Products

Log in Sign up

Mutable borrow too long when mutating in a loop

Ask Question

Asked 1 year, 1 month ago

Active 1 year, 1 month ago

Viewed 186 times

3

I have a piece of code that needs to store `String` s and access references to those strings. I first wrote it as follows:

```
struct Pool {
    strings : Vec<String>
}

impl Pool {
    pub fn new() -> Self {
        Self {
            strings: vec![]
        }
    }

    pub fn some_f(&mut self) -> Vec<&str> {
        let mut v = vec![];

        for i in 1..10 {
            let string = format!("{}", i);
            let string_ref = self.new_string(string);
            v.push(string_ref);
        }

        v
    }

    fn new_string(&mut self, string : String) -> &str {
        self.strings.push(string);
        &self.strings.last().unwrap()[..]
    }
}
```

This does not pass the borrow checker:

```
error[E0499]: cannot borrow `*self` as mutable more than once at a time
  --> src/main.rs:19:30
   |
14 |     pub fn some_f(&mut self) -> Vec<&str> {
   |                   - let's call the lifetime of this reference `'1`
...
19 |             let string_ref = self.new_string(string);
   |                              ^^^^ mutable borrow starts here in previous iteration of loop
...
23 |         v
   |         - returning this value requires that `*self` is borrowed for `'1`
```

So apparently the borrow-checker isn't smart enough to realize that the mutable borrow doesn't extend beyond the call to `new_string` . I tried separating the part that mutates the structure from retrieving references, arriving at this code:

```
use std::vec::*;

struct Pool {
    strings : Vec<String>
}

impl Pool {
    pub fn new() -> Self {
        Self {
            strings: vec![]
        }
    }

    pub fn some_f(&mut self) -> Vec<&str> {
        let mut v = vec![];

        for i in 1..10 {
            let string = format!("{}", i);
            self.new_string(string);
        }
        for i in 1..10 {
            let string = &self.strings[i - 1];
            v.push(&string[..]);
        }

        v
    }

    fn new_string(&mut self, string : String) {
        self.strings.push(string);
    }
}
```


This is semantically equivalent (hopefully) and does compile. However doing as much as combining the two `for` loops into one:

```
for i in 1..10 {
    let string = format!("{}", i);
    self.new_string(string);
    let string = &self.strings[i - 1];
    v.push(&string[..]);
}
```

gives a similar borrowing error:

```
error[E0502]: cannot borrow `*self` as mutable because it is also borrowed as immutable
  --> src/main.rs:19:13
   |
14 |     pub fn some_f(&mut self) -> Vec<&str> {
   |                   - let's call the lifetime of this reference `'1`
...
19 |             self.new_string(string);
   |             ^^^^^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
20 |             let string = &self.strings[i - 1];
   |                           ------------ immutable borrow occurs here
...
24 |         v
   |         - returning this value requires that `self.strings` is borrowed for `'1`
```

I have several questions:

1. Why is the borrow checker so strict as to extend the mutable borrow for the entire duration of the loop in this case? Is it impossible/very hard to analyse that the `&mut` passed to `new_string` does not leak beyond that function call?
2. Is it possible to fix this issue with custom lifetimes so that I can go back to my original helper that both mutates and returns a reference?
3. Is there a different, more Rust-idiomatic way that doesn't upset the borrow checker in which I could achieve what I want, i.e. have a structure that mutates and returns references to itself?

I found this question, but I don't understand the answer (is it a negative answer to #2? no idea) and most other questions have issues with explicit lifetime parameters. My code uses only inferred lifetimes.

rust reference lifetime mutable borrow-checker

Share
Improve this question
Follow

asked Nov 1 '20 at 1:18

V0ldek
8,297 20 43

2

Here is a very good explanation. Hopefully you will get your answer there
– Ibraheem Ahmed
Nov 1 '20 at 2:51

Add a comment

1 Answer

Active Oldest Votes

1

In this case the borrow checker is correct in not allowing this:

```
self.new_string(string);
let string = &self.strings[i - 1];
v.push(&string[..]);
```

`self.new_string` could result in all previous references that you pushed to `v` to become invalid, since it might need to allocate memory for `strings` and move its contents. The borrow checker catches this because the references you push to `v` need a lifetime to match `v`'s, so `&self.strings` (and therefore `&self`) must be borrowed for the whole method, which prevents your mutable borrow.

If you use two loops, there's no shared borrow active at the time that you call `new_string`.

