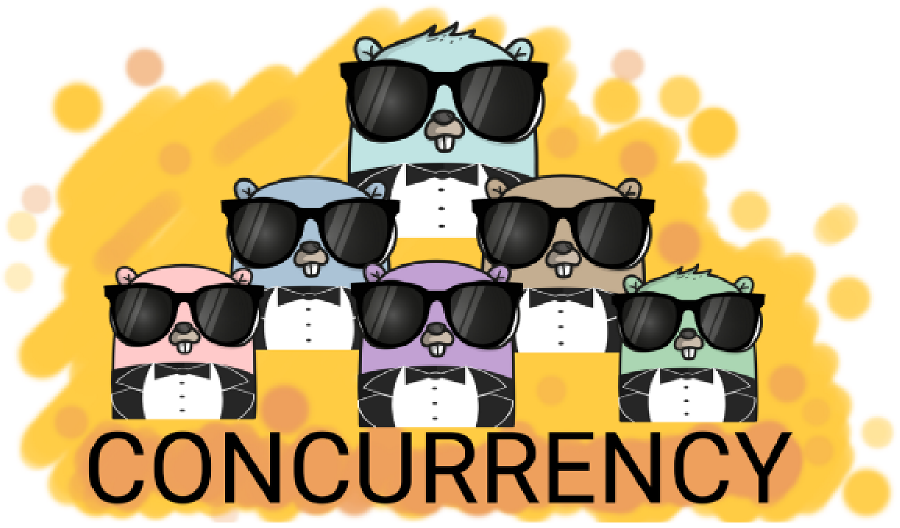
Go: A Tale of Concurrency (A Beginners Guide)



Gopher Concurrent Gang

A long time ago…

When the computers were invented, they needed instructions from their human creators. We created specific languages to talk to them, to tell them what to do, when to do and how to do. We would tell them step by step what their tasks were. For a time being this approach was working. the humans and their computers were very happy.

Soon the humans created something called the Cloud. It was basically a bunch of computers put together, which would help humans solve larger, complex problems. With a large array of computers, meant a large array of processors all waiting for tasks from their human creators. But our language of code was only suitable for one processor at a time. We would tell the computer what to do, one of the processors would pick up our instruction list and work on it—the rest of the processors were idle.

Now we needed to utilize the full extent of computing power that was available when we mashed array of computers together. The concept of concurrency comes into the picture here.

> How do we utilize all the cores on a computer to run our program faster?

Most of the languages were invented when the computers had only one processor, with the advent of Cloud, the languages had to be modified to enable concurrency. Even then, a new problem arose — our thinking was sequential, we all write code/logic in a sequential manner. How do we adapt our thinking in concurrent terms?

We started designing libraries, frameworks and methodologies to overcome this very problem. Then it struck us — Hey! Why don't we design a language that in and itself is adapted to concurrency, rather than change existing languages to enable concurrency.

Enter Go! The language which is best suited to solve problems concurrently.

This article aims to explain the primitives of Concurrency in Go.

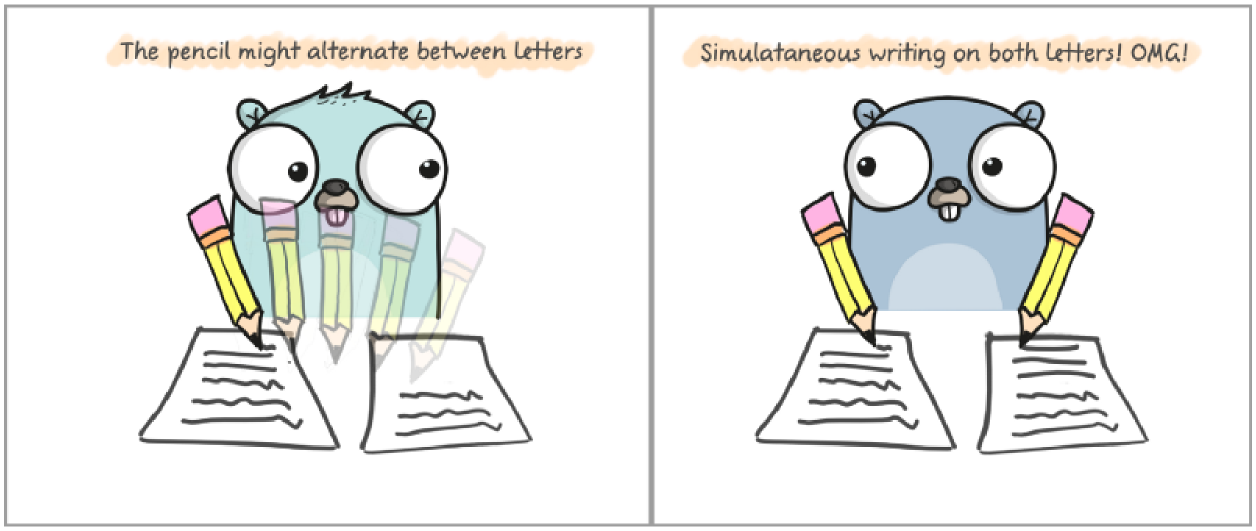
If you just need to learn the primitives and not bother about brief explanation of concepts, kindly scroll down to the tutorial part (Concurrency Guide).

