

How do I specify lifetime parameters in an associated type?

Asked 7 years ago Modified 2 years, 1 month ago Viewed 15k times

▲ I have this trait and simple structure:

52



```
use std::path::{Path, PathBuf};

trait Foo {
    type Item: AsRef<Path>;
    type Iter: Iterator<Item = Self::Item>;

    fn get(&self) -> Self::Iter;
}

struct Bar {
    v: Vec<PathBuf>,
}
```

I would like to implement the `Foo` trait for `Bar` :

```
impl Foo for Bar {
    type Item = PathBuf;
    type Iter = std::slice::Iter<PathBuf>;

    fn get(&self) -> Self::Iter {
        self.v.iter()
    }
}
```

However I'm getting this error:

```
error[E0106]: missing lifetime specifier
  --> src/main.rs:16:17
   |
16 |     type Iter = std::slice::Iter<PathBuf>;
   |                               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected lifetime parameter
```

I found no way to specify lifetimes inside that associated type. In particular I want to express that the iterator cannot outlive the `self` lifetime.


How do I have to modify the `Foo` trait, or the `Bar` trait implementation, to make this work?

[Rust playground](#)


rust lifetime

Share Edit Follow

edited Jan 26, 2018 at 16:00

 [Shepmaster](#)
353k 77 995 1251

asked Nov 16, 2015 at 11:47

 [mbt](#)
1,848 1 17 33

3 Answers

Sorted by: Highest score (default) ◆

There are a two solutions to your problem. Let's start with the simplest one:

There are a two solutions to your problem. Let's start with the simplest one.

Add a lifetime to your trait

60

```
trait Foo<'a> {  
    type Item: AsRef<Path>;  
    type Iter: Iterator<Item = Self::Item>;  
  
    fn get(&'a self) -> Self::Iter;  
}
```

This requires you to annotate the lifetime everywhere you use the trait. When you implement the trait, you need to do a generic implementation:

```
impl<'a> Foo<'a> for Bar {  
    type Item = &'a PathBuf;  
    type Iter = std::slice::Iter<'a, PathBuf>;  
  
    fn get(&'a self) -> Self::Iter {  
        self.v.iter()  
    }  
}
```

When you require the trait for a generic argument, you also need to make sure that any references to your trait object have the same lifetime:

```
fn fooget<'a, T: Foo<'a>>(foo: &'a T) {}
```

Implement the trait for a reference to your type

Instead of implementing the trait for your type, implement it for a reference to your type. The trait never needs to know anything about lifetimes this way.

The trait function then must take its argument by value. In your case you will implement the trait for a reference:

```
trait Foo {  
    type Item: AsRef<Path>;  
    type Iter: Iterator<Item = Self::Item>;  
  
    fn get(self) -> Self::Iter;  
}  
  
impl<'a> Foo for &'a Bar {  
    type Item = &'a PathBuf;  
    type Iter = std::slice::Iter<'a, PathBuf>;  
  
    fn get(self) -> Self::Iter {  
        self.v.iter()  
    }  
}
```

Your `fooget` function now simply becomes

```
fn fooget<T: Foo>(foo: T) {}
```

The problem with this is that the `fooget` function doesn't know `T` is in reality a `&Bar`. When you call the `get` function, you are actually moving out of the `foo` variable. You don't move out of the object, you just move the reference. If your `fooget` function tries to call `get` twice, the function won't compile.

If you want your `fooget` function to only accept arguments where the `Foo` trait is implemented for references, you need to explicitly state this bound:

```
fn fooget_twice<'a, T>(foo: &'a T)
where
    &'a T: Foo,
{}
```

The `where` clause makes sure that you only call this function for references where `Foo` was implemented for the reference instead of the type. It may also be implemented for both.

Technically, the compiler could automatically infer the lifetime in `fooget_twice` so you could write it as

```
fn fooget_twice<T>(foo: &T)
where
    &T: Foo,
{}
```

but it's not smart enough [yet](#).

For more complicated cases, you can use a Rust feature which is not yet implemented: [Generic Associated Types](#) (GATs). Work for that is being tracked in [issue 44265](#).

