# Paralelismo en Rust con Crossbeam

Por Elvi Mihai Sabau Sabau

# ¿Qué es 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.

# ¿Qué útiles provee Crossbeam?

La libreria crossbeam provee ( entre otros ) los siguientes útiles:

- Scope
- Channel
- Parker
- SharedLock
- WaitGroup

# Scope

Scope es un objeto que permite crear ámbitos para spawnear hilos. Cuando el bloque de código termina, este asegura que todos los hilos creados dentro del ámbito han finalizado.

```rust
use crossbeam::scope;

scope(|s| {
    s.spawn(|_| {
        println!("Hola desde un hilo hijo!");
    });

    s.spawn(|_| {
        println!("Hola desde otro hilo hijo!");
    });
}).unwrap();

println!("Hola desde el hilo principal.");
```

# 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 o "ilimitado".

¿ Qué sucede cuando no hay ningún mensaje para consumir ?

```rust
use crossbeam_channel::unbounded;

fn main(){

  // Creamos un canal sin límites 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!!".

}
```

# 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 o "ilimitado".

```rust
use crossbeam_channel::bounded;

fn main(){

  // 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 proceso porque el canal estaria lleno.
  s.send(5).unwrap();
}
```

# Channel

```rust
use crossbeam_channel::unbounded;

fn main(){

    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.
    println!("{}", r3.recv().unwrap()); // 10
    println!("{}", r2.recv().unwrap()); // 20
    println!("{}", r1.recv().unwrap()); // 30
}
```

# Channel

## Comunicación entre hilos ( canal sincronizado ).

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

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

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

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

# Parker

Parker es un útil usado para bloquear y desbloquear la ejecución de los subprocesos ( poner en espera o continuar procesos ).

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.

# 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.");
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.");
```

# Shared Lock

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

```
let lock = ShardedLock::new(5);

{   // Bloque de lectura, se puede tener más de un lector, una vez acabado el ámbito, los lectores se destruyen.
    let lector1_variable = lock.read().unwrap();
    let lector2_variable = lock.read().unwrap();
    println!("{}", lector1_variable); // 5
    println!("{}", lector2_variable); // 5
}

{   // Bloque de escritura, se puede tener solo un escritor, una vez acabado el ámbito, el escritor se destruye.
    let mut w = lock.write().unwrap();
    *w += 1;
    println!("{}", w); // 6
}

{
    let lector1_variable = lock.read().unwrap();
    println!("{}", lector1_variable); // 5
}
```

# Wait Group

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.

# Wait Group

```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 {

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

        s.spawn(move |_| {
            println!("El hilo {} hace cosas..", i); // Lógica del hilo.
            drop(wg);  // Dropeamos la referencia, dando a entender que ya no necesitamos esperar.
        });
    }

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

}).unwrap();
```

# Conclusión

¿ Preguntas ?