Programación Concurrente

Investigación y Análisis del Paralelismo en Rust usando Crossbeam.

2022 / 2023

Universitat d'Alacant
Universidad de Alicante

Elvi Mihai Sabau Sabau - 51254875L

7 de diciembre de 2022

Índice

1. Crossbeam

1.1. Que es crossbeam?

Crossbeam es una librería (crate) de Rust provee al desarrollador de ciertos útiles para facilitar la implementación de hilos concurrentes, sincronización, manejo y comunicación de y entre estos.

Esta herramientas facilitan el desarrollo de la programación concurrente usando Rust de una forma muchísimo más sencilla que usando las librerías por defecto del lenguaje.

2. Herramientas

Para ello, Crossbeam provee de ciertas herramientas y funciones. En la siguiente lista describiremos las funcionalidades más importantes de esta librería.

2.1. Scope

Scope crea un ámbito que permite el arranque de subprocesos, puede tomar prestado variables de los ámbitos padres, y lo más importante, asegura que una vez finalizado el ámbito, todos los subprocesos habrán finalizado y mergeado en el padre.

Todos los subprocesos secundarios que no se hayan unido manualmente se unirán automáticamente justo antes de que finalice la invocación de esta función. Si alguno de los subprocesos unidos ha entrado en pánico, se devuelve un Err que contiene los errores de los subprocesos que han entrado en pánico.

2.2. Channel

Channel es un sistema de mensajería entre procesos, y actúa como un broker creando un canal de tipo productor / consumidor. Este canal puede ser limitado, pero por defecto la cantidad de mensajes que se envían puede ser ilimitada (limitado a los recursos del sistema).

También permite la creación de canales multiproductor / multiconsumidor. La clonación del productor o consumidor solo crea un nuevo identificador, no crea un productor / consumidor separado con su propio canal de mensajes.

2.2.1. Tipos de Canales

Los canales se pueden crear usando dos funciones:

- Bounded: crea un canal de capacidad limitada, es decir, hay un límite para la cantidad de mensajes que puede contener a la vez.
- Unbounded crea un canal de capacidad ilimitada, es decir, puede contener cualquier número de mensajes a la vez.

2.2.2. Canales Compartidos

Se puede compartir lectores y escritores para compartir el canal de comunicación pasando un clon de cada objeto a diferentes hilos. También se pueden compartir pasando dichos objetos por referencia.

2.2.3. Desconexión

Cuando se desconectan todos los productores o todos los consumidores asociados con un canal, el canal se desconecta. No se pueden enviar más mensajes, pero aún se pueden recibir los mensajes restantes (mensajes encolados en el canal). Las operaciones de envío y recepción en un canal desconectado nunca se bloquean.

2.2.4. Iteradores

Los receptores se pueden utilizar como iteradores. Por ejemplo, el método Iter crea un iterador que recibe mensajes hasta que el canal se vacía y se desconecta, pero hay que considerar que la iteración puede quedar bloqueada la espera de que llegue el siguiente mensaje.

Estos iteradores nos pueden servir por ejemplo para mantener un registro de los mensajes enviados por el canal, o para automatizar ciertos aspectos de nuestra implementación.

Se puede crear un iterador sin bloqueo usando try_iter, que recibe todos los mensajes disponibles sin bloquear.

2.2.5. Selección

El macro select! permite definir un conjunto de operaciones para el canal. Si se definen varias operaciones, y estas están listas al mismo tiempo, se selecciona una al azar para su ejecución.

También es posible definir un caso predeterminado que se ejecuta si ninguna de las operaciones está lista, ya sea de inmediato o por un período de tiempo.

Se considera que una operación está lista si no tiene que bloquearse.

2.2.6. Canales extra

Hay además 3 tipos de canales especiales, que al generar el canal solo devuelve el productor.

- after: crea un canal que entrega un solo mensaje después de un cierto período de tiempo.
- tick: crea un canal que entrega mensajes periódicamente.
- never: crea un canal que nunca entregue mensajes.

Estos canales son muy eficientes porque los mensajes son lazyload en las operaciones de recepción.

2.3. Parker

Parker es un útil usado para bloquear y desbloquear la ejecución de los subprocesos (poner en espera o continuar procesos). Permite aparcar y desaparcar un proceso. También hay una opción para especificar el timeout del aparcamiento.

Al aparcar un proceso, parker pausa el proceso hasta que haya un token que consumir, unpark genera ese token, y esto permite al proceso continuar una vez que parker consuma el token.

Si no hay ningún token previo, parker se espera a recibir un token a consumir.

2.4. SharedLock

SharedLock es un útil que permite bloquear que acceso a lectura / escritura de una variable, y comprobar el estado actual de dichas variables.

Este bloqueo es equivalente a RwLock, pero las operaciones de lectura son más rápidas y las de escritura son más lentas.

