Rust Inspect Iterator: cannot borrow `*` as immutable because it is also borrowed as mutable

Asked 5 years, 8 months ago
Active 5 years, 8 months ago
Viewed 936 times

1

Why can't I `push` to this vector during `inspect` and do `contains` on it during `skip_while`?

I've implemented my own iterator for my own struct `Chain` like this:

```
struct Chain {
    n: u32,
}

impl Chain {
    fn new(start: u32) -> Chain {
        Chain { n: start }
    }
}

impl Iterator for Chain {
    type Item = u32;

    fn next(&mut self) -> Option<u32> {
        self.n = digit_factorial_sum(self.n);
        Some(self.n)
    }
}
```

Now what I'd like to do it `take` while the iterator is producing unique values. So I'm `inspect`-ing the chain and pushing to a vector and then checking it in a `take_while` scope:

```
let mut v = Vec::with_capacity(terms);
Chain::new(i)
    .inspect(|&x| {
        v.push(x)
    })
    .skip_while(|&x| {
        return v.contains(&x);
    })
```

However, the Rust compile spits out this error:

```
error: cannot borrow `v` as immutable because it is also borrowed as mutable [E0502]
...
borrow occurs due to use of `v` in closure
    return v.contains(&x);
           ^
previous borrow of `v` occurs here due to use in closure; the mutable borrow prevents subsequent moves, borrows, or modification of `v` until the borrow ends
    .inspect(|&x| {
        v.push(x)
    })
```

Obviously I don't understand the concept of "borrowing". What am I doing wrong?

iterator rust lazy-evaluation borrowing

Share
Improve this question
Follow
edited Apr 9 '16 at 4:42

asked Apr 9 '16 at 1:22
alt
12.3k 17 72 115

Likely the issue has to do with your attempt at closing over a variable that is somewhere in another scope (`v`). This is very important in Rust.. so we'll need to see where `v` comes from, how its declared and how you're referencing it within the current scope.
– Simon Whitehead
Apr 9 '16 at 1:46

@SimonWhitehead edited to include `v`. Must've left it out earlier.
– alt
Apr 9 '16 at 4:42

Add a comment

1 Answer

Active Oldest Votes

3

The problem here is that you're attempting to create both a mutable and an immutable reference to the same variable, which is a violation of Rust borrowing rules. And rustc actually does say this to you very clearly.

```
let mut v = Vec::with_capacity(terms);
Chain::new(i)
    .inspect(|&x| {
        v.push(x)
    })
    .skip_while(|&x| {
        return v.contains(&x);
    })
```

Here you're trying to use `v` in two closures, first in `inspect()` argument, second in `skip_while()` argument. Non-`move` closures capture their environment by reference, so the environment of the first closure contains `&mut v`, and that of the second closure contains `&v`. Closures are created in the same expression, so even if it was guaranteed that `inspect()` ran and dropped the borrow before `skip_while()` (which I is not the actual case, because these are iterator adapters and they won't be run at all until the iterator is consumed), due to lexical borrowing rules this is prohibited.

Unfortunately, this is one of those examples when the borrow checker is overly strict. What you can do is to use `RefCell`, which allows mutation through a shared reference but introduces some run-time cost:

```
use std::cell::RefCell;

let mut v = RefCell::new(Vec::with_capacity(terms));
Chain::new(i)
    .inspect(|x| v.borrow_mut().push(*x))
    .skip_while(|x| v.borrow().contains(x))
```

I *think* it may be possible to avoid runtime penalty of `RefCell` and use `UnsafeCell` instead, because when the iterator is consumed, these closures will only run one after another, not at the same time, so there should never be a mutable and an immutable references outstanding at the same time. It could look like this:

```
use std::cell::UnsafeCell;

let mut v = UnsafeCell::new(Vec::with_capacity(terms));
Chain::new(i)
    .inspect(|x| unsafe { (&mut *v.get()).push(*x) })
    .skip_while(|x| unsafe { (&*v.get()).contains(x) })
```

But I may be wrong, and anyway, the overhead of `RefCell` is not that high unless this code is running in a *really* tight loop, so you should only use `UnsafeCell` as a last resort, only when nothing else works, and exercise extreme caution when working with it.

Share
Improve this answer
Follow
answered Apr 9 '16 at 7:27

Vladimir Matveev
104k 30 254 274

This is a good explanation, however, both your solutions break the iterator for some reason. I can't figure it out. It's almost as if the `skip_while` isn't stopping the iteration in the way it usually does and it searches forever. Either that or it's over 100x slower (I waited ~1min and it never finished).
– alt
Apr 10 '16 at 6:31

I was able to implement a solution by moving all the `push` / `contains` logic into the iterator `next()` function: github.com/JacksonGariety/euler.rs/blob/master/src/…
– alt
Apr 10 '16 at 6:33

That's pretty efficient, however, I'd like to move the logic into a `skip_while` if possible. When I use both your implementations, the `skip_while` either skips all iterations or the iterator just runs forever.
– alt
Apr 10 '16 at 6:34

@JacksonGariety, I've just written an example program using this approach and understood that it can't work. When you ask an iterator chain for a next element, it works through the entire chain. So suppose you're calling `next()` on the above chain with `inspect()` and `skip_while()` (I think that it should actually be `take_while()` with an inverted condition, otherwise it doesn't make sense). An element is taken from the original iterator and written to the collection. And then the collection is queried for this element. Naturally, it is already there, so the iteration stops immediately.
– Vladimir Matveev
Apr 10 '16 at 14:23

1

@JacksonGariety therefore, your approach for internalizing the collection into the iterator is *the* way to go. Although I would use `HashSet` instead of `Vec` because `contains()` is much more efficient on `HashSet` than on `Vec`.
– Vladimir Matveev
Apr 10 '16 at 14:24

Show **5** more comments

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 iterator rust lazy-evaluation 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

Linked

1
Cannot borrow X as immutable because it is also borrowed as mutable in a mutable closure

Related

"the immutable borrow prevents mutable borrows" when pumping events with rust-sdl2

4
Cannot borrow '*x' as mutable because it is also borrowed as immutable

cannot borrow as immutable because it is also borrowed as mutable

Borrow errors for multiple borrows

Cannot borrow as mutable because it is also borrowed as immutable

Why does the Rust compiler not optimize code assuming that two mutable references cannot alias?

Hot Network Questions

How does a river freeze when the water keeps moving?

Given many questions as to whether Jesus was born on 25 December or not, I ask if the ambiguity in scripture is meant to teach us something?

What did John the Baptist say about Jesus that resulted in many Jews in Jesus' time to put faith in him? John 10:41-42

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

What was the plutonium for, that was stolen at the start of The Amazing Spider-Man 2?

more hot questions

Question feed