async/.await with async-std

Florian Gilcher

RustFest Barcelona

CEO and Rust Trainer
Ferrous Systems GmbH

Whoami

- Florian Gilcher
- https://twitter.com/argorak
- https://github.com/skade
- MD https://asquera.de, https://ferrous-systems.com
- Rust Programmer and Trainer: https://rust-experts.com
- Rustacean since 2013, team member since 2015

The async-rs/async-std project

async-std is a port of the Rusts std library into the async world.

It comes with its own executor and is based on *futures-rs*.

async-std is not new, it is the summary of 3 years of experience.

https://async.rs

Who?

async-std is was kicked off by Stjepan Glavina (Crossbeam, tokio), with Yoshua Wuyts (tide, surf) and me joining in early.

It is now developed by a global team.

Why?

- Stability: the Rust async ecosystem has been in flux for too long
- Ergonomics: should be and consistent to be used
- Accessibility: Comes with a book and *full* API docs
- Integration: fully integrates with the Rust ecosystem, most importantly futures-rs
- Speed: speed should come out of the box

Why?

- Stability: the Rust async ecosystem has been in flux for too long
- Ergonomics: should be and consistent to be used
- Accessibility: Comes with a book and *full* API docs
- Integration: fully integrates with the Rust ecosystem, most importantly futures-rs
- Speed: speed should come out of the box

The best library to get started with async/await.

Additional properties

- Small dependency tree
- Not overly generic
- Compiles fast

Synchronous functions

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_file(path: &str) -> io::Result<String> {
    let mut file = File::open(path)?;
    let mut buffer = String::new();
    file.read_to_string(&mut buffer)?;
    Ok(buffer)
}
```

Asynchronous functions

```
use async_std::fs::File;
use async_std::prelude::*;
use async_std::io;

async fn read_file(path: &str) -> io::Result<String> {
    let mut file = File::open(path).await?;
    let mut buffer = String::new();
    file.read_to_string(&mut buffer).await?;
    Ok(buffer)
}
```

User quote

We used async-std internally. We just replaced ”std” by ”async-std” and added ”async”/”await” at the right places.
– Pascal Hertleif (killercup)

async-std exports all types necessary for async programming, including re-exports of *std* library types.

If an *async-std* type exists, you should use that one over *std*.

Fun fact: did you know *std::path::Path* has functions that block?

Asynchronous functions

Rough desugar of the *async* keyword:

```
use async_std::fs::File;
use async_std::prelude::*;
use async_std::io;

fn read_file(path: &str) -> impl Future<Item=io::Result<String>> {
    let mut file = File::open(path).await?;
    let mut buffer = String::new();
    file.read_to_string(&mut buffer).await?;
    Ok(buffer)
}
```

What is .await?

```rust
use async_std::fs::File;
use async_std::prelude::*;
use async_std::io;

async fn read_file(path: &str) -> io::Result<String> {
    let mut file = File::open(path).await?;
    let mut buffer = String::new();
    file.read_to_string(&mut buffer).await?;
    Ok(buffer)
}
```

- .await marks points where we *wait for completion*.

Asynchronous functions

```rust
fn main() {
    let data = read_file("./Cargo.toml");
             //^^^^^^^^^^^^^^^^^^^^^^^^^
             //futures do nothing unless you `.await` or poll them
}
```

- Async functions generate futures when called.

How do we run our code?

Futures run using a *task*. There's multiple ways to get a task:

- blocking
- non-blocking
- blocking in the background

Multiple futures in one task run concurrently, tasks may run in parallel.

Concurrent vs parallel

- Concurrent: multiple processes run in a group, yielding to each other when they need to wait
- Parallel: multiple processes run next to each other, at the same time.

Blocking

Blocking is not a sharply definied term.

For the purpose of this presentation: if something *blocks*, it *blocks* the current parallel thread, blocking all other concurrent *tasks* on it.

block_on

```
use async_std::fs::File;
use async_std::prelude::*;
use async_std::io;
use async_std::task;

fn main() -> io::Result<()> {
    let contents = task::block_on(async {
        let mut file = File::open("Cargo.toml").await?;
        let mut buffer = String::new();
        file.read_to_string(&mut buffer).await?;
        Ok(buffer)
    });

    println!("{}", contents?);
}
```

This blocks the main thread, executes the future and wait for it to come back.

spawn

```rust
use async_std::task;

fn main() -> io::Result<()> {
    let task: JoinHandle<_> = task::spawn(async {
        let mut file = File::open("Cargo.toml").await?;
        let mut buffer = String::new();
        file.read_to_string(&mut buffer).await?;
        Ok(buffer)
    });

    task::block_on(async {
       println!("{}", task.await?);
    });
    Ok(())
}
```

This runs a background *task* and then waits for its completion, blocking the main thread.

JoinHandle

- JoinHandles function similar to *std::thread::JoinHandle*
- They are allocated in one go with the task they spawn
- They provide an easy future-based backchannel to the spawner
- JoinHandles resolve when the task completes

spawn_blocking

