Async/Await (Futures)

DanielAsensiRoch: **48776120C**

Índice

1. Rust

Rust es un lenguaje de programación compilado, de propósito general y multiparadigma que está siendo desarrollado por Fundación Rust. Es un lenguaje de programación multiparadigmático que soporta programación funcional pura, por procedimientos, imperativa y orientada a objetos.[9] **Rust fue elaborado para ser un lenguaje seguro, concurrente y práctico.** [5]

Rust funciona mediante la política del openSource es innegablemente un esfuerzo colectivo, y las contribuciones son bien recibidas, ya sean de aficionados o usuarios de producción, novatos o profesionales con experiencia.Su diseño se ha ido perfeccionando a través de las vivencias en el desarrollo del motor del navegador Servo y su propio compilador.

1.1. Propósito

Rust busca ser un buen lenguaje para la creación de grandes programas del paradigma **cliente/servidor** ejecutados en internet, por lo que sus creadores y la comunidad, han enfatizado en gran medida en la correcta distribución de la memoria, y en proporcionar al programador una concurrencia segura, evitando los punteros nulos o colgantes.

1.2. Los traits

La forma de clases que nos proporciona Rust son los **traits**, facilitando el polimorfismos con varios tipos de argumentos, añadiendo restricciones para escribir declaraciones de variables. Los traits son métodos con implementación o sin ella, que las estructuras o tipos deben de implementar, para cumplir ese trait.

1.3. Inferencia de tipos

Rust cuenta con **inferencia de tipos**, siempre y cuando las variables sean declaradas con **let**, además no es necesario que sean inicializadas con un valor por defecto.

1.4. Programación segura y no segura

Hay dos formas de escribir código en **Rust: Safe Rust y Unsafe Rust**. La diferencia es que el *"modo seguro"* impone restricciones adicionales al programador, para mantener el código funcionando, mientras que el *"modo inseguro"*, le da más autonomía al programador, pero el código es más fácil de que tenga un comportamiento inadecuado. Este modelo dual de Rust es una de sus mayores fortalezas, por ejemplo, en C++ nunca sabes que escribiste código inseguro, hasta que en algún momento el programa falla o se produce un agujero de seguridad.

2. Concurrencia

Rust tiene un sistema de tipo estático, que facilita la programación concurrente en **tiempo de compilación** y la administración de memoria al mismo tiempo. Este sistema asegura que no haya condiciones

de carrera, por lo que una vez compilado, estaremos seguros de que la gestión de la memoria compartida es correcta, y que no habrá fallos de segmentación.

2.1. Traits de concurrencia

Los siguientes traits nos permiten usar el sistema de tipos garantizando completamente la seguridad de las propiedades de nuestro código concurrente.

- **Send:** este trait le indica al compilador de rust, que cualquier elemento de este tipo puede transferir la pertenencia entre hilos de forma segura.
- **Sync:** este trait le indica al compilador de rust que cualquier elemento de este tipo, no posee la habilidad de introducir inseguridad en memoria, cuando es usado de forma concurrente por múltiples hilos de ejecución.

Además de los mencionados anteriormente tenemos a los tipos **Arc y Mutex**, los cuales nos serán útiles si queremos compartir una variable entre hilos. La biblioteca estándar cuenta con canales para la sincronización entre hilos.

2.2. Otras bibliotecas

Además de los manejos de threads de las bibliotecas estándar de Rust contamos con muchas otras conformadas por la comunidad, o por los propios desarrolladores principales, por ejemplo las siguientes:

1. **Crossbeam**
2. **Rayon**
3. **Coroutine-rs**
4. **Futures**

3. Modelos de Asincronía

En la siguiente sección, nos centraremos en esta última biblioteca (Future), que se centra en la programación asincrónica, que es el tema de nuestro documento. Pero antes debemos especificar que es la asincronía [5]

La programación asíncrona es un modelo de programación concurrente compatible con múltiples lenguajes de programación. Este tipo de programación nos permite realizar múltiples operaciones simultáneas en una pequeña cantidad de subprocesos del sistema operativo, mientras retenemos gran parte de la perspectiva de la programación síncrona estándar a través de los atributos asíncrono/en espera.

