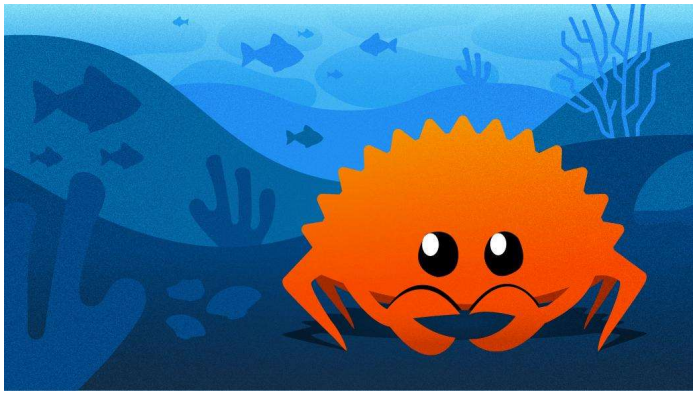
Futures and async/await

Asynchronous Programming

- Callbacks
- Multi-threading
- Future/Promise/Rx
- Coroutines

- ☑ Responsiveness
- ☑ Performance



Essentials

An async application should pull in at least two crates from Rusts ecosystem:

1. **futures**, an official Rust crate that lives in the rust-lang repository
2. A runtime of your choosing, such as **Tokio**, **async_std**, **smol**, etc.

```
[dependencies]
futures = { version = "0.3.*" }
tokio = { version = "1.24.*", features = ["full"] }
async-std = {
    version = "1.12.0",
    features = ["unstable"], ["attributes"] }
}
```


Futures

- Futures represent values that *should exist at some point in the future* as the result of an async operation.
- Concurrency primitives similar to **Promises** in Javascript.
- However, *Rust's Future is lazy* in that it does nothing until explicitly requested via such as `await`.
- Stepping stone to `async`/`await` which allows users to write *asynchronous code that looks like synchronous code***.**
- Futures arrived in stable Rust with version 1.36.0.
- Rust is a low-level language and doesn't include a runtime for scheduling async tasks.

async fn() syntax

- Use `async` in front of `fn`, **closure**, or a **block** to turn the code into a `Future`.
- `async` is a keyword to annotate an *async function*

```
async fn foo() -> u32 {
    42
}
```

which is just a syntactic sugar for:

```
fn foo_desugared -> impl Future<Output = u32> {
    async { 42 }
}
```

```
let f: impl Future<Output = u32> = foo();
let f: impl Future<Output = u32> = foo_desugared();
let f: impl Future<Output = u32> = async { 42 };
```


async/.await

- `async`/`.await` is Rust's built-in tool for *writing asynchronous functions that look like synchronous code*.
- Similar to `async`/`await` in Javascript and C#, etc.
  - Unlike Javascript, you need to pick a runtime to actually run your asynchronous code.
- `async` is zero-cost. You can use `async` without heap allocations and dynamic dispatch.
- `async`/`.await` is a simpler syntax for futures.

async/.await (cont'd)

- Async functions will not be run immediately, but will only be evaluated when the returned future is submitted to the executor via `.await` etc.

```
async fn hello_rusty() {
    println!("Hello, Rusty!");
}

async fn execute_async_function() {
    let future = hello_rusty(); // Nothing is printed
    future.await; // `future` is run and "hello, world!" is printed
}
```

.await is only allowed inside async functions and blocks

- `.await` yields control back to the calling thread to make other tasks progress.
- Rust does not let you declare `async` functions in traits, yet.

Runs a Future

- `block_on` run a future to completion on the current thread.
  - This function will block the caller until the given future has completed.

```
async fn hello_rusty() {
    println!("Hello, Rusty!");
}
```

```
use futures::executor;
executor::block_on(hello_rusty());
// "hello, world!" is printed
```

```
use async_std::task;
task::block_on(hello_rusty());
```

```
use tokio::runtime;
let rt = runtime::Runtime::new().unwrap();
rt.block_on(hello_rusty());
```

From Synchronous to Asynchronous

```
fn postItem(item: Item) {
    let token = requestToken();
    let post = createPost(token, item);
    showPost(post)
}

fn requestToken() -> Token {
    // makes request for a token & waits
    token // returns result when received
}

fn createPost(token: Token, item: Item) -> Post {
    // sends item to the server & waits
    return post // returns resulting post
}

fn showPost(post: Post) {
    // does some local processing of result
}
```


