Fast Rust Builds

Sep 4, 2021

It's common knowledge that Rust code is slow to compile. But I have a strong gut feeling that most Rust code out there compiles much slower than it could.

As an example, one fairly recent post says:

> With Rust, on the other hand, it takes between **15 and 45 minutes** to run a CI pipeline, depending on your project and the power of your CI servers.

This doesn't make sense to me. rust-analyzer CI takes **8** minutes on GitHub actions. It is a fairly large and complex project with 200k lines of own code and 1 million lines of dependencies on top.

It is true that Rust is slow to compile in a rather fundamental way. It picked "slow compiler" in the generic dilemma, and its overall philosophy prioritizes runtime over compile time (an excellent series of posts about that: 1, 2, 3, 4). But `rustc` is not a slow compiler — it implements the most advanced incremental compilation in industrial compilers, it takes advantage of compilation model based on proper modules (crates), and it has been meticulously optimized. Fast to compile Rust projects are a reality, even if they are not common. Admittedly, some care and domain knowledge is required to do that.

So let's take a closer look at what did it take for us to keep the compilation time within reasonable bounds for rust-analyzer!

Why Care About Build Times

One thing I want to make clear is that optimizing project's build time is in some sense busy-work. Reducing compilation time provides very small *direct* benefits to the users, and is pure accidental complexity.

That being said, compilation time is a *multiplier* for basically everything. Whether you want to ship more features, to make code faster, to adapt to a change of requirements, or to attract new contributors, build time is a factor in that.

It also is a non-linear factor. Just waiting for the compiler is the smaller problem. The big one is losing the state of the flow or (worse) mental context switch to do something else while the code is compiling. One minute of work for the compiler wastes more than one minute of work for the human.

It's hard for me to quantify the impact, but my intuitive understanding is that, as soon as the project grows beyond several thousands lines written by a single person, build times become pretty darn important!

The most devilish property of build times is that they creep up on you. While the project is small, build times are going to be acceptable. As projects grow incrementally, build times start to slowly increase as well. And if you let them grow, it might be rather hard to get them back in check later!

If project is already too slow to compile, then:

- Improving build times will be time consuming, because each iteration of "try a change, trigger the build, measure improvement" will take long time (yes, build times are a multiplier for everything, *including* build times themselves!)
- There won't be easy wins: in contrast to runtime performance, pareto principle doesn't work! If you write a thousand lines of code, maybe one hundred of them will be performance-sensitive, but each line will add to compile times!
- Small wins will seem too small until they add up: shaving off five seconds is a much bigger deal for a five minute build than for an hour-long build.
- Dually, small regressions will go unnoticed.

There's also a culture aspect to it: if you join a project and its CI takes one hour, then an hour-long CI is normal, right?

Luckily, there's one simple trick to solve the problem of build times …

The Silver Bullet

You need to care about build times, keep an eye on them, and fix them *before* they become a problem. Build times are a fairly easy optimization problem: it's trivial to get direct feedback (just time the build), there are a bunch of tools for profiling, and you don't even need to come up with a representative benchmark. The task is to optimize a particular project's build time, not performance of the compiler in general. That's a nice property of most instances of accidental complexity — they tend to be well defined engineering problems with well understood solutions.

The only hard bit about compilation time is that you don't know that it is a problem until it actually is one! So, the most valuable thing you can get from this post is this: if you are working on a Rust project, take some time to optimize its build today, and try to repeat the exercise once in a while.

Now, with the software engineering bits cleared, let's finally get to some actionable programming advice!

bors

I like to use CI time as one of the main metrics to keep an eye on.

Part of that is that CI time is important in itself. While you are not bound by CI when developing features, CI time directly affects how annoying it is to context switch when finishing one piece of work and starting the next one. Juggling five outstanding PRs waiting for CI to complete is not productive. Longer CI also creates a pressure to *not* split the work into independent chunks. If correcting a typo requires keeping a PR tab open for half a hour, it's better to just make a drive by fix in the next feature branch, right?

