Multiple lifetimes and move: assignment to borrowed `x` occurs here

Asked 3 years, 6 months ago
Active 3 years ago
Viewed 179 times

4

I have a struct with a function `next()` (similar to iterators but not an iterator). This method returns the next state after modification (preserving the original state). So: `fn next(&A) -> A`.

I started with a simple struct where I didn't need a lifetime (struct A in the example) and I extended it to add a reference to a new struct (struct B).

The problem is that I now need to specify the lifetime for my struct and for some reason my method `next()` refuses to work anymore.

I suspect that the lifetime of the new struct of every iteration is limited to the scope where it is created and I cannot move it outside of this scope.

Is it possible to preserve the behavior of my `next()` method?

Try it here

```
#[derive(Clone)]
struct A(u32);
#[derive(Clone)]
struct B<'a>(u32, &'a u32);

impl A {
  fn next(&self) -> A {
    let mut new = self.clone();
    new.0 = new.0 + 1;
    new
  }
}

impl<'a> B<'a> {
  fn next(&self) -> B {
    let mut new = self.clone();
    new.0 = new.0 + 1;
    new
  }
}

fn main() {
  let mut a = A(0);
  for _ in 0..5 {
    a = a.next();
  }

  let x = 0;
  let mut b = B(0, &x);
  for _ in 0..5 {
    b = b.next();
  }
}
```

The error is:

```
error[E0506]: cannot assign to `b` because it is borrowed
 --> src/main.rs:31:9
  |
31 |     b = b.next();
  |     ^^^^^^^^^^^^
  |     | |
  |     | borrow of `b` occurs here
  |     assignment to borrowed `b` occurs here
```

reference rust lifetime borrowing

edited Dec 18 '18 at 17:06
Peter Hall
41.4k 11 90 157

asked Jun 2 '18 at 10:11
Cecile
1,323 12 23

1 Answer

7

The problem is here:

```
impl<'a> B<'a> {
  fn next(&self) -> B {
    let mut new = self.clone();
    new.0 = new.0 + 1;
    new
  }
}
```

You didn't specify a lifetime for `B`, the return type of `next`. Because of Rust's lifetime elision rules, the compiler infers that you intended this:

```
impl<'a> B<'a> {
  fn next<'c>(&'c self) -> B<'c> {
    let mut new = self.clone();
    new.0 = new.0 + 1;
    new
  }
}
```

Which means that the return value may not outlive `self`. Or, put another way, `self` has to live longer than the `B` that is returned. Given the body of the function, this is a *completely unnecessary requirement* because those references are independent of each other. And it causes a problem here:

```
for _ in 0..5 {
  b = b.next();
}
```

You are overwriting a value that the borrow-checker thinks is still borrowed by the call to `next()`. Inside `next` we know that there is no such relationship – the lifetime annotations do not reflect the constraints of what you're actually doing.

So what are the lifetime bounds here?

1. The lifetimes of references to `B` are unrelated - each can exist without the other. So, to give the most flexibility to a caller, the lifetime of `B` should be different from the lifetime of the reference to `self` in `next`.
2. However, each `B` that is created with `next()` holds a reference to *the same* `u32` as is held by `self`. So the lifetime *parameter* that you give to each `B` must be the same.

Using explicitly named lifetimes, this is the result of combining both of those things:

```
impl<'a> B<'a> {
  fn next<'c>(&'c self) -> B<'a> {
    let mut new = self.clone();
    new.0 = new.0 + 1;
    new
  }
}
```

Note that — even though the *reference* to `self` here has lifetime `'c` — the type of `self` is `B<'a>`, where `'a` is the lifetime of the `&u32` inside. Just the same as the return value.

But actually, the `'c` can be elided. So it's really just the same as this:

```
impl<'a> B<'a> {
  fn next(&self) -> B<'a> {
    let mut new = self.clone();
    new.0 = new.0 + 1;
    new
  }
}
```

Share
Improve this answer
Follow
edited Jun 2 '18 at 21:27

answered Jun 2 '18 at 11:29

Peter Hall
41.4k 11 90 157

1

It's not only a good answer, it helps me understand better those lifetimes. Thanks a 100x times!
– Cecile
Jun 2 '18 at 13:05

Add a comment

Your Answer

Post Your Answer

By clicking "Post Your Answer", you agree to our terms of service, privacy policy and cookie policy

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

The Overflow Blog

- Sequencing your DNA with a USB dongle and open source code
- Don't push that button: Exploring the software that flies SpaceX rockets and...

Featured on Meta

- Providing a JavaScript API for userscripts
- Congratulations to the 59 sites that just left Beta

Related

List changes unexpectedly after assignment. Why is this and how can I prevent it?

6
cannot move out of borrowed content when unwrapping a member variable in a &mut self method

Cannot move out of borrowed content when trying to transfer ownership

Cannot move out of borrowed content when borrowing a generic type

Linking the lifetimes of self and a reference in method

Problems with Tuple's lifetime in rust.

6
Rust reference dropped here while still borrowed

Hot Network Questions

Text overflow in tabularx

Applied mathematics or Computer science PhD?

Why did the JWST solar array deploy early?

Have there been no deaths due to omicron in Africa?

`apt-mark showmanual` shows almost all packages. messed up?

more hot questions

Question feed