Thoughts on Rust bloat

Aug 21, 2019

I'm about to accept a PR that will increase druid's compile time about 3x and its executable size almost 2x. In this case, I think the tradeoff is worth it (without localization, a GUI toolkit is strictly a toy), but the bloat makes me unhappy and I think there is room for improvement in the Rust ecosystem.

Should we care?

For me, bloat in Rust is mostly about compile times and executable size. Compile time is on the top 10 list of bad things about the Rust development experience, but to some extent it's under the developer's control, especially by choosing whether or not to take dependencies on bloated crates.

Bloat is an endemic problem in software, but there are a few things that make it a particular challenge for Rust:

- Cargo makes it so easy to just reach for a crate.
- Generics, particularly monomorphization.
- Poor support for dynamic libraries.

One of the subtler ways compile times affect the experience is in tools like RLS.

It's going to vary from person to person, but I personally do care a lot. One of my hopes for xi-editor is that the core would be lightweight, especially as we could factor out concerns like UI. However, the release binary is now 5.9M (release build, Windows, and doesn't include syntax coloring, which is an additional 2.1M). I've done a bunch of other things across the Rust ecosystem to reduce bloat, and I'll brag a bit about that in this post.

Features vs bloat

Of course, the reason why I'm considering such a huge jump in compile times on druid is that I want localization, an important and complex feature. Doing it right requires quite a bit of logic around locale matching, Unicode, and natural language processing (such as plural rules). I don't *expect* a tiny crate for this.

One recent case we saw a similar tradeoff was the observation that the unicase dep adds

50k to the binary size for pulldown-cmark. In this case, the CommonMark spec demands Unicode case-folding, and without that, it's no longer complying with the standard. I understand the temptation to cut this corner, but I think having versions out there that are not spec-compliant is a bad thing, especially unfriendly to the majority of people in the world whose native language is other than English.

So, it's important not to confuse lean engineering with a lack of important features. I would say bloat is unneeded resource consumption beyond what's necessary to meet the requirements. Unicode and internationalization are a particularly contentious point, both because they actually do require code and data to get right, but also because there's a ton of potential for bloat.

Foundational crates should be lean

I would apply a higher standard to "foundational" crates, which are intended to be used by most Rust applications that need the functionality. Bloat in those is a reason *not* to use the dependency, or to fragment the ecosystem into different solutions depending on needs and tolerance for bloat.

I think a particular risk are crates providing generally useful features, ones that would definitely make the cut in a "batteries included" language. Some of these (bitflags, lazy_static, cfg-if, etc) are not very heavy, and provide obvious benefit, especially to make the API more humane. For others (rental, failure), the cost is higher and I would generally recommend not using them in foundational crates. But for your own app, if you like them, sure. I believe rental might be the most expensive transitive dependency for fluent, as I find it takes 27.3s (debug, Windows; 53.2s for release) for the crate alone.

~~I'm concerned about bloat in gfx-rs - about a minute for a debug build, and about 3M (Windows, quad example). For this reason (and stability and documentation), I'm leaning towards making the GPU renderers for piet use the underlying graphics APIs directly rather than using this abstraction layer. I've found similar patterns with other "wrapper" crates, including direct2d. But here the tradeoffs are complex.~~

[Update added afterwards re gfx-rs: In response to this post, the gfx-rs team very quickly landed major improvements to compile time, and kvark also pointed out that my methodology of using the quad example is not valid because it pulls in a bunch of other dependencies that not actually needed to run gfx-rs. I'm happy that there's been attention to this, and a lot of my concerns are alleviated, so will be looking much more closely at using gfx-rs for future GPU work. I have noticed compile time impact from other wrapper crates, so the general advice to take a careful look at these still stands.]

I don't have hard numbers yet, but I've found that the rust-objc macros produce quite bloated code, on the order of 1.5k per method invocation. This is leading me to consider rewriting the macOS platform binding code in Objective-C directly (using C as the

common FFI is not too bad), rather than relying on Rust code that uses the dynamic Objective-C runtime. I expect bloat here to affect a fairly wide range of code that calls into the macOS (and iOS) platform, so it would be a good topic to investigate more deeply.

Sharing

I sometimes hear that it's ok to depend on commonly-used crates, because their cost is amortized among the various users that share them. I'm not convinced, for a variety of reasons. For one, it's common that you get different versions anyway (the Zola build currently has two versions each of unicase, parking_lot, parking_lot_core, crossbeam-deque, toml, derive_more, lock_api, scopeguard, and winapi). Second, if generics are used heavily (see below), there'll likely be code duplication anyway.

That said, for stuff like Unicode data, it is quite important that there as few copies as possible in the binary. The best choice is crates engineered to be lean.

Proc macros

A particularly contentious question is proc macros. The support crates for these (syn and quote) take maybe 10s to compile, and don't directly impact executable size. It's a major boost to the expressivity of the Rust language, and we'll likely use them in druid, though have been discussing making them optional.

