rust-lang / **rfcs**

RFC: add futures and task system to libcore #2418

Open **aturon** wants to merge 2 commits into `rust-lang:master` from `aturon:async-trait`

Conversation 145 Commits 2 Checks 0 Files changed 1 +916 −0

aturon commented on 25 Apr • edited Member

Note: this is a heavily-revised, more conservative version of #2395

This RFC provides the library component for the first-class `async` / `await` syntax proposed in a companion RFC. It is intentionally minimal, including the smallest set of mechanisms needed to support async/await with borrowing and interoperation with the futures crate. Those mechanisms are:

- The task system of the futures crate, which will be moved into `libcore`
- A `Future` trait, which integrates the `PinMut` APIs with the task system to provide futures (i.e. asynchronous values).

Rendered

👍 37 🎉 26 ❤️ 20

aturon added the **T-libs** label on 25 Apr

aturon self-assigned this on 25 Apr

aturon referenced this pull request on 25 Apr
RFC: add futures to libcore #2395 Closed

withoutboats referenced this pull request on 25 Apr
async/await notation for ergonomic asynchronous IO #2394 Merged

`RFC: add Async trait and task system to libcore` `7d364aa`

aturon commented on 25 Apr • edited Member

Summary of differences from the original RFC:

- The main trait is called `Async` rather than `Future`, to align with `async` blocks; it now lives in `core::ops`.
- No `AsyncResult` in the proposal; this may be provided in the futures crate.
- No combinators are included in the proposal; they'll be provided completely out of tree in the futures crate.
- Futures 0.3 is a more conservative transition that immediately works on stable Rust, but is also compatible with async/await on nightly.
- There's a minimal path to stabilization that is based almost entirely on tech that has already been vetted for months or years.

❤️ 21

aturon referenced this pull request in **rust-lang-nursery/futures-rs** on 25 Apr
Convert basic futures combinators to futures-core 0.3 #980 Merged

ashfordneil commented on 25 Apr

Firstly, is `Executor` in this rfc a trait or an object? It is defined as a trait, but then used as an object in the `Context::new` function, I'm not sure if this was intentional due to the "final `Executor` trait" not being ready, but right now it's a little confusing.

Reviewers
carllerche ✓
eddyb
kennytm
RalfJung
nrc
pythonesque
aidanhs
YaLTeR
tmccombs
bugaevc

At least 1 approving review is required to merge this pull request.

Assignees
aturon

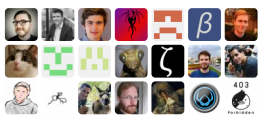
Labels
T-libs

Projects
None yet

Milestone
No milestone

Notifications

32 participants



and others

Secondly, why are `Waker`, `Context` and (possibly?) `Executor` all trait objects / structs instead of just traits? This introduces an extra layer of dynamic dispatch (on top of the probable boxing of `Async` that any executor would have to do) to poll any future that interacts with its context. As an alternative that would reduce the amount of dynamic dispatch, we could make `Async::poll` a method that is generic over `C: Context`.

As far as raw implementations of the `Async` trait are concerned, writing a generic `poll` method shouldn't be any harder than writing a `poll` method that uses trait objects in its context. Writers of `async` functions or end users of combinators shouldn't be affected at all by the change. The only potential issue I can see is how to make `Async` usable as a trait object (for an executor) with this change. I had a look at the problem, and I think it can be solved.

Basically, instead of the executor making a trait object out of `Async` and then working with those objects, the executor could define its own trait that has a monomorphised `poll` method (to work specifically with its implementation of `Context`) and work with a trait object of that. I've attached a playground that does just this below, and creation of the trait objects all compiles without issue so I *think* it would be fine.

https://play.rust-lang.org/?gist=079aa77b8679fcf80fd9d56cf7fd00cb&version=stable

Thinking on this, there could be some situations - `Async` aware mutexes come to mind - where you may still need to create trait objects of the `Waker` trait to store them. This change wouldn't prevent anyone from creating that trait object themselves if they needed to, but it wouldn't default to handing out trait objects, meaning that in the situations where a trait object isn't necessary, the trait object could be avoided.

Sorry for the wall of text, but what are your thoughts? At the moment I can't see any real disadvantage to doing this*, but there could be some big issue that I missed entirely, so I look forward to hearing other people's opinions on this.

*my playground doesn't use `Pin` yet so here's hoping that doesn't break everything 🤞

👍 2

✓ **carllerche** approved these changes on 25 Apr

View changes

As mentioned in the previous RFC, this is roughly in line with what I had hoped the end result would be 👍. I included a couple of nits / questions inline.

`text/0000-async.md`

```
143 +/// Any task executor must provide a way of signaling that a task it owns
144 +/// is ready to be `poll`ed again. Executors do so by providing a wakeup handle
145 +/// type that implements this trait.
146 +pub trait Wake: Send + Sync {
```

carllerche on 25 Apr **Member**

Why does this trait need to be moved into `core`? If `UnsafeWake` exists in core, then would it not be possible to leave this trait in a crate?

aturon on 25 Apr **Member**

Yes, this could live out of tree to start with. To be clear, though, I do expect more of the futures crate to move into core over time.

Reply…

`text/0000-async.md`

```
222 +/// Provides the reason that an executor was unable to spawn.
223 +pub struct SpawnErrorKind { .. }
224 +
225 +impl SpawnErrorKind {
```

carllerche on 25 Apr **Member**

Tokio currently requires an "at capacity" error variant.

aturon on 25 Apr **Member**

👍

Reply…

text/0000-async.md

```
331 +    /// spawn failures.
332 +    ///
333 +    /// NB: this will remain unstable until the final `Executor` trait
     is ready.
334 +    pub fn executor(&mut self) -> &mut BoxExecutor;
```

carllerche on 25 Apr **Member**

`BoxExecutor` is not defined anywhere. Is it possible to implement `FuturesUnordered` with this current definition?

Also, given the known issues* with this strategy, would it be wiser to punt on executor being tied to context?

* spawning in drop and spawning from a function.

aturon on 25 Apr **Member**

This should be `Executor`.

We can always delay on stabilizing this portion of the API if we are feeling uneasy. Remember, everything that goes into std goes on the *nightly* channel indefinitely; there's a separate process for actually committing it to stable.

Reply…

MajorBreakfast commented on 25 Apr • edited Contributor

A little downside of `Async` is that there is also the `async` keyword, so you can't simply create variables that are called "async".

```
fn main() {
    let async = print_async(); // No no
    println!("Hello from main");
    futures::block_on(async);
}
```

Edit: In my PR to the companion RFC, I've just called it `my_async` to avoid the problem.

aturon referenced this pull request in **rust-lang-nursery/futures-rs** on 25 Apr

Bring 0.3 branch up to date with new RFC #986 Closed

MajorBreakfast commented on 25 Apr • edited Contributor

Since the variable cannot be called "async", it could be consistently called "op" in the documentation. A consistent name would be good. "op" would be wonderfully short

```
let op = print_async(); // Can't call it "async"!
futures::block_on(op);
```

Edit:
Sry for bikeshedding, but this variable thing somehow really annoys me. (And it looked so promising!) It's inconvenient for everyday code° and it's inconvenient for the documentation.

° Example: In JavaScript, when I create a `Promise`, I usually call it "`promise`". This is non-imaginative, I know, but if I have only one it's also super clear and I don't have to come up with a name. Instead, I use the lowercase variant and can continue coding right away.

Alternative names for `Async`:

- `Task`: Like C#, it's even shorter and it is really descriptive of what it we intend it to represent
- `Promise`: Like JavaScript. Not as descriptive because "promise" implies a bit that it is already in progress. However, Rust's implementation is lazy.

I think the similarity between the `async` keyword and the `Async` trait was an idea with a really good intention. I thought for a long time that it'd be really nice if there was a symmetry here. I liked the previous "`Future`" well enough, though, so I never voiced that. But, now I start to see why no other language (that I'm aware of) decided to actually call it "`Async`". 🙂It's inconvenient because of the `async` keyword!

😕 3

MajorBreakfast referenced this pull request in **withoutboats/rfcs** on 25 Apr

Further changes #6 Closed

yasammez commented on 25 Apr

I am not bothered by the naming issue. Normally when I create a future in JS I either await it on the spot (no need for a binding) or gather multiple different things to await them at once later in which case I give them "business" names like customerPromise, basketPromise, etc.. I have never been tempted to name anything "async" :-)

👍 14 ❤️ 1

MajorBreakfast commented on 25 Apr • edited Contributor

@yasammez In real code you have that option, yes. But, in documentation where the intention is to stay brief and generic it's problematic. It'll bite us, I'm telling you

@alkis suggests "`Lazy`" as an alternative in the async/await RFC.

👎 1

rushmorem commented on 25 Apr

> A little downside of Async is that there is also the async keyword, so you can't simply create variables that are called "async".

@MajorBreakfast In that case, why not just name the variable `future`?

MajorBreakfast commented on 25 Apr Contributor

@rushmorem Unfortunately that'd be misleading. The new RFC standardizes `Future` under a different name to highlight the differences to `Future` from the futures crate (this was first proposed by **@aturon**). The two traits are very similar, but they're not the same. Good documentation has to be very precise in such matters to not confuse beginners

tmccombs commented on 25 Apr

`Task` is already taken for the overall task.

MajorBreakfast commented on 25 Apr • edited Contributor

@tmccombs Exactly. "overall task". The meaning of "overall task" would be clearer if it consisted of child tasks. Currently we have to explain what an `Async` is and then what a task is. But a task is really just a top level and scheduled `Async` ~~with `Output=()`~~. We could just call both tasks and be done with it 🙂

rpjohnst commented on 25 Apr

A task is more than simply an `Async<Output=()>` - it is a *scheduled* `Async` (with any `Output`).

glaebhoerl commented on 25 Apr Contributor

(It doesn't feel right to me to name a variable `async` in the first place because "async" (unlike "future") isn't even a noun.)

👍 15

aidanhs reviewed on 25 Apr — View changes

`text/0000-async.md` — Show outdated

MajorBreakfast commented on 25 Apr — Contributor

@rpjohnst I've edited my post above to reflect this. My point stands, though: Having two terms for this makes it just more complicated than it needs to be. I mean, it's easier to understand if we don't have that. One less thing to learn and such

@glaebhoerl Good point. " `reader_async` " vs " `reader_task` "... "async" just doesn't sound so well because it's an adjective

aturon commented on 25 Apr — Member

@MajorBreakfast I think that **@rpjohnst**'s point is that there are two distinct concepts here: async values, and tasks. A task is something that is being *actively run* by an executor, whereas an async value is by itself inert. Tasks generally work with async values internally, and are responsible for driving them to completion.

Understanding this distinction is critical to understanding Rust's unique approach to async programming, so I personally find having distinct terms for the concepts to be very helpful in teaching.

Regarding the larger question: I think that `Async` works very well as a name, both because of the very direct relationship to the construction form (`async` blocks) and because it is generally descriptive: `Async<Output = T>` is an async computation that produces a `T` . I understand the concern about examples in generic documentation, but my perspective is that docs where you're tempted to write `let future = ...` could probably be improved by giving a more meaningful name and computation anyway -- the same way that the best documentation avoids using `foo` / `bar` /etc in favor of more real names/examples.

👍 12

MajorBreakfast commented on 25 Apr • edited — Contributor

@aturon

> A task is something that is being actively run by an executor, whereas an async value is by itself inert.

Does this difference really merit its own terminology? I mean a "moving car" is still called a "car". No special term required to make the difference clear.

> my perspective is that docs where you're tempted to write let future = ... could probably be improved by giving a more meaningful name and computation anyway

I'd say it depends. Sometimes it'd just be irrelevant or confusing. For instance in the example code for `Future::map()` .

rpjohnst commented on 25 Apr

It's very similar to the distinction between "process" and "program." A task has extra data associated with it beyond the `impl Async` that it's running.

MajorBreakfast commented on 25 Apr • edited Contributor

@rpjohnst Yeah sure. But the user doesn't need to care about such implementation details because they're invisible. As far as the user is concerned it's a "running task". (Or a "running program" in your example)

👍 1

aturon commented on 25 Apr Member

@MajorBreakfast But that's just the thing: an `Async` is *not* a running task, and until it's driven by one it will do nothing. It's like the difference between a `FnOnce() -> ()` and a running thread.

👍 2

MajorBreakfast commented on 25 Apr • edited Contributor

> `Async` is *not* a running task

@aturon Yes. it'd be just a " `Task` " at first. True to the meaning of the english word "task": a piece of work. Then once it's driven by an executor, it becomes a "running task".

kennytm reviewed on 25 Apr View changes

`text/0000-async.md`

```
207 +    /// An `Ok` return means the executor is *likely* (but not guaranteed)
208 +    /// to accept a subsequent spawn attempt. Likewise, an `Err` return
209 +    /// means that `spawn` is likely, but not guaranteed, to yield an error.
210 +    fn status(&self) -> Result<(), SpawnErrorKind> {
```

kennytm on 25 Apr Member

The documentation means there's basically no guarantee whether the return value corresponds to the reality, as there are both false positive and negative. That means you can't reliably use `if x.status().is_xxx()` to check anything in either Ok or Err direction. What is the purpose of this function then?

carllerche on 25 Apr Member

Hints are useful for fast paths.

tmccombs on 25 Apr

Is there a reason that if status returns an Err, spawn isn't guaranteed to return an Err?

