Actors with Tokio

Published 2021-02-13

This article is about building actors with Tokio directly, without using any actor libraries such as Actix. This turns out to be rather easy to do, however there are some details you should be aware of:

1. Where to put the `tokio::spawn` call.
2. Struct with `run` method vs bare function.
3. Handles to the actor.
4. Backpressure and bounded channels.
5. Graceful shutdown.

The techniques outlined in this article should work with any executor, but for simplicity we will only talk about Tokio. There is some overlap with the spawning and channel chapters from the Tokio tutorial, and I recommend also reading those chapters.

Before we can talk about how to write an actor, we need to know what an actor is. The basic idea behind an actor is to spawn a self-contained task that performs some job independently of other parts of the program. Typically these actors communicate with the rest of the program through the use of message passing channels. Since each actor runs independently, programs designed using them are naturally parallel.

A common use-case of actors is to assign the actor exclusive ownership of some resource you want to share, and then let other tasks access this resource indirectly by talking to the actor. For example, if you are implementing a chat server, you may spawn a task for each connection, and a master task that routes chat messages between the other tasks. This is useful because the master task can avoid having to deal with network IO, and the connection tasks can focus exclusively on dealing with network IO.

I have also given a talk on this topic. Click here to view it.

The Recipe

An actor is split into two parts: the task and the handle. The task is the independently spawned Tokio task that actually performs the duties of the actor, and the handle is a struct that allows you to communicate with the task.

Let's consider a simple actor. The actor internally stores a counter that is used to obtain some sort of unique id. The basic structure of the actor would be something like the following:

```rust
use tokio::sync::{oneshot, mpsc};

struct MyActor {
    receiver: mpsc::Receiver<ActorMessage>,
    next_id: u32,
}
enum ActorMessage {
    GetUniqueId {
        respond_to: oneshot::Sender<u32>,
    },
}

impl MyActor {
    fn new(receiver: mpsc::Receiver<ActorMessage>) -> Self {
        MyActor {
            receiver,
            next_id: 0,
        }
    }
    fn handle_message(&mut self, msg: ActorMessage) {
        match msg {
            ActorMessage::GetUniqueId { respond_to } => {
                self.next_id += 1;

                // The `let _ =` ignores any errors when sending.
                //
                // This can happen if the `select!` macro is used
                // to cancel waiting for the response.
                let _ = respond_to.send(self.next_id);
            },
        }
    }
}

async fn run_my_actor(mut actor: MyActor) {
    while let Some(msg) = actor.receiver.recv().await {
        actor.handle_message(msg);
    }
}
```

Now that we have the actor itself, we also need a handle to the actor. A handle is an object that other pieces of code can use to talk to the actor, and is also what keeps the actor alive.

The handle will look like this:

```rust
#[derive(Clone)]
pub struct MyActorHandle {
    sender: mpsc::Sender<ActorMessage>,
}

impl MyActorHandle {
    pub fn new() -> Self {
        let (sender, receiver) = mpsc::channel(8);
        let actor = MyActor::new(receiver);
        tokio::spawn(run_my_actor(actor));
        Self { sender }
    }

    pub async fn get_unique_id(&self) -> u32 {
        let (send, recv) = oneshot::channel();
        let msg = ActorMessage::GetUniqueId {
            respond_to: send,
        };

        // Ignore send errors. If this send fails, so does the
        // recv.await below. There's no reason to check for the
        // same failure twice.
        let _ = self.sender.send(msg).await;
        recv.await.expect("Actor task has been killed")
    }
}
```

full example

Let's take a closer look at the different pieces in this example.

`ActorMessage`. The `ActorMessage` enum defines the kind of messages we can send to the actor. By using an enum, we can have many different message types, and each message type can have its own set of arguments. We return a value to the sender by using an oneshot channel, which is a message passing channel that allows sending exactly one message.