Share Edit Follow

edited Oct 5, 2020 at 17:36



vallentin

22.3k65677

answered Nov 17, 2015 at 11:46




oli_obk

26.9k47894

- I've chosen to use the first solution, since it seems to impose less burden from the point of view of `fooget` like functions. The trait is also more explicit w.r.t the second solution. – [mbrt](#) Nov 19, 2015 at 10:40

There are more than two solutions. See my answer for another one that does not have the disadvantages of these two, albeit kludgy in its own way. – [mzabaluev](#) Jan 12, 2019 at 23:53

1 Hello, do you know how to refer to the associated type `T::Item` in the second method where `T` does not implement `Foo` , only `&'a T: Foo` ? Edit: I fount it! It's `<&T as Foo>::Item` – [张实唯](#) Jan 11, 2020 at 8:13 

▲ Use a wrapper type

5 If the trait and all its implementations are defined in one crate, a helper type can be useful:



```
trait Foo {
    fn get<'a>(&'a self) -> IterableFoo<'a, Self> {
        IterableFoo(self)
    }
}

struct IterableFoo<'a, T: ?Sized + Foo>(pub &'a T);
```

For a concrete type that implements `Foo` , implement the iterator conversion on the `IterableFoo` wrapping it:

```
impl Foo for Bar {}
```

```
impl<'a> IntoIterator for IterableFoo<'a, Bar> {
    type Item = &'a PathBuf;
    type IntoIter = std::slice::Iter<'a, PathBuf>;
    fn into_iter(self) -> Self::IntoIter {
        self.0.v.iter()
    }
}
```

This solution does not allow implementations in a different crate. Another disadvantage is that an `IntoIterator` bound cannot be encoded into the definition of the trait, so it will need to be specified as an additional (and higher-rank) bound for generic code that wants to iterate over the result of

`Foo::get` :

```
fn use_foo_get<T>(foo: &T)
where
    T: Foo,
    for<'a> IterableFoo<'a, T>: IntoIterator,
    for<'a> <IterableFoo<'a, T> as IntoIterator>::Item: AsRef<Path>
{
    for p in foo.get() {
        println!("{}", p.as_ref().to_string_lossy());
    }
}
```

Associated type for an internal object providing desired functionality

The trait can define an associated type that gives access to a part of the object that, bound in a reference, provides the necessary access traits.

```
trait Foo {
    type Iterable: ?Sized;

    fn get(&self) -> &Self::Iterable;
}
```

This requires that any implementation type contains a part that can be so exposed:

```
impl Foo for Bar {
    type Iterable = [PathBuf];

    fn get(&self) -> &Self::Iterable {
        &self.v
    }
}
```

Put bounds on the reference to the associated type in generic code that uses the the result of `get` :

```
fn use_foo_get<'a, T>(foo: &'a T)
where
    T: Foo,
    &'a T::Iterable: IntoIterator,
    <&'a T::Iterable as IntoIterator>::Item: AsRef<Path>
{
    for p in foo.get() {
        println!("{}", p.as_ref().to_string_lossy());
    }
}
```

This solution permits implementations outside of the trait definition crate. The bound work at generic use sites is as annoying as with the previous solution. An implementing type may need an internal shell struct with the only purpose of providing the associated type, in case when the use-site bounds are not as readily satisfied as with `Vec` and `IntoIterator` in the example discussed.

Share Edit Follow

edited Feb 15, 2019 at 13:17

answered Jan 12, 2019 at 23:43



[mzabaluev](#)

422 5 12

That's an interesting solution, but it seems to me that it can only work if `IterableFoo` and `Bar` are defined in the same crate, right? So you can't use it in a generic trait, defined by your crate, that users of your crate would implement in their own... Or am I missing something? – [Pierre-Antoine](#) Jan 22, 2019 at 11:16

@Pierre-Antoine I have added another solution that permits off-crate implementations. – [mzabaluev](#) Jan 27, 2019 at 20:35



In future, you'll [want](#) an [associated type constructor](#) for your lifetime `'a` but Rust does not support that yet. See [RFC 1598](#)

1

Share Edit Follow



answered Apr 23, 2017 at 10:21



[Jeff Burdges](#)

4,156 22 46