Tokio: the future of asynchronous Rust

Dirkjan Ochtman

About me

 **@djc**  **@djco**

- https://dirkjan.ochtman.nl/
- Rust projects
  - Askama: type-safe compiled Jinja/Twig-like templates for Rust
  - Quinn: tokio-based QUIC implementation in Rust
  - tokio-imap & imap-proto: IMAP protocol client libraries
- Currently in between jobs. Would love to do some Rust freelancing (development, team kickstart, training)!

Started Rust 2 years ago.

Agenda

- Event-driven programming
- Rust
- Futures
- tokio
- tokio-codec
- async/await

Event-driven programming

- Frameworks
  - libuv
  - Node.js
- "Callback hell"
- Promises
- Green threads vs OS threads

Rust

- Three years since 1.0; new releases every six weeks
- 2018 **edition** coming in 5 weeks
  - async/await coming in 2019
- Three distinct groups of developers:
  - C/C++
  - Python/Ruby/JavaScript
  - Ocaml/Haskell/Clojure
- Not included: event-driven programming or async I/O

The code is a lie

Assorted code snippets are from alpha versions of the relevant libraries; current versions are different!

Future

"A future is a value that may not have finished computing yet. This kind of "asynchronous value" makes it possible for a thread to continue doing useful work while it waits for the value to become available."

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

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

Stream

"If `Future<Output = T>` is an asynchronous version of `T`, then `Stream<Item = T>` is an asynchronous version of `Iterator<Item = T>`. A stream represents a sequence of value-producing events that occur asynchronously to the caller."

```
trait Stream {
    type Item;
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Option<Self::Item>>;
}
```

Sink

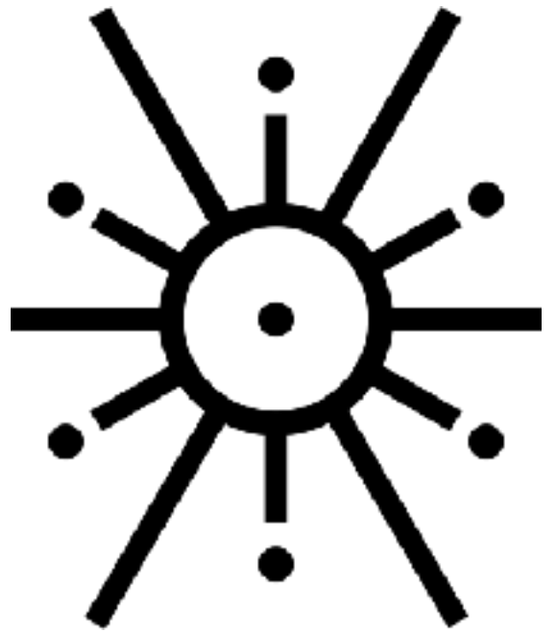
"A `Sink` is a value into which other values can be sent, asynchronously."

```
trait Sink {
    type SinkItem;
    type SinkError;
    fn poll_ready(self: [..], cx: [..]) -> Poll<Result<(), Self::SinkError>>;
    fn start_send(self: [..], item: Self::SinkItem) -> Poll<Result<[..]>>;
    fn poll_flush(self: [..], cx: [..]) -> Poll<Result<[..]>>;
    fn poll_close(self: [..], cx: [..]) -> Poll<Result<[..]>>;
}
```

tokio (1)

A runtime for writing reliable, asynchronous, and slim applications with the Rust programming language.

- Fast
- Reliable
- Lightweight



Fast: zero-cost abstractions, concurrency, non-blocking I/O. Reliable: ownership and type system, back pressure, cancellation. Lightweight: no garbage collector, modular.

tokio (2)

- A multithreaded, work-stealing based task scheduler
- A reactor backed by the operating system's event queue
- Asynchronous TCP and UDP sockets

Event queue: epoll (Linux), kqueue (BSD, including macOS), IOCP (Windows)

tokio (3)

- tokio-codec: Encoding and decoding protocol frames
- tokio-current-thread: Schedule futures on current thread
- tokio-executor: Task execution related traits and utilities
- tokio-fs: Filesystem (and standard in / out) APIs
- tokio-io: Asynchronous I/O related traits and utilities
- tokio-reactor: Event loop that drives I/O resources
- tokio-tcp: TCP bindings
- tokio-threadpool: Schedules futures on a thread pool
- tokio-timer: Time related APIs
- tokio-udp: UDP bindings
- tokio-uds: Unix Domain Socket bindings

tokio-codec (1)

- Utilities for encoding and decoding protocol frames
- Importantly, `Framed`
  - Adapters to go from streams of bytes to framed streams implementing `Sink` and `Stream`

tokio-codec (2)

```
struct Framed<T, U>;

trait Encoder {
    type Item;
    type Error;
    fn encode(&mut self, item: Self::Item, dst: &mut BytesMut)
        -> Result<(), Self::Error>;
}

trait Decoder {
    type Item;
    type Error;
    fn decode(&mut self, src: &mut BytesMut)
        -> Result<Option<Self::Item>, Self::Error>;
    ...
}

trait AsyncRead: Read {
    fn framed<T: Encoder + Decoder>(self, codec: T)
        -> Framed<Self, T> { ...}
    ...
}
```

async & await

- Specification in RFC 2394

```
async fn foo(arg: &str) -> usize { ... }
```

becomes

```
fn foo<'a>(arg: &'a str) -> impl Future<Output = usize> + 'a { ... }
```

await!() expands into

```
let mut future = IntoFuture::into_future($expression);
let mut pin = unsafe { Pin::new_unchecked(&mut future) };
loop {
    match Future::poll(Pin::borrow(&mut pin), &mut ctx) {
        Poll::Ready(item) => break item,
        Poll::Pending => yield,
    }
}
```

Minimal tokio server

```
extern crate tokio;
use tokio::io;
use tokio::net::TcpListener;
use tokio::prelude::*;

pub fn main() {
    let addr = "127.0.0.1:6142".parse().unwrap();
    let listener = TcpListener::bind(&addr).unwrap();
    let server = listener.incoming().for_each(|socket| {
        println!("accepted socket; addr={:?}", socket.peer_addr().unwrap());
        let connection = io::write_all(socket, "hello world\n")
            .then(|res| {
                println!("wrote message; success={:?}", res.is_ok());
                Ok(())
            });
        tokio::spawn(connection);
        Ok(())
    })
    .map_err(|err| {
        println!("accept error = {:?}", err);
    });
    println!("server running on localhost:6142");
    tokio::run(server);
}
```

Questions left to the end

- Are there any?