How to return a Rust closure that borrows state to use later?

Asked 2 years, 2 months ago
Active 2 years, 2 months ago
Viewed 172 times

2

I have a fairly simple bit of code. I have a feeling I need to use a lifetime to accomplish this but I'm stumped right now.

`parse_string` is a function that accepts a reference to a string, and returns a closure to be used later, here's the code:

```
fn main() {
    let parse_this = parse_string(&String::from("Hello!"));
    println!("{}", parse_this("goodbye!"));
}

fn parse_string(string: &String) -> impl Fn(&str) -> &String {
    return |targetString| {
        // pretend there is parsing logic
        println!("{}", targetString);
        return string;
    };
}
```

Compiler error:

```
error: cannot infer an appropriate lifetime
  --> src/main.rs:7:12
   |
6  |  fn parse_string(string: &String) -> impl Fn(&str) -> &String {
   |                                      ---------------------- this return type evaluates to the `'static` lifetime...
7  |      return |targetString| {
   |  ____________^
8  | |        // pretend there is parsing logic
9  | |        println!("{}", targetString);
10 | |        return string;
11 | |    };
   | |_____^ ...but this borrow...
   |

note: ...can't outlive the anonymous lifetime #1 defined on the function body at 6:1
  --> src/main.rs:6:1
   |
6  | / fn parse_string(string: &String) -> impl Fn(&str) -> &String {
7  | |    return |targetString| {
8  | |        // pretend there is parsing logic
9  | |        println!("{}", targetString);
10 | |        return string;
11 | |    };
12 | | }
   | |_^

help: you can add a constraint to the return type to make it last less than `'static` and match the anonymous lifetime #1 defined on the function body at 6:1
   |
6  | fn parse_string(string: &String) -> impl Fn(&str) -> &String + '_ {
   |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error[E0312]: lifetime of reference outlives lifetime of borrowed content...
  --> src/main.rs:10:16
   |
10 |        return string;
   |               ^^^^^^
   |
```

rust closures lifetime

Share
Improve this question
Follow

edited Oct 2 '19 at 19:21

Shepmaster
305k ● 59 ● 824 ● 1083

asked Oct 2 '19 at 19:03

SweetCoco
91 ● 1 ● 7

Add a comment

2 Answers

Active Oldest Votes

2

You have a number of compounding issues:

1. You need an explicit lifetime to connect the lifetime of the `string` argument to the lifetime of the return value of the returned closure. Right now, lifetime elision causes it to be inferred the same as the *argument* to the closure.
2. You cannot return a reference to the temporary through the function. It needs to be a distinct variable.
3. You have to move `string` into the closure to prevent taking another reference to it, which wouldn't live long enough.

Additionally...

1. `targetString` should be `target_string` to follow Rust idioms.
2. `return` should not be used at the end of a block to follow Rust idioms.
3. `&str` is generally preferred to `&String`

```
fn main() {
    let s = String::from("Hello!");
    let parse_this = parse_string(&s);
    println!("{}", parse_this("goodbye!"));
}

fn parse_string<'a>(string: &'a String) -> impl Fn(&str) -> &'a String {
    return move |target_string| {
        // pretend there is parsing logic
        println!("{}", target_string);
        string
    };
}
```

See also:

- Lifetimes for method returning iterator of structs with same lifetime
- Why is it discouraged to accept a reference to a String (&String), Vec (&Vec), or Box (&Box) as a function argument?
- Getting "temporary value dropped while borrowed" when trying to update an Option<&str> in a loop

Share
Improve this answer
Follow
edited Oct 2 '19 at 19:45

answered Oct 2 '19 at 19:31

Shepmaster
305k ● 59 ● 824 ● 1083

Add a comment

2

You need to add an explicit lifetime annotation to `parse_string` so that the compiler can tell which lifetimes are the same and which may be different.

`Fn(&str) -> &String` would be the type for a function that returns a `&String` of the same lifetime as the `&str` passed in; i.e., `for<'b> Fn(&'b str) -> &'b String`. You need to say that the `&String` returned has the same lifetime as the `&String` passed in to `parse_string`:

```
fn parse_string<'a>(string: &'a String) -> impl Fn(&str) -> &'a String {
```

Note that `Fn(&str)` doesn't have a lifetime annotation; this is because the lifetime of the `&str` passed into the *closure* is unrelated to the lifetime of the `&String` passed into `parse_string`.

In order to make `parse_string` compile, you need to make one more change. Closures try to borrow their environment if the compiler thinks it doesn't need to be moved. Your closure, which borrows `string`, can't be returned from the function where `string` is a local variable. To fix this, you `move` the captured variable into the closure:

```
    move |target_string| {
        // pretend there is parsing logic
        println!("{}", target_string);
        string
    }
```

It's idiomatic in Rust to omit the `return` in the last expression in a function.

Also note that `&String` is an unusual type because it offers no expressivity that `&str` does not provide. It is almost always a mistake to have `&String` in non-generic code. See Why is it discouraged to accept a reference to a String (&String), Vec (&Vec), or Box (&Box) as a function argument? for more information.

Putting it all together, here's how I'd write `parse_string`:

```
fn parse_string<'a>(string: &'a str) -> impl Fn(&str) -> &'a str {
    move |target_string| {
        // pretend there is parsing logic
        println!("{}", target_string);
        string
    }
}
```

Your `main` also needs a small tweak: `&String::from("Hello!")` takes a reference to a temporary `String` that will be dropped immediately at the end of the line, invalidating the reference. This is easily fixed by storing the `String` in a variable so it will not be dropped until the end of the scope:

```
fn main() {
    let hello = String::from("Hello!");
    let parse_this = parse_string(&hello);
    println!("{}", parse_this("goodbye!"));
}
```

Share
Improve this answer
Follow
edited Oct 2 '19 at 19:40

answered Oct 2 '19 at 19:32

trent formerly cl
20.1k ● 7 ● 42 ● 72

I'm actually not sure why `string` is reborrowed here instead of moved. Guess the compiler just isn't quite clever enough
– trent formerly cl
Oct 2 '19 at 19:43

Add a comment

Your Answer

Post Your Answer

Not the answer you're looking for? Browse other questions tagged rust closures lifetime or ask your own question.

Linked

Why is it discouraged to accept a reference to a String (&String), Vec (&Vec), or Box (&Box) as a function argument?

Getting "temporary value dropped while borrowed" when trying to update an Option<&str> in a loop

Lifetimes for method returning iterator of structs with same lifetime

Related

In PHP, what is a closure and why does it use the "use" identifier?

Rust: How to specify lifetimes in closure arguments?

How can I return None from a function that borrows from it's argument, or avoid needing to?

Factory method: instance does not live long enough

Why is function argument lifetime different to the lifetime of a binding inside a function?

1
What is the better way to wrap a FFI struct that owns or borrows data?

How to add lifetime argument to closure not returning a reference

1

How to specify lifetime for associated type that will be a closure argument?

can't use implemented trait in other file rust

Hot Network Questions

- How to increase white wine shelf life specifically bought for cooking?
- Efficent way to let objects appear/disappear
- What do I do when my boss is sabotaging interviews?
- What does the numbers mean in this guitar tab if they already gave the chords to play?
- Seeing oneself in an abstract painting

more hot questions

Question feed