The missing parts in Cargo

July 13, 2024 · 16 min

A cargo ship stagnated in March, 2021 (Julianne Cona / Instagram)

When people discuss the merits of Rust, they often mention its strict ownership rules, excellent diagnostics, and impressive performance. Cargo and the crates.io ecosystem frequently receive praise as well. Initially, when I started learning Rust, I couldn't understand why Cargo was so highly loved. Having extensive experience with JavaScript, I was accustomed to convenient package managers and couldn't grasp the enthusiasm—wasn't such a tool a given for any serious programming language? Surprisingly, not every programming language boas robust toolchain. While scripting languages often excel in this area, solutions for system-level programming is nearly empty.

A sweet meet in the universe

Cargo is incredibly user-friendly for most individual developers because it JUST WORKS. You can create a new package with `cargo new foo`, add dependencies using `cargo add`, and build with the blazing-fast Rust tool using `cargo build`. Cool! Then, you `cargo publish` to crates.io and share your project on `r/rust`. This process mirrors npm in the JavaScript ecosystem—download, build, publish. It's a seamless, one-stop shop without unnecessary boilerplate or system dependency hassles [1]. For individual developers, it's a blessed tool.

When Rust grows beyond Cargo

Although Cargo is an advanced package manager and build tool for pure Rust projects, it falls short for more complex, polyglot projects. Enterprise development environments often face resource constraints—no network access, limited access to pre-approved open-source projects, unusual linkage setups, outdated or customized C compiler toolchains, stringent security audits, and advanced but incompatible cache mechanisms.

Frustration mounts when developers discover Cargo's limitations for their projects. They either request specific features (sometimes broadly useful, sometimes not) or abandon Cargo altogether. This is disheartening, particularly as Cargo begins to lose users from some of the world's largest companies [2].

Most wanted features that never arrive

Examining issues with most thumb-ups and most comments reveals the community's needs. As of 2024-07-11, the rust-lang/cargo repository has 1,398 open issues—a manageable number, but nearly at its limit. Many issues appear to be duplicates with slight but essential differences, complicating the search for a general solution that covers various workflows.

Let's look at the features the community has wanted for Cargo to support ov
the years:

It's all about cache

Cargo uses two major types of caches:

- **Global Cache**: This cache stores downloaded dependency sources (`.crate` tarballs and Git repositories) under the `~/.cargo` directory. It never invalidates.
- **Local Cache**: This is a per-workspace-level cache for intermediate build artifacts (the `target` directory). This cache invalidates when a rebuild is needed. We will focus on this.

Cargo relies heavily on file modification times (mtime) reported by the operating system to determine cache freshness. However, this rebuild detection method is notoriously unreliable. For example, the clock may go backward, or the system mtime may have low precision, such as on Docker or Apple's HFS. Some developers have been exploring content-hash based solutions to address this issue, though the main challenge is performance. This could potentially be solved by reusing hashing results from rustc, but it requires significant investigation and cross-team communication.

Rust build times can be quite slow. To improve this, there is interest in reusing build artifacts between different projects for common crates like `syn` , `serde` , and `rand` . Although this seems logical, it is challenging. Cargo has a complex model for conditional compilation based on different compiler flags, Cargo features, and target platforms. The rebuild detection mechanism, known as the fingerprint, tracks these properties. If any of them changes, Cargo rebuilds. This means we need to track not only what to build but also how to build it. Without knowing "how," it's hard to provide a generalized fix for docker-cache layers.

Thus, simply reusing compiled artifact caches or sharing target-dirs is not very useful if we implement a basic cache-everything solution. We need a design that separates artifacts based on different combinations of flags, features, platforms, and configurations, providing an easy-to-use interface for users to define what to cache and how.

If your CI system generates a random path for each build, there's another issue. The seemingly static download sources will affect cache freshness by changing

the value of `CARGO_HOME`. This happens because the `CARGO_HOME` path is embedded in debug symbols.

The situation becomes even more complex when considering the non-determinism of build scripts and proc macros, but I will stop here for now.

Phases of a `cargo build`

Cargo wasn't designed to be a complete "build system". It was just a package manager that helps fetch dependencies from the internet, simply builds, and publishes them to crates.io. Okay, I guess I just stepped on a trap of defining what a build system should be. A build system, or build orchestrator, is software that generates a set of actions from user-provided build tasks. It can "optionally" execute these actions (Yes, so CMake is a build system in my mind).