But a bigger part is that CI gives you a standardized benchmark. Locally, you compile incrementally, and the time of build varies greatly with the kinds of changes you are doing. Often, you compile just a subset of the project. Due to this inherent variability, local builds give poor continuous feedback about build times. Standardized CI though runs for every change and gives you a time series where numbers are directly comparable.

To increase this standardization pressure of CI, I recommend following not rocket science rule and setting up a merge robot which guarantees that every state of the main branch passes CI. bors is a particular implementation I use, but there are others.

While it's by far not the biggest reason to use something like bors, it gives two benefits for healthy compile times:

- It ensures that every change goes via CI, and creates pressure to keep CI healthy overall
- The time between leaving `r+` comment on the PR and receiving the "PR merged" notification gives you an always on feedback loop. You don't need to specifically time the build, every PR is a build benchmark.

CI Caching

If you think about it, it's pretty obvious how a good caching strategy for CI should work. It makes sense to cache stuff that changes rarely, but it's useless to cache frequently changing things. That is, cache all the dependencies, but don't cache project's own crates.

Unfortunately, almost nobody does this. A typical example would just cache the whole of `./target` directory. That's wrong — the `./target` is huge, and most of it is useless on CI.

It's not super trivial to fix though — sadly, Cargo doesn't make it too easy to figure out which part of `./target` are durable dependencies, and which parts are volatile local crates. So, you'll need to write some code to clean the `./target` before storing the cache. For GitHub actions in particular you can also use Swatinem/rust-cache.

CI Workflow

Caching is usually the low-hanging watermelon, but there are several more things to tweak.

Split CI into separate `cargo test --no-run` and `cargo test`. It is vital to know which part of your CI is the build, and which are the tests.

Disable incremental compilation. CI builds often are closer to from-scratch builds, as changes are typically much bigger than from a local edit-compile cycle. For from-scratch builds, incremental adds an extra dependency-tracking overhead. It also significantly increases the amount of IO and the size of `./target`, which make caching less effective.

Disable debuginfo — it makes `./target` much bigger, which again harms caching. Depending on your preferred workflow, you might consider disabling debuginfo unconditionally, this brings some benefits for local builds as well.

While we are at it, add `-D warnings` to the `RUSTFLAGS` environmental variable to deny warning for all crates at the same time. It's a bad idea to `#![deny(warnings)]` in code: you need to repeat it for every crate, it needlessly makes local development harder, and it might break your users when they upgrade their compiler. It might also make sense to bump cargo network retry limits.

Read The Lockfile

Another obvious advice is to use fewer, smaller dependencies.

This is nuanced: libraries do solve actual problems, and it would be stupid to roll your own solution to something already solved by crates.io. And it's not like it's guaranteed that your solution will be smaller.

But it's important to realise what problems your application is and is not solving. If you are building a CLI utility for thousands of people of to use, you absolutely need clap with all of its features. If you are writing a quick script to run during CI, which only the team will be using, it's probably fine to start with simplistic command line parsing, but faster builds.

One *tremendously* useful exercise here is to read `Cargo.lock` (not `Cargo.toml`) and for each dependency think about the actual problem this dependency solves for the person in front of your application. It's very frequent that you'll find dependencies that just don't make sense at all, *in your context*.

As an illustrative example, rust-analyzer depends on `regex`. This doesn't make sense — we have exact parsers and lexers for Rust and Markdown, we don't need to interpret regular expressions at runtime. `regex` is also one of the heavier dependencies — it's a full implementation of a small language! The reason why this dependency is there is because the logging library we use allows to say something like:

```
RUST_LOG=rust_analyzer=very complex filtering expression
```

where parsing of the filtering expression is done by regular expressions.

This is undoubtedly a very useful feature to have for some applications, but in the context of rust-analyzer we don't need it. Simple `env_logger`-style filtering would be enough.

Once you identify a similar redundant dependency, it's usually enough to tweak `features` field somewhere, or to send a PR upstream to make non-essential bits configurable.