Async via Callbacks

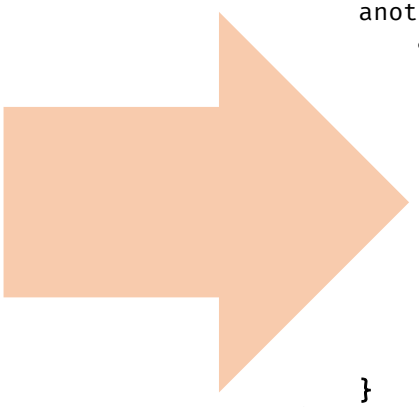
```
fn post_item(item: Item) {
    request_token(move |token| {
        create_post(token, item, move |post| {
            show_post(post);
        });
    });
}

fn request_token<F>(callback: F) where F: Fn(Token) + Send + 'static {
    background(Token, callback);
}

fn create_post<F>(token: Token, item: Item, callback: F)
where F: Fn(Post) + Send + 'static {
    background(Post, callback);
}

fn show_post(post: Post) {
    println!("post displyed");
}
```

Hard to read and harder to reason about

Handling exceptions makes it a real mess

Callback Hell

```
private fun loadData() {
    networkRequest { data ->
        anotherRequest(data) { data1 ->
            anotherRequest(data1) { data2 ->
                anotherRequest(data2) { data3 ->
                    anotherRequest(data3) { data4 ->
                        anotherRequest(data4) { data5 ->
                            anotherRequest(data5) { data6 ->
                                anotherRequest(data6) { data7 ->
                                    anotherRequest(data7) { data8 ->
                                        anotherRequest(data8) { data9 ->
                                            anotherRequest(data9) {
                                                // How many more do you want?
                                                println(it)
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```



Callback Hell

Async via Multi-threads

```
fn post_item(item: Item) {
    let (tx, rx) = std::sync::mpsc::channel();

    std::thread::spawn(move || {
        tx.send(request_token()).unwrap();
    });

    let (tx2, rx2) = std::sync::mpsc::channel();
    std::thread::spawn(move || {
        let token = rx.recv().unwrap();
        tx2.send(create_post(token, item)).unwrap();
    });
    std::thread::spawn(move || {
        let post = rx2.recv().unwrap();
        show_post(post);
    });
}
```

- Spawning threads can be expensive
- Codes can get quite complicated
- **Error-prone**

Async vis Future Combinators

```
fn post_item(item: Item) -> impl Future<Output = ()> {
    request_token()
        .then(move |token| create_post(token, item))
        .then(|post| {
            show_post(post);
            futures::future::ready(())
        })
}
```


No nesting indentation


Composable


Library-specific operators


Manually implemented futures

```
impl Future for RequestToken {
    type Output = Token;

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


Async via async/.await

```
async fn post_item(item: Item) {
    let token = request_token().await;          // suspension point
    let post = create_post(token, item).await; // suspension point
    show_post(post);
}
```

The async world is nicely sequential!

```
async fn request_token() -> Token {
    delay(2000); // simulate network delay
    Token
}

async fn create_post(token: Token, item: Item) -> Post {
    delay(2000); // simulate network delay
    Post
}

fn show_post(post: Post) {
    println!("post displyed");
}
```

Concurrency using async/.await

```
async fn loadImage(name: String): Image = { … }
fn combineImages(img1: &Image, &img2: Image): Image = { … }
```

```
// Non-blocking, but sequential code. How to make this code run concurrently?
async fn load_and_combine(name1: String, name2: String): Image {
    let task1 = loadImage(name1);
    let task2 = loadImage(name2);

    combineImages(task1.await, task2.await)
}
```

suspends until task is complete

Concurrency using async/.await

```
async fn loadImage(name: String): Image = { … }
fn combineImages(img1: &Image, &img2: Image): Image = { … }
```

```
// Non-blocking, but sequential code. How to make this code run concurrently?
async fn load_and_combine(name1: String, name2: String): Image {
    let task1 = loadImage(name1);
    let task2 = loadImage(name2);

    combineImages(task1.await, task2.await)
}
```

suspends until task is complete

Three Basic Operations to Figure Out

- How to start the runtime
- How to spawn a Future
- How to spawn blocking or CPU-intensive tasks

We will choose **tokio** as our runtime.

Async runtime - tokio

Tokio provides:

- Top-level ‘loop’
- Low-level resources like network sockets and timers (e.g., `tokio::net::TcpListener`, `tokio::fs`, `tokio::timer`)
- **Executor** schedules and drives futures to completion.
- **Reactor** interacts with OS (via crate Mio) to say ‘Wake me up in any of these file descriptors change state’ (epoll, kqueue , IO Completion Ports).
- Also handles waiting on non-OS events, e.g., receiving on channels.

Starting the runtime

- Explicitly instantiate the runtime, or use attribute macro `#[tokio::main]`.

