

Getting comfy with async await

Sanchayan Maity

Who

- ▶ Who am I?
 - ▶ Embedded Systems background
 - ▶ Prefer C, Haskell and Rust
 - ▶ Organize and speak at Rust and Haskell meet-ups in Bangalore
- ▶ Work?
 - ▶ Software Engineer @ **asymptotic**
 - ▶ Open source consulting firm based out of Bangalore and Toronto
 - ▶ Work on low level systems software centred around multimedia
 - ▶ GStreamer, PipeWire, PulseAudio
 - ▶ Language Polyglots

Agenda

- ▶ Future trait
- ▶ `async/await`
- ▶ Using futures/Runtime
- ▶ Working with multiple futures (`select`, `join`, `FuturesOrdered`)
- ▶ Streams
- ▶ Pitfalls
- ▶ `Pin/Unpin/pin_project`

Future¹

```
use std::future::Future;
use std::pin::Pin;
use std::task::Context;

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

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

¹Associated types

Example

```
async fn hello() {  
    println!("Hello from async");  
}
```

```
fn main() {  
    hello();  
    println!("Hello from main");  
}
```

Where's the future

```
async fn give_number() -> u32 {  
    100  
}
```

Sugar town²

```
fn give_number() -> impl Future<Output = u32> {
    GiveNumberFuture
}

struct GiveNumberFuture {}

impl Future for GiveNumberFuture {
    type Output = u32;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Self::Output> {
        Poll::Ready(100)
    }
}
```

²Syntactic sugar for Future

Runtimes



Runtimes³

- ▶ `futures::executor`
- ▶ `tokio`
- ▶ `smol-rs`
- ▶ `embassy`
- ▶ `glommio`
- ▶ `async-std`

³The state of Async Rust: Runtimes

Example

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

async fn hello() {
    println!("hello, world!");
}

fn main() {
    block_on(hello());
    println!("Hello from main");
}
```

Example

```
async fn hello() {  
    println!("Hello from async");  
}
```

```
#[tokio::main]  
async fn main() {  
    hello().await;  
    println!("Hello from main");  
}
```

Multiple futures

- ▶ `join`
- ▶ `join_all`
- ▶ `select`
- ▶ `select!`
- ▶ `select_all`
- ▶ `FuturesOrdered`
- ▶ `FuturesUnordered`
- ▶ `JoinSet`

join

```
use futures::future;

#[tokio::main]
async fn main() {
    let a = async { "Future 1" };
    let b = async { "Future 2" };
    let pair = future::join(a, b);

    println!("{:?}", pair.await);
}
```

join_all

```
use futures::future::join_all;
async fn hello(msg: String) -> String {
    msg
}

#[tokio::main]
async fn main() {
    let futures = vec![
        hello("Future 1".to_string()),
        hello("Future 2".to_string()),
        hello("Future 3".to_string()),
        hello("Future 4".to_string()),
    ];

    println!("{:?}", join_all(futures).await);
}
```

JoinSet

```
use tokio::task::JoinSet;

#[tokio::main]
async fn main() {
    let mut set = JoinSet::new();
    for i in 0..10 {
        set.spawn(async move { i });
    }

    while let Some(res) = set.join_next().await {
        println!("{}", res.unwrap());
    }
}
```

future::select

```
pub fn select<A, B>(future1: A, future2: B) -> Select<A, B>  
where  
    A: Future + Unpin,  
    B: Future + Unpin,
```


future::select

```
use futures::{future, future::Either, future::FutureExt, select};
use tokio::time::{sleep, Duration};

async fn task1(delay: u64) -> u64 {
    sleep(Duration::from_millis(delay)).await;
    delay
}

async fn task2(delay: u64) -> String {
    sleep(Duration::from_millis(delay)).await;
    "Hello".to_string()
}
```

future::select

```
#[tokio::main]
async fn main() {
    let t1 = task1(200u64).fuse();
    let t2 = task2(300u64).fuse();

    tokio::pin!(t1, t2);

    match future::select(t1, t2).await {
        Either::Left((value1, _)) => println!("{}", value1),
        Either::Right((value2, _)) => println!("{}", value2),
    };
}
```

futures::select!⁴

```
use futures::{future::FutureExt, pin_mut, select};
use tokio::time::{sleep, Duration};
async fn task(delay: u64) {
    sleep(Duration::from_millis(delay)).await;
}

#[tokio::main]
async fn main() {
    let t1 = task(300u64).fuse();
    let t2 = task(200u64).fuse();
    pin_mut!(t1, t2);
    select! {
        () = t1 => println!("task one completed first"),
        () = t2 => println!("task two completed first"),
    }
}
```

