How do I write an iterator that returns references to itself?

Asked 6 years, 7 months ago
Active 6 months ago
Viewed 19k times

40

9

I am having trouble expressing the lifetime of the return value of an `Iterator` implementation. How can I compile this code without changing the return value of the iterator? I'd like it to return a vector of references.

It is obvious that I am not using the lifetime parameter correctly but after trying various ways I just gave up, I have no idea what to do with it.

```
use std::iter::Iterator;

struct PermutationIterator<T> {
    vs: Vec<Vec<T>>,
    is: Vec<usize>,
}

impl<T> PermutationIterator<T> {
    fn new() -> PermutationIterator<T> {
        PermutationIterator {
            vs: vec![],
            is: vec![],
        }
    }

    fn add(&mut self, v: Vec<T>) {
        self.vs.push(v);
        self.is.push(0);
    }
}

impl<T> Iterator for PermutationIterator<T> {
    type Item = Vec<&'a T>;
    fn next(&mut self) -> Option<Vec<&T>> {
        'outer: loop {
            for i in 0..self.vs.len() {
                if self.is[i] >= self.vs[i].len() {
                    if i == 0 {
                        return None; // we are done
                    }
                    self.is[i] = 0;
                    self.is[i - 1] += 1;
                    continue 'outer;
                }
            }
```

(Playground link)

```
error[E0261]: use of undeclared lifetime name `'a`
 --> src/main.rs:23:22
  |
23 |     type Item = Vec<&'a T>;
  |                      ^^ undeclared lifetime
```

iterator rust lifetime

Share
Improve this question
Follow

edited Oct 2 '17 at 17:48
Shepmaster
305k • 59 • 824 • 1083

asked May 24 '15 at 9:54
elszben
415 • 1 • 4 • 6

possible duplicate of Iterator returning items by reference, lifetime issue – Chris Morgan May 24 '15 at 11:00

2

FYI, `loop { match i.next() { ... }}` is basically what `for v in i {}` desugars to. – Shepmaster May 24 '15 at 12:36

Add a comment

4 Answers

Active Oldest Votes

43

As far as I understand, you want want the iterator to return a vector of references into itself, right? Unfortunately, it is not possible in Rust.

This is the trimmed down `Iterator` trait:

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Item>;
}
```

Note that *there is no lifetime connection* between `&mut self` and `Option<Item>`. This means that `next()` method can't return references into the iterator itself. You just can't express a lifetime of the returned references. This is basically the reason that you couldn't find a way to specify the correct lifetime - it would've looked like this:

```
fn next<'a>(&'a mut self) -> Option<Vec<&'a T>>
```

except that this is not a valid `next()` method for `Iterator` trait.

Such iterators (the ones which can return references into themselves) are called *streaming iterators*. You can find more here, here and here, if you want.

Update. However, you can return a reference to some other structure from your iterator - that's how most of collection iterators work. It could look like this:

```
pub struct PermutationIterator<'a, T> {
    vs: &'a [Vec<T>],
    is: Vec<usize>
}

impl<'a, T> Iterator for PermutationIterator<'a, T> {
    type Item = Vec<&'a T>;

    fn next(&mut self) -> Option<Vec<&'a T>> {
        ...
    }
}
```

Note how lifetime `'a` is now declared on `impl` block. It is OK to do so (required, in fact) because you need to specify the lifetime parameter on the structure. Then you can use the same `'a` both in `Item` and in `next()` return type. Again, that's how most of collection iterators work.

Share
Improve this answer
Follow
edited May 24 '15 at 12:03

answered May 24 '15 at 10:56

Vladimir Matveev
104k 30 254 274

Does this mean that the iterator cannot return references, at all? I am not sure I fully understand the implications. You said that the iterator cannot return a reference into itself. What if I have another object storing the state and the iterator has to return reference into that object? How do I express the lifetime in that case?
– elszben
May 24 '15 at 11:19

@elszben, yes, it is possible to do the thing with a separate object for state. Please see my update on how to write lifetimes out in this case.
– Vladimir Matveev
May 24 '15 at 12:03

Thank you! I cut the thing into two pieces, now the Permutation object holds the vectors and the iterator has the mutable indices vector and a ref to the permutation and everything works as expected:)
– elszben
May 24 '15 at 14:53

Add a comment

10

@VladimirMatveev's answer is correct in how it explains *why* your code cannot compile. In a nutshell, it says that an Iterator cannot yield borrowed values from within itself.

However, it can yield borrowed values from something else. This is what is achieved with `Vec` and `Iter`: the `Vec` owns the values, and the the `Iter` is just a wrapper able to yield references within the `Vec`.

Here is a design which achieves what you want. The iterator is, like with `Vec` and `Iter`, just a wrapper over other containers who actually own the values.

