Can't borrow reference to structure in captured tree because it doesn't live long enough

Ask Question

Asked 1 year, 4 months ago
Active 8 months ago
Viewed 139 times

2

I have a tree structure with a node and children, and a loop from a GUI library which expects a function to run on each iteration. I'm struggling to get the borrow checker to let me keep a reference to the node I'm processing - it complains that `nodes` doesn't live long enough.

Here's a minimal reproduction:

```
#[derive(Debug)]
struct Node {
    value: u64,
    children: Vec<Node>,
}

fn run_loop<F>(mut handler: F)
where
    F: 'static + FnMut(),
{
    for _ in 0..500 {
        handler();
    }
}

fn main() {
    let nodes = vec![
        Node {
            value: 1,
            children: vec![Node {
                value: 3,
                children: vec![],
            }],
        },
        Node {
            value: 2,
            children: vec![],
        },
    ];
    let mut node = &nodes[0];

    run_loop(move || {
        println!("Node: {:?}", node);
        node = &node.children[0];
    });
}
```

```
error[E0597]: `nodes` does not live long enough
  --> src/main.rs:30:21
   |
30 |     let mut node = &nodes[0];
   |                    ^^^^^ borrowed value does not live long enough
31 |
32 | /     run_loop(move || {
33 | |         println!("Node: {:?}", node);
34 | |         node = &node.children[0];
35 | |     });
   | |______- argument requires that `nodes` is borrowed for `'static`
36 |   }
   |   - `nodes` dropped here while still borrowed
```

Rust Playground

What's the best way to make this work? I can't change the structure of `run_loop`. Ideally I wouldn't change the structure of `Node` (it's an object returned from a third-party library so while I could parse the object out into a new data structure, that wouldn't be elegant). Can I make the borrow checker happy with this just making changes in `main`?

rust lifetime borrow-checker

Share
Improve this question
Follow

edited Apr 7 at 14:11

Shepmaster
305k ● 59 ● 824 ● 1083

asked Jul 31 '20 at 13:06

Julian
2,243 ● 18 ● 20

2

This is the "very tricky" scenario because you'd need to move both `node` and `nodes` tied together, with some sort of guarantee that `node` will never outlive `nodes`. And of course, the funny thing is that you can't move `nodes` because it's borrowed by `node` anyway.
– Bartek Banachewicz

Jul 31 '20 at 13:20

I should add that really all I care about is the ability to walk through the nodes in the loop. If this were c++, I'd just want a a pointer to the current node. I'm pretty sure I can fix the issue by, for instance, taking all the nodes and wrapping them in `Rc`s, but I'm *hoping* there's a somewhat more elegant solution since in principle `nodes` can live as long as the closure and never needs to move/change.
– Julian
Jul 31 '20 at 13:35

1

If there's some clever way to initialize `node` inside the closure, that would work. I can't see any way to pass state to the closure without initializing it outside, and depending on capture, but if there's a way to do that I'd love to know it.
– Julian
Jul 31 '20 at 13:52

I'm not sure, but would making `node` an `Option<&Node>` and setting it to `Some(&nodes[0])` inside the loop work? (possibly with `-Z polonius=y` ?)
– Solomon Ucko
Jul 31 '20 at 16:46

Add a comment

2 Answers

Active Oldest Votes

4

it complains that nodes doesn't live long enough.

That's because it doesn't. The `run_loop` function requires its argument to live forever (`'static`). The `nodes` variable does not live forever, and consequently the closure that captures it does not live forever.

The easy fix would be to change `run_loop` to not require an argument that lives forever (by removing the `'static` constraint), but if you cannot do that then you can just make `nodes` live forever instead. You can do this by "leaking" it.

```
let nodes = vec![ /*...*/ ];
let nodes = Vec::leak(nodes);
let mut node = &nodes[0];
```

(Playground link)

At the moment, this requires nightly, but there is a similar leak function in Box in stable.

```
let nodes = vec![ /*...*/ ];
let nodes = Box::leak(nodes.into_boxed_slice());
let mut node = &nodes[0];
```

(Playground link)

Share
Improve this answer
Follow
edited Jul 31 '20 at 14:28

answered Jul 31 '20 at 14:21

cone snail
41 ● 2

I'm not sure leaking is the best solution here. The nodes don't actually have to live forever – you could also create a data structure that owns all the nodes and move that into the closure.
– Sven Marnach
Jul 31 '20 at 14:25

1

And for what it's worth, if you just want to leak the contents of the vector, it's better to use `Box:leak(nodes.into_boxed_slice())` . This avoids leaking the vector metadata as well, which you don't need anymore anyway.
– Sven Marnach
Jul 31 '20 at 14:28

Leaking looks like a good idea here. One more complication though. My simplified example used a `Vec` of nodes, but actually the object I have is this guy: docs.rs/sgf/0.1.5/sgf/sgf_node/struct.SgfCollection.html. I had (incorrectly) assumed whatever worked for a `Vec` , would work for this thin wrapper around a vec, but apparently not! Any thoughts on alternatives?
– Julian
Jul 31 '20 at 14:57

Add a comment

0

The leak solution didn't work for my actual use case, and in any case doesn't really represent the semantics of the situation or generalize very well (what if you don't want to leak the contents forever? Or what if it's not a vector you're working with?).

I ended up deciding that the best solution was just to do unsafe pointer manipulation:

```
let nodes = Box::pin(nodes);
let mut node_ptr = std::ptr::NonNull::from(&nodes[0]);

run_loop(move || {
    let node = unsafe { node_ptr.as_ref() };
    println!("Node: {:?}", node);
    node_ptr = std::ptr::NonNull::from(&(node.children[0]));
});
```

In my actual implementation I put both `nodes` and `node_ptr` in a single struct so that provides some guarantee that the nodes won't be dropped before `node_ptr`.

I'm going to leave this open since I'd love to see a solution that doesn't require `unsafe`, but am posting this here since for now at least it's the best I have.

Share
Improve this answer
Follow

answered Aug 9 '20 at 2:43

Julian
2,243 ● 18 ● 20

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 lifetime borrow-checker or ask your own question.

Related

"borrowed value does not live long enough" when using the builder pattern

Factory method: instance does not live long enough

Borrow in filter closure does not live long enough

Two mutable borrows happen on the same line?

Building a tree of vectors

Problems with Tuple's lifetime in rust.

How to avoid cloning parts when changing a mutable struct while recursion over that struct

Hot Network Questions

How to convince clan leaders and Party Cadres to give up their power?

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

Question on OEIS A000085

Most notable papers in Economics in 2021

Which Advent is it?

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