Sometimes it is a bigger yak to shave :) For example, rust-analyzer optionally use `jemalloc` crate, and its build script pulls in `fs_extra` and (of all the things!) `paste`. The ideal solution here would be of course to have a production grade, stable, pure rust memory allocator.

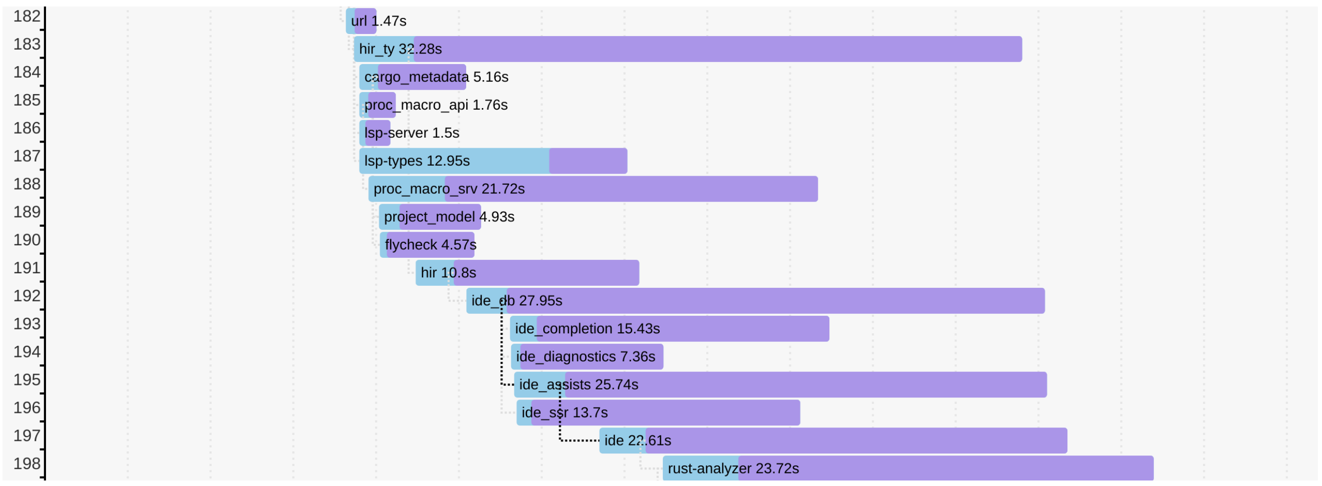
Profile Before Optimize

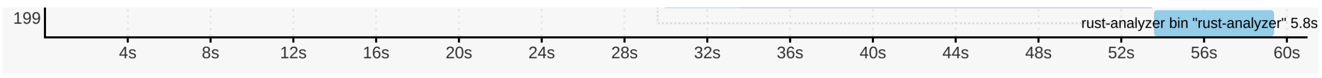
Now that we've dealt with things which are just sensible to do, it's time to start measuring before cutting. A tool to use here is `timings` flag for Cargo (documentation). Sadly, I lack the eloquence to adequately express the level of quality and polish of this feature, so let me just say ❤️ and continue with my dry prose.

`cargo build -Z timings` records profiling data during the build, and then renders it as a very legible and information-dense HTML file. This is a nightly feature, so you'll need the `+nightly` toggle. This isn't a problem in practice, as you only need to run this manually once in a while.

Here's an example from rust-analyzer:

```
$ cargo +nightly build -p rust-analyzer --bin rust-analyzer \
  -Z timings --release
```





Not only can you see how long each crate took to compile, but you'll also see how individual compilations where scheduled, *when* each crate started to compile, and its critical dependency.

Compilation Model: Crates

This last point is important — crates form a directed acyclic graph of dependencies and, on a multicore CPU, the shape of this graph affects the compilation time a lot.