⁴futures::select!

```
tokio::select!5
```

```
use tokio::time::{sleep, Duration};
async fn task(delay: u64) {
    sleep(Duration::from_millis(delay)).await;
}

#[tokio::main]
async fn main() {
    let t1 = task(300u64);
    let t2 = task(200u64);
    tokio::pin!(t1, t2);
    tokio::select! {
        () = t1 => println!("task one completed first"),
        () = t2 => println!("task two completed first"),
    }
}
```

⁵tokio::select!

```
loop tokio::select!
```

```
#[tokio::main]
```

```
async fn main() {
```

```
    let mut count = 0;
```

```
    let t1 = task(300u64);
```

```
    let t2 = task(200u64);
```

```
    tokio::pin!(t1, t2);
```

```
    loop {
```

```
        if count > 5 {
```

```
            break;
```

```
        }
```

```
        tokio::select! {
```

```
            () = &mut t1 => println!("task one completed first"),
```

```
            () = &mut t2 => println!("task two completed first"),
```

```
        }
```

```
        count += 1;
```

```
    }
```

```
}
```

loop futures::select!

```
#[tokio::main]
async fn main() {
    let mut count = 0;
    let t1 = task(300u64).fuse();
    let t2 = task(200u64).fuse();
    tokio::pin!(t1, t2);
    loop {
        if count > 5 {
            break;
        }
        futures::select! {
            () = &mut t1 => println!("task one completed first"),
            () = &mut t2 => println!("task two completed first"),
        }
        count += 1;
    }
}
```

Stream⁶

```
pub trait Stream {
    type Item;

    // Required method
    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;
}
```

⁶Guided tour of Streams

async-stream

```
fn zero_to_three() -> impl Stream<Item = u32> {
    stream! {
        for i in 0..3 {
            yield i;
        }
    }
}
```

```
#[tokio::main]
async fn main() {
    let s = zero_to_three();
    pin_mut!(s); // needed for iteration

    while let Some(value) = s.next().await {
        println!("got {}", value);
    }
}
```


`futures::select!` vs `tokio::select!`

- ▶ SO - What's the difference between `futures::select` and `tokio::select`?
- ▶ Provide `select!` macro

Multiple futures

- ▶ `FuturesUnordered`
- ▶ `FuturesOrdered`
- ▶ Must read
 - ▶ `FuturesUnordered` and the order of futures

Cancellation

- ▶ `futures::future::Abortable`

Pitfalls

- ▶ Blocking in `async`
 - ▶ `Async: What's blocking`
 - ▶ TLDR: Async code should never spend a long time without reaching an `.await`
- ▶ Cancellation safety
- ▶ Holding a `Mutex` across an `await`
- ▶ Must read
 - ▶ `Async cancellation: a case study of pub-sub in mini-redis`
 - ▶ `Yoshua Wuyts - Async Cancellation`
 - ▶ `Common mistakes with Rust Async`
 - ▶ `Rust tokio task cancellation patterns`
 - ▶ `for await and the battle of buffered streams`
 - ▶ `Mutex without lock, Queue without push: cancel safety in lilos`

Pinning

- ▶ Pin and suffering
- ▶ Pin, Unpin, and why Rust needs them

More references

- ▶ [Meetup code samples](#)
- ▶ [Tokio tutorial](#)
- ▶ [Tokio internals](#)
- ▶ [How Rust optimizes async/await - I](#)
- ▶ [How Rust optimizes async/await - II](#)

Questions

- ▶ Reach out on
 - ▶ Email: me@sanchayanmaity.net
 - ▶ Mastodon: [sanchayanmaity.com](https://mastodon.social/@sanchayanmaity)
 - ▶ Telegram: <https://t.me/SanchayanMaity>
 - ▶ Blog: sanchayanmaity.net