```
use std::iter::Iterator;

struct PermutationIterator<'a, T: 'a> {
    vs : Vec<&'a [T]>,
    is : Vec<usize>
}

impl<'a, T> PermutationIterator<'a, T> {
    fn new() -> PermutationIterator<'a, T> { ... }

    fn add(&mut self, v : &'a [T]) { ... }
}

impl<'a, T> Iterator for PermutationIterator<'a, T> {
    type Item = Vec<&'a T>;
    fn next(&mut self) -> Option<Vec<&'a T>> { ... }
}

fn main() {
    let v1 : Vec<i32> = (1..3).collect();
    let v2 : Vec<i32> = (3..5).collect();
    let v3 : Vec<i32> = (1..6).collect();

    let mut i = PermutationIterator::new();
    i.add(&v1);
    i.add(&v2);
    i.add(&v3);

    loop {
        match i.next() {
            Some(v) => { println!("{:?}", v); }
            None => {break;}
        }
    }
}
```

(Playground)

Unrelated to your initial problem. If this were just me, I would ensure that all borrowed vectors are taken at once. The idea is to remove the repeated calls to `add` and to pass directly all borrowed vectors at construction:

```
use std::iter::{Iterator, repeat};

struct PermutationIterator<'a, T: 'a> {
    ...
}

impl<'a, T> PermutationIterator<'a, T> {
    fn new(vs: Vec<&'a [T]>) -> PermutationIterator<'a, T> {
        let n = vs.len();
        PermutationIterator {
            vs: vs,
            is: repeat(0).take(n).collect(),
        }
    }
}

impl<'a, T> Iterator for PermutationIterator<'a, T> {
    ...
}

fn main() {
    let v1 : Vec<i32> = (1..3).collect();
    let v2 : Vec<i32> = (3..5).collect();
    let v3 : Vec<i32> = (1..6).collect();
    let vall: Vec<&[i32]> = vec![&v1, &v2, &v3];

    let mut i = PermutationIterator::new(vall);
}
```

(Playground)

(**EDIT**: Changed the iterator design to take a `Vec<&'a [T]>` rather than a `Vec<Vec<&'a T>>`. It's easier to take a ref to container than to build a container of refs.)

Share
Improve this answer
Follow
edited May 23 '17 at 10:31
Community Bot
1 1

answered May 24 '15 at 11:43
mdup
6,401 3 29 34

I want the Permutation object to own the vectors that hold the values, so I'll use values instead of refs there. I don't fully understand your motivation to limit that a specific vector can only be added once. Why is that useful? Anyway, thanks for the effort. It really helped me that so many versions got implemented:)
– elszben
May 24 '15 at 14:52

The motivation for my suggestion was to behave like other iterators in Rust's stdlib: the iterator is created all at once over the container, not in several steps. (e.g `myvec.iter()`). After one use, the iterator becomes consumed, i.e. unusable. Your `add()` design suggests the opposite. But that's not necessarily a bad thing :)
– mdup
May 24 '15 at 15:02

Add a comment

4

As mentioned in other answers, this is called a *streaming iterator* and it requires different guarantees from Rust's `Iterator`. One crate that provides such functionality is aptly called streaming-iterator and it provides the `StreamingIterator` trait.

Here is one example of implementing the trait:

```
extern crate streaming_iterator;

use streaming_iterator::StreamingIterator;

struct Demonstration {
    scores: Vec<i32>,
    position: usize,
}

// Since `StreamingIterator` requires that we be able to call
// `advance` before `get`, we have to start "before" the first
// element. We assume that there will never be the maximum number of
// entries in the `Vec`, so we use `usize::MAX` as our sentinel value.
impl Demonstration {
    fn new() -> Self {
        Demonstration {
            scores: vec![1, 2, 3],
            position: std::usize::MAX,
        }
    }

    fn reset(&mut self) {
        self.position = std::usize::MAX;
    }
}

impl StreamingIterator for Demonstration {
    type Item = i32;

    fn advance(&mut self) {
        self.position = self.position.wrapping_add(1);
    }

    fn get(&self) -> Option<&Self::Item> {
        self.scores.get(self.position)
```

Unfortunately, streaming iterators will be limited until *generic associated types* (GATs) from RFC 1598 are implemented.

Share
Improve this answer
Follow
answered Apr 6 '18 at 2:46

Shepmaster
305k • 59 • 824 • 1083

Add a comment

0

I wrote this code not long ago and somehow stumbled on this question here. It does exactly what the question asks: it shows how to implement an iterator that passes its callbacks a reference to itself.

It adds an `.iter_map()` method to `IntoIterator` instances. Initially I thought it should be implemented for `Iterator` itself, but that was a less flexible design decision.

I created a small crate for it and posted my code to GitHub in case you want to experiment with it, you can find it here.

WRT the OP's trouble with defining lifetimes for the items, I didn't run into any such trouble implementing this while relying on the default elided lifetimes.

Here's an example of usage. Note the parameter the callback receives is the iterator itself, the callback is expected to pull the data from it and either pass it along as is or do whatever other operations.

