Async Rust: Futures, Tasks, Wakers—Oh My!

Feb 5 2021

Rust Playground

Async is all the rage. Python, Rust, Go. Pick your language and chances are it's got some form of `async / await` going. If you've never worked with `async / await` before this can all be somewhat confusing, to put it mildly. In this post we'll take a high level overview of asynchronous programming in general before making our way to Rust, where we'll look at the current state of async—what it is, how to do it, and how it all works.

What Is Asynchronous Programming and Why Should We Care

Most programs are synchronous. They execute line by line in the order they're written. Whenever we encounter an operation that can't be completed immediately, we must wait until it finishes before we can proceed. Establishing a TCP connection, for example, requires several exchanges over a network, which can take some time, during which our program is unable to work on anything else. In other words, synchronous operations *block* the thread on which our program executes.

Asynchronous operations, on the other hand, are non-blocking by design. Operations that can't complete immediately are suspended to the "background" (more on this later), allowing the current thread to work on other tasks. Once an async operation finishes, the task is unsuspended and execution resumes where it left off. By working jointly on multiple tasks a program can avoid being idle. When one task can't progress because it's waiting on another resource (e.g. bytes from a socket), we simply switch to another task that can. This brings us to our first observation about asynchronous programming in general: we only benefit if there is other work to be done while we wait on an operation to finish.

For example, if we're running on a single thread and need to tackle a CPU-bound task, we have no choice but to wait until the task finishes before we can move onto the next one. Async won't help us here. However, if we're running on a thread pool, we could spawn the CPU-bound task onto another thread and call out to it asynchronously from the current thread, allowing us to do other work while we wait for it to finish. Which brings us to our second observation about asynchronous programming: it excels at any task that involves a lot of waiting. This could be a network call or a CPU-bound task as we've discussed, or a database query or RPC call. Whatever it is, if we have to wait, chances are async can help.

Concurrency and Parallelism

Before we dig into the meat and potatoes of async in Rust, let's make sure we're on the same page

regarding concurrency and parallelism. Concurrency refers to the act of working on multiple tasks at once, though not necessarily at the same time. For instance, if you alternate between doing the dishes and mopping the floor, you're working on both tasks concurrently but not in parallel. To do both in parallel you'd need two distinct individuals, one dedicated to each task. Parallelism is one way of achieving concurrency. Async is another, at least in some contexts. One of the advantages of asynchronous programming is that it allows you to work on many tasks concurrently without having to work on them in parallel via threads, which can be costly to spawn and run, depending on the implementation. As we'll see, async makes it possible to run many tasks concurrently on a single thread.

Async and the Future

Rust enables asynchronous programming via two keywords: `async` & `.await`. Functions marked as `async fn` are transformed at compile time into operations that can run asynchronously. Async functions are called just like any other function, except instead of executing when called, an async function returns a value representing the computation. This value is called a `Future`. In Rust a future is anything that implements the std::future::Future trait provided by the standard library. Every `async fn` implicitly returns a `Future`. For example, the async function

```
async fn hello() -> String {
    "hi, there!".to_string()
}
```

desugars to an ordinary function whose return type is an anonymous type implementing the future trait.

```
fn hello() -> impl Future<Output = String> {
    async {
        "hi, there!".to_string()
    }
}
```

The inner async block is a variant of Rust's ordinary block expressions which evaluates to an anonymous future type, similar to how closures return anonymous types implementing one of the `std::ops::Fn` traits.

Futures lie at the heart of async in Rust but are not the whole story. A key distinction between Rust and other languages like JS is that in Rust async operations are *lazy*. By themselves futures do nothing. They require a runtime, which provides, among other things, a task scheduler and an I/O driver backed

by the operating system's own event queue (epoll, kqueue, etc.). A very widely used runtime in Rust is provided by the tokio crate. Another is provided by async-std. All asynchronous behaviour in Rust occurs within the context of a runtime.

Now, you might be wondering why Rust doesn't just ship with a built-in runtime. The reason is Rust's commitment to zero-cost abstractions. Any runtime imposes a performance overhead that all programs must pay regardless of whether they require it. Rust chooses to instead only define asynchronous behaviour via standard library traits like `Future`, leaving the implementation to the larger Rust ecosystem.

⚠️ **Note:** An unfortunate side effect of this decision is that it forces crates to depend on a specific choice of runtime. For instance, reqwest requires `tokio` while surf requires `async_std`. This can be a big source of confusion for newcomers just looking to get the hang of async in Rust (i.e. me).

