Rust programming

Module E: Async and Rust for Web

Last time…

- Trait objects: dynamic dispatch
- Design patterns
- Anti-patterns

Any questions?

In this module

- Introduction Rust `async` programming
- The `Future` type
- The `async` and `await` keywords
- How to run futures using Tokio
- Introduction to the Axum web framework

Learning objectives

- Understand the mechanics of `async`/`await` from a high-level point of view
- Understand the reason of Rusts implementation of `async`
- Understand the trade-offs concerning Rust `async` programming
- Understand the mechanics behind `async` and `await`
- Work with `Future`s using the Tokio runtime
- Apply knowledge on `async` Rust in a web context

Module E

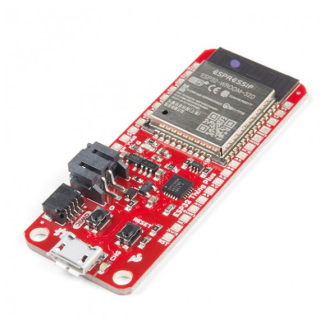
Async and Rust for Web

Content overview

- Rusts `async` implementation
- The `Future` trait
- `async` and `await`
- Running `Future`s with Tokio
- Rust for web with Axum

Async in Rust

Recap: Concurrency vs. Parallelism

Concurrency	Parallelism
Interleaves work	Parallelizes work
1 or more cores	2 or more cores
Waiting for events	Waiting for computation
	

Today, we're focusing on concurrency: *asynchronous programming*

What’s async?

- Concurrent programming model
- Very suitable for running large number of I/O bound tasks
  - like web servers!
- Look and feel* of synchronous code through `async`/`await` syntax

**Well, not perfectly. We’ll get to that*

Async vs OS Threads

	Async	OS Threads
Spawning & switching	Cheap	Expensive
Blocking is ok	No	Yes
Usage	I/O bound tasks (web servers)	CPU-bound tasks (Number crunching)
Reuse sync code	No	Yes

What Color is Your Function?

What `async` looks like in Rust

To get an idea

```rust
/// An async function
async fn run() -> anyhow::Result<()> {
    /// Await loading and parsing config file
    let config = load_config(CONFIG_PATH).await?;
    /// Await scraping job
    let data = scrape(&config.urls).await?;
    data.report();
    Ok(())
}

fn main() {
    // Set up a `tokio` runtime with default configurations
    let runtime = tokio::runtime::Runtime::new().unwrap();
    // Run a Future to completion
    runtime.block_on(run());
}
```

Question: What stands out to you? What strikes you as odd?

Async in Rust

- Revolve around `Future` trait (~like JS `Promise`, C# `Task`)
  → `async fn`s return `Future`s
- `Future`s are inert
- `async` is zero-cost
- No built-in runtime
- Single- or multithreaded execution
- Can be mixed with other concurrency models
- Relatively new and lacks some features and nice diagnostics

State of the `async` art

What you can expect doing `async` Rust

- Blazingly fast applications
- More interaction with advanced language features
- Compatibility issues (re: colored functions)
- Faster evolving ecosystem
- `async fn` in Traits not stable

But still a work in progress

Support of `async`

- Fundamental types and traits are in `std`
- `async`/`await` are native to the language
- Utilities/extensions in `futures` crate
- Async runtimes are third party

Example runtimes: `async-std`, `tokio`, `smol`

The `Future` trait

Foundation of async

A `VerySimpleFuture`

```rust
trait VerySimpleFuture {
    type Output;
    /// Do work and check if task is completed.
    /// Returns [Poll::Ready], containing the
    /// `Output` if task is ready,
    /// [Poll::Pending] if not
    fn poll(&mut self) -> Poll<Self::Output>;
}

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

```rust
struct VerySimpleAlarm {
    alarm_time: Instant,
}

impl VerySimpleFuture for VerySimpleAlarm {
    type Output = ();

    fn poll(&mut self) -> Poll<()> {
        if Instant::now() >= self.alarm_time {
            Poll::Ready(())
        } else {
            Poll::Pending
        }
    }
}
```

Executing `VerySimpleFuture`

```
fn main() {
    let mut first_alarm = VerySimpleAlarm {
        alarm_time: Instant::now()
            + Duration::from_secs(3)
    };
    let mut snooze_alarm = VerySimpleAlarm {
        alarm_time: Instant::now()
            + Duration::from_secs(5)
    };

    loop {
        if let Poll::Ready(_) = first_alarm.poll() {
            println!("Beep beep beep");
        }
        if let Poll::Ready(_) = snooze_alarm.poll() {
            println!("You're late for work!")
        }
    }
}
```

```
[pause...]
Beep beep beep
Beep beep beep
[... a few moments later...]
You're late for work!
Beep beep beep
You're late for work!
Beep beep beep
You're late for work!
[...ad infinitum]
```

It works!

Question: How can `VerySimpleFuture` be improved?

Limitation of `VerySimpleAlarm`

- Busy waiting
- How to signal the executor the future is *actually* ready to be polled?

Introduce a Waker

General idea:

- Run some callback to notify executor
- Have executor implement some job queue

A `SimpleFuture`