What I'd personally like to see is proc macros stabilize more and then be adopted into the language.

Use serialization sparingly

Digging into xi-editor, the biggest single source of bloat is serde, and in general the fact that it serializes everything into JSON messages. This was something of an experiment, and in retrospect I would say one of the things I'm most unhappy about. It seems that efficient serialization is not a solved problem yet. [Note also that JSON serialization is extremely slow in Swift]

Use polymorphism sparingly

The *particular* reason serde is so bloated is that it monomorphizes everything. There are alternatives; miniserde in particular yields smaller binaries and compile times by using dynamic dispatch (trait objects) in place of monomorphization. But it has other limitations and so hasn't caught on yet.

In general, overuse of polymorphism is a leading cause of bloat. For example, resvg switched from lyon to kurbo ~~for this reason~~ [Note added: RazrFalcon points out that the

big contribution to lyon compile times is proc macros, not polymorphism, and that's since been fixed]. We don't adopt the lyon / euclid ecosystem, also for this reason, which is something of a shame because now there's more fragmentation. When working on kurbo, I did experiments indicating there was no real benefit to allowing floating point types other than `f64`, so just decided that would be the type for coordinates. I'm happy with this choice.

Use async sparingly

For a variety of reasons, async code is considerably slower to compile than corresponding sync code, though the compiler team has been making great progress. Even though async/await is the shiny new feature, it's important to realize that old-fashioned sync code is still better in a lot of cases. Sure, if you're writing high-scale Internet servers, you need async, but there are a lot of other cases.

I'll pick on Zola for this one. A release build is over 9 minutes and 15M in size. (Debug builds are about twice as fast but 3-5x bigger). Watching the compile (over 400 crates total!) it's clear that its web serving (actix based) accounts for a lot of that, pulling in a big chunk of the tokio ecosystem as well. For just previewing static websites built with the tool, it might be overkill. That said, for this particular application perhaps bloat is not as important, and there are benefits to using a popular, featureful web serving framework.

As a result, I've chosen *not* to use async in druid, but rather a simpler, single-threaded approach, even though async approaches have been proposed.

Use feature gates

It's common for a crate to have some core functionality, then other stuff that only some users will want. I think it's a great idea to have optional dependencies. For example, xi-rope had the ability to serialize deltas to JSON because we used that in xi-editor, but that's a very heavyweight dependency for people who just want an efficient data structure for large strings. So we made that optional.

An alternative is to fragment the crate into finer grains; rand is a particular offender here, as it's not uncommon to see 10 subcrates in a build. We've found that having lots of subcrates often makes life harder for users because of the increased coordination work making sure versions are compatible.

Compile-time work

Another crate that often shows up in Rust builds is phf, an implementation of perfect hashing. That's often a great idea and what you want in your binaries, but it also accounts for ~13s of compile time when using the macro version (again bringing in two separate

copies of quote and syn). [Note added: sfackler points out that you can use phf-codegen to generate Rust source and check that into your repos.]

For optimizing compile times in unicode-normalization, I decided to build the hash tables using a custom tool, and check those into the repo. That way, the work is done only when the data actually changes (about once a year, as Unicode revs), as opposed to every single compile. I'm proud of this work, as it improved the compile time for unicode-normalization by about 3x, and I do consider that an important foundational crate.

Measure, measure, measure

Compile time and executable size are aspects of performance (even though often not as visible as runtime speed), and performance culture applies. Always measure, using tools like cargo-bloat where appropriate, and keep track of regressions.

A good case study for cargo-bloat is clap, though it's still pretty heavyweight today (it accounts for about 1M of Zola's debug build, measured on macOS).

There's also an effort to analyze binary sizes more systematically. I applaud such efforts and would love it if they were even more visible. Ideally, crates.io would include some kind of bloat report along with its other metadata, although using fully automated tools has limitations (for example, a "hello world" example using clap might be pretty modest, but one with hundreds of options might be huge).

Once you accept bloat, it's very hard to claw it back. If your project has multi-minute compiles, people won't even notice a 10s regression in compile time. Then these pile up, and it gets harder and harder to motivate the work to reduce bloat, because each second gained in compile time becomes such a small fraction of the total.

Conclusion

As druid develops into a real GUI, I'll be facing many more of these kinds of choices, and both compile times and executable sizes will inevitably get larger. But avoiding bloat is just another place to apply engineering skill. In writing this blog post, I'm hoping to raise awareness of the issue, give useful tips, and enlist the help of the community to keep the Rust ecosystem as bloat-free as possible.

As with all engineering, it's a matter of tradeoffs. Which is more important for druid, having fast compiles, or being on board with the abundance of features provided by the Rust ecosystem such as fluent? That doesn't have an obvious answer, so I intend to mostly listen to feedback from users and other developers.

Discuss

Lively discussion on Reddit, Hacker News, and lobste.rs.

Raph Levien's blog

Raph Levien's blog
raph.levien@gmail.com

raphlinus
raph

Blog of Raph Levien.