

Concurrencia en Go

Los procesos concurrentes en Go se lanzan añadiendo la palabra reservada “go” a una llamada a función, son las **goroutines**:

```
package main

import "fmt"

func main() {
    go fmt.Println("Hello, GDG Marbella!")
}
```

Como se podrá comprobar no se imprime ningún mensaje al ejecutar el programa. Esto es debido a lo que se conoce como **race condition** y ocurre cuando se desea que dos operaciones se desarrollen de forma secuencial pero su naturaleza concurrente no asegura que se respete esta secuencialidad. En este caso queremos que el primero se imprima el mensaje y luego termine el programa, y no al contrario, como ocurre en el anterior ejemplo.

¿CÓMO RESOLVER ESTA SITUACIÓN?

Existen varias opciones, pero la más corriente es utilizar una utilidad de sincronización llamada **WaitGroup**.

WaitGroup es un hito de espera que bloquea la ejecución de un proceso (wait) hasta que el número de procesos (add) que se le hayan notificado hayan concluido su tarea (done). Este es un caso simple en el que sólo esperamos a la conclusión de dos procesos (padre e hijo), pero realmente está pensado para la sincronización de más como veremos a continuación.

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup

    wg.Add(1)

    go func() {
        fmt.Println("Hello, GDG Marbella!")
        wg.Done()
    }()

    wg.Wait()
}
```

Para el paso de mensajes entre procesos Go nos ofrece una solución bastante intuitiva en su diseño: los **channels**. Podemos verlos como conductos que unen una función/proceso con otra/o. También vemos como se sincronizan dos procesos.

```
func main() {
    var wg sync.WaitGroup

    wg.Add(2)

    channel := make(chan string)

    go sender(channel, &wg)
    go receiver(channel, &wg)

    wg.Wait()
}

func sender(channel chan<- string, wg *sync.WaitGroup) {
    channel <- "Hello, GDG Marbella!"

    wg.Done()
}

func receiver(channel <-chan string, wg *sync.WaitGroup) {
    fmt.Println(<-channel)

    wg.Done()
}
```

Podemos querer también enviar varios mensajes a través de varios senders, para eso tenemos que iterar sobre un **slice** ... y sobre el propio canal para leer todos los mensajes.

```
func main() {
    var wg sync.WaitGroup

    channel := make(chan string)

    defer close(channel)

    messages := []string{
        "Hello, GDG Marbella!",
        "Hello, Gophers!",
        "Hello, World!",
    }

    for _, msg := range messages {
        wg.Add(1)

        go sender(msg, channel, &wg)
    }

    go receiver(channel)

    wg.Wait()
}

func sender(msg string, channel chan← string, wg *sync.WaitGroup)
{
    channel ← msg

    wg.Done()
}


func receiver(channel ←chan string) {
    for msg := range channel {
        fmt.Println(msg)
    }
}
```

Imaginemos ahora que queremos que nuestras goroutines accedan a un determinado recurso: queremos indexar nuestro slice de mensajes; la solución de al lado no es válida. Puede darse el caso que una goroutine quiera escribir mientras otra lo está haciendo. Para esto hemos de utilizar mutex, que nos permite bloquear un elemento mientras estemos escribiendo en él. Su uso más común es situándolo en un objeto que contenga el recurso:

```
type msgMap struct {  
    values map[int]string  
  
    sync.Mutex  
}
```

```
var wg sync.WaitGroup  
  
msgMap := make(map[int]string)  
  
messages := []string{  
    "Hello, GDG Marbella!",  
    "Hello, Gophers!",  
    "Hello, World!",  
}  
  
for i, msg := range messages {  
    wg.Add(1)  
  
    go func() {  
        msgMap[i] = msg  
  
        wg.Done()  
    }()  
}  
  
wg.Wait()  
  
for i := range messages {  
    fmt.Println(msgMap[i])  
}
```

Cada goroutine primero bloquea el objeto, escribe en él, y luego lo desbloquea para que la siguiente goroutine pueda escribir. Nuestro for anterior quedaría, con el uso del mutex, convertido a lo siguiente:



```
for i, msg := range messages {  
    wg.Add(1)  
  
    go func(msg string, index int) {  
        defer wg.Done()  
  
        msgsIndexed.Lock()  
        msgsIndexed.values[index] = msg  
        msgsIndexed.Unlock()  
    }(msg, i)  
}
```