How can I pass a reference to a stack variable to a thread?

Asked 6 years, 3 months ago

Active 1 month ago

Viewed 10k times

48

17

I'm writing a WebSocket server where a web client connects to play chess against a multithreaded computer AI. The WebSocket server wants to pass a `Logger` object into the AI code. The `Logger` object is going to pipe down log lines from the AI to the web client. The `Logger` must contain a reference to the client connection.

I'm confused about how lifetimes interact with threads. I've reproduced the problem with a `Wrapper` struct parameterized by a type. The `run_thread` function tries to unwrap the value and log it.

```
use std::fmt::Debug;
use std::thread;

struct Wrapper<T: Debug> {
    val: T,
}

fn run_thread<T: Debug>(wrapper: Wrapper<T>) {
    let thr = thread::spawn(move || {
        println!("{:?}", wrapper.val);
    });

    thr.join();
}

fn main() {
    run_thread(Wrapper::<i32> { val: -1 });
}
```

The `wrapper` argument lives on the stack, and its lifetime doesn't extend past `run_thread`'s stack frame, even though the thread will be joined before the stack frame ends. I'd could copy the value off the stack:

```
use std::fmt::Debug;
use std::thread;

struct Wrapper<T: Debug + Send> {
    val: T,
}

fn run_thread<T: Debug + Send + 'static>(wrapper: Wrapper<T>) {
    let thr = thread::spawn(move || {
        println!("{:?}", wrapper.val);
    });

    thr.join();
}

fn main() {
    run_thread(Wrapper::<i32> { val: -1 });
}
```

This will not work if `T` is a reference to a big object I don't want copied:

```
use std::fmt::Debug;
use std::thread;

struct Wrapper<T: Debug + Send> {
    val: T,
}

fn run_thread<T: Debug + Send + 'static>(wrapper: Wrapper<T>) {
    let thr = thread::spawn(move || {
        println!("{:?}", wrapper.val);
    });

    thr.join();
}

fn main() {
    let mut v = Vec::new();
    for i in 0..1000 {
        v.push(i);
    }

    run_thread(Wrapper { val: &v });
}
```

Which results in:

```
error: `v` does not live long enough
  --> src/main.rs:22:32
   |
22 |     run_thread(Wrapper { val: &v });
   |                                ^ does not live long enough
23 | }
   | - borrowed value only lives until here
   |
   = note: borrowed value must be valid for the static lifetime...
```

The only solution I can think of is to use an `Arc`.

```
use std::fmt::Debug;
use std::sync::Arc;
use std::thread;

struct Wrapper<T: Debug + Send + Sync + 'static> {
    arc_val: Arc<T>,
}

fn run_thread<T: Debug + Send + Sync + 'static>(wrapper: &Wrapper<T>) {
    let arc_val = wrapper.arc_val.clone();
    let thr = thread::spawn(move || {
        println!("{:?}", *arc_val);
    });

    thr.join();
}

fn main() {
    let mut v = Vec::new();
    for i in 0..1000 {
        v.push(i);
    }

    let w = Wrapper { arc_val: Arc::new(v) };
    run_thread(&w);

    println!("{}", (*w.arc_val)[0]);
}
```

In my real program, it appears that both the `Logger` and the connection object must be placed in `Arc` wrappers. It seems annoying that the client is required to box the connection in an `Arc` when it is internal to the library that the code is parallelized. This is especially annoying because the lifetime of the connection is guaranteed to be greater than the lifetime of the worker threads.

Have I missed something?

multithreading rust reference lifetime

Share
Improve this question
Follow

edited Nov 7 at 0:54

applemonkey496
522 ● 1 ● 6 ● 24

asked Sep 23 '15 at 23:00

Ned Ruggeri
896 ● 2 ● 7 ● 13

Add a comment

1 Answer

Active Oldest Votes

▲
48
▼

The thread support in the standard library allows the created threads to outlive the thread that created them; that's a good thing! However, if you were to pass a reference to a stack-allocated variable to one of these threads, there's no guarantee that the variable will still be valid by the time the thread executes. In other languages, this would allow the thread to access invalid memory, creating a pile of memory safety issues.

