Temporarily move out of borrowed content

Ask Question

Asked 6 years, 8 months ago

Active 6 months ago

Viewed 2k times

8

I'm tring to replace a value in a mutable borrow; moving part of it into the new value:

```
enum Foo<T> {
    Bar(T),
    Baz(T),
}

impl<T> Foo<T> {
    fn switch(&mut self) {
        *self = match self {
            &mut Foo::Bar(val) => Foo::Baz(val),
            &mut Foo::Baz(val) => Foo::Bar(val),
        }
    }
}
```

The code above doesn't work, and understandibly so, moving the value out of `self` breaks the integrity of it. But since that value is dropped immediately afterwards, I (if not the compiler) could guarantee it's safety.

Is there some way to achieve this? I feel like this is a job for unsafe code, but I'm not sure how that would work.

rust ownership borrow-checker

Share
Improve this question
Follow
edited Apr 10 '15 at 21:18

asked Apr 10 '15 at 21:12

azgult
464 ● 2 ● 9

1

If you add a `Copy` bound to `T`, your code actually works, although I obviously don't know if you're ok with that restriction.
– fjh
Apr 10 '15 at 21:44

Add a comment

3 Answers

Active Oldest Votes

5

Okay, I figured out how to do it with a bit of `unsafe` ness and `std::mem`.

I replace `self` with an uninitialized temporary value. Since I now "own" what used to be `self`, I can safely move the value out of it and replace it:

```
use std::mem;

enum Foo<T> {
    Bar(T),
    Baz(T),
}

impl<T> Foo<T> {
    fn switch(&mut self) {
        // This is safe since we will overwrite it without ever reading it.
        let tmp = mem::replace(self, unsafe { mem::uninitialized() });
        // We absolutely must **never** panic while the uninitialized value is around!

        let new = match tmp {
            Foo::Bar(val) => Foo::Baz(val),
            Foo::Baz(val) => Foo::Bar(val),
        };

        let uninitialized = mem::replace(self, new);
        mem::forget(uninitialized);
    }
}

fn main() {}
```

Share
Improve this answer
Follow

edited Sep 16 '17 at 20:05

Shepmaster
305k ● 59 ● 824 ● 1083

answered Apr 10 '15 at 22:05

azgult
464 ● 2 ● 9

4

This program will fail horribly if `T` has destructor. When you call `swap` you're replacing whatever is located at `self` with garbage. Then you reassign `*self`, and Rust will insert a call to destructor which would attempt to destroy the "old" value of `*self`, which is now garbage. For some reason playpen does not fail (but you can see double free there), but for me that program core dumps when I compile and run it locally.
– Vladimir Matveev
Apr 10 '15 at 22:26

1

This program more clearly demonstrates when and how a destructor is called. If your program was safe, it would be called only once, but it is called twice - the first time being erroneous.
– Vladimir Matveev
Apr 10 '15 at 22:31

Good catch. I believe that the modified version which uses `std::ptr::write` should be safe however.
– azgult
Apr 10 '15 at 22:41

1

no it's not safe, now you are actually dropping uninitialized memory. You need to `mem::forget` the `tmp` variable after the `write` call
– oli_obk
Nov 6 '15 at 9:23

2

`take` shows how to do this in a generic way, and also protects against unwinding (aborts instead).
– Stefan
Nov 25 '17 at 15:30

Show **2** more comments

4

> The code above doesn't work, and understandibly so, moving the value out of self breaks the integrity of it.

This is not exactly what happens here. For example, same thing with `self` would work nicely:

```
impl<T> Foo<T> {
    fn switch(self) {
        self = match self {
            Foo::Bar(val) => Foo::Baz(val),
            Foo::Baz(val) => Foo::Bar(val),
        }
    }
}
```

Rust is absolutely fine with partial and total moves. The problem here is that you do not own the value you're trying to move - you only have a mutable borrowed reference. You cannot move out of any reference, including mutable ones.