```rust
use async_std::task;

fn main() -> io::Result<()> {
    let task: JoinHandle<_> = task::spawn_blocking(async {
        let mut file = File::open("Cargo.toml");
        let mut buffer = String::new();
        file.read_to_string(&mut buffer);
        Ok(buffer)
    });

    task::block_on(async {
       println!("{}", task.await?);
    });

    Ok(())
}
```

The returned *JoinHandle* is exactly the same as for a blocking task.

spawn and spawn_blocking

```rust
fn main() {
    task::block_on(async {
        let mut tasks: Vec<task::JoinHandle<()>> = vec![];
        let task = task::spawn(async {
            task::sleep(Duration::from_millis(1000)).await;
        });
        let blocking = task::spawn_blocking(|| {
            thread::sleep(Duration::from_millis(1000));
        });
        tasks.push(task);
        tasks.push(blocking);
        for task in tasks {
            task.await
        }
    });
}
```

Async patters

- racing: 2 Futures are executed, we’re only interested in the first
- joining: 2 Futures are executed, we’re interested in the result of both

Racing

```rust
use async_std::task;
use async_std::prelude::*;
use surf::get;

type Error = Box<dyn std::error::Error + Send + Sync + 'static>;

async fn get(url: &str) -> Result<String, Error> {
    let mut res = surf::get(url).await?;
    Ok(res.body_string().await?)
}
fn main() -> Result<(), Error> {
    let first = async { get("https://mirror1.example.com/").await? };
    let second = async { get("https://mirror2.example.com/").await? };

    task::block_on(async {
        let data = first.race(second).await;
    });
}
```

Racing

```
fn main() -> Result<(), Error> {
    let first = async { get("https://mirror1.example.com/").await? };
    let second = async { get("https://mirror2.example.com/").await? };
    let first_handle = task::spawn(first);
    let second_handle = task::spawn(second);

    task::block_on(async {
        let data = first_handle.race(second_handle).await;
    });
}
```

Joining

```rust
use async_std::task;
use async_std::prelude::*;
use async_std::futures::join;
use surf::get;
fn main() -> Result<(), Error> {
    let first = async { get("https://mirror1.example.com/").await? };
    let second = async { get("https://mirror2.example.com/").await? };

    task::block_on(async {
        let (res1, res2) = join!(first, second).await;
        //..
    });
}
```

futures-rs also provides *join_all*, joining multiple futures

Streams are a fundamental abstraction around items arriving concurrently.

- In async-std, they take the place of Iterator
- They can be split, merged, iterated over

Example TCPListener

```
fn main() -> io::Result<()> {
    task::block_on(async {
        let listener = TcpListener::bind("127.0.0.1:8080").await?;
        println!("Listening on {}", listener.local_addr()?);

        let mut incoming = listener.incoming();

        while let Some(stream) = incoming.next().await {
            let stream = stream?;
            task::spawn(async {
                process(stream).await.unwrap();
            });
        }
        Ok(())
    })
}
```

Stream merging

```rust
fn main() -> io::Result<()> {
    task::block_on(async {
        let ipv4_listener = TcpListener::bind("127.0.0.1:8080").await?;
        let ipv6_listener = TcpListener::bind("[::1]:8080").await?;

        let ipv4_incoming = ipv4_listener.incoming();
        let ipv6_incoming = ipv6_listener.incoming();

        let mut incoming = ipv4_incoming.merge(ipv6_incoming);

        while let Some(stream) = incoming.next().await {
            let stream = stream?;
            task::spawn(async {
                process(stream).await.unwrap();
            });
        }
        Ok(())
    })
}
```

The sync module

- Comes with async-await ready versions of stdlib structures
- Mutex, Barrier, RwLock…

Mutex example

```rust
use async_std::sync::{Arc,Mutex};

let m = Arc::new(Mutex::new(0));
let mut tasks = vec![];

for _ in 0..10 {
    let m = m.clone();
    tasks.push(task::spawn(async move {
        *m.lock().await += 1;
    }));
}

for t in tasks {
    t.await;
}
assert_eq!(*m.lock().await, 10);
```

Futures-aware mutexes don’t block the thread, only yield the task and notify.

Channels

async-std channels are based on crossbeam channel:

- Multiple Producer, Multiple Consumer
- Always bounded
- Fast (faster than crossbeam-channels, the ones used in Servo)

Should cover all your generic use-cases.

Note: channels are currently unstable for API discussions.

Channels

```
use async_std::task;
use async_std::prelude::*;
use async_std::sync::channel;

struct Message;

fn main() {
    let (ping_send, ping_recv) = channel::<Message>(1);
    let (pong_send, pong_recv) = channel::<Message>(1);

    let node1 = async {
        while let Some(msg) = pong_recv.next().await {
            ping_send.send(Message).await
        }
    };
    let node2 = async {
        while let Some(msg) = ping_recv.next().await {
            pong_send.send(Message).await
        }
    };
```

Channels