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

fn main() {
    let mut rt = Runtime::new().unwrap();
    let future = app();

    rt.block_on(future);
}

async fn app() {
    println!("Hello, Tokio!");
}
```

```rust
#[tokio::main]
async fn main() {
    app().await;
}
```

```rust
fn main() {
    tokio::runtime::Builder
        ::new_multi_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(app())
}
```


Spawning a Future on the runtime

```rust
use std::time::Duration;
use tokio::time::sleep;

#[tokio::test]
async fn spawning_task() {
    use tokio::task;

    async fn some_task() {
        sleep(Duration::from_millis(1000)).await;
        println!("Hello from some_task");
    }

    let future = task::spawn(some_task()); // starts running now
    let result = future.await; // no need to await, if not necessary
}
```

Spawning blocking or CPU-intensive tasks

- You should avoid blocking or running CPU-intensive code in Futures themselves.
- Most of the code you write in async Rust will actually be executed in a Future.
- You should offload this work to a different thread, so that you avoid blocking the executor thread.
- In tokio, you can do this via `task::spawn_blocking`.

```rust
fn fibonacci(n: u32) -> u32 {
    match n {
        0 | 1 => n,
        v => fibonacci(v - 1) + fibonacci(v - 2),
    }
}
```


Spawning blocking or CPU-intensive tasks (cont's)

```rust
async fn foo() {
  for i in 0..10 {
    println!("Hello from foo: {i}");
      sleep(Duration::from_millis(500)).await;
  }
}
```

```rust
let fib = async { fibonacci(45) };
let (result, _) = tokio::join!(fib, foo());
println!("Result: {result}");
```

```rust
let fib = task::spawn_blocking(|| fibonacci(45));
let (result, _) = tokio::join!(fib, foo());
println!("Result: {}", result.unwrap());
```

Crate futures Runtime
(futures::future::*)

- Provides abstractions for asynchronous programming: Futures, Streams, Sinks, and Executors.
- Also provides a lot combinators (`map`, `then`, etc.) for future composition.
- `block_on` run a future to completion on the current thread.
  - This function will block the caller until the given future has completed.

```
use std::future::Future;
use futures::{ prelude::*, executor::block_on }

// Creates a future that is immediately ready with a value.
let future = future::ready(42);
let result = block_on(future);

// Create a future that is immediately ready with a success value.
let future = future::ok::<i32, &str>(42);

// Create a future that is immediately ready with an error value.
let future = future::err::<i32, &str>("error");
```


Futures in Depth

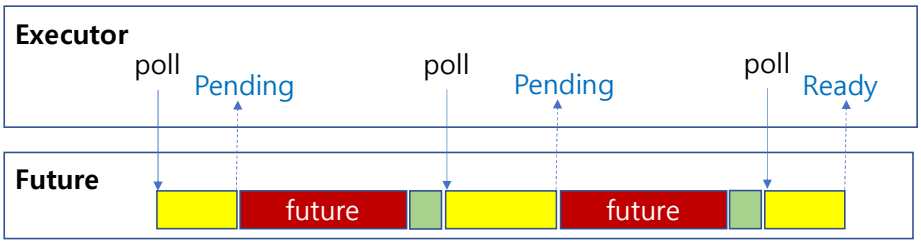
Trait std::future::Future

- `Future` is *polled* by an *executor* as many time as necessary until its value is ready.

```
trait Future {
  type Output;

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

Essentially a Callback to be called when the future is ready to progress.

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



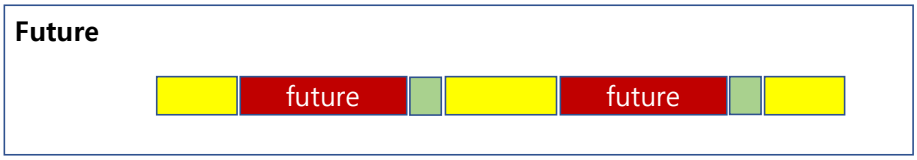
Executors and Runtime

- Runtimes normally consist of **Executor** and **Reactor**.
- Executor creates a *waker* and passes it to the `Future` when it is time to poll.
- Executor keeps invoking polling method of the `Future` whenever the passed *waker* notifies it, which will drive the future to completion..
- Reactor notifies the Executor of event completion via *waker*.
- There are no built-in runtime including executors in the standard library.
- Instead, runtimes are provided by community maintained crates (e.g., **tokio**, **async-std**, **smol**, etc.).
  - This allows different applications to choose different executors to make trade-offs that best meet the goals of a given applications.
  - Both single- and multi-threaded runtimes are available, which have different strengths and weaknesses.