Un ShardedLock está hecho internamente de una lista de shards, cada uno de los cuales es un RwLock que ocupa una sola línea de caché en memoria.

Al dividir el bloqueo en shards, las operaciones de lectura simultáneas en la mayoría de los casos elegirán diferentes fragmentos y, por lo tanto, actualizarán diferentes líneas de caché, lo que es bueno para la escalabilidad. Sin embargo, las operaciones de escritura necesitan hacer más trabajo y, por lo tanto, son más lentas de lo normal.

2.5. WaitGroup

Permite sincronizar grupos de hilos, y crear barreras de espera para cada grupo, de esta manera se pueden ejecutar varios grupos de subprocesos y controlar cuando finalizan.

Todos los subprocesos esperan a que otros lleguen a la Barrera. Con WaitGroup, cada subproceso puede optar por esperar a otros subprocesos o continuar sin bloquear.

Para esperar a más de un hilo, dicho hilo solo necesita una referencia al WaitGroup (con clone), y automáticamente WaitGroup detectará que habrá otro hilo al cual esperar. Después, para liberarse, solo se tiene que dropear la referencia del WaitGroup.

3. Ejemplos

A continuación explicaremos un par de ejemplos por cada útil.

3.1. Ejemplos por cada útil

3.1.1. Scope

En el siguiente fragmento de código se muestra un ejemplo sencillo de como crear un subproceso hijo dentro de un scope.

Con este ejemplo también recordamos que Scope también nos permite generar hilos que toman prestadas variables locales del stack.

```
use crossbeam_utils::thread::scope;
use std::thread;
use std::time::Duration;

fn main(){
 // Variable vector.
 let var_vec = vec![1, 2, 3];

 // Scope
 scope(|s| {

     // Hilo
     s.spawn(|_| {
         println!("Un proceso hijo tomando prestado la variable varVec: {:?}", var_vec);
     });

      // Hilo
     s.spawn(|_| {
         println!("Otro proceso hijo tomando prestado la variable varVec: {:?}", var_vec);
         println!("Me duermo 2 segundos...");
         thread::sleep(Duration::from_millis(2000));
     });

 }).unwrap();

 println!("Este mensaje aparecerá cuando los hijos acaben.");

}
```

3.1.2. Channel

En el siguiente fragmento creamos un canal, sin limites de mensajes en el canal, ‘s‘ y ‘r‘ son el sender y el receiver (productor / consumidor de nuestro canal).

```rust
use crossbeam_channel::unbounded;

// Creamos un canal sin limites de mensajes.
let (s, r) = unbounded();

// Mandamos un mensaje al canal.
s.send("Hola Mundo!!").unwrap();

// Recibimos el mensaje del canal.
println!("{}", r.recv().unwrap()); // "Hola Mundo!!".
```

En caso de querer un canal limitado, se haria de la siguiete manera:

```rust
// Crea un canal que solo puede tener 5 mensajes sin consumir.
let (s, r) = bounded(5);

// Solo se pueden mandar 5 mensajes sin quedarse bloqueado.
for i in 0..5 {
    s.send(i).unwrap();
}

// Otra llamada a "send" bloquearía el canal porque el canal estaria lleno.
// s.send(5).unwrap();
```

También, recordamos lo mencionado sobre los canales, la clonación de estas solo generará un nuevo acceso al mismo canal instanciado previamente:

```rust
use crossbeam_channel::unbounded;

let (s1, r1) = unbounded();

// Clonamos la referencia.
let (s2, r2) = (s1.clone(), r1.clone());
let (s3, r3) = (s2.clone(), r2.clone());

// Enviamos en orden s1, s2, s3.
s1.send(10).unwrap();
s2.send(20).unwrap();
s3.send(30).unwrap();

// Consumimos en orden, r3, r2, r1.
//  pero el orden de los mensajes consumidos
//  sigue siendo el mismo que cuando los producimos.
assert_eq!(r3.recv(), Ok(10));
assert_eq!(r1.recv(), Ok(20));
assert_eq!(r2.recv(), Ok(30));
```

Y lo mismo pasa con las referencias, independientemente desde que hilo se mande / lea el mensaje.

```rust
use crossbeam_channel::bounded;
use crossbeam_utils::thread::scope;

fn main() {
    /***
     * Un caso especial de channel con capacidad cero, que no puede contener ningún mensaje.
     * Las operaciones de envío y recepción deben ejecutarse
     *   al mismo tiempo para sincronizar y pasar el mensaje.
     */

    let (prod, cons) = bounded::<&str>(0);

    scope(|s| {
        // Crea un hilo que se espera a leer un mensaje, y envia otro.
        s.spawn(|_| {
            println!("{}", cons.recv().unwrap()); // => Envio mensaje al hilo hijo.
            prod.send("Envio mensaje al hilo principal.").unwrap();
        });

        // Envia un mensjae y se espera a leer.
        prod.send("Envio mensaje al hilo hijo.").unwrap();
        println!("{}", cons.recv().unwrap()); // => Envio mensaje al hilo hijo
    })
    .unwrap();
}
```

3.1.3. Parker

A continuación, un ejemplo de como pausar un proceso. Cada parker tiene una señal que por defecto está abstente.