Fortunately, we aren't limited to the standard library. At least two crates provide *scoped threads* — threads that are guaranteed to exit before a certain scope ends. These can ensure that stack variables will be available for the entire duration of the thread:

- crossbeam
- scoped-threadpool (legacy)

There are also crates that abstract away the low-level details of "threads" but allow you to accomplish your goals:

- rayon

Here are examples of each. Each example spawns a number of threads and mutates a local vector in place with no locking, no `Arc`, and no cloning. Note that the mutation has a `sleep` call to help verify that the calls are happening in parallel.

You can extend the examples to share a reference to any type which implements `Sync`, such as a `Mutex` or an `Atomic*`. Using these would introduce locking, however.

crossbeam

```
use crossbeam; // 0.6.0
use std::{thread, time::Duration};

fn main() {
    let mut vec = vec![1, 2, 3, 4, 5];

    crossbeam::scope(|scope| {
        for e in &mut vec {
            scope.spawn(move |_| {
                thread::sleep(Duration::from_secs(1));
                *e += 1;
            });
        }
    })
    .expect("A child thread panicked");

    println!("{:?}", vec);
}
```

rayon

```
use rayon::iter::{IntoParallelRefMutIterator, ParallelIterator}; // 1.0.3
use std::{thread, time::Duration};

fn main() {
    let mut vec = vec![1, 2, 3, 4, 5];

    vec.par_iter_mut().for_each(|e| {
        thread::sleep(Duration::from_secs(1));
        *e += 1;
    });

    println!("{:?}", vec);
}
```

> the client is required to box the connection in an `Arc` when it is internal to the library that the code is parallelized

Perhaps you can hide your parallelism better then? Could you accept the logger and then wrap it in an `Arc` / `Mutex` before handing it off to your threads?

Share
Improve this answer
Follow

edited May 25 at 13:17

answered Sep 24 '15 at 1:26

Shepmaster
305k ● 59 ● 824 ● 1083

1

Thanks so much for your reply! My solution was to make the `Logger` implement `Clone`, and have a field with type `Arc<Mutex<Connection>>`. Then the user could pass a clone of the logger to the threaded code. The user can't transfer ownership of the `Connection` to the threaded code (the user needs it for other purposes), so I don't see that it is possible for the threaded code to conveniently do the `Arc` and boxing on behalf of the user.
– Ned Ruggeri
Sep 24 '15 at 22:38

What do you do if the variable you are trying to pass around can't implement Clone/Copy? like a USB device handle from the rusb crate
– Brandon Ros
Jul 3 '20 at 1:15

@BrandonRos `Vec` doesn't implement `Copy` and `Clone` isn't used in these code examples. The code presented here works fine for such types.
– Shepmaster
Jul 3 '20 at 14:30

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 multithreading rust reference 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

9
Self has an anonymous lifetime but it needs to satisfy a static lifetime requirement

The lifetime of self parameter in Rust when using threads

Can you specify a non-static lifetime for threads?

6
Sharing read-only object between threads in Rust?

2
How can Rust be told that a thread does not live longer than its caller?

Parallel write to array with an indices array

How to share a pointer to a mutable variable to a thread?

Share function reference between threads in Rust

Do i have to create a copy of objects for threads need

Why is 'data' still borrowed at the end of my function?

See more linked questions

Related

3618
What are the differences between a pointer variable and a reference variable in C++?

How do I update the GUI from another thread?

When and how should I use a ThreadLocal variable?

How can I pass a parameter to a Java Thread?

How do I pass a variable by reference?

List changes unexpectedly after assignment. Why is this and how can I prevent it?

1415
How can I use threading in Python?

Android "Only the original thread that created a view hierarchy can touch its views."

Hot Network Questions

Repeating slices of an array incrementally
What is this large long-legged orange and black insect?
Company kept previous personal phone number
Which direction/constellation is the James Webb is headed towards
Best star for a Dyson sphere?
more hot questions

Question feed

STACK OVERFLOW

Questions
Jobs
Developer Jobs Directory
Salary Calculator
Help
Mobile

PRODUCTS

Teams
Talent
Advertising
Enterprise

COMPANY

About