```
task::block_on(async {
    let ping = task::spawn(node1);
    let pong = task::spawn(node2);
    ping.await;
    pong.await;
});
```

A piece of wisdom

Understanding *tasks* and *streams* is more important then understanding futures.

async-std provides the known and familiar interface of the Rust standard library with appropriate changes for async.

It avoids pitfalls by providing a full API surface around all async-critical modules.

Fully based on futures-rs

async-std integrates into the ecosystem very well!

- We full embrace the futures-rs library
- All types expose the relevant interfaces from futures-rs
- Not all, but the ones that are generally considered stable
- Others can be used through *use futures*
- Stream, AsyncRead, AsyncWrite, AsyncSeek

AsyncRead/Write/Seek

- AsyncRead: Read from a socket, asynchronously
- AsyncWrite: Write to a socket, asynchronously
- AsyncSeek: Write to a socket, asynchronously

tokio does implement (and change) their own, making them incompatible with the rest of the ecosystem.

Using async-std

- *applications* should use async-std directly
- *libraries* should use *futures-rs* as their interface
- Example: see *async-rs/async-tls*

Example

```
fn read_from_tcp(socket: async_std::net::TcpSocket) {
    // for applications
}

fn read_from_async<S>(sock: S)
    where
        S: futures::io::AsyncRead + Unpin {
    // for libraries
}
```

Lesser known executors

- Google Fuchsia
- bastion.rs
- wasm-bindgen-futures
- Some companies internal ones

async-std is meant for writing compatible libraries.

Speed

Sooooooooo. Benchmarks?

Preface

We believe there is a hyperfocus on benchmarks in the Rust community, at the cost of ergonmics and barring stabilisation.

Benchmarks are also often changing and we don't want to take part in a benchmark race.

Don't choose software by benchmarks alone!

File reading

```
Reading a 256K file:

tokio:     0.136 sec
async_std: 0.086 sec
```

https://github.com/jebrosen/async-file-benchmark

Benchmarks: Mutex creation

```
async_std::sync::Mutex:
test create         ... bench:           4 ns/iter (+/- 0)
futures_intrusive::sync::Mutex (default features, is_fair=true)
test create         ... bench:           8 ns/iter (+/- 0)
tokio::sync::Mutex:
test create         ... bench:          24 ns/iter (+/- 6)
futures::lock::Mutex:
test create         ... bench:          38 ns/iter (+/- 1)
```

Benchmarks: Mutex under contention

```
async_std::sync::Mutex:

test contention    ... bench:     893,650 ns/iter (+/- 44,336)

futures_intrusive::sync::Mutex (default features, is_fair=true)

test contention    ... bench:   1,968,689 ns/iter (+/- 303,900)

tokio::sync::Mutex:

test contention    ... bench:   2,614,997 ns/iter (+/- 167,533)

futures::lock::Mutex:

test contention    ... bench:   1,747,920 ns/iter (+/- 149,184)
```

Benchmarks: Mutex without contention

```
async_std::sync::Mutex:

test no_contention ... bench:     386,525 ns/iter (+/- 368,903)

futures_intrusive::sync::Mutex (default features, is_fair=true)

test no_contention ... bench:     431,264 ns/iter (+/- 423,020)

tokio::sync::Mutex:

test no_contention ... bench:     516,801 ns/iter (+/- 139,907)

futures::lock::Mutex:

test no_contention ... bench:     315,463 ns/iter (+/- 280,223)
```

Benchmarks: Tasks

name	tokio.txt ns/iter	async_std.txt ns/iter	speedup
chained_spawn	123,921	119,706	x 1.04
ping_pong	401,712	289,069	x 1.39
spawn_many	5,326,354	3,149,276	x 1.69
yield_many	7,640,958	3,919,748	x 1.95

(This is based on *Tokio 10x benchmarks*)

Channel ring benchmark

Send 1 message around a ring of n nodes, m times. Thanks, Joe!

- 0.9x slower compared to tokio
- 3x faster compared actix

Notice

For risks and side-effects of synthetic benchmarks, please consult your local Apple keynoter.

async-std is a fast, ergonomic, *futures-rs* base layer for asynchronous applications that.

An innovation space

- *JoinHandle*s were built in async-std and already adopted by others
- *single allocation task*s were invented in async-std and adopted by others

You can both innovate and commit to stability!

Roadmap

- 1.0 on Monday: stable release with all base functionality and runtime concerns
- ongoing: stabilisation of currently unstable library API
- ongoing: designing features that make async-std usable without the runtime
- provide additional libraries with similar guarantees
- 2.0: when new language features arrive or futures breakes base crates. We will provide update guides.

Let’s hack!

- Get started writing libraries on top!
- Challenge our benchmarks!
- Get started writing an application!
- Give opinions on our unstable API!

async-std is currently completely funded by Ferrous Systems.

https://opencollective.com/async-rs/

https://async.rs

Thank you!

- https://twitter.com/argorak
- https://github.com/skade
- https://speakerdeck.com/skade
- florian.gilcher@ferrous-systems.com
- https://ferrous-systems.com
- https://rust-experts.com