3.1. Los diferentes modelos concurrencia

Para comprender cómo encaja la programación asíncrona, en el campo más amplio de la programación paralela, es necesario describir brevemente los modelos concurrentes más populares, la programación asíncrona nos permite tener implementaciones poderosas, que son adecuadas para lenguajes de bajo nivel como Rust y ofrecen la mayoría de las ventajas ergonómicas de los hilos y corrutinas.

3.1.1. Hilos del sistema operativo

Los subprocesos del sistema operativo no requieren ningún cambio en el modelo de programación, lo que facilita mucho la expresión de la concurrencia. Sin embargo, la sincronización entre subprocesos puede ser difícil debido a la sobrecarga de alto rendimiento. Los grupos de subprocesos pueden reducir aproximadamente este costo, pero no son suficientes para admitir cargas de E/S masivas. [1]

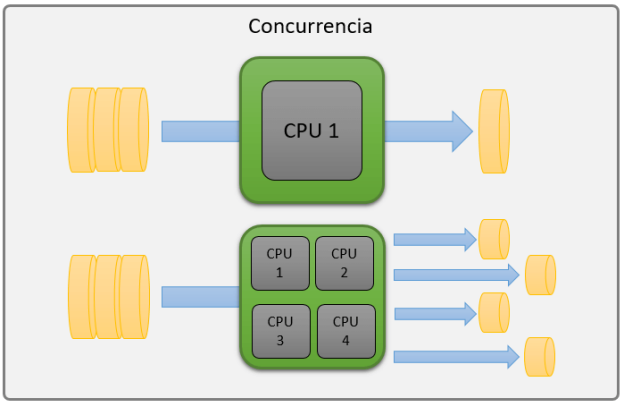

Figura 1: Hilos

3.1.2. Modelo del actor

El modelo de actor divide todos los cálculos concurrentes en **unidades denominadas actores**, que se comunican a través de mensajería como sistemas distribuidos. Este modelo se puede implementar de manera efectiva, pero plantea muchos desafíos, como el **control de flujo y la lógica repetitiva**. [4]

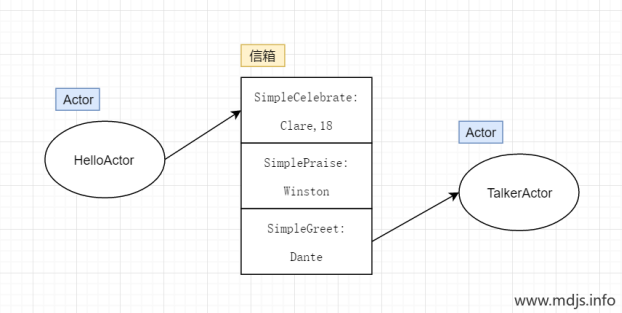

Figura 2: Actor

3.1.3. Dirigida a eventos

La programación basada en eventos puede ser muy poderosa cuando se usa con devoluciones de llamadas, pero tiende a proporcionar un flujo de control detallado y "no lineal". El flujo de datos y la propagación de errores a menudo son difíciles de entender.[3]

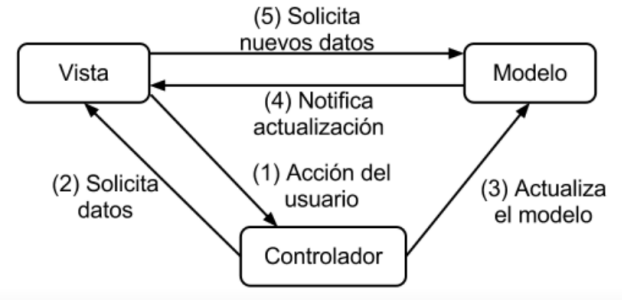

Figura 3: Eventos

3.1.4. Corrutinas

Como asíncronas, las rutinas pueden admitir una gran cantidad de tareas. Sin embargo, utilizan los detalles de bajo nivel que son importantes para los desarrolladores de sistemas y tiempos de ejecución personalizados.

