- Feature Name: dyn-trait-syntax
- Start Date: 2017-08-17
- RFC PR: rust-lang/rfcs#2113
- Rust Issue: rust-lang/rust#44662

Summary

Introduce a new `dyn Trait` syntax for trait objects using a contextual `dyn` keyword, and deprecate "bare trait" syntax for trait objects. In a future epoch, `dyn` will become a proper keyword and a lint against bare trait syntax will become deny-by-default.

Motivation

In a nutshell

The current syntax is often ambiguous and confusing, even to veterans, and favors a feature that is not more frequently used than its alternatives, is sometimes slower, and often cannot be used at all when its alternatives can. By itself, that's not enough to make a breaking change to syntax that's already been stabilized. Now that we have epochs, it won't have to be a breaking change, but it will still cause significant churn. However, impl Trait is going to require a significant shift in idioms and teaching materials all on its own, and "dyn Trait vs impl Trait" is much nicer for teaching and ergonomics than "bare trait vs impl Trait", so this author believes it is worthwhile to change trait object syntax too.

Motivation is the key issue for this RFC, so let's expand on some of those claims:

The current syntax is often ambiguous and confusing

Because it makes traits and trait objects appear indistinguishable. Some specific examples of this:

- This author has seen multiple people write `impl SomeTrait for AnotherTrait` when they wanted `impl<T> SomeTrait for T where T: AnotherTrait`.
- `impl MyTrait {}` is valid syntax, which can easily be mistaken for adding default impls of methods or adding extension methods or some other useful operation on the trait itself. In reality, it adds inherent methods to the trait object.
- Function types and function traits only differ in the capitalization of one letter. This leads to function pointers `&fn ...` and function trait objects `&Fn ...` differing only in one letter, making it very easy to mistake one for the other.

Making one of these mistakes typically leads to an error about the trait not implementing Sized, which is at best misleading and unhelpful. It may be possible to produce better error messages today, but the compiler can only do so much when most of this "obviously wrong" syntax is technically legal.

favors a feature that is not more frequently used than its alternatives

When you want to store multiple types within a single value or a single container of values, an enum is often a better choice than a trait object.

When you want to return a type implementing a trait without writing out the type's name--either because it can't be written, or it's too unergonomic to write--you should typically use impl Trait (once it stabilizes).

When you want a function to accept any type of value that implements a certain trait, you should typically use generics.

There are many cases where trait objects are the best solution, but they're not more common than all of the above. Usually trait objects become the best solution when you want to do two or more of the things listed above, e.g. you have an API that accepts values of types defined by external code, and it has to deal with more than one of those types at a time.

favors a feature that ... is sometimes slower

Trait objects typically require allocating memory and doing virtual dispatch at runtime. They also prevent the compiler from knowing the concrete type of a value, which may inhibit other optimizations. Sometimes these costs are unnoticeable in practice, or even optimized away entirely, but sometimes they have a significant impact on performance.

enums and impl Trait simply don't have these costs. It's strange that the more concise syntax gives you a feature that is often slower and rarely faster than its alternatives.

favors a feature that ... often cannot be used at all when its alternatives can

Many traits simply can't have trait objects at all, because they don't meet the object safety rules.

In contrast, impl Trait and generics work with any trait. It's strange that the more concise syntax gives you the feature that's least likely to compile.

impl Trait is going to require a significant shift in idioms and teaching materials all on its own

Today, when you want to return a type implementing a trait without writing out the type's name, you typically `Box` a trait object and accept the potential runtime cost. This includes most functions that return closures, iterators, futures, or combinations thereof. Most of those functions should switch to impl Trait once that syntax stabilizes and becomes the preferred idiomatic way of doing this, including many public API methods.

The way we teach the trait system will also have to change to describe impl Trait alongside all the existing ways of using traits via generics and trait objects, and explain when impl Trait is preferable to those and other options like enums. Moreover, the way we teach closures, iterators and futures will likely need to mention why impl Trait is useful for those types and use impl Trait in many examples, as well as when impl Trait isn't enough and you do need dyn Trait after all.