```rust
trait SimpleFuture {
    type Output;

    fn poll(&self, wake: fn()) -> Poll<Self::Output>;
}

pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl SimpleFuture for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {             // <-- Does syscall
            Poll::Ready(self.socket.read_buf())
        } else {
            self.socket.set_readable_callback(wake);    // <-- Does syscall
            Poll::Pending
        }
    }
}
```

Adapted from Asynchronous programming in Rust

Joining `SimpleFuture`s

```rust
pub struct Join<FutureA, FutureB> {
    a: Option<FutureA>,
    b: Option<FutureB>,
}

impl<FutureA, FutureB> SimpleFuture
    for Join<FutureA, FutureB>
where
    FutureA: SimpleFuture<Output = ()>,
    FutureB: SimpleFuture<Output = ()>,
{
    type Output = ();
```

```rust
    fn poll(&mut self, wake: fn())
        -> Poll<Self::Output>
    {
        if let Some(a) = &mut self.a {
            if let Poll::Ready(()) = a.poll(wake) {
                self.a.take(); // Drop future A
            }
        }
        if let Some(b) = &mut self.b {
            if let Poll::Ready(()) = b.poll(wake) {
                self.b.take(); // Drop future B
            }
        }
        if self.a.is_none() && self.b.is_none() {
            Poll::Ready(()) // Both futures dropped
        } else {
            Poll::Pending // A future is pending
        }
    }
}
```

Adapted from Asynchronous programming in Rust

And then...

```rust
pub struct AndThenFut<FutureA, FutureB> {
    first: Option<FutureA>,
    second: FutureB,
}

impl<FutureA, FutureB> SimpleFuture for AndThenFut<FutureA, FutureB>
where
    FutureA: SimpleFuture<Output = ()>,
    FutureB: SimpleFuture<Output = ()>,
{
    type Output = ();
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
        if let Some(first) = &mut self.first {
            match first.poll(wake) {
                Poll::Ready(()) => self.first.take(),
                Poll::Pending => return Poll::Pending,
            };
        }
        self.second.poll(wake)
    }
}
```

Adapted from Asynchronous programming in Rust

`SimpleFuture` takeaways

- Composing `SimpleFuture`s requires no heap allocations
- Composing `SimpleFuture`s requires no deeply nested callbacks

The `Future` is now!

```rust
pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

Question: what stands out to you?

- `&mut self` → `Pin<&mut Self>`: makes `Self` immovable
- `wake: fn()` → `cx: &mut Context<'_>`: contains a `Waker`

More on `Pin<&mut Self>` in the Rust async book

`async` and `await`

Expanding `async`

"`Futures` are cool, but why didn't I see them in the web scraper example?"

`async fn`s and `async` blocks are syntactic sugar generating `Future`s

```
async fn foo() -> u8 { 5 }
```

is equivalent to:

```
fn foo() -> impl Future<Output=u8> {
    async {
        5
    }
}
```

which is equivalent to:

```
fn foo() -> impl Future<Output=u8> {
    /// Create a future that is immediately ready
with a value.
    futures::future::ready(5)
}
```

Expanding `async` and `await`

```
let fut_one = /* ... */;
let fut_two = /* ... */;
async move {            // <-- generated Future takes ownership of referenced variables
    fut_one.await;
    fut_two.await;
}
```

Generates an opaque type implementing `Future`:

```
struct AsyncFuture {
    fut_one: FutOne,
    fut_two: FutTwo,
    state: State,
}
enum State {
    AwaitingFutOne,
    AwaitingFutTwo,
    Done,
}
```

This and the following is not the actually generated code, but it's a good mental model

Expanding `async` and `await` (2)

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

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        loop {
            match self.state {
                State::AwaitingFutOne => match self.fut_one.poll(/* - snip - */) {
                    Poll::Ready(()) => self.state = State::AwaitingFutTwo,
                    Poll::Pending => return Poll::Pending,
                }
                State::AwaitingFutTwo => match self.fut_two.poll(/* - snip - */) {
                    Poll::Ready(()) => self.state = State::Done,
                    Poll::Pending => return Poll::Pending,
                }
                State::Done => return Poll::Ready(()),
            }
        }
    }
}
```

Kind of looks like `AndThenFut`!

Adapted from Asynchronous programming in Rust

`async`/`await` expansion takeaways

- Rust generates state machines out of `async` blocks that implement `Future`
- You can `await` `Future`s
- Every `await` point introduces a new state
- Generated code may become very complex, but original is easy to follow

Running `Future`s

What's an `async` Runtime do?

- Spawn `Future`s
- Keep track of pending `Future`s
- Call `Future::poll` on each `Future` that can make progress
- Poll `Future`s on `Waker::wake` calls

Nice to have:

- Poll `Future`s on multiple threads
- Abstract over I/O

Crates depending on different runtime I/O abstractions be incompatible!

Many Runtime flavors

- `smol`: Small
- `async-std`: API resembles `std`
- `tokio`: Bery commonly used
- `embassy`: Embedded
- Create your own?

Note: crates may depend on a specific runtime!

Showcase: Tokio