```
use iter_map::IntoIterMap;

let mut b = true;

let s = "hello world!".chars().peekable().iter_map(|iter| {
    if let Some(&ch) = iter.peek() {
        if ch == 'o' && b {
            b = false;
            Some('0')
        } else {
            b = true;
            iter.next()
        }
    } else { None }
}).collect::<String>();

assert_eq!(&s, "hell0o w0orld!");
```

Because the `IntoIterMap` generic trait is implemented for `IntoIterator`, you can get an "iter map" off anything that supports that interface. For instance, one can be created directly off an array, like so:

```
use iter_map::*;

fn main()
{
    let mut i = 0;

    let v = [1, 2, 3, 4, 5, 6].iter_map(move |iter| {
        i += 1;
        if i % 3 == 0 {
            Some(0)
        } else {
            iter.next().copied()
        }
    }).collect::<Vec<_>>();

    assert_eq!(v, vec![1, 2, 0, 3, 4, 0, 5, 6, 0]);
}
```

Here's the full code - it was amazing it took such little code to implement, and everything just seemed to work smoothly while putting it together. It gave me a new appreciation for the flexibility of Rust itself and its design decisions.

```
/// Adds `.iter_map()` method to all IntoIterator classes.
///
impl<F, I, J, R, T> IntoIterMap<F, I, R, T> for J
//
where F: FnMut(&mut I) -> Option<R>,
      I: Iterator<Item = T>,
      J: IntoIterator<Item = T, IntoIter = I>,
{
    /// Returns an iterator that invokes the callback in `.next()`, passing it
    /// the original iterator as an argument. The callback can return any
    /// arbitrary type within an `Option`.
    ///
    fn iter_map(self, callback: F) -> ParamFromFnIter<F, I>
    {
        ParamFromFnIter::new(self.into_iter(), callback)
    }
}

/// A trait to add the `.iter_map()` method to any existing class.
///
pub trait IntoIterMap<F, I, R, T>
//
where F: FnMut(&mut I) -> Option<R>,
      I: Iterator<Item = T>,
{
    /// Returns a `ParamFromFnIter` iterator which wraps the iterator it's
    /// invoked on.
    ///
    /// # Arguments
    /// * `callback`  - The callback that gets invoked by `.next()`.
    ///                 This callback is passed the original iterator as its
    ///                 parameter.
    ///
    fn iter_map(self, callback: F) -> ParamFromFnIter<F, I>;
}
```

Share
Improve this answer
Follow
edited Jun 22 at 6:28

answered Jun 22 at 5:54

Todd
3,594 1 12 23

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 iterator 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

Linked

3
Lifetime problem when implementing Iterator with item type &str

2
How to create an iterator that allows mapping indices to mutable items in a slice

2
How do I generate an iterator based off some underlying data in a struct in Rust?

2
How to return a reference to a field in an Iterator?

2
Indexing a struct member Vec with a member HashMap in Rust

Generative iterator that gives out references to itself

Compilation error while trying to implement an iterator that borrows some data and owns other data

0
How can I split an iterator by a separator into an iterator of iterators?

0
How to access an array iteratively and return the result with slice?

1
Iterator with `&mut` items

See more linked questions

Related

How to implement an STL-style iterator and avoid common pitfalls?

How to convert an Iterator to a Stream?

How can I create my own data structure with an iterator that returns mutable references?

How to write a Rust function that takes an iterator?

Iterator returning a reference to itself

0
Return and consume an iterator of mutable references from a closure

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

0
Lifetime parameter problem in custom iterator over mutable references

How can I fix "cannot infer an appropriate lifetime for autoref" when implementing an iterator that returns mutable references?

Hot Network Questions

Seeing oneself in an abstract painting
Company kept previous personal phone number
Send Geometry nodes value into Shading tab
Is it possible to convert a taproot address into a native segwit address?
Why couldn't Smith absorb Neo in The Matrix Reloaded?

more hot questions

Question feed