- Feature Name: N/A
- Start Date: 2015-07-29
- RFC PR: rust-lang/rfcs#1242
- Rust Issue: N/A

Summary

This RFC proposes a policy around the crates under the rust-lang github organization that are not part of the Rust distribution (compiler or standard library). At a high level, it proposes that these crates be:

- Governed similarly to the standard library;
- Maintained at a similar level to the standard library, including platform support;
- Carefully curated for quality.

Motivation

There are three main motivations behind this RFC.

Keeping `std` small. There is a widespread desire to keep the standard library reasonably small, and for good reason: the stability promises made in `std` are tied to the versioning of Rust itself, as are updates to it, meaning that the standard library has much less flexibility than other crates enjoy. While we *do* plan to continue to grow `std`, and there are legitimate reasons for APIs to live there, we still plan to take a minimalistic approach. See this discussion for more details.

The desire to keep `std` small is in tension with the desire to provide high-quality libraries *that belong to the whole Rust community* and cover a wider range of functionality. The poster child here is the regex crate, which provides vital functionality but is not part of the standard library or basic Rust distribution -- and which is, in principle, under the control of the whole Rust community.

This RFC resolves the tension between a "batteries included" Rust and a small `std` by treating `rust-lang` crates as, in some sense, "the rest of the standard library". While this doesn't solve the entire problem of curating the library ecosystem, it offers a big step for some of the most significant/core functionality we want to commit to.

Staging `std`. For cases where we do want to grow the standard library, we of course want to heavily vet APIs before their stabilization. Historically we've done so by landing the APIs directly in `std`, but marked unstable, relegating their use to nightly Rust. But in many cases, new `std` APIs can just as well begin their life as external crates, usable on stable Rust, and ultimately stabilized wholesale. The recent `std::net` RFC is a good example of this phenomenon.

The main challenge to making this kind of "`std` staging" work is getting sufficient visibility, central management, and community buy-in for the library prior to stabilization. When there is widespread desire to extend `std` in a certain way, this RFC proposes that the extension can start its life as an external rust-lang crate (ideally usable by stable Rust). It also proposes an eventual migration path into `std`.

Cleanup. During the stabilization of `std`, a fair amount of functionality was moved out into external crates hosted under the rust-lang github organization. The quality and future prospects of these crates varies widely, and we would like to begin to organize and clean them up.

Detailed design

The lifecycle of a rust-lang crate

First, two additional github organizations are proposed:

- rust-lang-nursery
- rust-lang-deprecated

New cratess start their life in a `0.X` series that lives in the rust-lang-nursery. Crates in this state do not represent a major commitment from the Rust maintainers; rather, they signal a trial period. A crate enters the nursery when (1) there is already a working body of code and (2) the library subteam approves a petition for inclusion. The petition is informal (not an RFC), and can take the form of a discuss post laying out the motivation and perhaps some high-level design principles, and linking to the working code.

If the library team accepts a crate into the nursery, they are indicating an *interest* in ultimately advertising the crate as "a core part of Rust", and in maintaining the crate permanently. During the 0.X series in the nursery, the original crate author maintains control of the crate, approving PRs and so on, but the library subteam and broader community is expected to participate. As we'll see below, nursery crates will be advertised (though not in the same way as full rust-lang crates), increasing the chances that the crate is scrutinized before being promoted to the next stage.

Eventually, a nursery crate will either fail (and move to rust-lang-deprecated) or reach a point where a 1.0 release would be appropriate. The failure case will be determined by means of an RFC.

If, on the other hand, a library reaches the 1.0 point, it is ready to be promoted into rust-lang proper. To do so, an RFC must be written outlining the motivation for the crate, the reasons that community ownership are important, and delving into the API design and its rationale design. These RFCs are intended to follow similar lines to the pre-1.0 stabilization RFCs for the standard library (such as collections or Duration) -- which have been very successful in improving API design prior to stabilization. Once a "1.0 RFC" is approved by the libs team, the crate moves into the rust-lang organization, and is henceforth governed by the whole Rust community. That means in particular that significant changes (certainly those that would require a major version bump, but other substantial PRs as well) are reviewed by the library subteam and may require an RFC. On the other hand, the community has broadly agreed to maintain the library in perpetuity (unless it is later deprecated). And again, as we'll see below, the promoted crate is very visibly advertised as part of the "core Rust" package.