When does the task run?

- A **task** is a top-level future that has been submitted to an executor, and it is a unit of asynchronous computation.
- Some executor at a top level polls futures until progress can be made.
  - A very basic executor might poll futures in a loop until they complete.

Future {
  processing
  Future.await
  processing
  Future.await
  processing
}



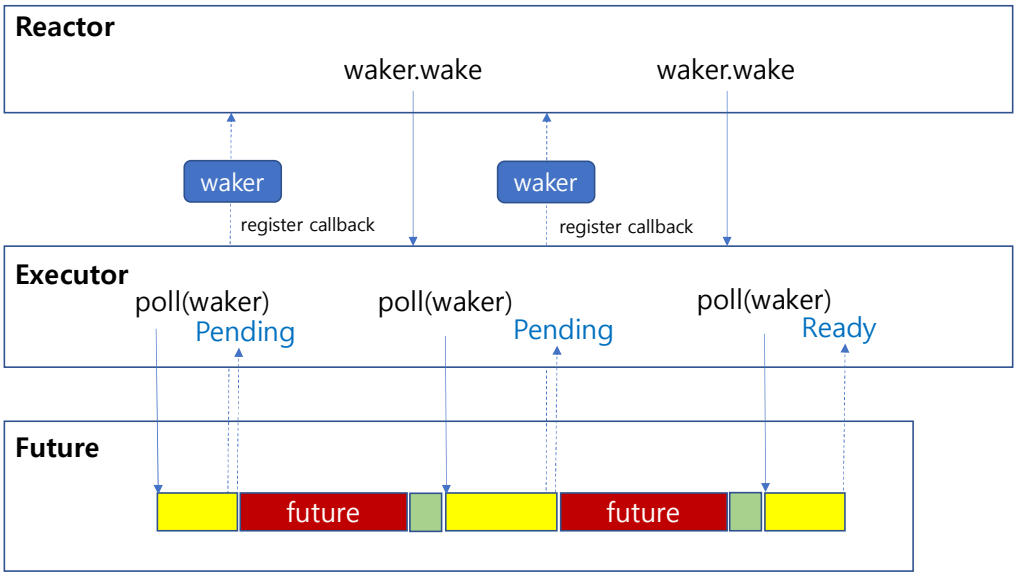
Simplified Top-level Executor

```
pub fn run_block_on<F: Future>(future: F) -> F::Output {
    pin_mut!(future); // Pins a value on the stack

    loop {
        // Create a waker
        let t = std::thread::current();
        let waker = Arc::new(ThreadWaker(t)).into();
        let mut cx = Context::from_waker(&waker);

        match future.as_mut().poll(&mut cx) {
            Poll::Ready(output) => return output,
            Poll::Pending => {
                std::thread::park();
            }
        }
    }
}
```

The .await expands roughly to this:

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


Future {
  processing
  Future.await
  processing
  Future.await
  processing
}



Implicitly Generated Future as State Machine

```rust
async fn foo() {
    println!("{:?}", starts);
    let result = io_read().await;
    println!("{:?}", result);
}
```

```rust
enum Status<F>
where
  F: Future<Output = [u8;10]>,
{
    Step0,
    Step1(F),
    Step2,
}
```

```rust
struct Foo {
    status: Status<IoRead>,
}

impl Foo {
    fn new() -> Self {
        Self {
            status: Status::Step0,
        }
    }
}

async fn io_read() -> [u8; 10] {
  let mut buffer = [0u8; 10];
  let mut b = "This string will be read"
              .as_bytes();
  // read up to 10 bytes
  b.read(&mut buffer);
  buffer
}
```


Implicitly Generated Future as State Machine

```rust
impl Future for Foo {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let mut status = unsafe {
            &mut self.get_unchecked_mut().status
        };

        match status {
            Status::Step0 => {
                println!("{:?}", starts);
                *status = Status::Step1(IoRead::new());
                cx.waker().wake_by_ref();
                Poll::Pending
            }
```