```
/// Set up a tokio Runtime and spawn the Future returned by `main`
#[tokio::main]
async fn main() {
    do_stuff.await();
}
```

It does stuff!

A simple TCP server

```rust
use tokio::net::{TcpListener, TcpStream};

/// Read a line, and reply with that line!
async fn handle_connection(socket: TcpStream) -> anyhow::Result<()> {
    let mut stream = BufReader::new(socket);
    let mut name = String::new();
    stream.read_line(&mut name).await?;

    stream.write_all(format!("Hello, {name}!").as_bytes()).await?;
    Ok(())
}

#[tokio::main]
async fn main() -> Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();

    loop {
        // The second item contains the IP and port of the new connection.
        let (socket, _) = listener.accept().await.unwrap();
        handle_connection(socket).await?;
    }
}
```

It works!

```
$ echo -e Ferris | nc localhost 6379
Hello Ferris!
```

Question: But does it scale?

Nope! Only one request at a time!

Spawning tasks is cheap!

```
#[tokio::main]
async fn main() -> Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();

    loop {
        // The second item contains the IP and port of the new connection.
        let (socket, _) = listener.accept().await.unwrap();
        handle_connection(socket).await?;
    }
}
```

becomes:

```
async fn main() -> Result<()> {
    loop {
        // The second item contains the IP and port of the new connection.
        let (socket, _) = listener.accept().await.unwrap();
        tokio::task::spawn(async {
            handle_connection(socket).await?;
            Ok::<_, anyhow::Error>(())
        });
    }
}
```

Rust for web

Are we web yet?

- "Yes! And it's freaking fast!"
- Several web frameworks exist
  - `rocket`
  - `actix-web`
  - `warp`
  - `axum`
  - ...lots more
- Several DB drivers and ORMs
- Much more!

Tip: have a look if you want to do web stuff in your final project

Axum demo: setting up server

```rust
use axum::{
    extract::{Path, State},
    response::Html,
    routing::get,
    Router,
};
use std::net::SocketAddr;

#[tokio::main]
async fn main() {
    // set up shared, mutable state.
    let app_state = Arc::new(Mutex::new(Vec::new()));
    // build our application with a route
    let app = Router::new()
        .route("/:name", get(handler))
        .with_state(app_state);
    // run it
    let addr = SocketAddr::from(([127, 0, 0, 1], 3000));
    println!("listening on {}", addr);
    axum::Server::bind(&addr)
        .serve(app.into_make_service())
        .await
        .unwrap();
}
```

Axum demo: request hander

```rust
/// A very long type name warrants a type alias
type AppState = State<Arc<Mutex<Vec<String>>>>;

async fn handler(
    Path(name): Path<String>,
    State(past_names): State<AppState>,
) -> Html<String> {
    let mut response = format!("<h1>Hello, {name}!</h1>");

    // Of course, locking here is not very fast
    let mut past_names = past_names.lock().await;

    if !past_names.is_empty() {
        response += "<h2>Names we saw earlier:</h2>";
        past_names
            .iter()
            .for_each(|name| response += &format!("<p>{name}</p>"))
    }

    past_names.push(name);

    Html(response)
}
```

Summary

- Async in Rust
- The `Future` trait
- `async` and `await` expansion
- Running `Future`s
- Rust for web

Tutorial time!

- Exercises E in 101-rs.tweede.golf
- If you haven't done so yet: project proposal deadline is today!
- Reach out if you want to discuss your proposal or need ideas!
- We'll provide feedback on proposals in coming days

Don't forget to `git pull`!

END

Bonus section: `Pin`

`async` and lifetime elision

`async fn`s which accept references, return a `Future` bound by argument lifetime:

```
async fn foo(x: &u8) -> u8 { *x }
```

is equivalent to:

```
fn foo_expanded<'a>(x: &'a u8) -> impl Future<Output = u8> + 'a {
    async move { *x }
}
```

- `async move` takes ownership of any variables it references
  - i.e. `x`, which itself is a *reference*
- The returned `impl Future` internally holds the references
- The returned `impl Future` must be `await`ed within `'a`

Self-referential structs

Consider:

```
async {
    let mut x = [0; 128];                           // <--
    let read_into_buf_fut = read_into_buf(&mut x);  // <-- Create future
    read_into_buf_fut.await;                        // <-- `await` future
    println!("{:?}", x);                            //
}
```

which becomes

```
struct ReadIntoBuf<'a> {
    buf: &'a mut [u8], // <-- reference to `Async::Future.x` below
}

struct AsyncFuture {
    x: [u8; 128],       // <-- referent
    read_into_buf_fut: ReadIntoBuf<'what_lifetime?>,
}
```

*Question: what happens when `AsyncFuture` is moved?

`Pin<T>`

- Wraps pointer types
- Guarantees values can't be moved (unless `T: Unpin`) using type system

More in Asynchronous Programming in Rust, and docs on `Pin`