Lifetime Errors using filter_map

Asked 6 years, 1 month ago

Active 6 years, 1 month ago

Viewed 476 times

1

I'm trying to use Iterator's `filter_map` function with a `HashMap` in Rust, but I can't get it to compile. Suppose I have a `HashMap` and a list of keys. For each key, if the map contains the key, I mutate the corresponding value in the map. For example, suppose the values were of type `i32` and I wanted to increment the appropriate values.

```
use std::collections::HashMap;

fn increment(map: &mut HashMap<i32, i32>, keys: &[i32]) {
    for value in keys.iter().filter_map(|index| map.get_mut(index)) {
        *value += 1;
    }
}

fn main() {
    let mut map = HashMap::new();
    map.insert(1,2);
    map.insert(4,5);
    increment(&mut map, &[0, 1, 2]);
    assert!(*map.get(&1).unwrap() == 3);
    assert!(*map.get(&4).unwrap() == 5);
}
```

This code gives me an error related to lifetimes:

```
<anon>:4:57: 4:71 error: cannot infer an appropriate lifetime for autoref due to conflicting requirements
<anon>:4         for value in keys.iter().filter_map(|index| map.get_mut(index)) {
                                                              ^~~~~~~~~~~~~~
<anon>:4:9: 6:10 note: in this expansion of for loop expansion
<anon>:4:9: 6:10 note: first, the lifetime cannot outlive the call at 4:8...
<anon>:4         for value in keys.iter().filter_map(|index| map.get_mut(index)) {
<anon>:5             *value += 1;
<anon>:6         }
<anon>:4:9: 6:10 note: in this expansion of for loop expansion
<anon>:4:9: 6:10 note: ...so that argument is valid for the call
<anon>:4         for value in keys.iter().filter_map(|index| map.get_mut(index)) {
<anon>:5             *value += 1;
<anon>:6         }
<anon>:4:9: 6:10 note: in this expansion of for loop expansion
<anon>:4:53: 4:71 note: but, the lifetime must be valid for the method call at 4:52...
<anon>:4         for value in keys.iter().filter_map(|index| map.get_mut(index)) {
                                                          ^~~~~~~~~~~~~~~~~~
<anon>:4:9: 6:10 note: in this expansion of for loop expansion
<anon>:4:53: 4:56 note: ...so that method receiver is valid for the method call
<anon>:4         for value in keys.iter().filter_map(|index| map.get_mut(index)) {
                                                          ^~~
<anon>:4:9: 6:10 note: in this expansion of for loop expansion
error: aborting due to previous error
```

Why am I getting this error, and what would be the best way to handle this situation using idiomatic Rust?

rust lifetime

Share
Improve this question
Follow

asked Nov 16 '15 at 2:52

Iceberg
366 ● 1 ● 11

Add a comment

2 Answers

Active Oldest Votes

1

✓

I've desugared your original code[†] in order to get a more precise error. I ended up with an error on the following method:

```
impl<'a, 'b> FnMut<(&'b i32,)> for MyClosure<'a> {
    extern "rust-call"
    fn call_mut(&mut self, (index,): (&'b i32,)) -> Option<&'a mut i32> {
        self.map.get_mut(index)
    }
}
```

The error is:

```
<anon>:21:18: 21:32 error: cannot infer an appropriate lifetime for autoref due to conflicting requirements [E0495]
<anon>:21         self.map.get_mut(index)
                           ^~~~~~~~~~~~~~
```

Usually, when you return a mutable reference from a function, it's bound to the lifetime of a parameter. The returned reference causes the value passed as a parameter to the function to still be considered borrowed, and by Rust's rules, you can't have take another borrow on that value until the first borrow goes out of scope. Hence, this program doesn't compile:

```
struct Foo {
    x: i32
}

impl Foo {
    fn x_mut(&mut self) -> &mut i32 { &mut self.x }
}

fn main() {
    let mut foo = Foo { x: 0 };
    let a = foo.x_mut();
    foo.x_mut(); // error: cannot borrow `foo` as mutable more than once at a time
}
```

The problem is that you're trying to return a mutable reference with lifetime `'a`, but that lifetime does not properly express the fact that you're actually borrowing from `MyClosure`. Therefore, the compiler would not consider the `MyClosure` borrowed after the call and would allow you to call the closure again, which could potentially return a mutable reference identical to one returned previously, leading to aliasing of mutable references, which is forbidden in safe Rust.