👍 1

carllerche on 25 Apr Member

Not all error statuses are permanent.

tmccombs on 25 Apr

Then maybe, SpawnErrorKind should have an API to determine whether or not the executor will/may eventually recover from the error.

👍 1

bugaevc on 27 Apr

If it's a hint, the naming should clearly indicate that; e.g. `status_hint()`

👍 6

Reply…

MajorBreakfast commented on 25 Apr Contributor

I've searched for `struct Task` and `trait Task` inside the futures crate:

```
struct Task { // local_pool.rs
    fut: Box<Future<Item = (), Error = Never>>,
    map: LocalMap,
}

struct Task { // thread_pool.rs
    spawn: Box<Future<Item = (), Error = Never> + Send>,
    map: LocalMap,
    exec: ThreadPool,
    wake_handle: Arc<WakeHandle>,
}
```

- " `Execution` " would be a fitting° name as well
- they are private

(° unless I'm mistaken. I'm only judging by what these structs contain. I am not familiar with the code)

Alternatively, we could also call it **"promise"**. That'd address the conflict with the `async` keyword as well and the word "promise" is mentioned just once in the whole crate. JavaScript devs would feel right at home.

Unless something important comes up, this is my final comment on this subject. I'm letting this rest now.
Just imagine me telling you "told you so" whenever you can't type `let async = ...` 🤓😅

BatmanAoD referenced this pull request on 26 Apr

Async IO #1081 Closed

kellytk commented on 26 Apr

@MajorBreakfast Although I don't feel sufficiently informed to make a meaningful contribution to this thread, I'm sympathetic of your sensitivity to precise naming and I have a suggestion. A possible trait name alternative for `Async` is `Defer`, with a conventional identifier of `deferred`.

I haven't used `Futures` yet so please pardon me if the following syntax is incorrect:

```
let deferred: impl Defer = async { ... };
```

For reference from https://en.wiktionary.org/wiki/deferred#English:
"(accounting) Whose value is not realized until a future date: e.g. annuities, charges, taxes and income, either as an asset or liability."

👍 3 👎 3 ❤️ 2

lxrec commented on 26 Apr Contributor

In Javascript-land, "deferred" specifically refers to an object that contains a promise and the resolve() method for that promise, so the promise can get resolved by code "outside" the usual promise chain, which ends up being critical for use cases like client-side caches of async-ly fetched data. So I would avoid that particular word.

Still, if all we're after is a throwaway variable name for toy examples where the particular async code being executed is irrelevant, I think `op`, `task`, `tsk`, `future`, `fut`, `promise`, `asynk`, and so on are all perfectly adequate. Personally I'd probably go with `op` just because it's the shortest and seems impossible to confuse with any language mechanism. But like **@yasammez**, I just don't think this is a serious issue that warrants any further bikeshedding. For the serious bikeshed about what to call the traits, I have no objections to the proposed `Async` / `Future` names.

👍 9

yasammez commented on 26 Apr

I am strongly against Defer because it is just one typo away from Deref and **that** would be very confusing.

👍 15

MajorBreakfast commented on 26 Apr • edited ▾ Contributor

(I've removed a comment here. It didn't have responses yet and the approach in it was flawed (accidentally introduced virtual dispatch). Doesn't make sense to talk about it. This is just an info in case you've got it via email or you've read it.)

tanriol commented on 27 Apr

There was a suggestion in the previous discussions that `Pending` should contain a token indicating you know its contract

```
/// A token that promises a future wakeup
///
/// By naming this struct, you promise that you have ensured
/// that the task will be notified when it can proceed.
///
/// If you poll an inner `Async`, you should reuse the token it returns.
/// However, if you have multiple inner `Async`s, be careful to poll
/// all of them that are active.
pub struct PromiseFutureWakeupIsScheduled;

pub enum Poll<T> {
    /// Represents that a value is immediately ready.
    Ready(T),

    /// Represents that a value is not ready yet.
    ///
    /// When a function returns `Pending`, the function *must* also
    /// ensure that the current task is scheduled to be awoken when
    /// progress can be made. It contains a zero-sized token
    /// that you should either get from an inner `Async` poll result
    /// or create manually if there's no inner `Async`.
    Pending(PromiseFutureWakeupIsScheduled),
}
```

Have the reasons not to do this been documented anywhere? Should it be documented as an alternative?

👍 6

eddyb commented on 27 Apr **Member**

@tanriol By making that struct not publicly constructable, we can enforce a bit more, nice!

bugaevc commented on 27 Apr

Another idea for the wakeup tokens I've seen is to make them non-ZST *in debug mode* to make them carry an explanation of where that token originates from, i.e. *how/where* the task was scheduled to be woken up; this can be (supposedly) very useful for debugging what a particular future is waiting for / blocked on.

👍 8

bugaevc reviewed on 27 Apr View changes

`text/0000-async.md` Show outdated

bugaevc reviewed on 27 Apr View changes

`text/0000-async.md`

```
115 +/// Indicates whether a value is available, or if the current task has
     been
116 +/// scheduled for later wake-up instead.
117 +#[derive(Copy, Clone, Debug, PartialEq)]
118 +pub enum Poll<T> {
```

bugaevc on 27 Apr

Should `Poll` impl `Try` ?

Reply…

tanriol commented on 27 Apr

@eddyb Then you need to somehow get a token if you need to schedule a wakeup manually. Maybe `Context::waker` could return `(&Waker, PromiseFutureWakeupIsScheduled)`.

eddyb commented on 27 Apr • edited — Member

~~**@tanriol** Why not return `PromiseFutureWakeupIsScheduled` from `Waker::wake` ?~~

tanriol commented on 27 Apr

@eddyb `Waker::wake` will be called before the next poll, while we need a token to return from this poll.

eddyb commented on 27 Apr • edited — Member

@tanriol Oh, I see, I got confused so the `Waker` is an object you could keep around (e.g. move to another thread) and use it inform the original event loop to run that specific future?
Shouldn't `Context::waker` return `Waker` by value then, since you'd always clone it?

👍 1

MajorBreakfast commented on 27 Apr • edited — Contributor

> Shouldn't Context::waker return Waker by value then, since you'd always clone it?

@eddyb I think it's more performant to return a reference. Then, it can be cloned only when required. E.g. cloning is not required if the `Async` is `Poll::Ready` immediately. (Edit: True, but then you don't need it at all. See the following comments for a real use case) The RFC indicates that `Waker` might use an `Arc` under the hood.

carllerche commented on 27 Apr — Member

@eddyb One does not always clone it. For example:

```
waker().wake() // yield.
```

MajorBreakfast commented on 27 Apr • edited — Contributor

@eddyb Uhm wait. What I said is not entirely correct.
@carllerche When would you do that? If it's already ready you'd return `Ready` when polled. If it's not ready you need to clone the waker to send it to another thread, right? Is it useful to divide the work into chunks? (yield from time to time, even if progress can still be made)

carllerche commented on 27 Apr — Member

@MajorBreakfast yes, it is very useful and is a pretty commonly used pattern today.

In cooperative multi tasking, you must not hold the thread for too long.

👍 2

seanmonstar commented on 27 Apr Contributor

@MajorBreakfast two cases where not cloning is useful:

1. A future has done some work, perhaps in a loop, and wants to allow yielding to any other futures on the thread, it'd just call `cx.waker().wake()` and return `Pending`, so the executor can poll others before coming back to this one.
2. If you have already saved a `Waker`, and you cannot make more progress, you may wish to check that the `cx.waker()` would wake the same task as the current one (futures 0.2 calls this `Waker::will_wake(other_waker)`), allowing to skip a clone. This one is important, as unnecessary cloning of the waker has appeared in server profiles.

👍 2

fredrikroos commented on 28 Apr • edited

> **@tanriol** By making that struct not publicly constructable, we can enforce a bit more, nice!

@eddyb Wouldn't that prevent any new implementations of leaf futures?

eddyb commented on 30 Apr Member

@fredrikroos How so? I think #2418 (comment) is a good way to get the token.

fredrikroos commented on 1 May

@eddyb Yes you are correct. I was confused by all the new terminology compared with futures 0.1, so I missed that part of the suggestion.

thomaseizinger commented on 2 May

A few thoughts on the variable name of `Async<T>`:

As noted, `async` is not a noun. One way of naming it would be `let async_t: Async<T> = ...`, which makes more sense from the perspective of the English language.
However, what comes to my mind when writing this is: Rust's view on variable shadowing.
In other languages, one would probably add such suffixes / prefixes in order to make it clear, what the variable holds, although it is actually some sort of type information. To my understanding, Rust encourages variable shadowing, so why not just name it after the actual value that is computed asynchronously, which in this case would be: `let t: Async<T> = ...`?
As soon as the value is computed, one can just shadow the previous value with the result. The compiler takes care of not mixing those up anyway.

The fact that it is some sort of type information discourages at least myself from including that in the variable name, because I also don't see myself writing: `let vec_names: Vec<String> = ...` but rather just `let names: Vec<String> = ...`. In addition, IDEs like CLion have features like type hints, which always display you the type of a variable if you don't explicitly state it, which is super convenient.

```
let bytes :Vec<u8> = responder.recv_bytes( flags: zmq::SNDMORE).unwrap();
```

👍 4

ghost commented on 2 May

I see nobody responded to / addressed the comment by **@ashfordneil** near the start of this thread.

I think those are valid concerns and I'd like some discussion or at least clarification around it. I tend to agree with it, but I might be misunderstanding something. Is there some reason why the API has to enforce trait objects everywhere?

I think that dynamic dispatch should be avoided/opt-in if possible, to avoid indirection and unnecessary performance overheads. It doesn't seem like it should be necessary in this case.

If this library API proposal can be changed so that it is done in terms of generics rather than trait objects, I think that would be better (as long as there are no major drawbacks). If it cannot, or if there are good reasons to use trait objects instead, I'd like to understand why, so please clarify what I am missing.

MajorBreakfast commented on 2 May Contributor

@rayvector I think that is because there is more than one kind of executor. Currently that's `LocalExecutor` and `ThreadPool`. At the time a future is created, it is not known which executor will be used on it. Dynamic dispatch is probably unavoidable in this case for compatibility with all possible executors. That said, I'd also like a clear explanation about where dynamic dispatch is used and why. This probably has been thought through thoroughly. I'm really curious about all this as well!

ashfordneil commented on 2 May

@MajorBreakfast There are multiple kinds of executors, but to the best of my knowledge each program is probably only going to pick one (type of) executor, and then put all of their ~~futures~~ asyncs in that. If you keep it a generic method, there is no need to know at creation time which executor will be used on it - each ~~future~~ async would have the capability to work with all executors as part of the `Async` trait.

sdroege commented on 2 May

> but to the best of my knowledge each program is probably only going to pick one (type of) executor, and then put all of their asyncs in that

@ashfordneil That's not necessarily true, e.g. I can easily imagine GTK applications using tokio and the GLib based executor. But each future/async is of course only ever spawned on a single executor

If you made `Context` generic over the waker/executor/etc, the generic type parameters would leak into basically everything (unless I'm missing something). Including all definitions of futures, combinators, functions taking futures, ... Usability-wise it wouldn't be very nice at all.

withoutboats commented on 2 May • edited Contributor

Really glad to see the libs side of things driving toward consensus. :) I'm mostly comfortable with the revisions in this RFC vs the old one, but I do have one quibble.

I think the change from `Future` to `Async` for the core trait is not a great move, mostly because of how it impacts the messaging around this system. We've built up a lot of understanding & awareness around the concept of "futures" for nearly two years now - ever since Zero-cost futures in Rust in 2016. If you don't follow the popularity & traction of different Rust blog posts like I do: this blog post was one of the most widely read posts about Rust **ever**, probably only the 1.0 announcement has gotten more attention. We've established, from then on, that the async IO story in Rust is built around futures.

And this appears in the code, not only in discourse. The crate that everyone will use to manipulate what in the RFC is called an `Async` is the `futures` crate. I expect to use the futures crate for utilities about futures, not utilities about "asyncs." Our documentation describes these types as "futures," and as we've seen on this thread, calling them "asyncs" is problematic - both because it isn't a noun and it isn't a keyword. The keyword thing impacts downstream users too: you can have a `future` module, but you can't have an `async` module.

So I think overall, we're going to want to continue to talk about these as futures, both to keep a clear messaging continuity and because its just more natural and convenient. I think its a mistake for the trait that defines a future to not be called `Future`.

I also think that the return type of an async function really doesn't have much in common with operator overloading, and using the precedence of that is not really a good analogy. I don't think its a mistake that we didn't call an `Iterator` a `ForLoop`, even though it controls the operation of that syntax. Unlike `Mul::mul`, which you would only call directly in very odd situations, it will still be quite normal to manipulate a future using one of the methods on it, instead of using a built-in syntax. This is the same as how an iterator works, which is why I think `Iterator` is much better precedent than the ops traits.

Finally, I wouldn't want to cause a confusion in which only some things that are futures are considered futures, because we have this second trait called `Future` (for the Unpin &mut futures) which not all futures implement. I'd be very worried about a situation in which only things which implement that trait are considered a "future," and its hard to explain how they connect with "asyncs," the things that async functions return. I much prefer using terminology which allows casual observers to understand everything as a single concept (in much the same way that even though we have three fn traits, everyone understands that closures are a unified concept and not three different things).

This means we'd need a different name for the &mut poll trait in the futures crate. I don't have any good ideas or strong opinions for what this should be called - `FutureMut` ? I don't know.