The potential of offloading build executions to other tools is essential. It makes transparent how build tasks should be executed with desired inputs and outputs, bringing a more deterministic and analyzable build. By separating the execution phase from a build, Cargo could be able to tell why a crate is rebuilt, without actually rebuilding it. Also, clearly no need to guess the test executable name with `jq` and `grep` magics.

To push it further, a build task with well-defined input/output could open a door for different kinds of pre/post build processing. Hmm… I shouldn't say pre/post processing. Tasks ought to be composable. Apart from the execution order, the interface of defining a build task should be pretty much the same, regardless of whether it is pre or post processing. Designing such an interface is unfortunately the most difficult part that slows down the design and development. For now, Cargo prefers TOML for build configuration. Its static property ensures Cargo only does things in a defined manner. When you are not on the happy path and need an escape hatch, Cargo provides a complete unsandboxed environment to do arbitrary things. Yes, that's called "build scripts".

These two approaches are at opposite ends of the spectrum. There is a huge gap in between that Cargo doesn't even look into. Why? Because developers who look at it often end up inventing a programming language (e.g., Nickel, Nix, and Starlark). Should Cargo evolve toward that direction? I don't know. There is a

proposal for sandboxing build scripts, but it's more like a patch for build scripts themselves, not a total solution for build task composability. Ed Page's post last year also provides an overview and potential solutions for it. It's short and worth a read.

Speaking of breaking a build into phases, Bazel and Buck2 are good examples. From my truly belief, by doing so, it also helps achieve distributed executions and remote caching for their use cases. It may not be a necessary feature for indie developers or small startups. Think about it: What if we solved the cache issue and someone just built a sharable cache service that benefits everyone's CI pipeline?

Build script is not a C package manager

Speaking of build scripts, they deserve credit for Rust's growing popularity. As a tool for system-level programming, Cargo is praised for its simplicity — just `cargo build` and you're set. Even if a package needs a missing C library, `build.rs` steps in to fetch and build it from source.

But convenience comes with trade-offs. A build script isn't a proper C package manager. Its imperative, dynamic nature can make dependency management tricky. Take the "Cargo feature unification" issue: once a `vendored` feature is on in the dependency graph, you can't turn it off. We could use a declarative approach like system-deps to handle this better. Yet, how do we model a powerful build system like CMake in TOML, a less flexible language? It all loops back to defining the interface of build tasks. Or maybe someone should create a C package manager that becomes mainstream, so Cargo can just call it?

The unconditional conditional compilation

Alright, let's dive into the less glamorous side.

Conditional compilation in Cargo revolves around "Cargo features". These features can:

- Toggle code blocks with corresponding `--cfg` flags for the compiler
- Activate optional dependencies

- Activate other features

This seems more powerful than traditional C's `#ifdef`, but actually not. To prevent excessive compilation overhead, the dependency resolver picks only one compatible version when a crate appears in the dependency graph multiple times, and each is within the SemVer-compatible versions defined. And because they are deemed compatible, Cargo gets one step further that unconditionally merges all activated features into a union of them. This is the "additive" property. You have no way to opt-out of this behavior. If you desperately need mutually exclusive features, you and downstream users of your crate likely hit the ground hard, as there is no simple way to support this (remember the `vendored` problem mentioned earlier). It will become a compatibility hazard if a crate doesn't respect SemVer-compatibility and the additive nature of Cargo features.

To make the situation worse, developers want to activate dependencies based on feature activations, which may in turn activate more features. How long will it take for a feature unification to converge if that is allowed? I don't really know. That said, it is a valid feature request. Just too hard to make the design right. The Cargo team has made several attempts to make feature resolution better, for example feature resolver v2. None of them is a clear win, as users need to know in which situations a feature resolution may be different. Those attempts even confuse maintainers of Cargo!

A piece of good news is that RFC 3416 was merged, allowing future extensions of feature metadata like public/private or unstable features. It will make feature resolution smarter with the sacrifice of software complexity, from the perspective of both tool users and maintainers.

Finger-crossed cross-compilation

When it comes to cross-compilation, people often highlight its built-in support through Rustup and Cargo. While this is true, they didn't tell you the full story.