For this to work, the `FnMut` implementation would have to be written this way:

```
impl<'a, 'b> FnMut<(&'b i32,)> for MyClosure<'a> {
    extern "rust-call"
    fn call_mut<'c>(&'c mut self, (index,): (&'b i32,)) -> Option<&'c mut i32> {
        self.map.get_mut(index)
    }
}
```

But this is not valid:

```
<anon>:19:5: 22:6 error: method `call_mut` has an incompatible type for trait:
 expected bound lifetime parameter ,
    found concrete lifetime [E0053]
<anon>:19     extern "rust-call"
<anon>:20     fn call_mut<'c>(&'c mut self, (index,): (&'b i32,)) -> Option<&'c mut i32> {
<anon>:21         self.map.get_mut(index)
<anon>:22     }
```

This is the same error that is generated when one tries to write a streaming iterator.

† Actually, this desugared code corresponds to the closure `move |index| map.get_mut(index)`. Your original closure would contain a `&mut &mut HashMap<i32, i32>` field, rather than a `&mut HashMap<i32, i32>` field.

Share
Improve this answer
Follow
edited Nov 16 '15 at 4:25

answered Nov 16 '15 at 4:07

Francis Gagné
50.1k ● 3 ● 138 ● 127

Add a comment

1

I don't know why you'd bring `filter_map` into the picture here. Simply iterating over the keys and setting the value is much more obvious to me:

```
fn increment(map: &mut HashMap<i32, i32>, keys: &[i32]) {
    for index in keys {
        if let Some(value) = map.get_mut(index) {
            *value += 1;
        }
    }
}
```

Your first solution has a massive problem — what happens if you have *the same key twice*? Since you are producing mutable references, you would then have an iterator that would have handed out the same mutable reference twice, introducing *aliasing*. This is forbidden in safe Rust.

I believe this to be the cause of your error message, but I'll admit to not fully pinning it to that issue. I do know that it would cause problems eventually though.

I don't see the aliasing effect. We are producing mutable references with `get_mut`, but the lifetimes of these references should not overlap as far as I can tell. Unless I'm missing something, it follows the same logic as your code.

To make it more obvious, here's your same code, with a `collect` tacked on:

```
let xs: Vec<_> = keys.iter().filter_map(|index| map.get_mut(index)).collect();
```

The important inner part is the same - you are producing an iterator that *might* contain multiple mutable references to the same item.

Note that the Rust compiler cannot (or at least does not) completely analyze your program to see that *in this case* you consume one value and throw it away, never holding on to multiple. All it can do is see that part of what you did *could* lead to this case and prevent you from doing it.

Conversely, the version above can *never* lead to aliasing because the mutable reference does not last beyond in `for` block.

Share
Improve this answer
Follow
edited Nov 16 '15 at 3:54

answered Nov 16 '15 at 3:34
Shepmaster
305k ● 59 ● 824 ● 1083

I don't see the aliasing effect. We are producing mutable references with get_mut, but the lifetimes of these references should not overlap as far as I can tell. Unless I'm missing something, it follows the same logic as your code.
– Iceberg
Nov 16 '15 at 3:47

Add a comment

Your Answer

Post Your Answer

Not the answer you're looking for? Browse other questions tagged rust lifetime 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

How can I explicitly specify a lifetime when implementing a trait?

"the type does not fulfill the required lifetime" when using a method in a thread

How can this instance seemingly outlive its own parameter lifetime?

1
Lifetime inference problem when implementing iterator with refs

How to understand the lifetime of function parameters and return values in Rust?

Understanding a lifetime issue

Implementing Index trait with lifetime

Hot Network Questions

Sample without replacement from 1 to N and stop when the value is less than the previous one

Alternatives to replace tandem single pole MWBC breakers?

Schrödinger's cat program

Why are nerves blocked even though potassium chanels are not blocked?

Do all observations arise from probability distributions?

more hot questions

Question feed

STACK OVERFLOW

Questions
Jobs
Developer Jobs Directory
Salary Calculator
Help
Mobile

PRODUCTS

Teams
Talent
Advertising
Enterprise

COMPANY

About
Press
Work Here
Legal
Privacy Policy
Terms of Service
Contact Us
Cookie Settings
Cookie Policy

STACK EXCHANGE NETWORK

Technology
Culture & recreation
Life & arts
Science
Professional
Business
API
Data

Blog
Facebook
Twitter
LinkedIn
Instagram