👍 13 😕 3 ❤️ 1

RalfJung reviewed on 2 May View changes

`text/0000-async.md` Show outdated

MajorBreakfast commented on 2 May Contributor

@withoutboats

> This means we'd need a different name for the &mut poll trait in the futures crate

- `FutureMut` : Deceptive. Suggests that `Future` is immutable which is not the case.
- `BoxFuture` : Deceptive. If the future is already `Unpin` , no actual boxing is required.
- `StableFuture` : Deceptive. The time will come when both it and `Future` are stable.
- `UnpinFuture` : Descriptive and to the point. Sounds a bit complicated, but that's good because `Future` is the default choice. `UnpinFuture` is only needed in edge cases which require `Unpin` .

eternaleye commented on 3 May • edited

I'm going to note that, *unless* `Poll<T>` 's `Pending` variant ends up carrying something, I'm not convinced it pulls its weight vs. `Option<T>` - its structure is exactly the same, and the fact that it's returned from `Async::poll()` is sufficient to define its meaning. In addition, `Option<T>` has a pretty good amount of useful functionality; figuring out what subset of that is worth duplicating to `Poll<T>` is a cost.

Moreover, there's a clean little correspondence between `Iterator` and `Async` that becomes much clearer if `Poll<T>` is replaced with `Option<T>` :

Trait	`Async`	`Iterator`
Produces	`Output`	`Item`
Advanced by	`poll()`	`next()`
Steady state	`None`	`Some(Item)`
Termination	`Some(Output)`	`None`

👍 6 👎 2

ashfordneil commented on 3 May

@sdroege I'm not suggesting making context generic over it's executor and waker, I'm suggesting the stdlib doesn't even define a context struct, and we instead leave it as a trait. The only place where we then need to be generic is in the poll method of Future. This generic is an easy adjustment to make, as the Context trait offers every function that the current Context struct has, and it is also self contained. As each Future implementer needs to be able to be polled with any context there is no extra generics / type information that needs to be part of the Future trait.

The playground I posted originally shows a little overhead to get generics to work with trait objects, but that boilerplate doesn't go beyond what the playground has, and is only necessary when implementing an executor. End users of futures don't need to worry about this at all.

withoutboats commented on 3 May • edited Contributor

@eternaleye that becomes worse when you consider Stream, which would then return an `Option<Option<Self::Item>>`. (And is it the outer option that means that its finished and the inner that means that its pending, or the reverse?)

👍 8

MajorBreakfast commented on 3 May • edited Contributor

The RFC mentions "once we can take `dyn` by value". What is meant by that? (Maybe a link to comment or blog post. I can't find anything about this)

kennytm commented on 3 May Member

@MajorBreakfast Likely #1909.

👍 1

aturon commented on 3 May Member

Following up on a few threads of discussion here.

The design of `Poll`

There are a couple of related questions here.

Should we change `Pending` to take "evidence" that a waker has been queued?

I thought this was a *fascinating* idea when I first saw it proposed, and the futures team has talked about it in some depth. However, as with many such uses of the type system, I think in the end the cost/benefit isn't there.

Benefits: the perceived benefit is that "lost wakeup" bugs are caught by the compiler -- a big deal! However, it's important to be clear about what is really guaranteed: that if a polled future returns `Pending`, *somewhere* in that future a waker was created. It doesn't help with cases where wakers need to be enqueued in multiple locations.

In other words, this technique mostly applies to "leaf operations" (like reading from a socket) that are responsible for actually enqueuing wakers directly. IME, these operations tend to be relatively straightforward, and in fact the bulk of their logic *is* handling wakeups. On the other hand, composite operations that may be polling a variety of underlying futures/streams/sinks are trickier to get right, but they don't benefit from this technique, because they don't handle the waker instances directly. (In fact, composite code is probably marginally harmed, because it must now route a `Pending` instance from one of its underlying `poll` calls to "prove" that queueing has occurred).

Costs. For leaf operations the costs are marginal because the control flow tends to be straightforward. For composite operations, however, "proving" that you've enqueued a waker will require stashing a token from one of the underlying operations, which could obscure control flow in complex situations.

So, on the whole, I don't think this change makes sense to bake in to the *foundational* layer. That said, it's quite possible to build this kind of functionality on top of futures as they are today, and it could be worth continuing to experiment to see if we can provide an "checked futures" abstraction with better tradeoffs.

What about debugging?

A related idea that's been floating around is that we could use the `Pending` variant, in debug mode, to stash debugging information about why a wait is occurring.

That debugging goal is a very good one, but it turns out it doesn't need to be coupled to `Poll` -- the key point is in acquiring a waker. At the Rust All Hands in Berlin, there was a discussion about how to provide such debugging infrastructure more generally -- making it possible to "turn on" in production as well. You can find some details at the tracking issue, and this is something the net WG is interested in pushing on as futures 0.3 takes shape.

Should we consider just using `Option` instead?

While `Poll` is indeed isomorphic to `Option`, I think we gain a lot of clarity by having a distinct enum with clearly-labeled variants. IIRC at some point early on with futures we did use `Option`, and, especially with streams, it became very difficult to make sense of control flow that was matching on layers of options.

At this point, we have quite a lot of experience with a custom `Poll` enum, and I don't think there's much to worry about in terms of growing its API.

Whether to make `Context` a trait

This is an interesting idea that has been raised in other guises previously. The problem is that this essentially "punts" on questions of interoperation.

For example, take a futures combinator like `then`, which combines two futures. For that combination to work, the two underlying futures need to agree on their context type, which means that you need to introduce that constraint at the combinator level; that happens across the board. More worryingly, when you use `impl Future` or `Box<Future>` you need to specify the context type as well, and again worry about tying them together. Aside from making signatures more complex across the board, this opens the door to incompatible subecosystems that use different context objects.

It's worth noting that the `Context` type itself is *not* a trait object, but it contains two (the executor and the waker). Dynamic dispatch costs are incurred only when spawning or waking -- where such costs are almost certainly dwarfed by other costs like atomics.

Moreover, in general we rely on the ability to vary the type underlying the context in futures that work together -- for example, by allowing some portion of a future tree to have a different default executor, or in `futures_unordered`, where we decorate the waker with information needed to do internal scheduling.

The name of the trait

I have some thoughts on this, but they relate to a bigger proposal I want to make in a separate comment, so I'll circle back on that later!

👍 2 ❤️ 8

aturon commented on 4 May • edited Member

So in parallel with this thread I've been working through these changes in a 0.3 branch. I hit some snags with the initial strategy, due (you guessed it!) to trait coherence issues. In particular, if we have both `Async` and `Future`, it's not possible to provide conversion traits in *either* direction that enable you to be fully generic over both options (i.e., to accept either an `Async` or a `Future`). That in turn means that APIs have to pick one or the other, and clients have to write manual `into_future` / `into_async` boilerplate.

For that and other reasons, I took a step back to see if there was a synthesis of the approaches we've been discussing -- and I think I found one!

Goals

Let me start with what I'm trying to achieve:

1. Provide **good ergonomics** for working with "unpinned" (moveable) futures, so that you can write manual futures impls in a similar style to futures 0.2.
2. Have a **single, unified `Future` trait** that provides the above, but *also* works for the async/await case. In particular, avoid introducing a hard ecosystem split and make it easy to write fully generic code.
3. Put us in a position to **quickly produce a futures 0.3** (at least a beta) that works on stable Rust *today*, so that we can start building ecosystem support ASAP.
4. At the same time, allow this same ecosystem to **work with async/await syntax on nightly ASAP**.
5. Make it possible to generalize APIs from working only with unpinned futures to working with arbitrary futures, **without causing breakage**.

As it stands, the RFC provides a good story for 1, 3, 4. What I propose below tries to get all five goals.

Core idea

The RFC as it stands roughly boils down to:

```
trait Async {
    type Output;
    fn poll(self: Pin<Self>, cx: &mut task::Context) -> Poll<Self::Output>;
}

trait Future {
    type Output;
    fn poll(&mut self, cx: &mut task::Context) -> Poll<Self::Output>;
}
```

with adapters that convert in both directions, with the `Async` to `Future` direction requiring boxing.

But really, these two traits are talking about the same *concept*, just with different ownership constraints being applied to the client (`&mut` vs `Pin`). Moreover, if you work with `&mut` , then you work with `Pin` . So we can instead do:

```
trait Future {
    type Output;
    fn poll(self: Pin<Self>, cx: &mut task::Context) -> Poll<Self::Output>;
    fn poll_mut(&mut self, cx: &mut task::Context) -> Poll<Self::Output> where Self: Unpin
        // by default, use `poll` if it is provided
        Pin::new(self).poll(cx)
    }
}
```

and ultimately provide a default impl of `poll` for `Unpin` types, so that you only need to provide `poll_mut` :

```
default impl<T: Unpin> Future for T {
    // this partial impl of the Future trait makes it possible to impl Future by providing
    // *only* a `poll_mut` method
    fn poll(self: Pin<Self>, cx: &mut task::Context) -> Poll<Self::Output> {
        self.poll_mut(cx)
    }
}
```

Note: this "default impl" feature is not fully working yet, but there's a decent workaround below, using macros; I just wanted to be clear about what the eventual definition could look like.

What this looks like in practice

To cut to the chase: I've worked through the above ideas in a preliminary branch, and have enough working to talk through what it looks like.

Note: this is not the maximally ergonomic version; I'll make some notes about that below.

Implementing a moveable future

Here's an implementation of the `map` combinator that works only with `Unpin` futures, and hence is implemented in the "old" style:

```
struct Map<A, F> {
    future: A,
    f: Option<F>,
}

unpinned! {
    impl<U, A, F> Future for Map<A, F> where
        A: Future + Unpin,
        F: FnOnce(A::Output) -> U,
    {
        type Output = U;

        fn poll_mut(&mut self, cx: &mut task::Context) -> Poll<U> {
            let e = match self.future.poll_mut(cx) {
                Poll::Pending => return Poll::Pending,
                Poll::Ready(e) => e,
            };
            let f = self.f.take().expect("cannot poll Map twice");
            Poll::Ready(f(e))
        }
}
```

The `unpinned` macro wraps a `Future` impl such that you only need to provide the `poll_mut` method, and it takes care of the rest; it's a bit of boilerplate that would eventually be unneeded (when `default impl` is fully working), but gets the job done for now.

The actual implementation code looks like what you'd expect for 0.2-style futures, with only one other notable aspect: the `Unpin` bound on the inner future. This is what allows you to call `poll_mut` on the inner future, but means that this version of map is only compatible with moveable futures.

TL;DR: to migrate to 0.3, just wrap with `unpinned!` and add `Unpin` bounds, and you're good. No unsafe code, just a bit of boilerplate (which we can improve on down the line).

Implementing a fully generic future

Now, supposing we'd migrated as above, we might now want to take the next step and let the map combinator work with pinned futures as well. *We can do this as a backwards-compatible change*, because we'll just be expanding the set of futures we work with.

```
impl<U, A, F> Future for Map<A, F>
    where A: Future, F: FnOnce(A::Output) -> U
{
    type Output = U;

    fn poll(mut self: Pin<Self>, cx: &mut task::Context) -> Poll<U> {
        /* ... */
    }
}
```

So here, we've dropped the extra `Unpin` bound, and instead of using `unpinned!`, we impl `poll` directly. **Because there's just a single `Future` trait, the resulting combinator works with *all* futures, whether pinned or not, without any extra conversions**.

TL;DR: code that initially restricts to `Unpin` can be generalized, avoiding the need for clients to box unpinned futures.

Over time, we expect the ergonomics of working with `Pin` to dramatically improve (work has already started on a custom derive that provides fully safe accessors). But in the meantime, being able to migrate to 0.3 *without* working with `Pin` directly, while being smoothly compatible with code that does, seems like a win.

Details

There are a bunch of details to making this all work -- particularly in order to make it compatible with *stable* Rust -- but these are all just implementation details in the futures crate, and I've pushed the branch far enough along to be confident that the whole thing holds together. If there's general agreement about this direction, I'll revise the RFC with further details and complete work on the branch.

Tradeoffs vs the current RFC

I think this proposal is very much in the spirit of this RFC, in terms of allowing for a "two phase" migration (first to 0.3, then to pinned futures). The tradeoffs are:

Pros:

- A smoother interop experience: no conversion boilerplate
- A simpler conceptual story: no `Future` vs `Async` confusion
- A smoother migration path: can transition code to drop `Unpin` bounds without breaking clients

Cons:

- Boilerplate when implementing unpinned futures
- Puts `Unpin` a bit more front-and-center, which may increase the learning curve

While I think the cons can be further mitigated, I think the key point is that this puts *slightly* more burden on core library authors (having to use a macro) while decreasing friction throughout the ecosystem.

👍 7 😕 2 ❤️ 5

MajorBreakfast commented on 4 May Contributor

@aturon This is truly great! Awesome 😁

One little nitpick: Could you rename `poll_mut -> poll_unpin`?
Rust already has a few of these `..._mut` functions, e.g. there's `RefCell::borrow()` and `RefCell::borrow_mut()` where the `..._mut` differentiates between immutable vs mutable.
With `Future::poll()` and `Future::poll_mut()` the `..._mut` differentiates between pinning vs no pinning. So, using the `..._mut` suffix as well is a bit misleading. I think it'd be better if `..._unpin` was used instead.

👍 5

carllerche commented on 4 May **Member**

The latest proposal is a step back IMO.