Tasks

The runtime doesn't interface with individual futures directly but instead groups them into *tasks*. A task is a lightweight, non-blocking unit of execution, similar to an OS thread, except instead of being managed by the OS scheduler, tasks are managed by the runtime. This pattern generally goes by the name *green threads*, after the java library that originally implemented them. Because tasks are cheap to create and run, a single OS thread can handle many tasks, enabling a program to run concurrently on a single thread (although a typical async program will likely have many, many tasks scheduled across multiple threads).

A key attribute of tasks is that they exhibit what is known as *cooperative multitasking*. A task is allowed to run until it *yields*, signaling to the runtime that it cannot currently progress any further. This allows the runtime to switch to another task that can progress. Tasks yield via calls to `.await`. Any invocation of `.await` within an async function or block yields control back to the runtime, which is then free to swap the current task for a different one.

When to `.await`

The most confusing thing about async in Rust is when to `.await`. Take the following example, which simulates a delayed response from, say, an http request or database query.

```
use tokio::{self, time};
use tokio::time::{Duration, Instant};

async fn slow_request(number: u32, ms: u64) {
    time::sleep(Duration::from_millis(ms)).await;
    println!("Response {} received!", number);
```

```
}

#[tokio::main]
async fn main() {
    slow_request(1, 2000);
    slow_request(2, 3000);
}
```

Although this code looks asynchronous, when we run it nothing happens. What gives? Well, even though we've configured our runtime with the `#[tokio::main]` macro, we haven't asked it to execute anything. Each call to `slow_request` returns a future which is simply dropped and never executed. To execute the futures returned by `slow_request` we must `.await` them.

```
#[tokio::main]
async fn main() {
    let start = Instant::now();
    slow_request(1, 2000).await;
    slow_request(2, 3000).await;
    println!("Finished in {} ms", start.elapsed().as_millis());
}
```

```
Response 1 received!
Response 2 received!
Finished in 5000 ms
```

Now both requests finish successfully. But there's a problem. Since we're attempting to make two concurrent requests, our program should take only as long as the longest request, however, our program takes 5 seconds even though the longest request only takes 3 seconds. The issue is that both futures are running on the same task, the one implicitly spawned by the runtime.

When we make the first call to `.await` the runtime is free to switch to another task, except there isn't any, so we're forced to wait until the first request finishes before we can make the second. In other words, our program executes *synchronously*, even though we're using `async / .await`. To execute both requests concurrently, we need to spawn another task.

```
#[tokio::main]
async fn main() {
    let start = Instant::now();
    tokio::spawn(slow_request(1, 2000));
    slow_request(2, 3000).await;
    println!("Finished in {} ms", start.elapsed().as_millis());
}
```

```
Response 1 received!
Response 2 received!
Finished in 3000 ms
```

Now our program runs in 3 seconds, as it should.

Running Multiple Tasks On A Single Thread

Let's build off of our previous example and prove that an asynchronous program can indeed run multiple tasks concurrently on a single thread. By default the `#[tokio::main]` macro uses a multi-threaded work stealing scheduler, however, we can ask it to utilize just a single thread by setting `flavor = "current_thread"`. We'll verify that this is indeed the case by re-writing `slow_request` to also print the id of the current thread.

```
use std::thread;

async fn slow_request(number:u32, ms: u64) {
    time::sleep(Duration::from_millis(ms)).await;
    println!("Response {} received on {:?}!", number, thread::current().id());
}

#[tokio::main(flavor = "current_thread")]
async fn main() {
    let start = Instant::now();

    let t1 = tokio::spawn(slow_request(1, 2000));
    let t2 = tokio::spawn(slow_request(2, 4000));
    let t3 = tokio::spawn(slow_request(3, 3000));
    let t4 = tokio::spawn(slow_request(4, 1000));
    tokio::join!(t1, t2, t3, t4);

    println!("Finished in {} ms", start.elapsed().as_millis());
}
```

```
Response 4 received on ThreadId(1)!
Response 1 received on ThreadId(1)!
Response 3 received on ThreadId(1)!
Response 2 received on ThreadId(1)!
Finished in 4000 ms
```

Note that we call `tokio::join!` here instead of simply `.await`ing the last task as in our previous example. This is because here the last request has the shortest delay, hence were we to `.await` the last task, it would finish before the others and execution would proceed to the next statement in `main`, the call to `println!`, which would complete immediately and terminate our program before the remaining tasks have finished. Calling `tokio::join!` lets the runtime know that we want to wait until *all* tasks have completed before moving on.