First of all, if you're in a pure Rust world, you are the luckiest person in the world. No need to deal with different compiler flags and linkers. No need to configu `target.<cfg>.rustflags` and find the dual behavior between build scripts and normal dependencies. While `target-applies-to-host` is a solution to this, it

never comes stabilized as it may break some subtle workflows around passing rustflags.

Beyond flag configurations, it's challenging to determine from `Cargo.toml` whether a package supports or requires building on certain platforms. While we have per-package-target, its semantics remain unclear, especially in relation to artifact dependencies and building the standard library (`build-std`). An even trickier part is the right timiing of filtering supported platforms. Should dependency resolution be aware of this? Should the lockfile track dependencies for supported platforms?

All of these questions above are not yet answered.

If all we have is a dependency resolver

As a package manager, selecting the correct dependency versions is the primary goal of Cargo. Cargo has its own ad-hoc dependency resolution algorithm that only a few people understand. Cargo made conditional compilation part of the resolver, so it can pick the correct set of optional dependencies and features within a valid version range. It understands `[patch]` because the resolver needs to pretend a patched dependency is from the original source. It knows the preferred Rust toolchain version in mind so that it can perform an MSRV-aware resolution.

While keeping in mind that Cargo's resolver and the entire community follow SemVer strictly, there is still a huge desire for allowing multiple SemVer-compatible versions in a dependency graph. Different strategies have been proposed, such as resolving to minimal versions. Restrictions like disallowing duplicate native lib linkage in one graph are looking for a lift. Besides, there is a potential need for support in resolving platform-specific dependencies.

Every feature request seems minimal. However, every problem becomes a version-solving problem if all we have is a dependency resolver. And that exacerbates the hard-to-maintain situation worse. The resolver is not an LLM, but it is still a myth to some maintainers how it actually works.

We should be glad that one of the Cargo maintainers decided to stand out and

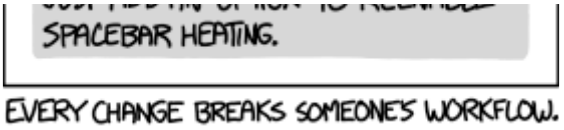
pursue a goal to modularize Cargo's ad-hoc resolver. To be more precise, it is letting the top-notch dependency resolver library `pubgrub` understand all shenanigans Cargo is doing right now. While this is a bold project, I am really looking forward to the outcome. You can track the progress by subscribing to this Zulip topic.

Stability with stagnation

Enough! We've talked too much about open issues. Let's step back and find out why the innovation seems to be stuck.

The stability guarantee of Cargo is both a blessing and a curse to the community. It's a blessing that we don't worry too much when running `rustup update` every six weeks. To make it worse, Cargo is one of the fastest-growing programming languages. Even an unstable nightly feature cannot be slightly removed due to the large adoption these days.

"Stability without stagnation" is a principle Rust holds. This is the gist of the 6-weeks "release train" model. However, people are so creative that inventing their own workflows to fix the inability of all the aforementioned missing features. That's nice, but Cargo is stuck in these implicit dependent relations, which makes maintainers[3] stressed out and reluctant to take risks on new stuff. You can see how the discussion RFC 3537 went, regardless of the intention of it was finding a middle ground to improve the current situation.



EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

Maybe because of the stability guarantee, people put a much higher bar for new features to be “perfect” and satisfy everyone. There is a summary of the Docker cache problem calling out:

> For a feature to be stablized in cargo, it needs to fit into the cohesive whole, meaning it needs to work without a lot of caveats. *It can’t be a second tier solution*

I have a dream. A dream that Cargo has its own release cadence, so it is free from the strict stability curse and can then ship major version releases.

Maximize compatibility with minimal compatibility

Well, it might be more than a dream to evolve Cargo without stagnation. A great example is `cargo-nextest`. As a non-official third-party Cargo plugin, `cargo-nextest` doesn’t need to hold the stability guarantee like other official Rust tools. Instead, it ships a performant test execution model that is deemed a breaking change if it were made to Cargo. It turns out that people love it and are willing to update their test code accordingly in exchange for test execution speed-up. Not to say that in most scenarios, `cargo-nextest` is just a drop-in replacement.