This is in fact one of the frequently requested features - a special kind of reference which would allow moving out of it. It would allow several kinds of useful patterns. You can find more here and here.

In the meantime for some cases you can use `std::mem::replace` and `std::mem::swap`. These functions allow you to "take" a value out of mutable reference, provided you give something in exchange.

Share
Improve this answer
Follow

answered Apr 10 '15 at 22:01

Vladimir Matveev
104k ● 30 ● 254 ● 274

It makes little sense for me to require the caller of the method to own `Foo`, when it is possible to implement it using only a `&mut`. Ownership shouldn't be required in this case, as the integrity of `self` can be guaranteed.
– azgult
Apr 10 '15 at 22:14

@azgult and that's exactly the reason why a lot of people request `&own`-like pointer (see the links to RFC and issue I provided) - because such thing in fact *do* require ownership (*only* owner can move values around).
– Vladimir Matveev
Apr 10 '15 at 22:19

Add a comment

3

mem::uninitialized has been deprecated since Rust 1.39, replaced by MaybeUninit.

However, uninitialized data is not required here. Instead, you can use `ptr::read` to get the data referred to by `self`.

At this point, `tmp` has ownership of the data in the enum, but if we were to drop `self`, that data would attempt to be read by the destructor, causing memory unsafety.

We then perform our transformation and put the value back, restoring the safety of the type.

```
use std::ptr;

enum Foo<T> {
    Bar(T),
    Baz(T),
}

impl<T> Foo<T> {
    fn switch(&mut self) {
        // I copied this code from Stack Overflow without reading
        // the surrounding text that explains why this is safe.
        unsafe {
            let tmp = ptr::read(self);

            // Must not panic before we get to `ptr::write`

            let new = match tmp {
                Foo::Bar(val) => Foo::Baz(val),
                Foo::Baz(val) => Foo::Bar(val),
            };

            ptr::write(self, new);
        }
    }
}
```

More advanced versions of this code would **prevent** a panic from bubbling out of this code and instead cause the program to abort.

See also:

- replace_with, a crate that wraps this logic up.
- take_mut, a crate that wraps this logic up.
- Change enum variant while moving the field to the new variant
- How can I swap in a new value for a field in a mutable reference to a structure?

Share
Improve this answer
Follow
edited Jun 4 at 20:52

answered Feb 24 '20 at 18:42

Shepmaster
305k ● 59 ● 824 ● 1083

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 ownership borrow-checker 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

Is there a safe way to temporarily retrieve an owned value from a mutable reference in Rust?

How can I swap out the value of a mutable reference, temporarily taking ownership?

moving out of borrowed content safely

2
Is there some unsafe way to take ownership of a contained value in order to mutate it?

How can I replace an element in a vector with another element based on the original

0
How to replace a struct in a vector using self consuming / mutator method

0
How can I apply an arbitrary in-place operation to a &mut reference?

0
How to reorder the elements of an array in-place?

0
Is there a clean way in Rust to mutate a `&mut` by replacing its value?

0
Is it possible to take temporary ownership of a struct field?

See more linked questions

Related

Can't borrow File from &mut self (error msg: cannot move out of borrowed content)

Cannot move out of borrowed content / cannot move out of behind a shared reference

6
cannot move out of borrowed content when unwrapping a member variable in a &mut self method

Get an enum field from a struct: cannot move out of borrowed content

"Cannot move out of borrowed content" while summing command line arguments

Cannot move out of borrowed content and Builder pattern

0
&self move field containing Box - Move out of borrowed content

How can I replace the value inside a Mutex?

Cannot move out of borrowed content for Box::into_raw on self.attribute

Hot Network Questions

What does "Graecōs Argōs" in this sentence mean? (LLpsI)

Split polyline to equal parts using QGIS

Company kept previous personal phone number

Bit Rot within LUKS Encryption

How does a river freeze when the water keeps moving?

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