Futures in Rust

Copenhagen Rust Group

Alice Ryhl

May 2019

What is a future?

A future consists of three things:

1. The state of a task.
2. A function that can poll the task.
3. A way to notify when the task is ready to be polled.

What is a future not?

A future is different from many similar concepts.

- Not a Javascript promise.
- Not a thread handle.

A future *must* be polled.

Why are futures hard?

1. Futures are not the same as promises.
2. Futures are like iterators, but there is no analogy to the for loop (yet).
3. Futures only return a single item, so the difference between chaining and mapping is subtle.
4. Futures need a runtime to poll them. A while loop is not good enough.

What is tokio?

Tokio provides the runtime and provides some leaf futures: IO and timers.

The Runtime

Manages a thread pool that polls the futures. Only polls futures that have notified that they are ready.

The Reactor

The Reactor is tokio's solution to:

- Non-blocking file IO is very limited. The `aio_*` functions in the C standard library work by doing synchronous IO in a thread pool.
- A future runs no code when it is not polled. How would a timer notify when the future is ready?

The Future trait

```
pub enum Async<T> { Ready(T), NotReady }
type Poll<T, E> = Result<Async<T>, E>;

pub trait Future {
    type Item;
    type Error;
    fn poll(&mut self) -> Poll<Self::Item, Self::Error>;
}
```

Example

It's time to make some futures.

```
use tokio::fs::read;
use futures::Future;

let future = read("data.txt")
    .map(|vec| vec.len())
    .map(|length| println!("{} bytes.", length))
    .map_err(|err| eprintln!("{}", err));

tokio::run(future);

println!("tokio::run is blocking");
```

Example

Let's look at the example in detail.

```
use tokio::fs::read;
use futures::Future;

let future = read("data.txt")
// ^-- Future<Item = Vec<u8>, Error = std::io::Error>
    .map(|vec| vec.len())
    .map(|length| println!("{} bytes.", length))
    .map_err(|err| eprintln!("{}", err));

tokio::run(future);

println!("tokio::run is blocking");
```

Example

Let's look at the example in detail.

```
use tokio::fs::read;
use futures::Future;

let future = read("data.txt")
    .map(|vec| vec.len())
//    ^-- Future<Item = usize, Error = std::io::Error>
    .map(|length| println!("{} bytes.", length))
    .map_err(|err| eprintln!("{}", err));

tokio::run(future);

println!("tokio::run is blocking");
```

Example

Let's look at the example in detail.

```rust
use tokio::fs::read;
use futures::Future;

let future = read("data.txt")
    .map(|vec| vec.len())
    .map(|vec| println!("{} bytes.", vec.len()))
//    ^-- Future<Item = (), Error = std::io::Error>
    .map_err(|err| eprintln!("{}", err));

tokio::run(future);
//    ^-- Only accepts Future<Item = (), Error = ()>

println!("tokio::run is blocking");
```

Example

Let's look at the example in detail.

```
use tokio::fs::read;
use futures::Future;

let future = read("data.txt")
    .map(|vec| vec.len())
    .map(|vec| println!("{} bytes.", vec.len()))
    .map_err(|err| eprintln!("{}", err));
//    ^-- Future<Item = (), Error = ()>

tokio::run(future);

println!("tokio::run is blocking");
```

Running several futures

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

let future1 = read("data.txt")
    .map(|vec| println!("{} bytes (1).", vec.len()))
    .map_err(|err| eprintln!("{}", err));

let future2 = read("data.txt")
    .map(|vec| println!("{} bytes (2).", vec.len()))
    .map_err(|err| eprintln!("{}", err));

let mut runtime = Runtime::new()?;
runtime.spawn(future1);
runtime.spawn(future2);
println!("runtime.spawn is not blocking");

runtime.shutdown_on_idle().wait().unwrap();
println!("but this is");
```

Chaining futures

A common source of confusion is the difference between `map` and `and_then`.

```
let future = read("data.txt")
    .and_then(|vec| {
        write("data_copy.txt", vec)
    })
    .map(|vec| println!("{} bytes copied.", vec.len()))
    .map_err(|err| eprintln!("{}", err));

tokio::run(future);
```

Chaining futures

A common source of confusion is the difference between `map` and `and_then`.

```
let future = read("data.txt")
    .and_then(|vec| {
        write("data_copy.txt", vec)
//        ^-- This returns a Future!
//            Had we used map, we would end up with a
//            Future<Item=Future<...>, ...>
    })
    .map(|vec| println!("{} bytes copied.", vec.len()))
    .map_err(|err| eprintln!("{}", err));

tokio::run(future);
```

Chaining futures

A common source of confusion is the difference between `map` and `and_then`.

