Concurrency and channels in Go

In this lesson, we explore concurrency in Go and the use cases for using goroutines and channels.

Prerequisites

- Basic understanding of Go syntax
- Basic understanding of concurrency

Session 1

What is Concurrency?

In simple terms, Concurrency is the ability of computers to execute multiple tasks serially, while giving the user the impression that tasks are executed parallelly. For example, modern-day computers allow video chat, internet browsing, and application upgrades all at the same time by interleaving tasks with one another.

A modern day computer has multiple cores, where each core can run independent tasks parallely. Each core can only run one task at a time, but is able to interleave multiple processes in a given amount of time.

How can Concurrency be achieved in Go?

While computers today have the ability to do concurrent tasks, the applications developed aren't automatically made concurrent by using serial programming techniques. For example, the code below is not serial and doesn't get the two names concurrently.

```
func getName() string {
     firstName := getFirstNameFromDatabase()
     secondName := getSecondNameFromDatabase()
     return firstName + " " + secondName
}
```

So, applications now-a-days need to be made concurrent explicitly by using the language's concurrency paradigms.

Concurrency in Go is achieved through *goroutines*. The Go syntax allows for goroutines to be created easily. Goroutines are very light-weight (~2KB) and millions of them can be run on a machine at the same time. While the internal implementation of a goroutine is abstracted away from a Go developer, internally, there is a many-to-one relationship between goroutines and OS threads, with the Go scheduler mapping several goroutines on one or few threads.

The following sample code shows how goroutines are created in Go:

```
func doTasks() {
     go func() {
          fmt.Printf("Do task 1")
     }()

     go func() {
            fmt.Printf("Do task 2")
      }()

      fmt.Printf("Dispatched tasks.")
}
```

The above function creates two goroutines and exits without waiting for the completion of the two tasks. By creating goroutines, it hands over the responsibility of executing these two tasks to the goroutine scheduler, which will make sure that those tasks will be executed at a later time.

If the above piece of code is run by itself, the program might exit before executing the goroutines at all. To ensure the timely execution of the goroutines, Go has wait groups that can be used to wait for goroutines. Below is a way to ensure that goroutines finish execution before *doTasks* returns:

```
func doTasks() {
      var wg sync.WaitGroup
      wg.Add(1)
      go func() {
            defer wg.Done()
            fmt.Printf("Do task 1")
      }()
      wg.Add(1)
      go func() {
            defer wg.Done()
            fmt.Printf("Do task 2")
      }()

      fmt.Printf("Dispatched tasks.")

      wg.Wait()

      fmt.Printf("Done tasks")
}
```

Sharing memory among goroutines

Goroutines can run independently only if they don't share memory. To share memory, goroutines need to synchronise amongst themselves using synchronization libraries provided by Go. If memory is shared among goroutines without synchronization, it may lead to race conditions which are typically hard to debug. Below is an example of how goroutines without synchronization can lead to data races.

```
func raceExample() {
      n := 0

      go func() {
            for i := 0 ; i < 100000 ; i++ {
                  n++
            }
      }()

      go func() {
            for i := 0 ; i < 100000 ; i++ {
                  n++
            }
      }()

      time.Sleep(time.Second)
      fmt.Printf("%d\n", n)
}
```

If the above example was executed serially, *n* would be 200000. However, the increments are done concurrently and each increment involves various steps (read, increment and write). These steps are interleaved, eventually leading to *n* being less than 200000.

As with any other language, Go provides a way to lock memory access using `mutex`. Below is an example of how goroutines can use mutex.

```
func fixRaceExample() {
      n := 0
              l := &sync.Mutex{}
      go func() {
            for i := 0 ; i < 100000 ; i++ {
                                        l.Lock()
                  n++
                                        l.Unlock()
            }
      }()

      go func() {
            for i := 0 ; i < 100000 ; i++ {
                                        l.Lock()
                  n++
                                        l.Unlock()
            }
      }()
```

```
    time.Sleep(time.Second)
    fmt.Printf("%d\n", n)
}
```

In the above example, *n* is 200000 as the 2 goroutines are synchronised in the way they access the variable *n*.

Channels

Channels are first-class citizens in Go i.e they are initialised and garbage collected just like any other primitive variable (`int`, `string` etc.). In simple terms, channels can be thought of as an in-memory thread-safe queue to which elements can be pushed into and popped from.

Below is how we can read from and write into channels.

```
ch := make(chan int)

// write 1 to channel
ch <- 1

// receive from channel
<-ch
```

Buffered/Un-buffered Channels

By default, a channel is unbuffered in Go. Below is how to declare an unbuffered channel:

```
newCh := make(chan int)
```

A write to an unbuffered channel is blocking. Therefore, a write to an unbuffered channel without a corresponding `read` currently running could lead to the `write` being blocked forever.

To prevent such a deadlock, Go allows for declaring buffered channels. Below is how we can initialise a buffered channel with a buffer size of 5.

```
newCh := make(chan int, 5)
```

A buffered channel implies that `newCh` can hold a maximum of 5 `ints`. Any write to a buffer that is already full will be blocked until a reader reads the channel.

Channel Use-cases

Just like mutexes, channels are used for synchronisation among goroutines. However, channels can be used to synchronise goroutines in a lot more ways than locking/unlocking a mutex. Below is a common use-case where channels come in handy:

```
func producerConsumer() {
	ch := make(chan int, 5)

	// Producer
	go func() {
		num := 0
		for {
			num++
			if num == 10 {
				close(ch)
				break
			}

			<- time.After(1 *time.Second):
			ch <- 1
		}
	}()

	// Consumer

	go func() {
		for {
			select {
			case _, ok := <-ch:
				fmt.Println("Received work")
				if !ok {
					return
				}
			}
		}
	}()

	time.Sleep(100 * time.Second)
}
```

The above program has two goroutines - *producer* and *consumer* that communicate through channels. Once the producer is done producing tasks into the channel, it closes the channel. Using an additional parameter (*ok* in above example), the channel closure notification is received by the consumer I.

Common pitfall using Go concurrency and channels

Receiving from closed channel

Consider the seemingly harmless piece of code below:

```
func incorrectRecv(recvCh chan int) {
	for {
		select {
			case res := <-ch:
				fmt.Printf("Received %d", res)
		}
	}
}
```

In the example above, the function receives continuously from the *recvCh* channel. Now, consider the scenario that the producer closes the channel after sending a few results. What do you think will happen to the receiver when the channel is closed?

In the case the channel is closed, the `for` loop will run infinitely with the select statement returning the zero value immediately. Thus, reading in such a way will unintentionally hog the CPU.

Accidentally Shared Variables

A novice Go programmer would expect the below piece of code to print integers in ascending order:

```
func main() {
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go func() {
			fmt.Printf("%d ", i)
			wg.Done()
		}()
	}

	wg.Wait()
}
```

However, all the goroutines above share the same variable *i* by closure. Hence, it is highly unlikely for the above program to print digits in ascending order and highly likely for a lot of the goroutines to print the same number. A quick fix is to pass the variable as a parameter to the goroutine instead of sharing it by closure.

```
func main() {
	var wg sync.WaitGroup
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go func(i int) {
			fmt.Printf("%d ", i)
			wg.Done()
		}(i)
	}

	wg.Wait()
}
```

Send to closed channel

A send to a closed channel results in a panic. So, it is the developer’s responsibility to ensure that their code does not send to a closed channel as doing so can be catastrophic.