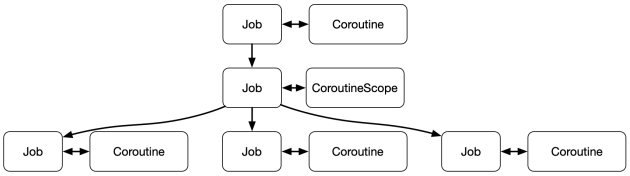

Figura 4: Corrutinas

4. Async/Await con Future

Future nos permite desarrollar funciones asíncronas, como si fueran funciones síncronas implementando los atributos **async /await**, transformando el bloque en cuestión en una máquina de estado. Al contrario que los bloques síncronos, los bloques desarrollados con future, cederán el control del hilo para que otros Futures puedan ejecutarse.

Figura 5: Logo Future RS

4.1. Instalación de Futures

Para instalar las dependencias de Futures y poder usar todos sus traits, debemos añadir las siguientes lineas a nuestro archivo **Cargo.toml**, este archivo de configuración es responsable de almacenar información sobre paquetes, nombres, Versión, información del autor y versión de Rust.

```
[dependencies]
futures = "0.3"
```

4.2. Creación función asíncrona

Para la creación de nuestras funciones asíncronas usaremos la siguiente sintaxis:

```
fn main() {
    async fn do_something() { /* ... */ }
}
```

4.2.1. Uso de Block_on, .await y Join

Esta función asíncrona nos devolverá un valor **Future**, para que algo suceda deberemos ejecutar dicho Future en un **Ejecutor**, **Ejemplo de Hello World**:

```
use futures::executor::block_on;
    async fn hello_world() {
    println!("hello, world!");
}

fn main() {
    let future = hello_world(); // No se imprime nada
    block_on(future); // `Future` se ejecuta y "hello, world!" se imprime.
}
```

block_on bloquea el hilo actual, hasta que el Future proporcionado se haya ejecutado hasta el final. Otros ejecutores proporcionan un comportamiento más complejo, como programar múltiples futuros en el mismo hilo.

Dentro de una función asíncrona, podemos usar **.await**, esto nos servirá para esperar a la finalización de otro tipo que implemente el trait Future, como la salida de otra función asíncrona. A diferencia de block_on **.await no bloqueará el hilo actual**, este esperará asíncronamente a que se complete el Future, lo que permite que otras tareas de dicho trait se ejecuten con normalidad. Al igual que **.await** tenemos **join** el cual tiene el mismo funcionamiento que el anteriormente mencionado **.await** pero con la particularidad de que puede esperar varios futures simultaneamente.

4.2.2. Ejemplo del poeta

Para entender el uso y diferencias de **block_on y .await** representaré un pequeño ejemplo.

Pongamos el ejemplo de un poeta este puede realizar una serie de acciones con los poemas que recibe de sus superiores, como son: **Aprender el poema, Recitar el poema y representarlo**, para representar en forma de código estas acciones definiremos las siguientes funciones:

```
async fn aprender_poema() -> Poema { /* ... */ }
async fn recitar_poema(poema: Poema) { /* ... */ }
async fn representar() { /* ... */ }
```

Podríamos hacer que nuestro poeta realizara las anteriores funciones individualmente utilizando el ya mencionado **block_on**.

```
fn main() {
    let poema = block_on(aprender_poema());
    block_on(recitar_poema(poema));
    block_on(representar());
}
```