This is slow to compile, as all the crates need to be compiled sequentially:

```
A -> B -> C -> D -> E
```

This version is much faster, as it enables significantly more parallelism:

```
   +-  B  -+
  /         \
A  ->  C  ->  E
  \         /
   +-  D  -+
```

There's also connection between parallelism and incrementality. In the wide graph, changing `B` doesn't entail recompiling `C` and `D`.

The first advice you get when complaining about compile times in Rust is: "split the code into crates". It is not *that* easy — if you ended up with a graph like the first one, you are not winning much. It is important to architect the applications to look like the second picture — a common vocabulary crate, a number of independent features, and a leaf crate to tie everything together. The most important property of a crate is which crates it doesn't (transitively) depend on.

Another important consideration is the number of final artifacts (most typically binaries). Rust is statically linked, so, if two different binaries use the same library, each binary contains a separately linked copy of the library. If you have `n` binaries and `m` libraries, and each binary uses each library, then the amount of work to do during the linking is `m * n`. For this reason, it's better to minimize the number of artifacts. One common technique here is BusyBox-style Swiss Army knife executables. The idea is that you can hardlink the same executable as several files with different names. The program then can look at the zeroth command line argument to learn the name it was invoked with, and use it effectively as a name of a subcommand. One cargo-specific gotcha here is that, by default, each file in `./examples` or `./tests` folder creates a new executable.

Compilation Model: Macros And Pipelining

But Cargo is even smarter than that! It does pipelined compilation — splitting the compilation of a crate into metadata and codegen phases, and starting compilation of dependent crates as soon as the metadata phase is over.

This has interesting interactions with procedural macros (and build scripts). `rustc` needs to run procedural macros to compute crate's metadata. That means that procedural macros can't be pipelined, and crates using procedural macros are blocked until the proc macro is fully compiled to the binary code.

Separately from that, procedural macros need to parse Rust code, and that is a relatively complex task. The de-facto crate for this, `syn`, takes quite some time to compile (not because it is bloated — just because parsing Rust is hard).

This generally means that projects tend to have `syn` / `serde` shaped hole in the CPU utilization profile during compilation. It's relatively important to use procedural macros only where they pull their weight, and try to push crates before `syn` in the `cargo -Z timings` graph.

The latter can be tricky, as proc macro dependencies can sneak up on you. The problem here is that they are often hidden behind feature flags, and those feature flags might be enabled by downstream crates. Consider this example:

You have a convenient utility type — for example, an SSO string, in a `small_string` crate. To implement serialization, you don't actually need derive (just delegating to `String` works), so you add an (optional) dependency on `serde`:

```toml
[package]
name = "small-string"

[dependencies]
serde = { version = "1" }
```

SSO string is a rather useful abstraction, so it gets used throughout the codebase. Then in some leaf crate which, eg, needs to expose a JSON API, you add dependency on `small_string` with the `serde` feature, as well as `serde` with derive itself:

```toml
[package]
name = "json-api"

[dependencies]
small-string = { version = "1", features = [ "serde" ] }
serde = { version = "1", features = [ "derive" ] }
```

The problem here is that `json-api` enables the `derive` feature of `serde`, and that means that `small-string` and all of its reverse-dependencies now need to wait for `syn` to compile! Similarly, if a crate depends on a subset of `syn`'s features, but something else in the crate graph enables all features, the original crate gets them as a bonus as well!

It's not necessarily the end of the world, but it shows that dependency graph can get tricky with the presence of features. Luckily, `cargo -Z timings` makes it easy to notice that something strange is happening, even if it might not be always obvious what *exactly* went wrong.

There's also a much more direct way for procedural macros to slow down compilation — if the macro generates a lot of code, the result would take some time to compile. That is, some macros allow you to write just a bit of source code, which feels innocuous enough, but expands to substantial amount of logic. The prime example is serialization — I've noticed that converting values to/from JSON accounts for surprisingly big amount of compiling. Thinking in terms of overall crate graph helps here — you want to keep serialization at the boundary of the system, in the leaf crates. If you put serialization near the foundation, then all intermediate crates would have to pay its build-time costs.

