Consume Vector inside closure without cloning

Asked 12 months ago
Active 12 months ago
Viewed 71 times

2

I have this data structure.

```
let bucket = HashMap<&str, Vec<&str>>
```

Given

```
let cluster = Vec<&str>
```

I want to expand it from the `Vec` s on Bucket and **I can guarantee that I will just access each key value pair once and the `&str` in `cluster` are always a key in `bucket`.**

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

fn main() {
    let mut bucket: HashMap<&str, Vec<&str>> = HashMap::new();
    bucket.insert("a", vec!["hello", "good morning"]);
    bucket.insert("b", vec!["bye", "ciao"]);
    bucket.insert("c", vec!["good"]);
    let cluster = vec!["a", "b"];
    let cluster2 = vec!["c"];
    let mut clusters = [cluster, cluster2];
    clusters.iter_mut().for_each(|cluster| {
        // I don't like this clone
        let tmp = cluster.clone();
        let tmp = tmp.iter().flat_map(|seq| bucket[seq].
            clone() // I really don't like this other clone
        );
        cluster.extend(tmp);
    });
    println!("{:?}", clusters);
}
```

This compiles but what I really want to do is drain the vector on `bucket` since I know I won't access it again.

```
let tmp = tmp.iter().flat_map(|seq| bucket.get_mut(seq).
    unwrap().drain(..)
);
```

That gives me a compiler error:

```
error: captured variable cannot escape `FnMut` closure body
  --> src/main.rs:13:45
   |
4  |     let mut bucket: HashMap<&str, Vec<&str>> = HashMap::new();
   |         ---------- variable defined here
...
13 |         let tmp = tmp.iter().flat_map(|seq| bucket.get_mut(seq).
   |                                        - ^-----
   |                                        | |
   | _______________________________________|_| variable captured here
   | |                                      |
   | |                                      inferred to be a `FnMut` closure
14 | |           unwrap().drain(..)
   | |__________________________________^ returns a reference to a captured variable which escapes the closure body
   |
   = note: `FnMut` closures only have access to their captured variables while they are executing...
   = note: ...therefore, they cannot allow references to captured variables to escape
```

Do I need to go unsafe? How? And more importantly, is it reasonable to want to remove that `clone`?

rust closures lifetime unsafe

Share
Improve this question
Follow

edited Dec 30 '20 at 11:50
user4815162342
116k ● 13 ● 205 ● 272

asked Dec 30 '20 at 10:37
carrasco
158 ● 11

"Do I need to go unsafe?" - This is almost never the answer. If the compiler is telling you that you are borrowing mutably and immutably at the same time, getting around that with unsafe will quite often lead to undefined behaviour.
– Peter Hall
Dec 30 '20 at 10:59

@PeterHall The thing is that the `Vec` *clusters* are created from the keys of *bucket* in the real program (and a key is just put once), but the compiler does not know that so it says, "hey, bucket, which is mutated, outlives the `FnMut` closure, that's very bad, you can't reuse it!". That's why I think that I need to go `unsafe` but there may be a safe solution to this where the compiler does not yell at me.
– carrasco
Dec 30 '20 at 11:08

Add a comment

1 Answer

Active Oldest Votes

2

You can eliminate `bucket[seq].clone()` using `std::mem::take()` :

```
let tmp = tmp.iter().flat_map(
    |seq| std::mem::take(bucket.get_mut(seq).unwrap()),
);
```

That will transfer ownership of the existing `Vec` and leave an empty one in the hash map. Since the map remains in a well-defined state, this is 100% safe. Since the empty vector doesn't allocate, it is also efficient. And finally, since you can guarantee that you no longer access that key, it is correct. (Playground.)

As pointed out in the comments, an alternative is to remove the vector from the hash map, which also transfer the ownership of the vector:

```
let tmp = tmp.iter().flat_map(|seq| bucket.remove(seq).unwrap());
```

The outer `cluster.clone()` cannot be replaced with `take()` because you need the old contents. The issue here is that you cannot extend the vector you are iterating over, which Rust doesn't allow in order to implement efficient pointer-based iteration. A simple and effective solution here would be to use indices instead of iteration (playground):

```
clusters.iter_mut().for_each(|cluster| {
    let initial_len = cluster.len();
    for ind in 0..initial_len {
        let seq = cluster[ind];
        cluster.extend(std::mem::take(bucket.get_mut(seq).unwrap()));
    }
});
```

Of course, with indexing you pay the price of indirection and bound checks, but rustc/llvm is pretty good at removing both when it is safe to do so, and even if it doesn't, indexed access might still be more efficient than cloning. The only way to be sure whether this improves on your original code is to benchmark both versions on production data.

Share
Improve this answer
Follow
edited Dec 30 '20 at 13:13

answered Dec 30 '20 at 10:48
user4815162342
116k 13 205 272

This is exactly what I wanted. I had the vague idea that this `std::mem` existed, but I don't know how to use it confidently. Time to study it, I guess. Thank you so much!
– carrasco
Dec 30 '20 at 12:19

1

@carrasco Glad it helped - although the borrow checker sometimes appear daunting, you very rarely *need* unsafe when writing ordinary code. (You might need it to efficiently implement efficient data structures from scratch.) Also note that `std::mem::take()` is just a convenience wrapper for `std::mem::replace()` , which is the real beast. :)
– user4815162342
Dec 30 '20 at 12:23

1

You can use `HashMap::remove` instead of `mem::take` for values that don't have a `Default` implementation.
– Aplet123
Dec 30 '20 at 12:45

@Aplet123 Thanks, I've now updated the answer to mention it. (`std::mem::take()` is still useful for cases where you cannot easily remove, and few people know of it so I like to advertise it.)
– user4815162342
Dec 30 '20 at 13:14

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 closures lifetime unsafe 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

What is the difference between a 'closure' and a 'lambda'?

Access to Modified Closure

JavaScript closure inside loops – simple practical example

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

What is a practical use for a closure in JavaScript?

When does a closure implement Fn, FnMut and FnOnce?

How to declare a lifetime for a closure argument?

Use function pointer parameter inside closure Rust

Keeping a value by reference inside a closure

Hot Network Questions

What does this entry on the Rocinante's pilot quick-menu mean?

Why is 20m band waterfall showing signals every 50 kHz?

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

Is the science in "Don't Look Up" realistic?

Why is my reasoning incorrect - probability?

more hot questions

Question feed