> Have a single, unified Future trait that provides the above, but also works for the async/await case. In particular, avoid introducing a hard ecosystem split and make it easy to write fully generic code. (This is the new thing)

I disagree with having this as a goal. As I think about it, I am believe multiple traits are better suited as long as there is the appropriate interop.

Also, the separate associated types for `Item` and `Error` has been lost. As the benefits of this has been discussed at length earlier, I will not rehash.

This is roughly what I would like to see:

```
trait Async {
    type Output;

    fn poll(self: Pin<Self>, cx: &mut task::Context) -> Poll<Self::Output>;
}

trait Future: Unpin {
    type Item;
    type Error;

    fn poll(&mut self, cx: &mut task::Context) -> PollResult<Self::Item, Self::Error>;
}

impl<T: Future> Async for T {
    type Output = Result<T::Item, T::Error>;

    fn poll(self: Pin<Self>, cx: &mut task::Context) -> Poll<Self::Output> {
        Future::poll(self.get_mut(), cx)
    }
}

impl<T: Async> Future for BoxPin<T> {
    // impl
}
```

Note that `Future` is bound by `Unpin`.

Two separate traits

It will remain common to implement "future" by hand. I really do not think that it should be recommended that anyone default to `unsafe` code to work with `Pin<Self>`. This means the `Unpin` is required for `Future` to be the trait that is implemented by hand. Doing this also allows a blanket impl of `Async` for `T: Future` that requires no allocations or anything.

Of course, then given:

```
struct MyFuture<T> {
    inner: T,
}

impl<T: Future> Future for MyFuture<T> { ... }
```

One cannot create a `MyFuture` with a `T: Async`. **this is a feature** as this guards against using unsafe. If a `T: Async` needs to be passed into `MyFuture`, then it can be placed in a `BoxPin` (or whatever it is) to ensure it doesn't move... at this point it implements `Future`.

Smallest increments

Today, there is an ecosystem that works based on futures 0.1. I believe the most prudent way forward would be to keep what is known to work today while enabling `async / await`. Let the ecosystem form patterns around `async / await` and extract abstractions from that.

By moving `Async`, and only that trait into `std`, this is the smallest change that is needed to unblock this other feature. The existing ecosystem can be updated to use the new `Future: Unpin` trait allowing for interop and then the ecosystem can evolve based on the new capabilities provided.

Libraries vs. applications

> I think the key point is that this puts slightly more burden on core library authors

There is no line between library authors and applications. All applications include "library" like components. This proposal would put the burden on all.

👍 4

aturon commented on 4 May • edited — Member

@carllerche

I only have a brief moment, but just to reply to a couple of points:

> Also, the separate associated types for Item and Error has been lost. As the benefits of this has been discussed at length earlier, I will not rehash.

To clarify: the `Error` issue is orthogonal, and what I had in mind was to follow the strategy we reached consensus on before (and that's currently in the RFC) -- having a trait like `TryFuture` that includes the associated `Error` type and has a blanket impl.

> I am believe multiple traits are better suited **as long as there is the appropriate interop.** *(emphasis mine)*

```
impl<T: Future> Async for T
```

This is the key point: *if* we could have this impl, the situation would indeed be much better!

But sadly, this is what I mean about running into problems with coherence: a blanket impl like this conflicts with other important blanket impls for `Async` / `Future`, e.g. those for lifting over `Box` etc. I haven't found any way to make it work.

That's what I was getting at in saying that we end up not being able to write fully generic code, and instead have to force explicit `into_async` / `into_future` helpers everywhere you cross the "boundary".

> The existing ecosystem can be updated to use the new Future: Unpin trait allowing for interop and then the ecosystem can evolve based on the new capabilities provided.

The core problem I'm trying to highlight is that, in practice, we end up with a split with no migration path -- because there's no way to be generic over both `Future` and `Async`, it's not possible to write libraries that work transparently with both. Users have to manually convert even in the direction that doesn't require allocation.

> Today, there is an ecosystem that works based on futures 0.1. I believe the most prudent way forward would be to keep what is known to work today while enabling async / await.

I agree! I believe that the proposal I'm outlining gets us the closest to that goal, because it makes it possible to continue writing futures in the 0.1 style, use them in an async/await context, and gradually generalize libraries over time.

> It will remain common to implement "future" by hand. I really do not think that it should be recommended that anyone default to unsafe code to work with Pin.

I agree that implementing future by hand will remain important, and that it's not acceptable for that to commonly involve unsafe code. What both the current RFC and this new proposal aim for is a two-pronged approach:

- Retain the ability to write futures in the `&mut` style indefinitely.
- Over time, gain the ability to work with `Pin` without any unsafe code.

But the key issue I'm struggling with is interop and migration; I think it would be a grave mistake to end up with a split between these two styles that makes migration backwards incompatible, but without the ability to have a blanket impl, that's where we land. The revised proposal is trying to thread that needle.

> There is no line between library authors and applications. All applications include "library" like components. This proposal would put the burden on all.

To claify, I wasn't distinguishing between libs and apps, but rather *core* libs which are heavily writing manual future impls vs other libs where interop with async/await is more important.

In any case, the burden here is basically a macro invocation (in terms of stable Rust today), and with a bit more work in rustc it becomes essentially nothing: once we can provide the `default impl` I mention, you'll be able to implement `Future` providing only `poll_mut` with no other fanfare.

👍 3 ❤️ 3

aturon commented on 4 May **Member**

Oh, one other thing:

> One cannot create a MyFuture with a T: Async. this is a feature as this guards against using unsafe.

I didn't quite follow this -- IIUC, the same is true with e.g. the first map example I gave, which requires the future you give it to be `Unpin`.

aturon commented on 4 May **Member**

Here's a playground link illustrating the coherence problem. These impls are incoherent because a downstream crate could have:

- `struct Downstream` that implements `Future`
- `impl Async for Box<Downstream>`

which leads to *two* competing impls of `Async` for `Box<Downstream>`: one via the blanket impl, and one via the direct impl.

MajorBreakfast commented on 4 May Contributor

> ```
> impl<T: Future> Async for T
> ```
>
> This is the key point: if we could have this impl, the situation would be much better!

@aturon I think your new solution is much better instead! It addresses the problems **@withoutboats** lists his comment. The new solution keeps the messaging about futures a lot clearer. With it, there is only one `Future` trait and two ways to implement it `poll()` and `poll_mu...unpin()`. Much easier to understand!

YaLTeR reviewed on 4 May **View changes**

`text/0000-async.md`

```
206 +    ///
207 +    /// An `Ok` return means the executor is *likely* (but not
     guaranteed)
208 +    /// to accept a subsequent spawn attempt. Likewise, an `Err` return
209 +    /// means that `spawn` is likely, but not guaranteed, to yield an
     error.
```

YaLTeR on 4 May

`spawn => spawn_obj` ?

Reply…

`text/0000-async.md` Show outdated

`text/0000-async.md`

```
312 +pub struct Context<'a> { .. }
313 +
314 +impl<'a> Context<'a> {
```

```
315 +    pub fn new(waker: &'a Waker, executor: &'a mut Executor) ->
    Context<'a>
```

YaLTeR on 4 May

Missing `;` in the end.

Reply…

text/0000-async.md

```
160 +/// A `Waker` is a handle for waking up a task by notifying its executor
     that it
161 +/// is ready to be run.
162 +///
163 +/// This handle contains a trait object pointing to an instance of the
     `UnsafeWake`
```

YaLTeR on 4 May

It could make sense to mention that `UnsafeWake` is discussed further down in the RFC as to not get people confused about a type that doesn't appear until much further on.

Reply…

text/0000-async.md

```
545 +    /// to `wake` on the returned handle should be equivalent to calls
     to
546 +    /// `wake` on this handle.
547 +    ///
548 +    /// # Unsafety
```

YaLTeR on 4 May

Usually these sections are called `Safety` rather than `Unsafety`.

Reply…

carllerche commented on 4 May **Member**

@aturon Re your snippet, would it not work if `Async` is in `core` and `impl<T: Async> Async for Box<T> {}` exists in the same crate as `Box` ?

carllerche commented on 4 May **Member**

> I didn't quite follow this -- IIUC, the same is true with e.g. the first map example I gave, which requires the future you give it to be Unpin.

It's a question of defaults and what the API pushes you to do. Requiring an extra bound (`Unpin`) to be able to avoid `unsafe` is pushing implementors away from this path.

seanmonstar commented on 4 May Contributor

These are the goals that I have in mind when looking at this feature proposal:

1. We all came to Rust with the **promise of safety**. Yes, speed too. But without the safety promise, we'd just have C/C++. So, when introducing this new feature, it is paramount that we don't sacrifice safety.
2. While safety is a requirement, it's clear that *some* unsafety may needed to eek out performance. That's fine, but the default, easier way of doing things **must be safe first**. In the same way that users of Rust should reach for a `Box<Foo>` instead of `*mut Foo` , we should push users by default to using the safe way of building futures.

I'm not against speed. I want speed. But, I also have a healthy respect of reducing unsafe code. The thought "what's wrong with a little unsafe code that many people can analyze" is exactly the reasoning engineers use with their C/C++ projects, and **every single one** of them has security vulnerabilities.

If I apply those goals to the revised proposal, I see these issues:

- `Pin` and `Unpin` require very complex reasoning around `unsafe` code usage.
- The "safe" methods appear second-class, with suffixes and extra bounds, almost encouraging users to prefer to use the unsafe versions.

I feel that instead, the proposed feature should encourage using the safe mechanisms, like `&mut self`, by default. You want to implement a `Future`? Easy, safe. If someone feels clever enough, then they can reach for the escape hatch, but it is *they* who should be making the extra effort. Those wanting to be safe shouldn't feel they are being bothered because of it.

The Error type

> what I had in mind was to follow the strategy we reached consensus on before (and that's currently in the RFC) -- having a trait like `TryFuture` that includes the associated `Error` type and has a blanket impl.

I disagree that we reached consensus. Several people still feel that removing the error associated type is a net negative. This RFC also does *not* mention that the removal of the error type from `Future` had been decided on. Here's some quotes directly from the RFC text:

> The 0.3 release will continue to provide a `Future` trait that resembles the one in 0.2.

The `Future` trait in 0.2 has an `Error` associated type, so it seems fair to assume it would continue to do so, unless mentioned otherwise. It wasn't mentioned otherwise. There was this line, however:

> The question of whether to build in an Error type for Future, and other such details, will be tackled more directly in the futures repo.

So, if we're backing up and instead putting `Future` *back* into this RFC, then it seems we need to determine *if* there is an `Error` associated type.

The reasons against it's removal have been outlined plenty in the previous RFC. However, there are valid reasons for it as well. I believe that cons of an `Error` type can be reduced. Specifically, with a default associated type of `type Error = !`, with `async` being able to notice if the return type is `T` or `Result<T, E>`, the error type can nearly be invisible to people who don't need it.

Now then, I'll outline two proposals here, one that has two different traits, and one that tries to combine them, applying all that I've written up so far.

Proposal 1: `Async` and `Future`

Since the revised proposal mentions relying on specialization, it seems fair to consider what else specialization could be used for in. While it seems that we cannot have blanket impls mapping `Async`s to `Future`s and `Future`s to `Async`s, reading the specialization RFC, it certainly *should* be possible. Consider:

```
trait Async {
    type Output;

    fn poll_pin(self: Pin<Self>, cx: &mut Context) -> Poll<Self::Output>;
}

trait Future {
    type Item;
    type Error = !;

    fn poll(&mut self, cx: &mut Context) -> Poll<Result<Self::Item, Self::Error>>;
}

// A default blanket impl
impl<T: Future> Async for T {
    type Output = Result<T::Item, T::Error>;

    fn poll_pin(self: Pin<Self>, cx: &mut Context) -> Poll<Self::Output> {
        self.poll(cx)
    }
}

// Per specialization, Box<T> is more specific than T
impl<T: Async> Async for Box<T> {
    // ...
}

// In some other crate:
```

```
// Per specialization, Box<Foo> is more specific than Box<T> and T
impl Async for Box<Foo> {
    // ...
}
```

That this might not be possible *today* isn't the issue. If the above is incorrect, then the specialization RFC should be updated to clarify, *or* actually it should just be made to be correct, since otherwise is non-intuitive of specialization.

Notably in this proposal:

- Users reach for `Future` first. It is safe, and there are no additional speed bumps for them to do so. The method is called `poll`, because it's the normal way of doing things.
- Users don't worry about `Pin`, and don't worry about `Unpin`.
- Users wanting to go dragon hunting can resort to implementing `Async` themselves. The method is `poll_pin`, because it is unsafe to use, and should not actually be the easier method to write.

Proposal 2: just `Future`

If we really wanted a single trait, we could try to go that route. It would use similar concepts to the revised proposal, notably providing `default impl`s so you only need to implement either `poll` or `poll_pin`. However, I would change things to emphasize reaching for the safe interface first.

- Without an `unpinned` macro. Or at the least, if one is absolutely needed to allow turning on specialization in nightly, then just name the macro `future`.
- The methods should be `poll(&mut self)` and `poll_pin(self: Pin<Self>)`.
- A user shouldn't need to write `Future + Unpin`.
- There should be `type Error = !` in the trait.
- There would be `default impl`s both ways, so you would only write either `poll` or `poll_pin`.

Needing all of those things seems difficult to do for a single trait. Maybe even impossible. I think with improvements to specialization, and not worrying about how to name "asyncs", having two traits might end up being required. But I'm open to a single trait, if the points above can be accomplished.