All that being said, an interesting side-note here is that procedural macros are not *inherently* slow to compile. Rather, it's the fact that most proc macros need to parse Rust or to generate a lot of code that makes them slow. Sometimes, a macro can accept a simplified syntax which can be parsed without `syn`, and emit a tiny bit of Rust code based on that. Producing valid Rust is not nearly as complicated as parsing it!

Compilation Model: Monomorphization

Now that we've covered macro issues at the level of crates, it's time to look closer, at the code-level concerns. The main thing to look here are generics. It's vital to understand how they are compiled, which, in case of Rust, is achieved by monomorphization. Consider a run of the mill generic function:

```rust
fn frobnicate<T: SomeTrait>(x: &T) {
    ...
}
```

When Rust compiles this function, it doesn't actually emit machine code. Instead, it stores an abstract representation of function body in the library. The actual compilation happens when you *instantiate* the function with a particular type parameter. The C++ terminology gives the right intuition here — `frobnicate` is a "template", it produces an actual function when a concrete type is substituted for the parameter `T`.

In other words, in the following case

```rust
fn frobnicate_both(x: String, y: Widget) {
  frobnicate(&x);
  frobnicate(&y);
}
```

on the level of machine code there will be two separate copies of `frobnicate`, which would differ in details of how they deal with parameter, but would be otherwise identical.

Sounds pretty bad, right? Seems like that you can write a gigantic generic function, and then write just a small bit of code to instantiate it with a bunch of types, to create a lot of load for the compiler.

Well, I have bad news for you — the reality is much, much worse. You don't even need different types to create duplication. Let's say we have four crates which form a diamond

```
    +- B -+
   /       \
A           D
   \       /
    +- C -+
```

The `frobnicate` is defined in `A`, and is used by `B` and `C`

```rust
// A
pub fn frobnicate<T: SomeTrait>(x: &T) { ... }

// B
pub fn do_b(s: String) { a::frobnicate(&s) }

// C
pub fn do_c(s: String) { a::frobnicate(&s) }

// D
fn main() {
  let hello = "hello".to_owned();
  b::do_b(&hello);
  c::do_c(&hello);
}
```

In this case, we only ever instantiate `frobincate` with `String`, but it will get compiled twice, because monomorphization happens *per crate*. `B` and `C` are compiled separately, and each includes machine code for `do_*` functions, so they need `frobnicate<String>`. If optimizations are disabled, rustc can share template instantiations with dependencies, but that doesn't work for sibling dependencies. With optimizations, rustc doesn't share monomorphizations even with direct dependencies.

In other words, generics in Rust can lead to accidentally-quadratic compilation times across many crates!

If you are wondering whether it gets worse than that, the answer is yes. I *think* the actual unit of monomorphization is codegen unit, so duplicates are possible even within one crate.

Keeping an Eye on Instantiations

Besides just duplication, generics add one more problem — they shift the blame for compile times to consumers. Most of the compile time cost of generic functions is borne out by the crates that use the functionality, while the defining crate just typechecks the code without doing any code generation. Coupled with the fact that at times it is not at all obvious what gets instantiated where and why (example), this make it hard to directly see the footprint of generic APIs

Luckily, this is not needed — there's a tool for that! `cargo llvm-lines` tells you which monomorphizations are happening in a specific crate.

Here's an example from a recent investigation:

```
$ cargo llvm-lines --lib --release -p ide_ssr | head -n 12
 Lines           Copies         Function name
  -----           ------         -------------
  533069 (100%)  28309 (100%)  (TOTAL)
   20349 (3.8%)    357 (1.3%)  RawVec<T,A>::current_memory
   18324 (3.4%)    332 (1.2%)  <Weak<T> as Drop>::drop
   14024 (2.6%)    332 (1.2%)  Weak<T>::inner
   11718 (2.2%)    378 (1.3%)  core::ptr::metadata::from_raw_parts_
   10710 (2.0%)    357 (1.3%)  <RawVec<T,A> as Drop>::drop
    7984 (1.5%)    332 (1.2%)  <Arc<T> as Drop>::drop
    7968 (1.5%)    332 (1.2%)  Layout::for_value_raw
    6790 (1.3%)     97 (0.3%)  hashbrown::raw::RawTable<T,A>::drop_
    6596 (1.2%)     97 (0.3%)  <hashbrown::raw::RawIterRange<T> as
```