Futures In Depth

Rust Playground

So far we've taken a look at tasks, futures, and the runtime. We know that tasks are made up of futures and are submitted to the runtime for execution, and that each call to `.await` causes the current task to yield to the runtime, allowing the runtime to switch to another task. But how does the runtime know when to return to the original task? To answer this question, we need to take a closer look at the Future trait. This section is adapted from the superb *Async In Depth* chapter from the tokio tutorial.

```rust
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

The `Future` trait consists of a single method `poll`, which returns an enum `Poll` that tells the runtime if a future has completed or is still in progress.

```rust
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

`Pin` enables borrowing across `.await` points and `Context` provides a reference to a `Waker`, which we'll return to shortly.

If a future has finished executing, `poll` returns `Poll::Ready(val)`, where `val` is the value returned by the future. On the other hand, if a future is not able to complete (usually because it's waiting on some other resource), `poll` returns `Poll::Pending`, which informs the runtime that the future will complete at a later time and that `poll` should be called again.

But when should we poll again? The simplest solution would be to loop over all spawned tasks and poll them repeatedly, and this does indeed work, albeit not terribly efficiently because the majority of the time a future will return `Poll::Pending`. Let's see how it's done and then work towards a better solution. We'll implement a `Delay` future and modify `slow_request` to use `Delay` instead of `tokio::time::sleep` to simulate the delayed response. For good measure we'll also track how many times `Delay` is polled.

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use std::time::{Duration, Instant};

struct Delay {
    when: Instant,
    polled: u32,
}

impl Delay {
    fn new(duration: Duration) -> Delay {
        Delay {
            when: Instant::now() + duration,
            polled: 0,
        }
    }
}

impl Future for Delay {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        self.polled += 1;
        if Instant::now() >= self.when {
            println!("Polled {} times!", self.polled);
            Poll::Ready(())
        } else {
            cx.waker().wake_by_ref();
            Poll::Pending
        }
    }
}
```

Our implementation of `poll` is more or less straightforward, with the exception of the call to `waker`, which we'll examine in a second. Each call to `poll` simply checks if enough time has passed, returning `Poll::Ready` if it has and `Poll::Pending` if it hasn't. Let's see how it runs.

```
async fn slow_request(number: u32, ms: u64) {
    Delay::new(Duration::from_millis(ms)).await;
    println!("Response {} received!", number);
}

#[tokio::main]
async fn main() {
    let start = Instant::now();
    slow_request(1, 500).await;
    println!("Finished in {} ms", start.elapsed().as_millis());
}
```

```
Polled 243598 times!
Response 1 received!
Finished in 500 ms
```

Yikes. That's a lot of polls. The issue is that our implementation of `poll` causes the runtime to repeatedly poll an instance of `Delay`, which is a shame because we know at the moment a `Delay` is created that it will not return `Poll::Ready` until enough time has passed. It'd be much better if we could let the runtime know to only call `poll` again when a future is ready to progress. For example, were we reading bytes from a socket, we'd only want to poll the future when the socket has received new bytes to be read.

Enter Wakers. A `Waker` is a handle for waking a task by notifying the runtime that it is ready to be executed. In our implementation of `poll`, we get a reference to a future's `Waker` via its `Context` and immediately call `.wake()` to re-schedule the task for execution, resulting in a busy loop. Instead, let's spawn a timer thread to sleep until we know a `Delay` is ready to complete. This should result in `poll` only being called twice, once when the `Delay` is first `.await`ed and once when the delay has expired.

```
impl Future for Delay {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        self.polled += 1;

        if Instant::now() >= self.when {
            println!("Poll called {} times!", self.polled);
            Poll::Ready(())
```

```
        } else {
            let waker = cx.waker().clone();
            let when = self.when;

            thread::spawn(move || {
                let now = Instant::now();
                if now < when {
                    thread::sleep(when - now);
                }
                waker.wake();
            });

            Poll::Pending
        }
    }
}
```

Running `main` again gives:

```
Poll called 2 times!
Response 1 received!
Finished in 500 ms
```

Nice, right?

Conclusion

We've taken a long look at async in Rust, starting with futures, which must be submitted to the runtime for execution by `.await`ing them, and ending with `poll` and `Waker`, which are the means by which the runtime knows how to suspend and resume tasks. Along the way we took a high level overview of asynchronous programming in general, incldiuing one of its main advantages—the ability to run multiple tasks concurrently on a single thread—which we saw in action when we ran several tasks concurrently on a single-threaded `tokio` runtime.

As we've seen, asynchronous programming isn't always straightforward. We hope this post gives comfort to fellow new Rustaceans just looking to get the hang of async in Rust. Rust's design goal is for asynchronous programming to be as easy as synchronous programming, but with greater expressivity. While I can't say we're there yet, Rust's async story is one I look forward to reading more about in the future.

programming