Ideally, introducing dyn Trait won't create much additional churn on top of impl Trait, since these idiom shifts and documentation rewrites can account for both of those changes together.

"dyn Trait vs impl Trait" is much nicer for teaching and ergonomics than "bare trait vs impl Trait"

There's a natural parallel between the impl/dyn keywords and static/dynamic dispatch that we'll likely mention in The Book. Having a keyword for both kinds of dispatch correctly implies that both are important and choosing between the two is often non-trivial, while today's syntax may give the incorrect impression that trait objects are the default and impl Trait is a more niche feature.

After impl Trait stabilizes, it will become more common to accidentally write a trait object without realizing it by forgetting the impl keyword. This often leads to unhelpful and cryptic errors about your trait not implementing Sized. With a switch to dyn Trait, these errors could become as simple and self-evident as "expected a type, found a trait, did you mean to write impl Trait?".

Explanation

The functionality of `dyn Trait` is identical to today's trait object syntax.

`Box<Trait>` becomes `Box<dyn Trait>` .

`&Trait` and `&mut Trait` become `&dyn Trait` and `&mut dyn Trait` .

Migration

On the current epoch:

- The `dyn` keyword will be added, and will be a contextual keyword
- A lint against bare trait syntax will be added

In the next epoch:

- `dyn` becomes a real keyword, uses of it as an identifier become hard errors
- The bare trait syntax lint is raised to deny-by-default

This follows the policy laid out in the epochs RFC, where a hard error is "only available when the deprecation is expected to hit a relatively small percentage of code." Adding the `dyn` keyword is unlikely to affect much code, but removing bare trait syntax will clearly affect a lot of code, so only the latter change is implemented as a deny-by-default lint.

Drawbacks

- Yet another (temporarily contextual) keyword.
- Code that uses trait objects becomes slightly more verbose.
- `&dyn Trait` might give the impression that `&dyn` is a third type of reference alongside `&` and `&mut`.
- In general, favoring generics over trait objects makes Rust code take longer to compile, and this change may encourage more of that.

Rationale and Alternatives

We could use a different keyword such as `obj` or `virtual`. There wasn't very much discussion of these options on the original RFC thread, since the motivation was a far bigger concern than the proposed syntax, so it wouldn't be fair to say there's a consensus for or against any particular keyword.

This author believes that `dyn` is a better choice because the notion of "dynamic" typing is familiar to a wide variety of programmers and unlikely to mislead them. `obj` is likely to incorrectly imply an "object" in the OOP sense, which is very different from a trait object. `virtual` is a term that may be unfamiliar to programmers whose preferred languages don't have a `virtual` keyword or don't even expose the notion of virtual/dynamic dispatch to the programmer, and the languages that do have a `virtual` keyword usually use it to mean "this method can be overridden", not "this value uses dynamic dispatch".

We could also use a more radical syntax for trait objects. `Object<Trait>` was suggested on the original RFC thread but didn't gain much traction, presumably because it adds more "noise" than a keyword and is arguably misleading.

Finally, we could repurpose bare trait syntax for something other than trait objects. It's been frequently suggested in the past that impl Trait would be a far better candidate for bare trait syntax than trait objects. Even this RFC's motivation section indirectly argues for this, e.g. impl Trait does work with all traits and does not carry a runtime cost, unlike trait objects. However, this RFC does not propose repurposing bare trait syntax yet, only deprecating and removing it. This author believes dyn Trait is worth adding even if we never repurpose bare trait, and repurposing it has some significant downsides that dyn Trait does not (such as creating the possibility of code that compiles in two different epochs with radically different semantics). This author believes the repurposing debate should come later, probably after impl Trait and dyn Trait have been stabilized.

Unresolved questions

- How common are trait objects in real code? There were some requests for hard data on this in the original RFC thread, but none was ever provided.
- Does introducing this contextual keyword create any parsing ambiguities?
- Should we try to write out how The Book would teach impl Trait vs dyn Trait in the future?