Generator Pattern

Let's study our first pattern of concurrency, which is about generators that return channels as returning argument.

In this chapter, we'll look at some concurrency patterns that can come in handy while we write concurrent code.

Let's begin with generators. Generators return the next value in a sequence each time they are called. This means that each value is available as an output before the generator computes the next value. Hence, this pattern is used to introduce parallelism in our program.

Have a look at the simple example below:

```
package main

import (
        "fmt"
)

func foo() <-chan string {
        mychannel := make(chan string)

        go func() {
                for i := 0; ; i++ {
                        mychannel <- fmt.Sprintf("%s %d", "Counter at : ", i)
                }
        }()

        return mychannel // returns the channel as returning argument
}

func main() {
        mychannel := foo() // foo() returns a channel.

        for i := 0; i < 5; i++ {
                fmt.Printf("%q\n", <-mychannel)
        }

        fmt.Println("Done with Counter")
}
```

In the code above, we use a generator as a function which returns a channel. The `foo()` function, when invoked on **line 20**, calls a goroutine which sends a string onto the channel on **line 12**. The string contains the value of `i` which is incremented in every iteration before being sent to the channel. As the goroutine executes concurrently, the program returns `mychannel` immediately to the main routine. Then `mychannel` receives the data on the channel one by one on **line 23** as it is being sent by the goroutine. In conclusion, both the goroutine and the main routine can execute concurrently as we print the value of the counter as soon as we receive it while the goroutine simultaneously computes the next value.

I hope you are with me so far. Let's extend this pattern to a concept where we think of the channel as a handle on a service.

```
package main

import "fmt"

func updatePosition(name string) <-chan string {
	positionChannel := make(chan string)

	go func() {
		for i := 0; ; i++ {
			positionChannel <- fmt.Sprintf("%s %d", name , i)
		}
	}()

	return positionChannel
}

func main() {
	positionChannel1 := updatePosition("Legolas :")
	positionChannel2 := updatePosition("Gandalf :")


	for i := 0; i < 5; i++ {
		fmt.Println(<-positionChannel1)
		fmt.Println(<-positionChannel2)
	}

	fmt.Println("Done with getting updates on positions.")
}
```

In the code above, we are getting updates on the position of Legolas and

Gandalf using the `updatePosition` function. We again launch the goroutine

from inside the function, which is the key thing to notice in both the examples we have seen so far. Thus by returning a channel, the function enables us to communicate with the service it provides which, in this case, is giving position updates.

Moreover, we can have more than one instance of the function, which can also be seen in the example above, as we get position updates for both Legolas and Gandalf.

However, we still have a slight problem, which is when the following statements are blocking each other:

```
    fmt.Println(<-positionChannel1)
        fmt.Println(<-positionChannel2)
```

We will resolve this issue in the upcoming lessons.

Now, let’s move to a more interesting example where we print the Fibonacci sequence up to the nth term.

```
package main

import "fmt"

func fibonacci(n int) chan int {
    mychannel := make(chan int)
    go func() {
        k := 0
        for i, j := 0, 1; k < n ; k++ {
            mychannel <- i
            i, j = i+j,i

        }
            close(mychannel)
    }()
    return mychannel
}

func main() {

    for i := range fibonacci(10) {
       //do anything with the nth term while the fibonacci()
       //is computing the next term
        fmt.Println(i)
    }
}
```

I will not be going in depth to explain the logic of the code given above and only discuss it from a concurrent perspective. The crux of the solution lies in the part where we return a channel from the function on **line 16** to the main routine on **line 21** which allows the `fibonacci()` function to communicate each and every term to the main routine as soon as it is computed in the function body. In this way, when we print a term in the Fibonacci sequence, this pattern enables us to do whatever we want to do with each term while it is concurrently computing the next term. Isn't that great?

I hope you enjoyed this lesson so stay tuned as there are many more to come. In the next lesson, we will have a look at another important pattern: Fan-In, Fan-Out.