The async Rust ecosystem

Florian “Florob” Zeitz

2019-12-04

Motivation

- some applications require lots of concurrent I/O
- (web-)servers sometimes 100s to 1000s blocking operations
- creating a thread per operation uses up resources

$\Rightarrow$ ideally we would handle multiple operations on one thread; after all their concurrent not parallel

Blocking Operations

Blocking Operation

A blocking operation is one that when waiting for some event, such as a resource becoming available or the completion of an I/O operation, halts the thread it's executing on.

- most I/O operations are blocking by default
- a blocked thread is not scheduled by the kernel
- blocked threads still take up memory though

Non-Blocking Operations

Non-Blocking Opeation

A non-blocking operation is one that when it would halt a thread returns with an appropriate error instead.

- some I/O operations can be set to non-blocking mode
- one could repeatedly attempt an operation testing whether it can progress
- some (blocking) APIs exist to check whether an operation can be performed on one of several resources

Async(hornous) Operations

Asynchronous Operation

An asynchronous operation is one that executes independently of the main process.

- often build around an event loop
- operations are enqueued, events fired upon completion

⇒ we want an abstraction for operations that finish sometime in the `Future`

Simplified Future API


```
1  trait Future {
2      type Output;
3      fn poll(&mut self) -> Poll<Self::Output>;
4  }
5
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```

- async operations are modeled by the `Future` trait
- `poll` returns `Poll::Pending` while the operation is still in progress
- `poll` returns `Poll::Ready(T)` when the operation has finished

Why is this in libstd?

- started out in `futures-rs`, with more provided functions
- having it in libstd standardizes the trait across the ecosystem
- language features (**`async`**/**`.await`**) can rely on it

Executors

- `Future`s don't start executing upon creation (i.e. are lazy)
- execution starts upon the first call to `poll()`
- the component calling `poll` to drive `Future`s to completion is know as an *executor*
- executors don't have to actually poll the `Future`, instead `Future`s are required to notify the executor when progress can be made

Example: async-std's Executor


```
1 let stream_fut = TcpStream::connect(("koeln.ccc.de", 23u16));
2 let mut stream = task::block_on(stream_fut).unwrap();
3 let mut buffer = [0u8; 1024];
4 let len = task::block_on(stream.read(&mut buffer)).unwrap();
5 println!("{:?}", std::str::from_utf8(&buffer[..len]));
```

- `task::block_on()` executes a future to completion in a blocking manner
- there is also `task::spawn()` to execute features in the background
- obviously starting a single feature on an executor is not useful, we need *composition*

Composition

- `Future`s have to be composable
- complete A, then B based on A's result
- complete A and B, return both results
- complete A or B, return the first one that completes
- ...

A then B: Manually?


```
1  enum AthenB<A, B, F> {
2      First(A, Option<F>),
3      Second(B),
4  }
```

A then B: Manually?


```
5  impl<A, B, F> Future for AthenB<A, B, F>
6      where A: Future, B: Future, F: FnOnce(A::Output) -> B
7  {
8      type Output = B::Output;
9      fn poll(&mut self) -> Poll<B::Output> {
10         loop {
11             match self {
12                 AthenB::First(a, f) => match a.poll() {
13                     Poll::Pending => return Poll::Pending,
14                     Poll::Ready(r) => {
15                         let f = f.take().unwrap();
16                         *self = Second(f(r));
17                     }
18                 }
```

A then B: Manually?


```
18                          AthenB::Second(b) => return b.poll(),
19                      }
20                  }
21              }
22          }
```

- this is *pseudocode* (e.g. still using the simplified interface)
- actual implementation looks similar though
- quite bothersome to write

A then B: FutureExt::then()


```
1 trait FutureExt {
2     fn then<Fut, F>(self, f: F) -> Then<Self, Fut, F> where
3         F: FnOnce(Self::Output) -> Fut,
4         Fut: Future,
5 }
```

- part of `futures-rs`
- convenient API for very simple cases
- often hard/impossible to use due to borrowing

A then B: FutureExt::then()


```
1 let mut buffer = [0u8; 1024];
2 let stream_fut = TcpStream::connect(("koeln.ccc.de", 23u16));
3 let read_fut = stream_fut.then(|stream| {
4     stream.unwrap().read(&mut buffer)
5 }); // Error: neither buffer nor stream live long enough
6 let len = task::block_on(read_fut).unwrap();
7 println!("{:?}", std::str::from_utf8(&buffer[..len]));
```

A then B: async blocks