Pero esto no sería óptimo para el negocio, por lo que nuestro poeta ha pensado que puede representar, bailar y aprender un poema todo a la vez, pero no puede recitar un poema que este aun no ha aprendido, si quisieramos representar lo siguiente en código deberíamos realizar dos funciones asíncronas que pudieran ejecutarse simultáneamente.

```
async fn aprender_y_recitar() {
   let poema = aprender_poema().await;
   recitar_poema(poema).await;
}

async fn async_main() {
   let f1 = aprender_y_recitar();
   let f2 = representar();
   futures::join!(f1, f2);
}
```

```
fn main() {
block_on(async_main());
}
```

En la función ***aprender_y_recitar()*** esperamos a aprender el poema antes de recitarlo y utilizaremos el .await, para evitar el bloqueo del hilo de representar a la vez que recitamos.

En la función ***async_main()*** si estamos temporalmente bloqueados en el Future de ***aprender_y_recitar*** el future de ***representar*** se hará cargo del hilo actual. Si ***representar*** se bloquea entonces ***aprender_y_recitar*** retomará el control. En el caso de que ambos se bloqueen, ***async_main()*** también se bloqueará y por lo tanto se cederá al ***ejecutor*** el control.

En el ejemplo puesto el aprendizaje de un poema debe ocurrir antes de poder recitarlo, pero tanto el aprendizaje como la recitación del mismo puede ocurrir exactamente al mismo tiempo que la representación.

En el caso de que utilizáramos el ya mencionado **block_on(aprender_poema()** en lugar de **aprender_poema().await** en la función **aprender_y_recitar()**, el hilo no podrá hacer nada más mientras se ejecuta la función de **aprender_poema()**, lo que haría imposible que nuestro poema se representara al mismo tiempo.

Al utilizar **.await** en el future de nuestra función **aprender_poema()**, permitimos que otras tareas tomen el control del hilo actual si **aprender_poema()** se bloquea. Esto nos hará posible poder ejecutar múltiples Futures hasta su finalización de forma concurrente en el mismo hilo.

4.2.3. Bloques Asíncronos

Otra forma de implementar código asíncrono es utilizar los bloques asíncronos, los cuales al igual que las funciones, devolverán un valor que implementa el rasgo **Future**.

```
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
async {
   let x: u8 = foo().await;
   x + 5
}
}
```

foo() devuelve un tipo que implementa **'Future Output = u8'** además, **foo().await** dará como resultado un valor de tipo 'u8'

5. Future Trait

El rasgo Futuro representa un cómputo asíncrono, que es un valor que puede no estar completo todavía. Este tipo de "valor asíncrono"permite que los subprocesos continúen ejecutando código mientras espera a que el valor esté disponible.

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

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

Los dos componentes principales que podemos observar son: **type Output**; que es el valor que devuelve el trait si existe, y la función **poll**; la cual contiene dos parámetros **Self y Context**, devolviéndonos un tipo enumerado llamado Poll para saber si el valor se encuentra listo o pendiente.

- El primer parámetro **Self** nos permitirá crear Futures inamovibles, es decir constantes, usando **Pin** ya que es necesario para habilitar las funciones async/await.
- El segundo parámetro **cx** se utiliza el tipo Context para tener acceso a un valor de tipo Waker, para así saber que tarea debe ejecutarse específicamente.

Cuando Future aún no está listo, la función de Poll devuelve Poll::Pending y almacena una copia de Waker copiada en el Contexto actual (el tipo de Waker se usa para decirle al ejecutor que la tarea relacionada debe realizarse usando wake( )).

Este Waker se despierta una vez que el Future puede avanzar, e intenta evaluar (poll) el Future otra vez, lo que puede o no obtener un valor final.

5.1. Join! esperar varios futures

La macro **Join!** nos permitirá esperar a que varios Futures que se ejecutan al mismo tiempo se completen. Esta macro la usamos en el ejemplo del poeta para proceder a la espera de los Futures f1 y f2 simultáneamente. Como hemos explicado la razón de usar el **join** es porque sino lo hicieramos de esta manera f2 no se podrá ejecutar hasta que se termine de ejecutar f1.