```
let future = read("data.txt")
    .and_then(|vec| {
        write("data_copy.txt", vec)
    })
//  ^-- Future<Item=Vec<u8>>
//      write resolves to the buffer,
//      since it takes ownership of it.
    .map(|vec| println!("{} bytes copied.", vec.len()))
    .map_err(|err| eprintln!("{}", err));

tokio::run(future);
```

The difference between `map` and `and_then`

The difference can be seen in the type definitions. The `map` method is defined as

```
fn map<F, U>(self, f: F) -> Map<Self, F> where
    F: FnOnce(Self::Item) -> U,
```

The `Map` type has the following `Future` impl:

```
impl<U, A, F> Future for Map<A, F> where
    A: Future,
    F: FnOnce(A::Item) -> U
{
    type Item = U
    type Error = A::Error
}
```

The difference between `map` and `and_then`

The difference can be seen in the type definitions. The `and_then` method is defined as

```
fn and_then<F, B>(self, f: F) -> AndThen<Self, B, F> where
    F: FnOnce(Self::Item) -> B,
    B: IntoFuture<Error = Self::Error>,
```

The `AndThen` type has the following `Future` impl:

```
impl<A, B, F> Future for AndThen<A, B, F> where
    A: Future,
    B: IntoFuture<Error = A::Error>,
    F: FnOnce(A::Item) -> B,
{
    type Item = B::Item
    type Error = B::Error
}
```

Join and select

There are two combinators for combining futures.

Join

The `join` combinator creates a future, that resolves once two futures have finished. The result is a tuple of the two results.

Select

The `select` combinator waits on two futures, and returns the result of the one that completes first.

Join example

```rust
let future1 = read("data.txt");
let future2 = read("data2.txt");
let future = future1.join(future2)
    .and_then(|(vec1, vec2)| {
        let mut vec = vec1;
        vec.extend(&vec2[..]);
        write("concat.txt", vec)
    })
    .map(|vec| println!("{} bytes written.", vec.len()))
    .map_err(|err| eprintln!("{}", err));

tokio::run(future);
```

Then

The `then` method deserves extra mention.

- Both `map` and `map_err` in one operation.
- Both `and_then` and `or_else` in one operation.
- Allows using `?` in the closure.

```
fn then<F, B>(self, f: F) -> Then<Self, B, F> where
    F: FnOnce(Result<Self::Item, Self::Error>) -> B,
    B: IntoFuture,
    Self: Sized
```

Notice that `Result` implements `IntoFuture`!

Stream

There is also a `Stream` trait.

```
pub trait Stream {
    type Item;
    type Error;
    fn poll(&mut self) -> Poll<Option<Self::Item>,
                                       Self::Error>;
}
```

Not a fundamental type

A stream is not a fundamental type in the same way `Future` is.

- You can't run a `Stream`.
- Will always be wrapped in a `Future`.
- Will not be in `std`.

Stream combinators

A `Stream` has many combinators similar to the ones on `Future`, and a collection of combinators similar to those on `Iterator`.

- The methods `map`, `and_then`, `map_err`, `or_else` and `then` perform the operation on each element or error.
- The methods `filter`, `chain`, `skip_while`, `take_while`, `zip` and so on perform the same operation as they would on an `Iterator`.
- The methods `for_each`, `fold` and `collect` are the main ways to turn a `Stream` into a `Future`.
- Two ways to merge: `select` and `merge`.

Flattening, `and_then` and `Streams`

If you want to turn a `Future` into a `Stream` with some sort of mapping, you can use `map` followed by `flatten_stream`.

This is common when using the hyper crate:

- A connection in hyper starts with a `ResponseFuture`.
- The future resolves to the headers and a stream of `Chunks`.

`and_then` and `flatten`

The `and_then` method could also be replaced by a `map` followed by a `flatten`, but you will probably never need to do this.

A stream of streams can be flattened using the `flatten` stream combinator.

Creating futures manually

Instead of putting futures together, you can manually implement `Future` on your own types.

We will look at an example of how to do this. The example future will collect data from an internet connection, and pass it to serde to decode the received json.

The future of futures

A rebuild of the future system is being worked on. It involves some changes:

- The `Future` trait will be moved into the standard library.
- A new async fn feature is added.
- The waker system is reworked.

The new Future trait

Added to the standard library:

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

Changes:

- No error type!
- `Pin` around `self`.
- A `Context` variable (contains the waker).
- Combinators not provided by `std`.

The async fn feature

The new async fn feature let's the compiler turn your imperative code into a state machine.

```
async fn get_url<T>(url: &str) -> Result<T> {
    let response_future = /* ... */;
    let (parts, body) = response_future.await?;
    let mut vec = Vec::with_capacity(parts.content_length);

    for chunk in body.await? {
        vec.extend(&chunk[..]);
    }

    serde_json::from_slice(&vec[..])?
}
```