A Neophyte's Introduction to the async/await landscape in Rust

The truth of the matter is, the async/await features in other programming languages seem to be more straightforward than it is in Rust. For example in JavaScript, you have the syntax that you use, and that is about it. But in Rust, there seem to be more moving parts.

Tokio? Future? There is a futures crate? Runtimes? How do all these fit into the async/await picture in Rust?

This is a short post to help clarify some of the terms a developer would encounter when they start exploring asynchronous programming in Rust. It would explain the main moving parts, which are: the standard library's `std::future::Future`, the `futures` crates, and asynchronous runtimes like `tokio` and `async-std`.

Let get started:

std::future::Future

The crux of asynchronous programming is to maximize computing resources, especially IO-bound computation. One of the ways this is done, is to code in such a way that the computation is not directly executed but written in a suspended form that can be later executed or paused, or retried.

The type used to express such a computation is `std::future::Future`. For example, a print statement that is not executed directly can be written is as follows:

```rust
use std::future::Future;
fn future_print() -> impl Future<Output = ()> {
    async {
        println!("Hello World!");
    }
}
```

The return type `impl Future<Output = ()>` means this is a function that returns any type that implements the `std::future::Future` trait.

To fully understand the syntax which might be a bit obscure, check the Trait section of the Rust book.

The `std::future::Future` is part of the standard library and there is no need to depend on any external crate to have access to it.

To confirm that the `future_print` function does not execute directly, let us try and call it in `main` to see what happens:

```rust
use std::future::Future;

fn future_print() -> impl Future<Output = ()> {
    async {
        println!("Hello World!");
    }
}

fn main() {
    future_print();
}
```

The output should be something similar to:

```
warning: `async_await` (bin "async_await") generated 2 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/async_await`

Process finished with exit code 0
```

As you can see, there is no *Hello World* printed to the console. Confirming that the `std::future::Future` denotes computation that has been described but that is not executed directly and this is the crux of asynchronous programming in Rust.

In practice, there is usually no need to use the `std::future::Future` type directly as the return type. Rust has a syntactic sugar in the form of the `async` keyword that makes the code less noisy. The above code snippet can be re-written as:

```
async fn future_print() {
    println!("Hello World!");
}

fn main() {
    future_print();
}
```

This is less noisy, even though it is essentially the same as the verbose code that directly uses the `std::future::Future` type.

The question now is, how do a method that returns a future be executed? To answer this we look at the next piece of the async/await story in Rust. The `futures` crate, which can be found here

futures crate.

Formally `futures-rs.` This crate provides the other core foundational components needed for asynchronous programming in Rust.

The components it provides, are not included in the standard library like `std::future::Future`, and hence to bring them in, the crate needs to be defined as a dependency in `Cargo.toml`. It can be done as follows:

```
[dependencies]
futures = "0.3.21" # change to the current version
```

Even though it is not included in the standard library it is worth noting that the `futures-rs` crate is developed by the Rust core team, as can be seen from the fact that the code repository is under the rust-lang namespace on Github

The `futures-rs` crates include key trait definitions like `Stream`, as well as utilities like `join!`, `select!`, and various futures combinator methods that enable expressive asynchronous control flow.

This post won't look at the utilities like `join!`, `select!`, instead we go back to the initial question we are yet to answer: We have a function that returns a `Future` how do we run it?

To run a `Future` in Rust, there is a need for an asynchronous runtime. This runtime is usually called the *executor*. The executor is a machinery that is tasked with the management of all the resources required for executing futures. It takes a computation described as a `Future` and ensures it is executed asynchronously.

The `futures-rs` crate comes with such an executor which can be accessed via `futures::executor::block_on`.

The following code snippet shows its usage:

```
use futures::executor::block_on;

async fn future_print()  {
    println!("Hello world");
}

fn main() {
    block_on(future_print());
}
```

Now if the following code is executed, the output should look like the following:

```
/Users/dadepoaderemi/.cargo/bin/cargo run --color=always --package async_await --bin async_await
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/async_await`
Hello world

Process finished with exit code 0
```

Confirming that we can execute the future code with the usage of `futures::executor::block_on`

At this point, we have looked at `std::future::Future` which is part of the standard library and is used to describe computation to be executed later by a runtime. We also looked at the `futures-rs` crate, which provides such a runtime in the form of the `futures::executor::block_on` executor.

The only problem is that the `futures::executor::block_on` executor is quite primitive and not powerful enough to be used in a real life production environment.

To fill the gap for more powerful, feature-rich asynchronous runtimes, various crates have spurned up within the Rust ecosystem. Some popular ones include: Tokio, async-std, and smol.

They all do the same thing: Provide a sophisticated and more powerful runtime that executes asynchronous computations.

We will use `Tokio` to demonstrate. Update the dependencies as follows:

```
[dependencies]
futures = "0.3.21" # change to the current version
tokio = { version="1.18.2", features = ["full"] } # change to the current version
```

Note that using external runtimes like `tokio` does not necessarily remove the need for the `futures-rs` crate, as it contains useful utilities that can still be used alongside the runtime.

Using the `tokio` runtime would look something like this:

```rust
use tokio::runtime::Runtime;

async fn future_print() {
    println!("Hello world")
}

fn main() {
    Runtime::new().unwrap().block_on(future_print());
}
```

This works. The only problem though is that this is a bit verbose. To make the code less noisy, we use another Rust language keyword related to async/await, the `.await`, and a macro from Tokio: `#[tokio::main]`.

In code, it looks like this:

```rust
async fn future_print() {
    println!("Hello world")
}

#[tokio::main]
async fn main() {
    future_print().await;
}
```

Running the above code would print something like this into the console:

```
/Users/dadepoaderemi/.cargo/bin/cargo run --color=always --package async_await --bin async_await
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
```

```
     Running `target/debug/async_await`
Hello World!
```

Seeing the *Hello World!* confirms that we were able to use a runtime provided by Tokio runtime to execute the computation.

Summary

Using async/await in Rust is not as straightforward as just using the language syntax as it is with languages like JavaScript or C# etc that have similar asynchronous constructs.

More moving parts might not be obvious to the beginner Rust developer. This post gave a quick overview of the crucial components:

The `std::future::Future` type is part of the standard library that forms the crux of the asynchronous programming model as it is used to denote computation that can be executed asynchronously.

The `futures-rs` crates, maintained by the Rust team, provide more of the foundations for asynchronous programming in Rust. It comes with a simple executor that can be used as a runtime to execute asynchronous code.

And finally, we looked at Tokio, as an example of an external crate that provides a more sophisticated asynchronous runtime that is usually used in a production-like environment.

Other well known crates providing asynchronous runtimes include smol, and async-std.

In summary, the core async/await landscape can be described as comprising the standard library type of `std::future::Future` which denotes asynchronous computation. The `futures-rs` crate provides additional foundational components: traits, macros utilities, etc and then finally the 3rd party runtimes that are external to the standard library which provide sophisticated executors used for running asynchronous code.

This post does not attempt to go into details about how to use async/await in Rust. It seeks to just be the introductory text that explains the lay of the land. To learn more about asynchronous programming in Rust, you can check the following resources:

- Crust of Rust: async/await
- Async/await in Rust: Introduction
- Asynchronous Programming in Rust

Posted by dade at Sunday, May 29, 2022

Labels: learning rust, rust