👍 6

MajorBreakfast commented on 5 May — Contributor

@seanmonstar

- Safety: I agree with you that Rust should always encourage saftey
- Associated error type:
  - I don't buy the argument to keep it for compatibility reasons. Compatibility isn't important. It's all about stabilizing what is best. Whether there should be an associated error type or not should be solely determined by other arguments.
  - Reusing `Result` in the async ecosystem establishes nice parallels in the language that make the language feel like a cohesive whole. I prefer having only one associated type (`Output`) for this reason.
  - **@aturon** proposed a way for how `FutureResult` can work Could you comment on that? I don't think you've commented on that proposal yet
- Proposal 1: This reintroduces the downsides that **@aturon**'s proposal fixes. I like the one trait solution more because it's conceptually simpler and achieves more or less the same
- Proposal 2:
  - Point 1: The macro is only a temporary solution. So you're mainly proposing to rename the macro. I don't have an opinion on this ^^'
  - Point 2: Renaming the methods could make sense, yes. Implementing `poll_pin()` only makes sense when implementing a combinator, i.e. pinning is required. Most manually implemented futures aren't combinators and don't need pinning.
  - Point 3: AFAICT `Future + Unpin` is only required when writing a combinator like in **@aturon**'s example
  - Point 4: Is about associated error types. See above
  - Point 5: Both of **@aturon**'s examples only implement one of the two methods

👍 1 ❤️ 1

seanmonstar commented on 5 May — Contributor

@MajorBreakfast

> I don't buy the argument to keep it for compatibility reasons.

I wasn't arguing that it should be there for compatibility reasons at all. Rather, it seems incredibly common (in the context of networking, it is always) to need an error for a future. By not having one, you make all of async networking in Rust harder. Ideally, there'd be a really great reason to do so. But I haven't seen that great reason yet. It seems mostly to be about "purity", since the other problems are dealt with as I mentioned. Is there some other issue with that I haven't addressed?

Additionally, many other futures/promises systems in other languages include the concept of errors. I'd even go so far as to say Rust not including an error in its `Future` makes it stick out weirdly.

> Compatibility isn't important.

Saying compatibility isn't important isn't a great stance. Breaking hundreds of thousands of lines of code of people already using this stuff in production should be done with care. Not that it cannot happen, but it should be a) be a really good reason, and b) have a simple-ish way to upgrade.

> proposed a way for how FutureResult can work Could you comment on that?

It seems that proposal has been somewhat dropped, so I haven't picked it up myself. The landscape of trait aliases seems to be very fuzzy. I would only be satisfied with a solution that uses one where we can guarantee the exact way trait aliases will play out, which seems hard since they aren't implemented.

> Proposal 1: This reintroduces the downsides that @ aturon's proposal fixes.

No. I was highlighting that with specialization, the problems don't actually exist. If they exist in the currently nightly *implementation* of specialization, oh well. But the RFC says they should work. So, if the proposals are going to mentioned specialization (as aturon's revision already does), then we should be fair and consider how it can work with 2 traits.

With specialization and blanket impls, a pinned `Async` *is* a `Future`, and a `Future` *is* an `Async`. No weird conversions are needed.

> Most manually implemented futures aren't combinators and don't need pinning.

In my experience, I have not found this to be true. In fact, I feel the reverse is the case. I find that I have very few leaf futures, and more often write futures that depend on others (so, a de-facto combinator).

> AFAICT `Future + Unpin` is only required when [...]

My objection is to that requirement. There shouldn't be one at all, by default. A user should be able to write `Future` safely, without needing to clarify anything about pins. In the same way that you only need to write `unsafe` when you're actually doing something unsafe.

👍 2

jimmycuadra commented on 5 May

A potential way to avoid the "asyncs" naming problem would be to call the trait `PinFuture` instead of `Async`.

tmccombs commented on 5 May

> Rather, it seems incredibly common (in the context of networking, it is always) to need an error for a future

It may be common, perhaps even more common than not having a result, and I agree there should definitely be ergonomic ways of dealing with an `Async<Other=Result<_,_>>`. However, if Future does have an Error type, and poll returns a `Poll<Result<_,_>>`, then working with a Future that *doesn't* have an error case is significantly more awkward than it would be to work with a `Future<Output=Result<_,_>>` if Future doesn't have an Error type.

In particular, if poll always gives you a `Result`, then for Futures that don't have error cases you'll end up with a bunch of `unwrap()` calls with (hopefully) comments saying things like "This future never gives an Errors, so unwrap() will never panic."

👍 6

MajorBreakfast commented on 5 May • edited

Contributor

@seanmonstar

Associated error type

> it seems incredibly common (in the context of networking, it is always) to need an error for a future. By not having one, you make all of async networking in Rust harder.

This assumes that it is actually harder. I think it is equally easy to use with **@aturon**'s proposal.

> It seems that proposal has been somewhat dropped, so I haven't picked it up myself.

I don't think it has been dropped. The new minimal RFC only adds what is directly required for `async` / `await` to core/std. But this RFC is only the first step. As I see it, the only thing that has changed is that it will live in the futures crate at first, like other stuff, the combinators for instance, that doesn't fall into this category. Later, once the dust has settled, some parts of the futures crate will gradually be added to core/std. `FutureResult` and basic combinators for it are probably not far down the list.

Pinning

> A user should be able to write Future safely, without needing to clarify anything about pins.

> we should be fair and consider how it can work with 2 traits.

We should consider both. I agree.

- The two trait names sound so wildly different. Can be solved, as **@jimmycuadra** says, by calling them `Future` and `PinFuture`
- I don't want explicit conversions to be necessary. Does your code require conversions? How do I make an `Async` into a `Future` ? (not shown)
- **@carllerche**'s code has `Unpin` as super trait for `Future` . Yours does not. Why?
- **@aturon**'s proposal uses a macro to produce code that works immediately. Can something similar be done for your approach as well?

RalfJung commented on 5 May

Contributor

@aturon

```
trait Future {
    type Output;
    fn poll(self: Pin<Self>, cx: &mut task::Context) -> Poll<Self::Output>;
    fn poll_mut(&mut self, cx: &mut task::Context) -> Poll<Self::Output> where Self: Unpin
        // by default, use `poll` if it is provided
        Pin::new(self).poll(cx)
    }
}

default impl<T: Unpin> Future for T {
    // this partial impl of the Future trait makes it possible to impl Future by providing
    // *only* a `poll_mut` method
    fn poll(self: Pin<Self>, cx: &mut task::Context) -> Poll<Self::Output> {
        self.poll_mut(cx)
    }
}
```

So if I just use the `default impl` , I have `poll` defined in terms of `poll_mut` and vice versa... isn't there some danger here of accidentally creating a future that will just overflow the stack when called?

pythonesque commented on 5 May • edited

Contributor

> Over time, we expect the ergonomics of working with Pin to dramatically improve (work has already started on a custom derive that provides fully safe accessors). But in the meantime, being able to migrate to 0.3 without working with Pin directly, while being smoothly compatible with code that does, seems like a win.

@aturon Can you elaborate on where this is happening, out of interest? I am also working on one and I'd rather not duplicate effort, but the obvious approach (a method per field) breaks down when you consider enums, which I actually think are going to be a pretty common thing to want to `Pin`, and I'm also not sure how to make `Drop xor PinDrop` a reality in the presence of specialization (but I might be running into weird issues, since I don't have a good grasp on the current implementation of specialization that well).

pythonesque commented on 5 May Contributor

@tmccombs

> In particular, if poll always gives you a Result, then for Futures that don't have error cases you'll end up with a bunch of unwrap() calls with (hopefully) comments saying things like "This future never gives an Errors, so unwrap() will never panic."

I'm not too invested in that side of things (though I think not having an `Error` is probably the way to go), but with `never_type` you can have the error part of a `Result` be `!` and then everything works great(you can do `let Ok(foo) = something.result_that_never_fails()`). So I'm not sure if that's actually so important.

pythonesque commented on 5 May • edited Contributor

@carllerche @seanmonstar I agree that if `Pin`'d types can't really be used safely, it's barely worth having pinning at all, especially in a popular API. In particular, if most of the obviously safe cases need unsafe code, it's going to be a problem. But I think that is a reason to put a ton of work into making common usecases for pinning safe (`#[derive(PinFields)]` giving you wholly safe field and variant accesses), not a reason to avoid pinning. If you can fix things so that you don't need unsafe code when you use pinned types, then people don't need to pass around an `Unpin` bound and most of the associated ergonomic issues go away.

👍 1

plietar commented on 5 May • edited

@aturon It is possible to avoid the coherence issues by not using a blanket `<T: Async + Unpin> Future for T`. Instead you only really need a manual implementation for `PinBox<T>` (and maybe one more for stack pinning).

The following definitions seem to work fine to me. Except for one line in the blanket async implementation (that lives in libcore anyway), no unsafe is needed. Existing uses of Future remain as is.

```
trait Future {
    type Item;
    fn poll(&mut self) -> Poll<Self::Item>;
}

trait Async {
    type AsyncItem;
    fn poll_async(self: Pin<Self>) -> Poll<Self::AsyncItem>;
}

impl <T: Future> Async for T {
    type AsyncItem = T::Item;
    fn poll_async(mut self: Pin<Self>) -> Poll<Self::AsyncItem> {
        unsafe { Pin::get_mut(&mut self) }.poll()
    }
}

impl <T: Async> Future for PinBox<T> {
    type Item = T::AsyncItem;
    fn poll(&mut self) -> Poll<Self::Item> {
        self.as_pin().poll_async()
    }
}
```

An async function can await a future thanks to the blanket Async implementation. An Async object can be converted into a Future using `PinBox::new`.

```
fn spawn<F: Future>(f: F) { ... }
async fn do_stuff() { ... }
```

```
fn main() {
    spawn(PinBox::new(do_stuff()));
}
```

IntoFuture

Given how a lot of APIs in the futures ecosystem already use IntoFuture, I was hoping to leverage that by adding a default implementation for `<T: Async>`. Unfortunately this doesn't quite work, as specialization of multiple items is somewhat unusable today. But here's the idea anyway:

```
trait IntoFuture {
    type Item;
    type Future: Future<Item=Self::Item>;
    fn into_future(self) -> Self::Future;
}

impl <T: Async> IntoFuture for T {
    default type Item = T::AsyncItem;
    default type Future = PinBox<T>;
    default fn into_future(self) -> Self::Future {
        PinBox::new(self)
    }
}

impl <T: Future> IntoFuture for T {
    type Item = Self::Item;
    type Future = Self;
    fn into_future(self) -> Self::Future {
        self
    }
}
```

❤️ 1

tmccombs commented on 5 May

@pythonesque

> you can do `let Ok(foo) = something.result_that_never_fails()`

This doesn't work on nightly (https://play.rust-lang.org/?gist=40cf4d8b97303aa98ab1b8c402d11972&version=nightly&mode=debug), is there an RFC that would allow this? Also, that would require a temporary variable in places where you would normally use method chaining.

pythonesque commented on 5 May • edited Contributor

@tmccombs

> This doesn't work on nightly

I was puzzled by this initially since it definitely has worked for me before; I then realized that the old behavior has been pushed behind a second feature flag, `exhaustive_patterns`. It still works with that enabled.

> Also, that would require a temporary variable in places where you would normally use method chaining.

No, that's not necessary: https://play.rust-lang.org/?gist=2ed555220138f1aa8ac08219366b539a&version=nightly&mode=debug. One could make that a method on `Result` pretty easily.

This pattern works very nicely in practice, at least in my experience. I've used the never type a lot in code that had a lot of Results in it but only occasionally needed them. Again, I'm not actually arguing that we should keep a `Result` in `Future`, as I haven't used them; only pointing out that it should be possible to avoid runtime cost or panics with either solution (provided that the exhaustive patterns stuff is actually stabilized--I'm a little perturbed that it was postponed, since without it what should be API decisions based on ergonomics, like this one, become much more contentious).

carllerche commented on 6 May Member

@pythonesque `#[derive(PinFields)]` is not sound and afaik (unless there have been recent developments), there is no way forward to make it sound. i.e., you would still need `unsafe` .

pythonesque commented on 6 May • edited Contributor

@carllerche I believe I have a way to make it sound, I just hadn't shared it with anyone yet (except **@RalfJung** briefly in person) so I was surprised to hear that someone else was already working on it. I can share more details if you want, I just wanted to wait until I had a full implementation so I could present the whole thing in one piece (and make sure it sufficed for what I want it for, which is safe intrusive collections--until I looked at the RFC comments for the new async proposals I didn't realize people were expressing many of the same concerns about pinning that I had for that use case).

Edit: Here's a link to a prototype: rust-lang/rust#49150 (comment).

❤️ 1

aturon commented on 7 May **Member**

Lot's of great commentary, thanks all for driving things forward! Since there are a variety of perspectives at play, I think it'd be helpful to take stock of the design constraints we're collecting -- and in particular, which constraints are broadly agreed on, and which are up in the air. I'm hoping this can help consolidate the progress we've made, and highlight places we need to keep drilling in to.

Personal note: I'm at a managerial retreat all week, so may not be able to comment quickly.

Design goals and constraints

I'm going to break down what I consider to be the most important constraints we've built up so far. **This is a vital consensus check**, so please speak up and let the thread know the points of agreement and disagreement.