Los Futures en Rust no harán nada hasta que se activen usando .await. Esto significa que si sólo usamos **.await**, dichos futures se ejecutarán en serie en lugar de ejecutarlos al mismo tiempo.

El valor devuelto por join! es una tupla que contiene la salida de cada Future pasado como parámetro.

5.2. Select! ejecución de múltiples futures

La macro **select!** nos permitirá ejecutar múltiples Futures simultáneamente, lo que permite al usuario responder tan pronto como se ejecute un Future, en el siguiente ejemplo extraído de la documentación oficial de rust podemos ver un ejemplo de uso de este select:

```
#![allow(unused)]
fn main() {
    use futures::{
        future::FutureExt, // for `.fuse()`
        pin_mut,
        select,
    };

    async fn task_one() { /* ... */ }
    async fn task_two() { /* ... */ }

    async fn race_tasks() {
        let t1 = task_one().fuse();
        let t2 = task_two().fuse();

        pin_mut!(t1, t2);

        select! {
            () = t1 => println!("task one completed first"),
            () = t2 => println!("task two completed first"),
        }
    }
}
```

La función anterior ejecutará tanto t1 como t2 al mismo tiempo. Cuando finalice t1 o t2, el controlador correspondiente llamará a println! y la función finalizará sin completar la tarea restante. [8]

6. Tiempos de ejecución asíncronos

A diferencia de otras características de Rust, no puedes simplemente escribir await en el código y ejecutarlo. Debe usar un tiempo de ejecución asíncrono como **Tokio o async-std**. Estos son bibliotecas utilizadas para ejecutar aplicaciones asíncronas. Suelen agrupar un reactor (bucle de eventos que controla todos los recursos de E/S) con uno o más ejecutores.

Los reactores nos proporcionan mecanismos de suscripción para eventos externos como las entradas y salidas asíncronas, la comunicación entre procesos y temporizadores.

Los suscriptores suelen ser futuros que representan operaciones de E/S de bajo nivel. Los ejecutores manejan la planificación y ejecución de tareas. Además, realizan un seguimiento de las tareas suspendidas y en ejecución, evalúan los futures hasta su finalización y activan las tareas cuando pueden avanzar.

6.1. Los runtime más populares

En rust disponemos de varios runtimes asíncronos, entre los cuales podemos listar los siguiente por orden de popularidad.[6]

1. **Tokio** es uno de los tiempos de ejecución más antiguos y ampliamente utilizados, y probablemente el más utilizado en la producción. Tiene un ejecutor altamente eficiente, personalizable y flexible. Es de destacar que los futuros no tienen que pasar a un hilo de fondo para su ejecución, lo cual es excelente para el rendimiento, sino que requiere reglas bastante estrictas.

2. **Async-std y Smol** son dos tiempos de ejecución basados en componentes smol-rs. Async-std está diseñado para estar lo más cerca posible de la biblioteca estándar sincrónica, mientras que Smol está diseñado para ser más mínimo. Son ampliamente utilizados y listos para la producción. A diferencia de Tokio, el ejecutor Smol siempre usa hilos de fondo para ejecutar futuros que sacrifica algún rendimiento potencial para una mejor usabilidad.

3. **Glommio** es un tiempo de ejecución especializado basado en la filosofía de hilo por núcleo e implementado usando io_uring. Está diseñado principalmente para el disco IO. A diferencia de Tokio y async-std, Glommio no es un tiempo de ejecución asíncrono de propósito general.

4. **Embassy** es un tiempo de ejecución diseñado específicamente para el desarrollo integrado. En particular, evita la asignación y no requiere un heap.

6.2. Tokio

Figura 6: Logo Tokio

Tokio es una librería que nos permite ejecutar código de entrada/salida de forma asíncrona, usando futuros por debajo. Son especialmente útiles, la parte de red de Tokio y sus ejecutores correspondientes.