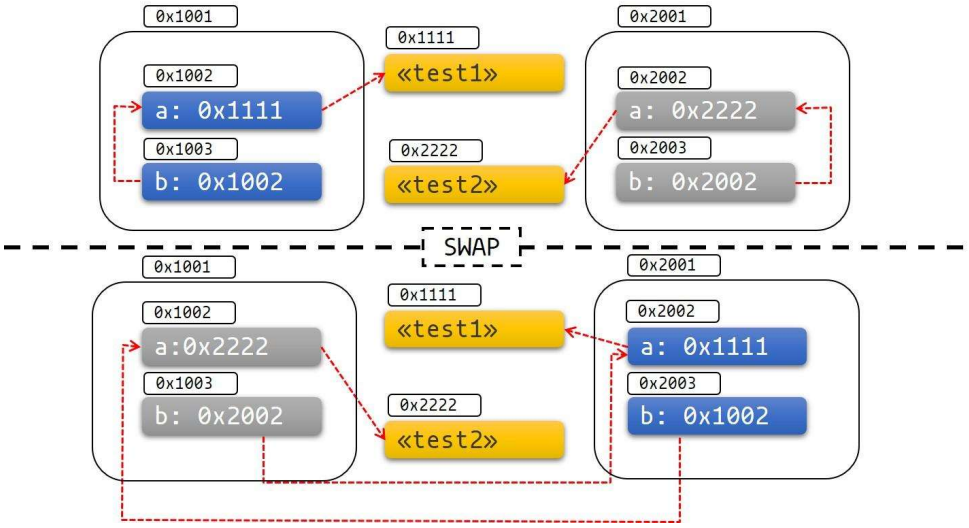
```
                Status::Step1(ref mut ioread) => {
                    match Pin::new(ioread).poll(cx) {
                        Poll::Ready(result) => {
                            *status = Status::Step2(result);
                            cx.waker().wake_by_ref();
                            Poll::Pending
                        }
                        Poll::Pending => {
                            cx.waker().wake_by_ref();
                            Poll::Pending
                        }
                    }
                }
                Status::Step2(result) => {
                    println!("{:?}", result);
                    Poll::Ready(())
                }
            }
        }
    }
```


Self-referential Types

```
struct Test {
  a: String,
  b: *const String,
}

impl Test {
  fn new(txt: &str) -> Self {
    Test {
      a: String::from(txt),
      b: std::ptr::null(),
    }
  }

  fn init(&mut self) {
    self.b = &self.a as *const String;
  }

  fn a(&self) -> &str { &self.a }
  fn b(&self) -> &String { unsafe { &*(self.b) } }
}
```

```
let mut test1 = Test::new("test1");
test1.init();
let mut test2 = Test::new("test2");
test2.init();

println!("a: {}, b: {}", test1.a(), test1.b());
println!("a: {}, b: {}", test2.a(), test2.b());
// Use the swap() function to swap the two
std::mem::swap(&mut test1, &mut test2);
println!("a: {}, b: {}", test1.a(), test1.b());
println!("a: {}, b: {}", test2.a(), test2.b());
```

Self-referential Types



Pinning and boxing

- Rust does not know whether a type can be safely moved.
- Types that shouldn't be moved must be wrapped inside `Pin<T>`.
- Most types are `Unpin`ned types. They implement the trait `Unpin` and can be freely moved within memory.
- If a type is wrapped inside `Pin<T>` and the wrapped type is `!Unpin`, it is not possible to get a mutable reference out of it.
- Futures created by the **async** keyword are `!Unpin` and thus must be pinned.

Implicitly Generated Future as State Machine

```
async fn foo() {
    println!("{:?}", starts);
    let mut buffer = [u8; 10];
    io_read(&mut buffer).await;
    println!("{:?}", buffer);
}

enum Status<F>
where
  F: Future<Output = ()>,
{
    Step0,
    Step1(F),
    Step2,
}
```

```
struct Foo {
    buffer: [u8; 10],
    status: Status<IoRead>,
}
impl Foo {
    fn new() -> Self {
        Self {
            buffer: [0u8; 10],
            status: Status::Step0,
        }
    }
}
async fn io_read(mut buf: &mut [u8; 10]) {
  let mut b = "This string will be read"
            .as_bytes();
  // read up to 10 bytes
  b.read(&mut buffer[..]);
}
```


Implicitly Generated Future as State Machine

```
impl Future for Foo {
  type Output = ();

  fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
    let mut future = unsafe { self.get_unchecked_mut() };

    match future.status {
      Status::Step0 => {
        println!("{:?}", starts);
        status = Status::Step1(IoRead::new(&mut future.buffer as *mut _));
        cx.waker().wake_by_ref();
        Poll::Pending
      }
```

```
                Status::Step1(ref mut ioread) => {
                    match Pin::new_unchecked(ioread).poll(cx) {
                        Poll::Ready(()) => {
                            *status = Status::Step2;
                            cx.waker().wake_by_ref();
                            Poll::Pending
                        }
                        Poll::Pending => {
                            cx.waker().wake_by_ref();
                            Poll::Pending
                        }
                    }
                }
                Status::Step2 => {
                    println!("{:?}", future.buffer);
                    Poll::Ready(())
                }
            }
        }
    }
```