In the success story of `cargo-nextest`, its maintainers got a space to experiment on different design ideas, as well as gain some adoptions for feedback. Is it possible to achieve that for other parts of Cargo? To answer the question, first, we need to figure out the minimal set of functionalities Cargo must provide to be compatible with the crates.io ecosystem. Calling out being compatible with crates.io is because, to be honest, Cargo is nothing if there is no such ecosystem. If we want to see a wide adoption of our `cargo-nextbuild`, `cargo-nextrun`, or else, we would like to maximize compatibility with the crates.io ecosystem. You don’t ever want to recreate a whole new ecosystem in Rust, trust me.

Let's see what the minimal set of functionalities a Cargo-compatible tool needs to have to be free from stagnation.

> Note that a Cargo-compatible tool doesn't necessarily need to be done from scratch. It can be a wrapper of Cargo or use cargo-the-library.

Matching the result of dependency resolution

In an ideal world, a published crate on crates.io is guaranteed to be buildable. Other developers can fetch its source and build it flawlessly. This guarantee is upheld with the Cargo ad-hoc dependency resolver, and their contract is written in the form of `dependencies` and `features` tables in `Cargo.toml`.

If we're going to build a new dependency resolver, we don't want to fall into a situation where the old resolver found a solution for a package, whereas the new resolver can't. That makes the package unbuildable, hurting the compatibility.

It is acceptable that two dependency resolvers find different solutions, as long as those solutions are valid for both resolvers.

In summary, a Cargo-compatible tool must produce dependency resolution results that are valid in Cargo, and vice versa [4]. This also includes correctly parsing dependency information[5] from `Cargo.toml` and `Cargo.lock`.

Matching the behavior of running build scripts

In a Cargo package, every dependency is statically known, with one exception: Running build scripts. By the nature of build scripts being able to run arbitrary code, they are considered "dynamic dependencies". Cargo doesn't know what will be produced until a build script has run.

Cargo has a set of build script instructions that defines corresponding behavior before, during, and after running a build script. For example, a Cargo-compatible tool must not run if the path given by `cargo::rerun-if-changed=PATH` has no change. When a `cargo::rustc-env=VAR=VALUE` instruction is emitted, the env var must be set for the compiler invocation of the crate-being-built.

That is to say, a Cargo-compatible tool must exactly match the behavior of

running build scripts, including:

- Determine whether a rerun is needed.
- Correctly parse the emitted build script instructions.
- Configure the compiler invocation based on emitted instructions.

Setting environment variables for crates

This one is relatively straightforward. Cargo sets environment variables for compiler invocations and build script runs. These variables are either package metadata from `Cargo.toml` or necessary information helping build scripts do dirty jobs. The tricky part is that some variables are pretty Cargo-centric. For example, it's odd that a non-Cargo tool setting a path to cargo because a crate requiring the `CARGO` environment variable to call `cargo` executable recursively. However, it doesn't really make the situation worse, as build scripts are already able to do anything.

Closing

In this note, we reviewed the current state of Cargo. There are a bunch of feature requests never done. They stagnate because of the pursuit of perfection or fear of breaking unnoticed workflows. They often have a way too large design space with only a tiny place to experiment.

We then looked into a way to break the stagnation. Taking `cargo-nextest` as an example, we need to ensure a minimal compatible interface is implemented for a Cargo-compatible tool when experimenting with new ideas. Surprisingly, the minimal compatible layer is still of a reasonable size, though we might miss some important aspects that should be covered.

So, are we ready for a new adventure?

Discuss on Reddit r/rust

1. This is a success story for build scripts that they just vendor everything C

dependencies, however, silently. It is ironically that it is now considered a failure because silent vendoring is not acceptable from several aspects. See system-deps#97. ↩

2. Apparently, Google and Meta don't really use Cargo. Projects like Nix and Rust-for-Linux are willing to roll out their own build system for Rust. ↩
3. At least for me, I often feel incapable of merging a PR, even when it seemed completely harmless but actually broke 3rd-party plugin users. ↩
4. This is actually written in the 2024H2 Project Goals "Extend pubgrub to match cargo's dependency resolution". Thanks again to the owner of the goal! ↩
5. Thankfully, not all fields in `Cargo.toml` are needed for the minimal interface. In Cargo, the core fields are defined in the `Summary` struct. They construct necessary info for the resolver to work. ↩

Rust

NEXT »
如何成為正港倫敦人