```
let mut buffer = [0u8; 1024];
let data_fut = async {
    let mut stream = TcpStream::connect(
        ("koeln.ccc.de", 23u16)
    ).await.unwrap();
    let len = stream.read(&mut buffer).await.unwrap();
    std::str::from_utf8(&buffer[..len])
};
let data = task::block_on(data_fut);
println!("{:?}", data);
```

- `async` blocks create anonymous `Future`s
- `.await` resolves a `Future` within the block by "bubbling up" any `Poll::Pending` results

A then B: async fn


```
async fn get_data() -> String {
    let mut buffer = vec![0u8; 1024];
    let mut stream = TcpStream::connect(
        ("koeln.ccc.de", 23u16)
    ).await.unwrap();
    let len = stream.read(&mut buffer).await.unwrap();
    buffer.truncate(len);
    String::from_utf8(buffer).unwrap_or(String::new())
}

let data = task::block_on(get_data());
println!("{}", data);
```

A then B: async fn (desugared)


```rust
fn get_data() -> impl Future<Output=String> {
    async {
        let mut buffer = vec![0u8; 1024];
        let mut stream = TcpStream::connect(
            ("koeln.ccc.de", 23u16)
        ).await.unwrap();
        let len = stream.read(&mut buffer).await.unwrap();
        buffer.truncate(len);
        String::from_utf8(buffer).unwrap_or(String::new())
    }
}
```

A and B: future::join()


```
1  use futures::future;
2  async fn get_data(port: u16) -> String {
3      let mut buffer = vec![0u8; 1024];
4      let mut stream = TcpStream::connect(
5          ("koeln.ccc.de", port)
6      ).await.unwrap();
7      let len = stream.read(&mut buffer).await.unwrap();
8      buffer.truncate(len);
9      String::from_utf8(buffer).unwrap_or(String::new())
10 }
11
12 let (real, nick) = task::block_on(
13     future::join(get_data(23), get_data(31337))
14 );
15 println!("{}", real);
16 println!("{}", nick);
```

A and B: join!()


```
use futures::join;
async fn get_data(port: u16) -> String {
    let mut buffer = vec![0u8; 1024];
    let mut stream = TcpStream::connect(
        ("koeln.ccc.de", port)
    ).await.unwrap();
    let len = stream.read(&mut buffer).await.unwrap();
    buffer.truncate(len);
    String::from_utf8(buffer).unwrap_or(String::new())
}

let (real, nick) = task::block_on(async {
    join!(get_data(23), get_data(31337))
});
println!("{}", real);
println!("{}", nick);
```

Real Future API


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

- `Pin` makes a type immovable
- pinned types can be self referential (e.g. referencing a buffer in a future)
- `Context` (for now) is only used to get a `Waker`

Waker

- once a `Future` returns `Poll::Pending` an executor does not have to automatically call `poll()` again
- `poll()` is only guaranteed to be called again once `Waker::wake()` is invoked, a `Future` has to arrange for this
- most efficient way to achieve this is often integration with the executor
- the waking mechanism is executor specific, i.e. it provides a function table to be used by the `Waker`

futures-rs

- crate the `Future` trait originally was developed in
- staring with version 0.3 re-exports `Future` from `libstd`
- still home of many useful extension traits, functions and macros

Tokio


```rust
async fn say_hello() {
    println!("Hello, world!");
}

#[async_std::main]
async fn main() {
    say_hello().await;
}
```

- one of the first Rust async libraries
- its `Futures` only work when run under the Tokio runtime
- runtime can't be started independently of the executor

async-std

- one-to-one port of libstd to async I/O
- replacing `std` with `async_std` should (almost) just work
- Waker runtime independent of executor
- `[async_std::main]` attribute to make `main` an **`async fn`**

hyper

- *the* HTTP crate for Rust
- default I/O and runtime is based on tokio
- provides support for implementing HTTP clients and servers
- somewhat higher level wrappers like `reqwest` exist

The Future

- ecosystem relying more on stable `Future` trait
- `Stream` trait: similar to `Future`, but continuously producing values
- **`async fn`**s in traits

Resources

- Async Book `https://rust-lang.github.io/`
- async-std Book `https://book.async.rs/`
- `https://docs.rs/futures`

Thank you for your attention. Any questions?

https://babelmonkeys.de/~florob/talks/RC-2019-12-04-rust-async.pdf