Relation to futures 0.1

- **Error types**: working with error-producing futures should be roughly as ergonomic as in 0.1
  - It *must* be easy to (1) require a future to have an error and (2) project the item/error type as associated types.
- **Manual `poll` impls**: writing a `poll` impl should be roughly as ergonomic as in 0.1.
  - It *must* be possible to write a `poll` method whose signature and body is *identical* to 0.1, modulo error-related changes
  - Such impls *must not* require unsafe code, except where they would in 0.1.

Async/await integration

- **Support async/await**: the APIs must support async/await notation (the impetus for this RFC in the first place)
- **Zero-cost abstraction**: using async/await notation should not incur a performance penalty
  - It *must* be possible to work with async/await-generated futures without extra allocations
- **Compatibility**: "traditional" and async/await-style futures should interoperate seamlessly, rather than introducing an ecosystem split
  - It *must* be possible to write abstractions that work with both
  - It *must* be possible to *introduce* such abstraction backwards-compatibly

Vetting and stabilization

- **Easy migration**: we should be able to quickly produce a futures 0.3 release with an easy migration from 0.1/0.2
  - It *must* work on stable Rust today, while providing async/await support on nightly
  - It *must* be possible for crates to optionally support 0.3 under a flag
- **Decoupling**: the decisions on various aspects of the design should ideally be decoupled, so that they do not block each other
  - E.g., ideally we can stabilize async/await support even if we're not yet ready to stabilize `Pin` (an

idea elaborated in the RFC)

Some points of uncertainty and disagreement

Probably **the** most important thing that is "up in the air" is the long-term story around `Pin`, in terms of safety and ergonomics (and what that should imply about the futures design). I think it's important that *all of us* recognize that `Pin` is in its early days, and we can't be sure how things will go. Here are three plausible outcomes:

- **Scenario 1**: We never find a way to make programming with `Pin` safe/ergonomic/easy; `&mut`-based `poll` remains dominant for most manual cases, except for core combinators or cases where the allocation truly matters.
- **Scenario 2**: We fully work out the `Pin` story, perhaps as `&pin` or through a derive, such that working with pinned types feels much like working with `&mut`, and rarely requires unsafe code.
- **Scenario 3**: We find various safe abstractions for working with `Pin`, but they remain somewhat clunky or patchy, so in practice the ecosystem contains a mix of `Pin` and `Unpin` implementations.

So put differently: `Pin` is wildly unsafe today, but the story could be very different tomorrow -- or not! None of us can say definitively right now, because it's an active area of research. (And as a procedural point, discussion of the details should happen on the tracking issue as much as possible.)

This leads me to prefer a "conservative" design, i.e. one that (1) meets the constraints above and (2) does not strongly commit us to any of the three scenarios above. In other words, I think we should strive for a design where, no matter how things with `Pin` play out, we won't have major regrets.

This all comes back to some of the discussion around defaults etc -- basically, does the design push you toward `&mut` or `Pin`. In the spirit above, I think we should try to put these on as equal footing as we can manage.

Some meta/procedural points

RFC process

It's worth remembering: this is an RFC discussion, **with lots of stages left before stabilization**. We often punt on finalizing names or defaults until we have more real-world experience. That said, the situation with futures is more complicated, since we want to be able to use 0.3 on stable. At the same time, the companion async/await RFC is essentially ready to merge.

I propose that we:

- Let the async/await RFC proceed, which implies that we'll land *something* in nightly `std`
- Keep *this* RFC open even after an initial version is in nightly, as a way to keep discussion going as we gain experience. Potentially we could use this RFC for ultimate stabilization.
- Produce a 0.3-beta as the first major stage of vetting, allowing us to prepare integration under flags throughout the ecosystem before committing to anything
- Keep this RFC and the nightly/0.3 impl in sync as we experiment.

Give and take

A final point: there are a lot of people deeply invested in this space with disparate goals and expectations. I believe strongly in the Rust community's ability to "bend the curve", to keep working together until we can find a solution that overcomes major tradeoffs to meet conflicting needs. But to get there, it's important for us all to reflect on which of our personal constraints are *most important*, and where we can afford to budge a little bit. This is why, for example, I phrased things above in terms of "roughly as ergonomic"; a curve-bending solution is probably going to come at *some* cost *somewhere*, but it's important to keep that cost in perspective.

The potential for Rust to provide zero-allocation async/await with safe borrowing, *and* have it seamlessly work with manually-written futures that are also purely safe code -- it's mind-blowing, and goes far beyond things like the in-the-works C++ proposals in this space. Let's work together and figure out how to ship this thing!

👍 14 ❤️ 20

jimmycuadra commented on 7 May

Most of the technical details of `Pin` are over my head, but as an enthusiastic Rust *user*, I do have a few thoughts I want to share:

- I agree with all of the goals and constraints listed by **@aturon** above, with the exception that I don't feel strongly about futures 0.3 pre-releases needing to work on stable Rust. Given that futures 0.2 was deemed a "checkpoint release" which users won't really end up using because it's not fully-embraced by downstream libraries and because we know 0.3 is imminent anyway, I think it'd be fine for 0.3 to focus on the end product expected when the necessary language features are stabilized. Everyone is staying on futures 0.1 until then, anyway.
- Due to the highly volatile nature of this RFC, I think it should be labeled as an eRFC to make it more clear that we really need more experience using the proposed changes in practice to determine how effective they will be. The fact that this is already the second RFC opened, and that there have been two completely different proposals within this second RFC, shows that things are moving faster than I think they should. Normally this is the kind of thing that would have had a pre-RFC thread in the internals forum so that the real RFC wasn't so controversial and needing to be revised like this. Since that's all happened already and we can't go back, just calling this an eRFC seems like a better compromise.
- I understand the value of setting a deadline (in this case the goal of Rust 2018 in September) in order to push work forward, but again the rate of change here is troubling, knowing the deadline is on the horizon. There are several unstable features that will need to be stabilized by Rust 2018 for this RFC to work, and there are other languages features (specifically specialization) that might dramatically change the design of this RFC if we had them. I think rather than trying to ship async/await now, we should be taking stock of all the first-order dependencies of a really solid proposal for async/await and make *those* the goals for Rust this year. I know the pressure is on for Rust to have good support for async network programs (believe me, I can't wait for it either), but I'd feel much better about the future of Rust if I knew we were going to get a bunch of more foundational language features like impl Trait in traits, specialization, `Pin`, and perhaps GATs, and that async/await and futures using the "multiple trait" design would be built on all the benefits those would bring.

👍 18

aturon commented on 7 May — **Member**

@jimmycuadra

Thanks much for sharing your perspective. I give some point-by-point below, but I think the core message is: fast iteration, discussion and experimentation is an important part of how we refine designs -- it's why we have the RFC + nightly process. The high stakes moment is *stabilization*, and we should try to go into *that* discussion with as much experience as we can.

> I think it'd be fine for 0.3 to focus on the end product expected when the necessary language features are stabilized. Everyone is staying on futures 0.1 until then, anyway.

The last sentence is the key one, I think. The design in this RFC, as well as the alternatives being discussed, all share the property that you can work with `&mut`-style futures without *any* new language features. That makes it plausible to get portions of the ecosystem moved over prior to final stabilizations in `core`, which I think is a prudent step for vetting the design.

> The fact that this is already the second RFC opened, and that there have been two completely different proposals within this second RFC, shows that things are moving faster than I think they should.

I definitely understand your concern! That said, I think it's good -- and not uncommon -- for *discussion* to move quickly, with revisions and new RFCs; it happens pretty frequently. What ultimately matters is what we actually ship, i.e. stabilize.

I think there's a sort of similar dynamic to the quote above here: by moving relatively quickly in evolving the initial design and getting a 0.3 beta together, **we can move more slowly when it comes to stabilization**. That is, we'll have more time to actually experience the affects of the design.

> Since that's all happened already and we can't go back, just calling this an eRFC seems like a better compromise.

I think this is essentially the process I was suggesting: you can view the companion async/await RFC as also being the "eRFC" for landing an initial futures API in nightly. Then we keep *this* RFC open as we gain experience and evolve the design, ultimately using this RFC as the stabilization point for the APIs.

> I understand the value of setting a deadline (in this case the goal of Rust 2018 in September) in order to push work forward, but again the rate of change here is troubling, knowing the deadline is on the horizon.

I totally get this; the Edition creates a sense of urgency that has both good and bad aspects, and makes it more tempting to rush toward stabilization. But remember, this discussion is about *what we land in nightly* to learn more, not yet about stabilization. The sooner we can start getting hands-on experience, the more confidence we can have in a final design.

Also, to be super clear: while it's an *aspiration* to ship async/await with Rust 2018, the Edition process is explicitly not a "feature-based release" process, and if push comes to shove we will ship it without async/await, if we collectively feel we need more time. But for the time being, I think it's useful to *try*.

> There are several unstable features that will need to be stabilized by Rust 2018 for this RFC to work, and there are other languages features (specifically specialization) that might dramatically change the design of this RFC if we had them

Just to clarify: part of the design here is to not require *any* new features to be stabilized for the `&mut` portion of the RFC to work. The only other unstable feature is the `Pin` types, and as outlined in the RFC, we can decouple stabilization there from landing async/await support in general. **IOW, it's possible to ship async/await without *any* other stabilizations**.

In terms of other features changing the possible design: I agree that it's really important to keep this in mind, and I think the thread has been doing a good job of exploring the space.

❤️ 12

withoutboats referenced this pull request in **rust-lang/rust** on 8 May

Tracking issue for async/await (RFC 2394) #50547 Open

0 of 5 tasks complete

DDOtten commented on 9 May • edited

To be honest I do not really get the need for two functions or two traits for that mater. If we just use this version of `Future`

```
pub trait Future {
    type Output;

    fn poll(self: Pin<Self>, ctx: &mut Context) -> Poll<Self::Output>;
}
```

then you only need an `Unpin` bound to make `Pin` work just like `&mut` .
Because `DerefMut` is implemented for every `Unpin` value you would never really have to think about the difference between `Pin` and `&mut` .
I admit this will make the unsafe version seem like the default but I don't think that would be wrong.
You should not require `Unpin` in your api except for cases where it is really necessary.

Another question we should consider is learnability.
We are already adding a lot to `std` and the addition of two almost identical traits can be very confusing to new users.
I must admit that it took me a while to see why we would do this.
Even if we have blanked `impl` 's both ways we would see two different trait bounds throughout the ecosystem.

aturon commented on 9 May Member

I wanted to follow up on a couple of technical issues, to make sure we're all clear on some of the mechanics. (Sorry for the terseness, I'm stealing a few minutes from my week of meetings...)

The coherence/specialization issue

> Re your snippet, would it not work if Async is in core and impl<T: Async> Async for Box {} exists in the same crate as Box?

Unfortunately, no. The problem is with downstream impls that could be written, regardless of where the primary impls live.

> While it seems that we cannot have blanket impls mapping Asyncs to Futures and Futures to Asyncs, reading the specialization RFC, it certainly should be possible.

Yes, specialization should ultimately be able to do the kind of thing you have in mind, though it would probably require full intersection impls. It definitely seems worth working through a full version of this variant, and seeing what pieces would be possible today, how it relates to stabilizing specialization, etc.

> Since the revised proposal mentions relying on specialization

Two things to clarify here:

- The revised proposal doesn't *require* specialization; it's possible to use a simple macro for the time being, and is forward-compatible with adding the default impl
- I realize this is confusing, but the `default impl` feature is *related* to specialization but is different: it's purely about refining default methods for trait impls, rather than providing multiple *full* impls. While this may not seem like a big distinction, it has one very important aspect: none of the soundness issues we've struggled with apply here. IOW, it's a much more sure bet that we can stabilize `default impl` in the near future than specialization proper. That said, I think the above point is the more important one -- that the `default impl` is not a critical thing, but rather an improvement over using a macro.

> So if I just use the default impl, I have poll defined in terms of poll_mut and vice versa... isn't there some danger here of accidentally creating a future that will just overflow the stack when called?

Only if you literally provide *no* methods. I don't think this is an issue worth worrying about.

RalfJung commented on 9 May — Contributor

> Only if you literally provide no methods. I don't think this is an issue worth worrying about.

Right, that's the case I've been thinking about. Usually Rust catches this. It's not a deal breaker, of course, but IMHO shows that the entire thing is a little fragile/hacky.

pythonesque reviewed on 10 May — View changes

`text/0000-async.md` — Show outdated

bstrie referenced this pull request in **rust-lang/rust** on 12 May

Tracking issue for Pin APIs (RFC 2349) #49150 — Open

1 of 3 tasks complete

aturon commented on 14 May • edited — Member

There haven't been a ton of replies about the goal-setting post, but my sense is that the design goals are broadly agreeable. So I wanted to dive back in to the design space on that basis, and in particular pick up on one of **@seanmonstar**'s proposals.