You can see that it's not the mutable borrow being extended that's the issue, in this (completely useless) version of the loop, which compiles:

```
for i in 1..10 {
    let string = format!("{}", i);
    self.new_string(string);
    let mut v2 = vec![];
    let string = &self.strings[i - 1];
    v2.push(&string[..]);
}
```

As for a more idiomatic way, the Vec class is free to invalidate references in mutable operations, so you can't do what you want in safe rust. You wouldn't want to do it with a c++ vector either, even if the compiler lets you, unless you preallocate the vector and manually ensure you never push more elements than what you initially allocated. Obviously rust doesn't want you to be manually verifying the memory safety of your program; the size of a preallocation is not visible in the type system, and cannot be checked by the borrow checker, so this approach is not possible.

You cannot solve this even if you use a fixed sized container like `[String; 10]`. In that case there could be no allocation, but what would actually make this safe is the fact that you're never updating an index from which you've already taken a reference. But rust has no concept of partial borrows from containers, so there's no way to tell it "there's a shared borrow up to index n so it's ok for me to do a mutable borrow from index n + 1".

If you really need a single loop for performance reasons, you'll need to preallocate, and use an unsafe block, eg.:

```
struct Pool {
    strings: Vec<String>,
}

const SIZE: usize = 10;

impl Pool {
    pub fn new() -> Self {
        Self {
            strings: Vec::with_capacity(SIZE),
        }
    }

    pub fn some_f(&mut self) -> Vec<&str> {
        let mut v: Vec<&str> = vec![];

        // We've allocated 10 elements, but the loop uses 9, it's OK as long as it's not the other way around!
        for i in 1..SIZE {
            let string = format!("{}", i);
            self.strings.push(string);
            let raw = &self.strings as *const Vec<String>;
            unsafe {
                let last = (*raw).last().unwrap();
                v.push(last);
            }
        }

        v
    }
}
```

A possible alternative is to use `Rc`, though if your reason for wanting a single loop is performance, using the heap + the runtime costs of reference counting might be a bad tradeoff. Here's the code in any case:

```
use std::rc::Rc;

struct Pool {
    strings: Vec<Rc<String>>,
}

impl Pool {
    pub fn new() -> Self {
        Self { strings: vec![] }
    }

    pub fn some_f(&mut self) -> Vec<Rc<String>> {
        let mut v = vec![];

        for i in 1..10 {
            let string = format!("{}", i);
            let rc = Rc::new(string);
            let result = rc.clone();
            self.strings.push(rc);
            v.push(result);
        }

        v
    }
}
```

Share
Improve this answer
Follow
edited Nov 3 '20 at 9:47

answered Nov 1 '20 at 16:40

Diego Veralli
898 ● 5 ● 12

Thanks, that actually makes sense. Is there a way to fix this with a layer of indirection? So if instead of taking references to the interior of the Vec I'd store the Strings on a heap and keep `Box` versions in the Vec? What lifetime would the `&str` slices taken from such boxed strings have then?
– V0ldek
Nov 1 '20 at 17:20

I don't think a Box would help you here, the data might be on the heap but the lifetimes and borrows of the box variables would be the same as the locals you have now.
– Diego Veralli
Nov 2 '20 at 16:58

But that would fix the underlying issue, right? If the `Vec` reallocating memory was only moving the `Box` objects and not the actual `String` objects then a reference to that `String` would not be invalid after said reallocation. And I don't need to borrow the box, I only need the underlying reference. I'm not saying that I see how to lifetime-annotate that, but I don't see why that wouldn't be possible with another layer of indirection from the memory-management standpoint.
– V0ldek
Nov 2 '20 at 18:09

The `Box` owns the underlying data, and you couldn't have two boxes pointing to the same part of the heap, so any solution involving a box would end up making a copy of the string. Better to just copy the string yourself in that case. But you can do what you want using `Rc` references. I'll add the details to the answer.
– Diego Veralli
Nov 3 '20 at 9:33

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 rust reference lifetime mutable borrow-checker 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

Linked

Cannot borrow as mutable because it is also borrowed as immutable

8
Mutable borrow in a loop

Related

5
Borrow vs mutable borrow strange failure in lifetimes

Why does linking lifetimes matter only with mutable references?

13
Why doesn't a mutable borrow of self change to immutable?

6
Save mutable reference for later even when aliased

8
Mutable borrow in a loop

Conflicting lifetime requirements when mapping mutable references with self

2
Rust zero copy lifetime handling

Why does the borrow checker disallow a second mutable borrow even if the first one is already out of scope?

Erronous mutable borrow (E0502) when trying to remove and insert into a HashMap

Hot Network Questions

Applied mathematics or Computer science PhD?

Repeating slices of an array incrementally

Does anyone know what this fan blower thingy is?

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

Do all observations arise from probability distributions?

more hot questions

Question feed