It shows, for each generic function, how many copies of it were generated, and what's their total size. The size is measured very coarsely, in the number of llvm ir lines it takes to encode the function. A useful fact: llvm doesn't have generic functions, its the job of `rustc` to turn a function template and a set of instantiations into a set of actual functions.

Keeping Instantiations In Check

Now that we understand the pitfalls of monomorphization, a rule of thumb becomes obvious: do not put generic code at the boundaries between the crates. When designing a large system, architect it as a set of components where each of the components does something concrete and has non-generic interface.

If you do need generic interface for better type-safety and ergonomics, make sure that the interface layer is thin, and that it immediately delegates to a non-generic implementation. The classical example to internalize here are various functions from `str::fs` module which operate on paths:

```rust
pub fn read<P: AsRef<Path>>(path: P) -> io::Result<Vec<u8>> {
  fn inner(path: &Path) -> io::Result<Vec<u8>> {
    let mut file = File::open(path)?;
    let mut bytes = Vec::new();
    file.read_to_end(&mut bytes)?;
    Ok(bytes)
  }
  inner(path.as_ref())
}
```

The outer function is parameterized — it is ergonomic to use, but is compiled afresh for every downstream crate. That's not a problem though, because it is very small, and immediately delegates to a non-generic function that gets compiled in the std.

If you are writing a function which takes a path as an argument, either use `&Path`, or use `impl AsRef<Path>` and delegate to a non-generic implementation. If you care about API ergonomics enough to use impl trait, you should use `inner` trick — compile times are as big part of ergonomics, as the syntax used to call the function.

A second common case here are closures: by default, prefer `&dyn Fn()` over `impl Fn()`. Similarly to paths, an `impl`-based nice API might be a thin wrapper around `dyn`-based implementation which does the bulk of the work.

Another idea along these lines is "generic, inline hotpath; concrete, outline coldpath". In the once_cell crate, there's this curious pattern (simplified, here's the actual source):

```rust
struct OnceCell<T> {
  state: AtomicUsize,
  inner: Option<T>,
}

impl<T> OnceCell<T> {
  #[cold]
  fn initialize<F: FnOnce() -> T>(&self, f: F) {
    let mut f = Some(f);
    synchronize_access(self.state, &mut || {
      let f = f.take().unwrap();
      match self.inner {
        None => self.inner = Some(f()),
        Some(_value) => (),
      }
    });
  }
}

fn synchronize_access(state: &AtomicUsize, init: &mut dyn FnMut())
  // One hundred lines of tricky synchronization code on atomics.
}
```

Here, the `initialize` function is generic twice: first, the `OnceCell` is parametrized with the type of value being stored, and then `initialize` takes a generic closure parameter. The job of `initialize` is to make sure (even if it is called concurrently from many threads) that at most one `f` is run. This mutual exclusion task doesn't actually depend on specific `T` and `F` and is implemented as non-generic `synchronize_access`, to improve compile time. One wrinkle here is that, ideally, we'd want an `init: dyn FnOnce()` argument, but that's not expressible in today's Rust. The `let mut f = Some(f) / let f = f.take().unwrap()` is a standard work-around for this case.

Conclusions

I guess that's it! To repeat the main ideas:

Build times are a big factor in the overall productivity of the humans working on the project. Optimizing this is a straightforward engineering task — the tools are there. What might be hard is not letting them slowly regress. I hope this post provides enough motivation and inspiration for that! As a rough baseline, 200k line Rust project somewhat optimized for reasonable build times should take about 10 minutes of CI on GitHub actions.

Discussion on /r/rust.

This post is a part of One Hundred Thousand Lines of Rust series.