In the example above, we match on the enum inside a `handle_message` method on the actor struct, but that isn't the only way to structure this. One could also match on the enum in the `run_my_actor` function. Each branch in this match could then call various methods such as `get_unique_id` on the actor object.

Errors when sending messages. When dealing with channels, not all errors are fatal. Because of this, the example sometimes uses `let _ =` to ignore errors. Generally a `send` operation on a channel fails if the receiver has been dropped.

The first instance of this in our example is the line in the actor where we respond to the message we were sent. This can happen if the receiver is no longer interested in the result of the operation, e.g. if the task that sent the message might have been killed.

Shutdown of actor. We can detect when the actor should shut down by looking at failures to receive messages. In our example, this happens in the following while loop:

```
while let Some(msg) = actor.receiver.recv().await {
    actor.handle_message(msg);
}
```

When all senders to the `receiver` have been dropped, we know that we will never receive another message and can therefore shut down the actor. When this happens, the call to `.recv()` returns None, and since it does not match the pattern `Some(msg)`, the while loop exits and the function returns.

`#[derive(Clone)]` The `MyActorHandle` struct derives the `Clone` trait. It can do this because `mpsc` means that it is a multiple-producer, single-consumer channel. Since the channel allows multiple producers, we can freely clone our handle to the actor, allowing us to talk to it from multiple places.

A run method on a struct

The example I gave above uses a top-level function that isn't defined on any struct as the thing we spawn as a Tokio task, however many people find it more natural to define a run method directly on the `MyActor` struct and spawn that. This certainly works too, but the reason I give an example that uses a top-level function is that it more naturally leads you towards the approach that doesn't give you lots of lifetime issues.

To understand why, I have prepared an example of what people unfamiliar with the pattern often come up with.

```
impl MyActor {
    fn run(&mut self) {
        tokio::spawn(async move {
            while let Some(msg) = self.receiver.recv().await {
                self.handle_message(msg);
            }
        });
    }

    pub async fn get_unique_id(&self) -> u32 {
        let (send, recv) = oneshot::channel();
        let msg = ActorMessage::GetUniqueId {
            respond_to: send,
        };

        // Ignore send errors. If this send fails, so does the
        // recv.await below. There's no reason to check for the
        // same failure twice.
        let _ = self.sender.send(msg).await;
        recv.await.expect("Actor task has been killed")
    }
}

... and no separate MyActorHandle
```

The two sources of trouble in this example are:

1. The `tokio::spawn` call is inside run.
2. The actor and the handle are the same struct.

The first issue causes problems because the `tokio::spawn` function requires the argument to be `'static`. This means that the new task must own everything inside it, which is a problem because the method borrows `self`, meaning that it is not able to give away ownership of `self` to the new task.

The second issue causes problems because Rust enforces the single-ownership principle. If you combine both the actor and the handle into a single struct, you are (at least from the compiler's perspective) giving every handle access to the fields owned by the actor's task. E.g. the `next_id` integer should be owned only by the actor's task, and should not be directly accessible from any of the handles.

That said, there is a version that works. By fixing the two above problems, you end up with the following:

```rust
impl MyActor {
    async fn run(&mut self) {
        while let Some(msg) = self.receiver.recv().await {
            self.handle_message(msg);
        }
    }
}

impl MyActorHandle {
    pub fn new() -> Self {
        let (sender, receiver) = mpsc::channel(8);
        let actor = MyActor::new(receiver);
        tokio::spawn(async move { actor.run().await });

        Self { sender }
    }
}
```

This works identically to the top-level function. Note that, strictly speaking, it is possible to write a version where the `tokio::spawn` is inside `run`, but I don't recommend that approach.

Variations on the theme

The actor I used as an example in this article uses the request-response paradigm for the messages, but you don't have to do it this way. In this section I will give some inspiration to how you can change the idea.

No responses to messages

The example I used to introduce the concept includes a response to the messages sent over a `oneshot` channel, but you don't always need a response at all. In these cases there's nothing wrong with just not including the `oneshot` channel in the message enum. When there's space in the channel, this will even allow you to return from sending before the message has been processed.