Promotion to 1.0 requires first-class support on all first-tier platforms, except for platform-specific libraries.

Crates in rust-lang may issue new major versions, just like any other crates, though such changes should go through the RFC process. While the library subteam is responsible for major decisions about the library after 1.0, its original author(s) will of course wield a great deal of influence, and their objections will be given due weight in the consensus process.

Relation to `std`

In many cases, the above description of the crate lifecycle is complete. But some rust-lang crates are destined for std. Usually this will be clear up front.

When a std-destined crate has reached sufficient maturity, the libs subteam can call a "final comment period" for moving it into `std` proper. Assuming there are no blocking objections, the code is moved into `std`, and the original repo is left intact, with the following changes:

- a minor version bump,
- *conditionally* replacing all definitions with `pub use` from `std` (which will require the ability to `cfg` switch on feature/API availability -- a highly-desired feature on its own).

By re-routing the library to `std` when available we provide seamless compatibility between users of the library externally and in `std`. In particular, traits and types defined in the crate are compatible across either way of importing them.

Deprecation

At some point a library may become stale -- either because it failed to make it out of the nursery, or else because it was supplanted by a superior library. Nursery and rust-lang crates can be deprecated only through an RFC. This is expected to be a rare occurrence.

Deprecated crates move to rust-lang-deprecated and are subsequently minimally maintained. Alternatively, if someone volunteers to maintain the crate, ownership can be transferred externally.

Advertising

Part of the reason for having rust-lang crates is to have a clear, short list of libraries that are broadly useful, vetted and maintained. But where should this list appear?

This RFC doesn't specify the complete details, but proposes a basic direction:

- The crates in rust-lang should appear in the sidebar in the core rustdocs distributed with Rust, along side the standard library. (For nightly releases, we should include the nursery crates as well.)

- The crates should also be published on crates.io, and should somehow be *badged*. But the design of a badging/curation system for crates.io is out of scope for this RFC.

Plan for existing crates

There are already a number of non- `std` crates in rust-lang. Below, we give the full list along with recommended actions:

Transfer ownership

Please volunteer if you're interested in taking one of these on!

- rlibc
- semver
- threadpool

Move to rust-lang-nursery

- bitflags
- getopts
- glob
- libc
- log
- rand (note, @huonw has a major revamp in the works)
- regex
- rustc-serialize (but will likely be replaced by serde or other approach eventually)
- tempdir (destined for `std` after reworking)
- uuid

Move to rust-lang-deprecated

- fourcc: highly niche
- hexfloat: niche
- num: this is essentially a dumping ground from 1.0 stabilization; needs a complete re-think.
- term: API needs total overhaul
- time: needs total overhaul destined for std
- url: replaced by https://github.com/servo/rust-url

Drawbacks

The drawbacks of this RFC are largely social:

- Emphasizing rust-lang crates may alienate some in the Rust community, since it means that certain libraries obtain a special "blessing". This is mitigated by the fact that these libraries also become owned by the community at large.

- On the other hand, requiring that ownership/governance be transferred to the library subteam may be a disincentive for library authors, since they lose unilateral control of their libraries. But this is an inherent aspect of the policy design, and the vastly increased visibility of libraries is likely a strong enough incentive to overcome this downside.

Alternatives

The main alternative would be to not maintain other crates under the rust-lang umbrella, and to offer some other means of curation (the latter of which is needed in any case).

That would be a missed opportunity, however; Rust's governance and maintenance model has been very successful so far, and given our minimalistic plans for the standard library, it is very appealing to have *some* other way to apply the full Rust community in taking care of additional crates.

Unresolved questions

Part of the maintenance standard for Rust is the CI infrastructure, including bors/homu. What level of CI should we provide for these crates, and how do we do it?