A revised single-trait approach

Based on feedback from the thread, here's a revised and more complete version of the single trait idea:

```
trait Future {
    type Output;

    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output> where Self: Unpin { ... }
    fn poll_pin(self: Pin<Self>, cx: &mut Context) -> Poll<Self::Output>;
}

// easy way to use `poll` to implement `poll_pin` until `default impl` works:

macro_rules! poll_pin ...

// add this once partial impls are available
default impl<T: Unpin> Future for T {
    // this partial impl of the Future trait makes it possible to impl Future by providing
    // *only* a `poll` method
    fn poll_pin(self: Pin<Self>, cx: &mut task::Context) -> Poll<Self::Output> {
        self.poll(cx)
    }
}

type TryPoll<T, E> = Poll<Result<T, E>>;

trait TryFuture {
```

```
    type Item;
    type Error;

    // this is just used to provide an `into_future` adapter; eventually we will be able
    // to make `Future` a super-trait
    fn try_poll_pin(self: Pin<Self>, cx: &mut Context) -> TryPoll<Self::Item, Self::Error>;
}

impl<F, T, E> TryFuture for F where F: Future<Output = Result<T, E>> { ... }
```

Here are some general notes:

- The `poll_pin` macro is meant to be used directly in the impl body, so that you impl `poll` and then also write `poll_pin!()` to get a version of `poll_pin` that uses `poll`
  - This is only needed until we can have the `default impl`, which is something we could do relatively soon (it's not actually specialization)
- The naming scheme here pushes you more toward `&mut self`-based methods, but is ergonomic for both.
- The `TryFuture` trait is essentially an alias for `Future<Output = Result<T, E>>`, except that it gives you associated types. You never implement it directly, but only use it in bounds when you want to restrict to `Result` and project types. (This is the solution from the earlier RFC).
  - There's a bit of boilerplate when you have `T: TryFuture` and you need to plug it in where something is expecting `Future` -- you have to use `.into_future()`. As with the macro situation above, this is only a temporary papercut, which can be eliminated once some further trait system improvements land.

Evaluation

My hope is that we're narrowing in on a design that balances well between some disparate use cases and priorities. Going back to the design goals I laid out before:

- **Error types**: working with error-producing futures should be roughly as ergonomic as in 0.1
  - 👍 For the eventual setup where `TryFuture: Future`, since then the only regression is the extra three characters for `Try`.
  - 👎 For the initial setup, there's a papercut when using a `TryFuture` when a `Future` is expected; it requires an explicit conversion call.
- **Manual poll impls**: writing a poll impl should be roughly as ergonomic as in 0.1.
  - 👍, especially after the `default impl` is possible. Using a macro before that point is pretty minimal overhead.
  - See below for thoughts on explicit `Unpin` bounds
- **Support async/await**: the APIs must support async/await notation
  - 👍
- **Zero-cost abstraction**: using async/await notation should not incur a performance penalty
  - 👍
- **Compatibility**: "traditional" and async/await-style futures should interoperate seamlessly, rather than introducing an ecosystem split
  - 👍, because of there's a single trait, compatibility comes down to whether you impose the additional `Unpin` bound or not, and you can impose it at first then relax the restriction later, back-compat
- **Easy migration**: we should be able to quickly produce a futures 0.3 release with an easy migration from 0.1/0.2
  - 👍

Overall, I believe this design retains the core ergonomics of futures 0.1/0.2, with some papercuts initially but a clear path for eliminating them.

This design is also very close to hitting all the points **@seanmonstar** was asking for, except for one thing: you still need to bound by `Future + Unpin` if you rely on moveability. Let's examine that further.

The role of `Unpin`

~~First: I would really like to find a better name than `Unpin`. I think of it as more like `Move`, meaning that the movement of the data structure is never restricted, event when pinned.~~ **EDIT**: after re-reading the threads about this, I agree that `Unpin` is probably the best choice. But in any case, the conceptual point is the same: it means the value is moveable, much like `Send` means it's sendable.

I'm pushing for a mental model that separates out the concerns of *being a future* on the one hand, and *being moveable* on the other -- hence the `Future + Unpin` bound. Code that can work with unmoveable futures is strictly more general, and hence has fewer bounds (just `Future`).

My gut feeling is that this extra `+ Unpin` feels bad right now because `Unpin` feels new and niche. But I think this will change over time, as we develop ways of working with pinned data in purely safe code, and as people begin to use `async` notation. As a community, once we've grown accustomed to this distinction, I think it will feel just like having a `+ Send` bound (which used to be baked into futures, but is now treated as an orthogonal concern).

Relatedly, using `Unpin` as a "knob" like `Send` also allows us to avoid multiplying the set of traits -- i.e., `Stream + Unpin` or `Sink + Unpin` etc all make sense, and so we don't have to double up on each of the futures-related traits.

wdyt?

Is this getting in the ballpark for folks? In particular, do you buy the argument that this design minimizes the ergonomic regression (especially over time), while accomplishing a host of other goals?

👍 13

pythonesque commented on 14 May • edited ▾ Contributor

@aturon

I think I mostly like the proposal. Just a couple of comments:

> The naming scheme here pushes you more toward &mut self-based methods, but is ergonomic for both.

I guess I am a bit worried that people will unnecessarily restrict themselves to `poll` implementations when it's not needed. I am mostly okay with this though because you also (I think?) have to write the `where Self: Unpin` bound wherever it's not implied by the type it's being implemented for; I expect that once `poll_pin` can be done with minimal ceremony in safe code people will gravitate back towards it for increased generality (just as people mostly got rid of unnecessary `'static` bounds on concurrency primitives once `Send` no longer required it).

I will ask: once you have the `Self: Unpin` bound, given that `poll_pin` will have a `DerefMut` and `Deref` implementation, why do you actually need `poll` to be a trait method? Can you just make it an inherent impl for `T: Future + Unpin` (so people don't have to bother constructing a `PinMut` explicitly in order to use it)? That way everyone just implements `poll_pin`, but you can *call* `poll` unless you have a `!Unpin` future. Maybe this was very controversial earlier and I missed it, but it seems like the only advantage for having a `poll` method on the trait that takes `&mut self` at this point is so people don't have to see the word `Pin` two times.

> First: I would really like to find a better name than Unpin. I think of it as more like Move, meaning that the movement of the data structure is never restricted, event when pinned.

Unfortunately I think this is already causing confusion in the other RFC... `Move` suggests a type that can move, and `!Move` suggests one that can't. But that is not what `Unpin` means. `Unpin` means that, *once pinned*, a type can still move. The distinction is really important because it means that even immovable generators can still be used like normal Rust types until the moment they're pinned in place. It is true that you could instead interpret `Move` as a "can always move" and `!Move` as "can't always move", but I don't think that's the natural interpretation, and even that interpretation doesn't suggest that the type has anything to do with pinning. I'm worried that that would make it very unclear that implementing (or not) `Pin` accessors should be related to a `Move` blanket impl.

👍 1 ❤️ 1

MajorBreakfast commented on 15 May Contributor

@aturon Perfect!

Some thoughts about `TryFuture` (this is AFAIK the first proposal that uses this name):

- It's short
- It's descriptive, but arguably not as descriptive as `ResultFuture` because `poll()` literally returns a `Result`
- OTOH there's considerable percendence in std for usage of "try" in such scenarios

- `TryStream` and `TrySink` sound good.
  - It's likely that we'll get these in the future, right?
  - BTW I just realized that `FutureResult` would have lead to `StreamResult` and `SinkResult` which would have been a naming disaster IMO ^^'
- Conclusion: I think I like `TryFuture` as trait name!

👍 2 ❤️ 2

jimmycuadra commented on 15 May

I take credit for `TryFuture` ! Happy to see that being used, of course. :}

@aturon's latest proposal looks good to me, though 'd like some clarification on the timeline for removing the papercuts (the `poll_pin` macro and `TryFuture::try_poll_pin`). Is the goal to have these removed prior to stabilization? Should that (and the features required to remove them) be added to the RFC?

👍 1

cramertj commented on 16 May • edited ▾ Member

After more discussion with **@aturon**, **@carllerche**, and **@seanmonstar**, I'd like to offer an amendment to this proposal that I think offers the advantages of `fn poll(self: PinMut, ...)` without the negative impact on manual `Future` implementations.

The basic idea is that, rather than having `Future::poll` take `PinMut<Self>` , we make `Future::poll` take `&mut self` (as today). Rather than implementing `Future` for `async fn` s directly, we implement `Future for PinMut<'a, async fn type>` . We then introduce a trait called `PinFuture` which refers to types which implement `Future` for `PinMut<Self>` . Here's how it looks:

```
pub trait Future: Unpin {
    type Output;
    fn poll(&mut self, cx: &mut task::Context) -> Poll<Self::Output>;
}

impl<'a, T: Future> Future for PinMut<'a, T> {
    type Output = T::Output;
    fn poll(&mut self) -> Poll<Self::Output> {
        (**self).poll()
    }
}

pub trait PinFuture where for<'a> PinMut<'a, Self>: Future {
    type Output;
}

impl<F: Future> PinFuture for F {
    type Output = <Self as Future>::Output;
}

// Note: we'd really *like* for that last impl to be
// impl<F> PinFuture for F where for<'a> PinMut<'a, F>: Future>
// Unfortunately, there's no way to go from that to an associated type output.
// It'd be nice if there was a way to require `for<'a, 'b> PinMut<'a, F>::Output == PinMut<
// and project out on that type, but that isn't possible currently.
// If we put `Future` in core and `PinFuture` in futures-rs, we can fix this backwards-comp
// (for std) if/when it becomes possible.
```

With this strategy, none of the combinators need any unsafe code. Manual `Future` impls don't change from what we have today.

To use `async fn` with APIs that accept `T: Future` , users could put the result of the `async fn` in a `PinBox` or in a `PinMut` (via a `pin!` macro like this one). APIs which choose to work on `T: PinFuture` and handle pinning internally could do so and would be compatible with both `Future` and `PinFuture` types.

One interesting bit you may have noticed is that `Future` now has an `Unpin` bound. This is necessary for the `impl<'a, T: Future> Future for PinMut<'a, T>` , which is necessary in order for `T: Future` to implement `PinFuture` . Under the current `unsafe trait Unpin` , this would mean that generic structs that implemented `Future` would need to unsafely implement `Unpin` . I propose instead that we make `Unpin` a *safe* trait, and that manual `Future` implementations should opt-into `Unpin` where necessary (such as in generic structs and types that contain other `!Unpin` types).

It's also worth noting that users who implement `Future` for `PinMut` of their own types in order to have custom self-referential futures (or ones which operate directly on `PinFuture` types rather than `Future` types) will also need to implement `PinFuture` for their types manually. In order to prevent this, we'd really like for that last impl in the code block above to be `impl<F> PinFuture for F where for<'a> PinMut<'a, F>: Future>`. Unfortunately, there's no way to go from that to an associated type output. It'd be nice if there was a way to require `for<'a, 'b> PinMut<'a, F>::Output == PinMut<'b, F>::Output` and project out on that type, but that isn't possible currently. If we put `Future` in core and `PinFuture` in futures-rs, we can fix this backwards-compatibly (for std) if/when it becomes possible.

This is a lot of information, but TL;DR:

- `Future` combinators become safe to implement manually without requiring any `T: Unpin` bounds or worrying about `Pin` / `PinMut` / `Unpin` at all.
- `async fn` return types implement `PinFuture` and can safely be used as `Future` types by either allocating in `PinBox` or stack-pinning with a `pin!` macro.
- We can make futures 0.3 usable on stable before `Pin` is stable by feature-gating `PinFuture`.

P.S. there is a bug in the trait system right now that stops these bounds from working correctly, but this is clearly a bug and should be resolved ASAP. See issue rust-lang/rust#50792

👍 12 😄 1 🎉 3 ❤️ 6

aturon commented on 16 May **Member**

There are a couple of key points about this new idea that are worth elaborating on:

- The big insight here is that we can use `PinMut`, and not just `PinBox`, as a way of interfacing between the two worlds. **That means that, in the context of an async block, you can piggyback on the eventual allocation for the task when pinning**.
  - In particular, async blocks make it possible to have internal borrowing within a future. We can use this together with `PinMut` to essentially pin a piece of data to the "ambient" allocation that the whole task will provide.
  - For something like `select` that involves moving futures around, the point is that the `PinMut` *itself* will be moved around, while the actual data for the future will stay put within the task's ambient data.
- Syntax/macros like `await`, `for await` etc can build in pinning so that you don't have to explicitly construct a `PinMut`.
- Part of the impetus for this idea was realizing that `PinMut` would be needed far beyond the `Future` trait -- if we want the full benefits, it would need to appear even in traits like `AsyncRead`. But with this new approach we have a full separation of concerns: APIs are written against `&mut self` and `Future`, etc., and when you want to work with pinned structures, you can layer appropriate mechanisms *on top*.
- This also makes it trivial to provide an initial 0.3 version that works on stable Rust today.

👍 3 🎉 4 ❤️ 8

aturon commented on 16 May **Member**

Good news: I think we can simplify this even more:

- Drop the `PinFuture` trait altogether
- Drop the `Unpin` super-trait bound

Given the ability to pin internally within a task, we only really need what `PinFuture` was providing when we actually get to the point of task creation, so we can just inline it:

```
pub fn spawn<F: 'static + Send>(&mut self, f: F)
   where F: for<'a> PinMut<'a, F>: Future<Output = ()>
```

Thus for manual futures, an `Unpin` requirement is only needed at the top-most level: for futures actually being spawned.

👍 7 ❤️ 1

tmandry commented on 16 May • edited

> We can use this together with PinMut to essentially pin a piece of data to the "ambient" allocation that the whole task will provide.

How exactly would this work with combinators? ~~I assume they must work with `PinFuture` directly to support this.~~

Otherwise, we would be hiding the fact that data is pinned from combinator types. For a type like `Select<A, B>`, we'll swap `A` (our inner future state) for `PinMut<'a, A>`. But then the state of our future can't live within the `Select`, it must live elsewhere, i.e. in a separate allocation.

