike many other task runners like java ant, or javascript gruntjs or gulp, writing tasks for cargo-make does not involve coding using cargo-make specific API. Instead, you can use any cargo plugin which automatically brings you a wealth of powerful commands and tools in your disposal.

Running scripts

Another task capability is running scripts. So now, flows can be mixture of scripts and other commands or cargo plugins.

In the following example we copy the documentation generated by the cargo doc command from the target folder to the main folder.

```
[tasks.copy-apidocs]
description = "Copies the generated documentation to the docs/api
directory."
script = [
    "mkdir -p ./docs/api",
    "mv ./target/doc/* ./docs/api"
]
```

So while it is a nice script, we run into an issue of portability. For windows, this script won't work (unless you have bash installed).

cargo-make has the means to solve this portability issue but we will cover it in later articles. For now, this is an example of how to define a task which executes a script.

Task Dependencies

Sometimes tasks require other actions to be finished before they start. For example, we need to first generate documentation before copying it to the docs folder.

To support it, tasks can define dependencies which are basically names of other

tasks to execute before executing the original task action.

```
[tasks.docs]
description = "Generate rust documentation."
command = "cargo"
args = ["doc", "--no-deps"]

[tasks.copy-apidocs]
description = "Copies the generated documentation to the docs/api
directory."
dependencies = ["docs"]
script = [
    "mkdir -p ./docs/api",
    "mv ./target/doc/* ./docs/api"
]
```

The *copy-apidocs* task is the same as we had before except one change, we now added the **dependencies** attribute and added the *docs* as a value there.

When we run the *copy-apidocs* task it will first run the *docs* task and only afterwards it will invoke the script defined in the copy-apidocs task.

A task is never executed more than once (there are several ways to workaround that using aliases and run_task explained in later sections/articles), which is a very common concept shared by most task runners.

So let's say we have the following Makefile.toml:

```
[tasks.A]
dependencies = ["B", "C"]

[tasks.B]
dependencies = ["D"]

[tasks.C]
dependencies = ["D"]

[tasks.D]
script = [
    "echo hello"
]
```

In this scenario, A depends on B and C, and both B and C are dependent on D.

Task D however will not be invoked twice.

If we run

```
cargo make A
```

The output of the execution will look something like this:

```
[cargo-make] info - Task: A
[cargo-make] info - Setting Up Env.
[cargo-make] info - Running Task: D
[cargo-make] info - Execute Command: "sh" "/tmp/cargo-make/
CNuU47tIix.sh"
hello
[cargo-make] info - Running Task: B
[cargo-make] info - Running Task: C
[cargo-make] info - Running Task: A
```

Defining dependencies is the basic way in cargo-make to define flows.

I can define a flow of unrelated tasks which are not really dependent on each other and can be executed own their own using a wrapper task which doesn't invoke any action, just defines the flow, for example:

```
[tasks.my-flow]
dependencies = [
    "format",
    "build",
    "test"
]
```

Ignoring Task Errors

When a task fails due to any error, the entire flow will stop. Sometimes we want to have optional tasks that should not break our build in case they fail. To mark tasks that should continue in case of errors, we use the **force** attribute as follows:

```
[tasks.unstable_task]
force = true
```

There is so much more in cargo-make and I'll deep dive into the many more features in the coming articles.

In the next article I'll be talking about extending tasks, platform overrides and aliases.

Hope you enjoyed this article and give cargo-make a try.

Rust | Rustlang | Software Development | Build Tool

Follow

Written by Sagie Gur-Ari

23 followers · 6 following

Software Engineer, today working mostly in web development.

Responses (3)