Proporciona los componentes básicos necesarios para escribir aplicaciones de red. Brinda la flexibilidad para apuntar a una amplia gama de sistemas, desde grandes servidores con docenas de núcleos hasta pequeños dispositivos integrados.

6.2.1. Ventajas de Tokio

- **Fiable** Las API de Tokio son seguras para la memoria, seguras para los hilos y resistentes al mal uso. Esto ayuda a prevenir errores comunes, como colas ilimitadas, desbordamientos de búfer y inanición de tareas.
- **Rápido** Sobre la base de Rust, Tokio proporciona un programador de subprocesos múltiples que roba el trabajo. Las aplicaciones pueden procesar cientos de miles de solicitudes por segundo con gastos generales mínimos.
- **Fácil** async/await reduce la complejidad de escribir aplicaciones asíncronas. Junto con las utilidades de Tokio y el ecosistema vibrante, escribir aplicaciones es muy fácil.
- **Flexible** Las necesidades de una aplicación de servidor difieren de las de un dispositivo integrado. Aunque Tokio viene con valores predeterminados que funcionan bien fuera de la caja, también proporciona las perillas necesarias para ajustar la melodía en diferentes casos.

6.2.2. Como usar Tokio

1. Lo primero que deberemos hacer es instalar mini-redis lo cual no ayudará a probar nuestros clientes y construirlos, también ejecutaremos el servidor de mini-redis

```
cargo install mini-redis
mini-redis-server
```

2. Añadiremos las dependencias de Tokio a nuesto archivo Cargo.toml justo debajo de [dependencies]

```
tokio = { version = "1", features = ["full"] }
mini-redis = "0.4"
```

El siguiente bloque de código irá en nuestro main.rs

```
use tokio::io::AsyncWriteExt;
use tokio::net::TcpStream;
use std::error::Error;

#[tokio::main] // incluir esta macro para poder ejecutar nuestro código en el runtime de Tokio
pub async fn main() -> Result<(), Box<dyn Error>> {
    // Tokio TcpStream, completamente asíncrono.
    let mut stream = TcpStream::connect("127.0.0.1:6142").await?;
    println!("created stream");

    let result = stream.write(b"Hola mundo\n").await;
    println!("wrote to stream; success={:?}", result.is_ok());

    Ok(()) // enum Result, contiene el valor si todo ha ido bien
}
```

En este ejemplo se muestra un cliente simple que abre la conexión con el protocolo TCP usando el módulo de Tokio **tokio::net::TcpStream**, escribe *Hola mundo* y cierra la conexión.

6.2.3. Tareas de Tokio

El módulo **tokio::task** proporciona herramientas importantes para trabajar con tareas. **Spawn y el tipo JoinHandle**, se utiliza para programar una nueva tarea en el runtime de Tokio y esperar los resultados de una tarea generada. **Tokio::sync** es un módulo contiene las primitivas de sincronización para usarlo cuando necesitamos comunicar o compartir datos, incluyendo **Channels** para el envió de variables entre tareas, **Mutex** para controlar el acceso compartido y los valores mutables, **Barrera Asíncrona**, para evitar que varias tareas se sincronicen antes de iniciar el cálculo.