~~So if we wish to embed our state in a combinator, it must work with `PinFuture`. Which, of course, is not a bad trade-off for what it buys us.~~

aturon commented on 16 May **Member**

@tmandry So the key point is that `PinMut<'a, A>` will implement `Future` itself for async values (i.e. the result of calling an async fn or making an async block). That means that once you've pinned such a thing, you can use it with combinators.

The new idea, though, is that **this doesn't require additional allocation**, assuming that this combinator is being consumed by an enclosing async block. You can pin it on "the stack" from the perspective of `async` blocks, which actually means that it's pinned to the task allocation that the async block is used in.

👍 1

aturon commented on 16 May **Member**

Getting back to dropping `PinFuture`, I just recalled that the original impetus for it was for the analog to `impl Future` for async blocks. So we may still need it at some form, but the key point is that it doesn't need to play an important role in the ecosystem -- it's pure shorthand -- and in particular we don't need to be blocked by the current compiler bugs around it.

aturon commented on 16 May **Member**

Here's an example, to help clarify what this might look like:

```
let a = async { /* some code */ };
let b = async { /* some code */ };
pin!(a, b);
select(a, b) // this produces a Future with borrows
```

The resulting future will have internal borrows, but as long as *it* in turn is used within an `async` block, that's fine:

```
// this produces a 'static PinFuture
async {
    let a = async { /* some code */ };
    let b = async { /* some code */ };
    pin!(a, b);
    await!(select(a, b))
}
```

❤️ 7

tmandry commented on 16 May • edited

@aturon All clear now.

So, what the outer async block is effectively doing is creating a struct with our pinned async state and our combinator state, side-by-side. The combinator operates happily on PinMut references to that async state, which it can move at will.

This is exciting!

MajorBreakfast commented on 16 May • edited Contributor

This new proposal is exciting!

@cramertj

> I propose instead that we make Unpin a safe trait, and that manual Future implementations should opt-into Unpin where necessary (such as in generic structs and types that contain other !Unpin types).

Can you explain what you mean with "a safe trait" with a code example? I don't quite understand this passage.

This wasn't mentioned, but I think it is important: I think `PinFuture` types (types where `PinMut<'a, Self>: Future`) should **not** implement `IntoFuture` for "convenience". Such an implementation would need to box it. Combinators commonly accept `IntoFuture` which would make the boxing effectively implicit. The choice between pinning and boxing should always be explicit. I'm just saying in case the idea comes up ;)

👍 1

pythonesque commented on 16 May • edited Contributor

@cramertj

Am I correct in thinking that your proposal is that people mostly take pinned futures by (pinned) reference, rather than by value? For some reason I had been under the impression that people didn't want to do that--it certainly adds an extra indirection, but I thought there was some other problem with that. Maybe it was because the stack pinning API wasn't available yet?

> I propose instead that we make Unpin a safe trait, and that manual Future implementations should opt-into Unpin where necessary (such as in generic structs and types that contain other !Unpin types).

This would both be a major change, and cannot possibly be sound with the current definition of `Unpin`. It is very easy to cause unsafety by opting into `Unpin`--especially if pin projections become safe (which is required for pinning to be usable in a lot of scenarios). I realize that for the particular case of futures you want to get around this by storing `PinMut`s, which are themselves `Unpin`, but that is not sufficient for cases where you need to store pinned types by value, which is pretty much always the case for intrusive collections. So, I am happy to hear **@aturon** say it's not needed except at the spawn point (when presumably the concrete type will usually be mostly available).

👍 2 ❤️ 1

Thomasdezeeuw referenced this pull request in **rust-lang-nursery/futures-rs** on 16 May

0.3 #1010 Merged

RalfJung commented on 16 May Contributor

> This would both be a major change, and cannot possibly be sound with the current definition of Unpin. It is very easy to cause unsafety by opting into Unpin--especially if pin projections become safe (which is required for pinning to be usable in a lot of scenarios).

From what I can tell, if `impl Unpin` opts out of safe projections, it actually cannot cause issues *without other unsafe code*.

Actually, that's pretty much a requirement because people can write `impl Drop` in safe code, which performs an unpin, and it should not cause issues without other unsafe code.

pythonesque commented on 16 May • edited ▾

Contributor

> From what I can tell, if impl Unpin opts out of safe projections, it actually cannot cause issues without other unsafe code.

`impl Unpin` opting out of safe projections is definitely not something I remember coming up in the other RFC and it would make the current documentation for `PinMut::map` even more misleading. I guess I would be okay with it though, as long as it's actually checked by the compiler. It would certainly reduce the amount of spurious `unsafe` in people's code which would probably make them more at ease with using `Unpin`. I am a little worried that you could safely implement `Unpin` and then write unsafe code that was perfectly fine if it weren't for the safe implementation--but then, I suppose that safe code needing to preserve guarantees for unsafe code to be correct happens a lot in Rust and is usually considered okay as long as it's restricted to code defining the type.

RalfJung commented on 16 May

Contributor

> impl Unpin opting out of safe projections is definitely not something I remember coming up in the other RFC

Absolutely, that's why this is a new proposal.

> I am a little worried that you could safely implement Unpin and then write unsafe code that was perfectly fine if it weren't for the safe implementation

Yeah, me too. However, the same applies for safely implementing `Drop`.

bugaevc commented on 16 May

> Actually, that's pretty much a requirement because people can write impl Drop in safe code, which performs an unpin, and it should not cause issues without other unsafe code.

Oh wow, this is indeed a soundness issue with the whole pinning API, isn't it?

pythonesque commented on 16 May • edited ▾

Contributor

@RalfJung

> Absolutely, that's why this is a new proposal.

Actually, this seems very problematic to me. You can't necessarily know whether a type implements `Unpin` or not since it's conditional in many cases (and you can't condition on `!Unpin`). How are you proposing this would work?

> Oh wow, this is indeed a soundness issue with the whole pinning API, isn't it?

Yes, but there's been a lot of discussion in the other thread about it--it seems resolvable.

RalfJung commented on 16 May

Contributor

@bugaevc FTR, that "other thread" is rust-lang/rust#49150 (comment)

👍 2

 `Revamp RFC to one-trait approach` `0ded7b7`

 **aturon** changed the title from **RFC: add Async trait and task system to libcore** to **RFC: add futuresand task system to libcore** 21 days ago

 **aturon** changed the title from **RFC: add futuresand task system to libcore** to **RFC: add futures and task system to libcore** 21 days ago

aturon commented 21 days ago

Member

Apologies for the recent silence! I've just pushed a significant update to the RFC, after a bunch more experimentation on multiple 0.3 branches.

TL;DR: the revision returns to a single-trait approach, but with a streamlined way of avoiding pinning that does not require any macros or unsafe code. It also includes a greatly expanded rationale/alternatives/drawbacks section, which in particular explains the three core approaches we've explored to incorporating `PinMut` (which, NB, is largely orthogonal from the approach to errors).

As the updated text explains, the primary motivation for moving back to the one-trait approach for now is that it's conservative: it's forward-compatible with a two-trait approach later on, if we deem that necessary (and I believe all the stakeholders are open to that). We could also consider landing the second trait experimentally, to gain experience with both for comparison's sake.

The HRTB-based approach, which I prototyped sort of fell apart due to compiler limitations and lack of a clear win over a two-trait-with-blanket-impl approach.

The blanket impl approach, for its part, looks more plausible now: we've determined that the issues with `Box` / `&mut` are *specific to those two types*, and are tied up with the `#[fundamental]` attribute in a way that can very likely be addressed or worked around. I think it's quite plausible that we'll end up there in the end.

For the time being, I think we've hit diminishing returns on the RFC discussion around `PinMut`, and we're best off pushing on experiments to gain more data. To that end, the 0.3 branch has been partially ported to match the revised proposal; there's more work to be done, but nothing fundamental. In the latest round of work, I also experimented with a new, macro-driven approach to working with `PinMut` that essentially simulates what a safe derive would provide; **@pythonesque** is working on an implementation thereof. You can see an example here; the idea is that none of the unsafe declarations above it will be needed once the derive is available.

I plan to continue pushing through this branch to completion. Meanwhile, **@cramertj** and **@withoutboats** have been making steady progress on implementing async/await. These two lines of work should intercept each other Soon™, at which point we'll have a full story to play with.

Meanwhile: because we've focused so much on the `PinMut` story, we haven't talked as much on this thread about the tradeoffs around errors. In the revised RFC, I'm more explicit about introducing a `TryFuture` trait, and talk about the tradeoffs as I see them. Please let me know if I've missed anything!

👍 11 ❤️ 8

ebfull commented 19 days ago

> In the long run, though, once we can take `dyn` by value, we would deprecate `spawn_obj` and add a default `spawn` method:

This should be accompanied by a link to the tracking issue for unsized rvalues which is what I think this is referring to. Also, I think the `Future` API depends on custom self types (which hasn't been merged yet), which should be mentioned. (I'm just trying to follow along and having more links really helps.)

👍 5 ❤️ 1

cramertj referenced this pull request in **rust-lang/rust** 19 days ago

Add Future and task system to the standard library #51263 Merged

bors added a commit to rust-lang/rust that referenced this pull request 14 days ago

`Auto merge of #51263 - cramertj:futures-in-core, r=aturon` ··· ✗ `37bd8b0`

bors added a commit to rust-lang/rust that referenced this pull request 13 days ago

`Auto merge of #51263 - cramertj:futures-in-core, r=aturon` ··· ✓ `19d0b53`

nrc reviewed 13 days ago View changes

`text/0000-async.md`

```
151 +```
```

```
152 +
153 +In general async values are not coupled to any particular executor, so
     we use a trait
154 +object to handle waking:
```

nrc 13 days ago **Member**

When you say 'trait object' I was expecting to see a `Box<Wake>` or something. Why do we need a `Waker` struct as well as the `Wake` trait? And how do the two relate - do we expect `Waker` to implement `Wake` or a `Waker` to have a `Box<Wake>` field or something else?

nrc 13 days ago **Member**

I think this gets clear later in the RFC but it would be nice to clarify what is meant here

Reply…

nrc reviewed 13 days ago View changes

text/0000-async.md

```
217 +}
218 +
219 +/// Provides the reason that an executor was unable to spawn.
220 +pub struct SpawnErrorKind { .. }
```

nrc 13 days ago **Member**

Could we use an enum rather than a struct? Since we're in core we can use an `unstable` variant to prevent exhaustive matching.

eddyb 12 days ago **Member**

We have `#[non_exhaustive]` nowadays, which is even better than an unstable variant.

Reply…

aidanhs reviewed 12 days ago View changes

text/0000-async.md

```
362 +    /// of data on a socket, then the task is recorded so that when data
     arrives,
363 +    /// it is woken up (via `cx.waker()`). Once a task has been woken
     up,
364 +    /// it should attempt to `poll` the future again, which may or may
     not
365 +    /// produce a final value at that time.
```

aidanhs 12 days ago **Member**

(reiterating comment that was lost in RFC update): As I said in the last RFC, I think it's important to make it explicit that it's not an error to poll a future even before a registered 'interest' has caused a task wakeup (otherwise you can't be sure you can implement join/select).

Reply…

nrc reviewed 12 days ago View changes

text/0000-async.md

```
440 +    ///
441 +    /// This method is a stopgap for a compiler limitation that prevents
     us from
442 +    /// directly inheriting from the `Future` trait; in the future it
     won't be
443 +    /// needed.
```

nrc 12 days ago **Member**

It would be useful to note exactly what feature is missing which prevents this

eddyb 12 days ago **Member**

I'm assuming (up)casting trait objects to one of their supertraits?

Reply…

nrc reviewed 12 days ago **View changes**

text/0000-async.md

```
460 +by `TryFuture`), and to obtain the success/error types (by using the
    `Item` and
461 +`Error` associated types).
462 +
463 +Similarly, `PollResult<T, E>` is a type alias for `Poll<Result<T, E>>`.
```

nrc 12 days ago **Member**

This is probably pedantic, but 'similarly' seems wrong here - it is a completely different mechanism

Reply…

MajorBreakfast commented 6 days ago • edited — Contributor

I think there should be a `spawn_local_obj()` method on the `Executor` trait that spawns a `LocalTaskObj` (same as `TaskObj`, but not `Send`).

The current `Executor` definition has no no way to spawn a `!Send` future. However I think we want this ability. Example use case: Futures that contain `Rc`s. A future that contains an `Rc` is not `Send`. `spawn_local_obj()` would enable us to run these futures concurrently on the current thread (i.e. concurrency but no parallelism).

sdroege commented 6 days ago

> I think there should be a spawn_local_obj() method on the Executor trait that spawns a LocalTaskObj (same as TaskObj, but not Send).
>
> The current Executor definition has no no way to spawn a !Send future. However I think we want this ability. Example use case: Futures that contain Rcs. A future that contains an Rc cannot be Send. spawn_local_obj() would enable us to run these futures concurrently on the current thread (i.e. concurrency but no parallelism).

The problem with that is that it prevents implementing `Executors` that could schedule execution on arbitrary threads of a thread pool... as somehow you always need to be able to handle the non-`Send` futures on the same thread.

I a separate trait for this could make IMHO, or just having API on the specific `Executor` instances. Like `CurrentThread` in tokio.

cramertj commented 6 days ago — Member

@MajorBreakfast See my comment here about why `spawn_local` can't and doesn't need to be built-in. Many (most) multithreaded executors don't have the ability to create thread-locked tasks.

MajorBreakfast commented 6 days ago — Contributor

@sdroege @cramertj Thanks for your replies!

I appreciate that this has been thought about in such great detail.