Park paraliza el proceso hasta que dicha señal esté disponible, y la consume. Unpark genera una nueva señal para que sea consumida por Park.

Por ello al realizar unpark, le estaremos asignado al proceso la señal, y después al hacer park, dicha señal ya se habrá consumido.

```rust
use std::thread;
use std::time::Duration;
use crossbeam_utils::sync::Parker;

let p = Parker::new();
let u = p.unparker().clone();

// Crea una señal que será consumida por el siguiente p.park().
u.unpark();

println!("Hilo Principal: Me voy a aparcar.");

// Consume la señal y permite que el proceso continue,
//   si no hubiera una señal, el proceso se quedaria
//   aparcado hasta que una señal fuera generada por u.unpark().
p.park();

println!("Hilo Principal: Estoy aparcado!");

thread::spawn(move || {
    println!("Hilo Hijo: Me espero 2 segundos.");
    thread::sleep(Duration::from_millis(2000));
    println!("Hilo Hijo: Desaparco el hilo principal.");
    u.unpark();
});

// El proceso se queda aparcado hasta que algún otro hilo genere
//   una señal para desaparcar.
p.park();

println!("Hilo Principal: Ya estoy de vuelta.");
```

3.1.4. SharedLock

Esta librería permite crear variables con bloqueos de estados de escritura o lectura, y denegar el acceso a estas cuando están en uso por otros procesos, hasta que se desbloquean. Permite crear muchos lectores, pero solo un escritor.

```rust
use crossbeam_utils::sync::ShardedLock;

// Variable con bloqueos.
let lock = ShardedLock::new(5);

// Se pueden crear más de un lector.
{
    let lector1Variable = lock.read().unwrap();
    let lector2Variable = lock.read().unwrap();
    println!("{}", lector1Variable); // 5
    println!("{}", lector2Variable); // 5
} // Los lectores dejan de existir una vez fuera del ámbito.

// Solo se puede tener un escritor por ámbito.
{
    let mut w = lock.write().unwrap();
    *w += 1;
    println!("{}", w); // 6
} // El escritor deja de existir una ve fuera del ámbito.

```

3.1.5. WaitGroup

Permite crear barreras de espera para grupos de subprocesos, al clonar un WaitGroup, este automáticamente se espera a dicho clon, todos los clones y el original tiene que estar dropeados para que wg.wait() pueda continuar.

```rust
use crossbeam_utils::sync::WaitGroup;
use crossbeam_utils::thread::scope;

// Creamos un wg ( barrera ).
let wg = WaitGroup::new();

scope(|s| {

    // Creamos 4 hilos.
    for i in 0..4 {

        // Creamos otra referencia a la MISMA barrera.
        let wg = wg.clone();

        s.spawn(move |_| {

            // Logica del hilo.
            println!("El hilo {} hace cosas..", i);

            // Dropeamos la referencia, dando a entender que ya no necesitamos esperar.
            drop(wg);
        });
    }

    // Se bloque hasta que todas las referencias de dicha barrera ( wg ) estén dropeadas
    //    para continuar.
    wg.wait();
    print!("Todos los procesos han finalizado. :)");

}).unwrap();

```

4. Comparativas con otras librerías

En contraste con las herramientas por defecto que nos provee rust para realizar implementaciones de hilos concurrentes, y manejo de hilos, podemos concluir que crossbeam añade una capa de confort además de otros útiles que nos facilitan el control y las funcionalidades que queremos tener a la hora de manejar hilos.

También es de mencionar que la mayoría de herramientas que substituyen a las herramientas propias del lenguaje son intuitivas de usar por el hecho de que estos utiles han sido desarrollados usando las funcionalidades básicas que provee el propio lenguaje para el manejo de hilos, por ejemplo, sharedLock se podría decir que es una extensión de RwLock, además del spawn de scope comparado con la herramienta thread de Rust.

5. Conclusiones

Lo mejor de Crossbeam se podría decir que es su amplio abanico de útiles con toda clase de funcionalidades, y que estas herramientas no se limitan solo al manejo y uso de hilos, sino se que se pueden usar para otro tipo de proyectos también.

5.1. Conclusión personal.

Personalmente, las herramientas que más me han fascinado ha sido scope y channel, ya que estas 2 en especial proveen de unas funcionalidades que facilita muchísimo la implementación de la concurrencia y la comunicación entre hilos incluso comparándolo con librerías de otros lenguajes como JS o C++.