6.2.4. Ejemplo de servidor echo

Ahora vamos a ir con un ejemplo más interesante, ahora el cliente y el servidor estarán conectados durante un tiempo, mandando el cliente un mensaje y el servidor respondiendo. Pero queremos que haya varios clientes simultáneamente. Usando futuros y Tokio podemos hacerlo en un mismo hilo (al estilo Node.js).[7]

```
extern crate tokio;
extern crate tokio_io;
extern crate futures;

use futures::prelude::*;
use tokio_io::AsyncRead;
use futures::Stream;
use tokio_io::codec::*;
use std::net::*;
use tokio::prelude::*;

fn main(){
    let socket = SocketAddr::new(IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1)), 8080);
    let listener = tokio::net::TcpListener::bind(&socket).unwrap();
    let server = listener.incoming().for_each(|socket|{
       let (writer,reader) = socket.framed(LinesCodec::new()).split();
       let action = reader
       .map(move |line|{
           println!("ECHO: {}",line);
           line
       })
       .forward(writer)
       .map(|_|{
        })
       .map_err(|err|{
           println!("error");
       });
       tokio::spawn(action);
       Ok(())
    }).map_err(|err|{
        println!("error = {:?}",err);
    });
    tokio::run(server);
}
```

Ahora a cada socket de cliente asignamos una tarea, que se encarga, de forma independiente, de ir leyendo línea a línea (gracias a LineCodec). A partir de aquí se programa en forma de stream, un patrón muy común en la programación con futuros [2]

Workflow del servidor:

1. En el primer map imprimimos la línea en el servidor y la seguimos pasando por el stream.

2. El siguiente paso es escribir en writer, con forward imprimimos y mandamos esa línea al cliente. Forward a su vez devuelve una tupla con datos (útil para seguir haciendo cosas).

3. Como no los necesitamos, hacemos un map cuya única finalidad sea descartar los valores y finalmente un map_err para capturar posibles errores.

4. Una vez hecho esto tenemos un futuro listo para ser ejecutado. Iniciamos la tarea con spawn, pasando a esperar a por un nuevo cliente.

5. Ahora, en el servidor podemos manejar varios clientes a la vez, cada uno en una tarea distinta, dentro de un mismo hilo.

7. Conclusiones

Las conclusiones obtenidas tras indagar en el uso y creación de código asíncrono en rust, es que este se utiliza especialmente para aplicaciones en internet del estilo cliente / servidor, para tratar la posibilidad de que muchos clientes accedan simultáneamente a un recurso.

Al igual que en el resto de lenguajes el código asíncrono en rust tiene una curva de aprendizaje vertical cuando se comienza a intentar implementar, ya que el concepto de la asincronía es complejo de entender, pero una vez empiezas a indagar en el uso de los runtime esta curva se suaviza exponencialmente, volviéndose amigable para el programador, el cual solo tendrá que pensar de manera asíncrona durante las primeras partes de la implementación.

Las peculiaridades que más me han llamado la atención de la asincronía en rust es el indispensable uso de los runtime si se quiere compilar y ejecutar código asíncrono, otra de las peculiaridades que me han llamado la atención es que no contamos, como si es el caso de otros lenguajes como JavaScript, con el conocido Event Loop, es decir que nuestro Future no se activa hasta que utiliza el .await.

Referencias

[1] Eric C Cooper y Richard P Draves. "C threads". En: (1988).

[2] Zak Cutner, Nobuko Yoshida y Martin Vassor. "Deadlock-free asynchronous message reordering in rust with multiparty session types". En: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2022, págs. 246-261.

[3] Ted Faison. *Event-Based Programming*. Springer, 2006.

[4] Carl Hewitt. "Actor model of computation: scalable robust information systems". En: *arXiv preprint arXiv:1008.1459* (2010).

[5] Rust Lang. *The Rust Programming Language*. 2020.

[6] Shing Lyu. "What Else Can You Do with Rust?" En: *Practical Rust Projects*. Springer, 2020, págs. 237-250.

[7] Arash Sal Moslehian. "An Experimental Integration of io uring and Tokio: An Asynchronous Runtime for Rust". Tesis doct. Ferdowsi University of Mashhad.

[8] Tokio runtime. *Tokio.rs*. 2022.

[9] Wikipedia. *Rust (lenguaje de programación) — Wikipedia, La enciclopedia libre*. [Internet; descargado 16-octubre-2022]. 2022. URL: https://es.wikipedia.org/w/index.php?title=Rust_(lenguaje_de_programaci%C3%B3n)&oldid=146614273.