You should still make sure to use a bounded channel so that the number of messages waiting in the channel don't grow without bound. In some cases this will mean that sending still needs to be an async function to handle the cases where the `send` operation needs to wait for more space in the channel.

However there is an alternative to making `send` an async method. You can use the `try_send` method, and handle sending failures by simply killing the actor. This can be useful in cases where the actor is managing a `TcpStream`, forwarding any messages you send into the connection. In this case, if writing to the `TcpStream` can't keep up, you might want to just close the connection.

Multiple handle structs for one actor

If an actor needs to be sent messages from different places, you can use multiple handle structs to enforce that some message can only be sent from some places.

When doing this you can still reuse the same `mpsc` channel internally, with an enum that has all the possible message types in it. If you *do* want to use separate channels for this purpose, the actor can use `tokio::select!` to receive from multiple channels at once.

```
loop {
    tokio::select! {
        Some(msg) = chan1.recv() => {
            // handle msg
        },
        Some(msg) = chan2.recv() => {
            // handle msg
        },
        else => break,
    }
}
```

You need to be careful with how you handle when the channels are closed, as their `recv` method immediately returns None in this case. Luckily the `tokio::select!` macro lets you handle this case by providing the pattern `Some(msg)`. If only one channel is closed, that branch is disabled and the other channel is still received from. When both are closed, the else branch runs and uses `break` to exit from the loop.

Actors sending messages to other actors

There is nothing wrong with having actors send messages to other actors. To do this, you can simply give one actor the handle of some other actor.

You need to be a bit careful if your actors form a cycle, because by holding on to each other's handle structs, the last sender is never dropped, preventing shutdown. To handle this case, you can have one of the actors have two handle structs with separate `mpsc` channels, but with a `tokio::select!` that looks like this:

```
loop {
    tokio::select! {
        opt_msg = chan1.recv() => {
            let msg = match opt_msg {
                Some(msg) => msg,
                None => break,
            };
            // handle msg
        },
        Some(msg) = chan2.recv() => {
            // handle msg
        },
    }
}
```

The above loop will always exit if chan1 is closed, even if chan2 is still open. If chan2 is the channel that is part of the actor cycle, this breaks the cycle and lets the actors shut down.

An alternative is to simply call abort on one of the actors in the cycle.

Multiple actors sharing a handle

Just like you can have multiple handles per actor, you can also have multiple actors per handle. The most common example of this is when handling a connection such as a TcpStream, where you commonly spawn two tasks: one for reading and one for writing. When using this pattern, you make the reading and writing tasks as simple as you can — their only job is to do IO. The reader task will just send any messages it receives to some other task, typically another actor, and the writer task will just forward any messages it receives to the connection.

This pattern is very useful because it isolates the complexity associated with performing IO, meaning that the rest of the program can pretend that writing something to the connection happens instantly, although the actual writing happens sometime later when the actor processes the message.

Beware of cycles

I already talked a bit about cycles under the heading “Actors sending messages to other actors”, where I discussed shutdown of actors that form a cycle. However, shutdown is not the only problem that cycles can cause, because a cycle can also result in a deadlock where each actor in the cycle is waiting for the next actor to receive a message, but that next actor wont receive that message until its next actor receives a message, and so on.

To avoid such a deadlock, you must make sure that there are no cycles of channels with bounded capacity. The reason for this is that the send method on a bounded channel does not return immediately. Channels whose send method always returns immediately do not count in this kind of cycle, as you cannot deadlock on such a `send`.

Note that this means that a oneshot channel cannot be part of a deadlocked cycle, since their `send` method always returns immediately. Note also that if you are using `try_send` rather than `send` to send the message, that also cannot be part of the deadlocked cycle.

Thanks to matklad for pointing out the issues with cycles and deadlocks.