Before we begin, let's resolve the basic confusion we have encountered

What is Concurrency? Is it the same as Threaded? Parallel? Asynchronous?

Concurrency in its simplest form can be defined as doing multiple things at once. Imagine writing two letters with both hands when you have one pencil, you write a sentence with your right hand on one letter, then write a sentence with your left hand on other letter. You might keep alternating hands, until both the letters are complete. — ***Doing multiple tasks, by sharing time, resources etc.***

Parallelism means starting the tasks, completing the tasks together. Imagine writing the letters with both hands with a pencil in each hand without interruption. This would require a lot of brain power and practice. — ***Doing multiple tasks "at the same time"***


The pencil might alternate between letters
Simulataneous writing on both letters! OMG!

Concurrency and Parallelism

In the programmers world, we define the concurrency as the code property, and the parallelism as machine property. Meaning the developer will write a concurrent code, the machine in the end decides whether it will run parallelly or not.

> A Programmer writes a Concurrent Code.

> A Programmer prays the code runs in parallel.

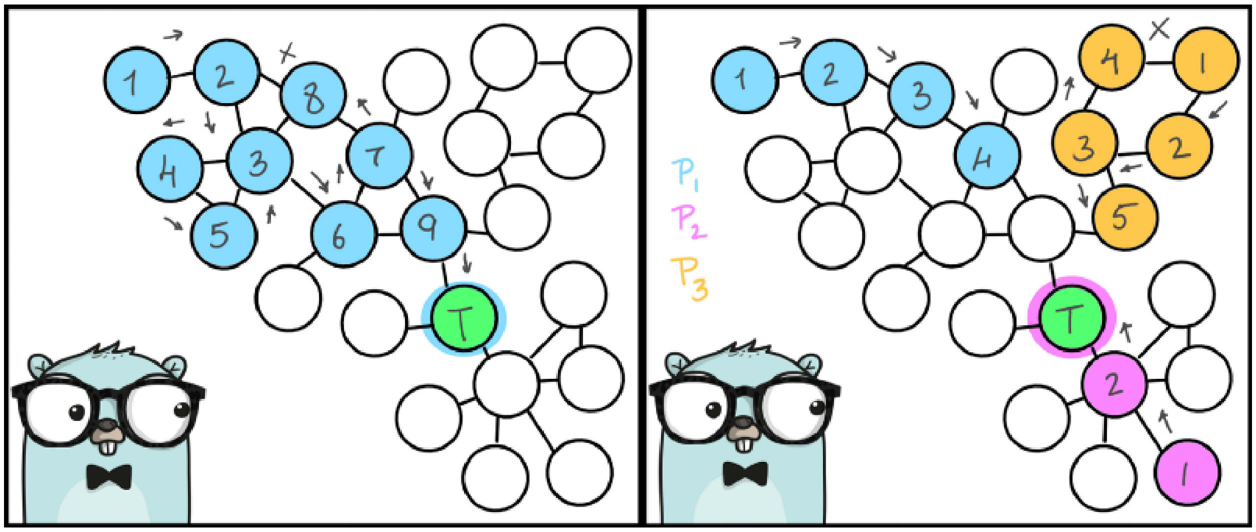
Thread Programming is usually how we implement the concurrency.

Asynchronous Programming is a coding style that gives us the impression that things are being done concurrently *(Might or might not be done, behind the scenes)*

Wait a minute! Why is this important? And Why should I care?

Being a developer used to be simple. You would learn logic, you would learn a coding language, you would write a program to tell the dumb machine what to do. The code would be a sequential set of operations performed by the machine. Most of us still rely on the good old sequential thinking to design logic and code.

Sequential programming isn't bad, but you cannot utilize the full power of what a multiple core provides. Imagine you design a graph search algorithm, your algorithm starts at a single point in graph and works it's way towards the target node. Now think how much faster it would be if you had 4 cores, performing search operations on different parts of graph simultaneously.



Awesomeness Unleashed!

With the advent of cloud computing, the processing power has increased vastly. What you can achieve with multiple cores, greatly overshadows what you can achieve with a single core. Now is the time to take advantage of the computing power.

OK! But why Go?

Traditionally programming concurrency—has been done using threads, most programming languages offer a thread library to implement concurrency. You would do something like this

1. Think how a program might utilize threads
2. Implement a code which utilizes threads
3. Find out that you are not getting the desired output, due to various reasons like deadlocks, invalid memory access, no synchronous operations, no communication between threads …
4. Figure out what is the real issue—fix the issue, check again
5. Repeat step 3 and 4 until the program runs as you desire
6. Watch the program become so complex, you couldn't explain it, even if you tried to.

More importantly the Thread/Concurrent operations in all languages are dependent on some common factors like

- How the language handles OS threads ?
- What's the size of the thread stack available ?
- How efficient is the library in handling it's thread pool ?
- How many cores would be available for thread operations ?

Go offers the following advantages in Concurrency

- Lightweight Threads
- No OS Threads, instead we get virtual threads multiplexed onto OS threads
- Channels for communicating between concurrent parts
- Cleaner and simpler implementation than most other languages
- Go Runtime and Scheduler work hand in hand to manage the implementation efficiently.

A Concurrency Guide

Now armed with the knowledge—that Go is the Way for Concurrency in future. Let's understand the basic primitives of it.

We have 3 basic concepts in implementing concurrency in Go:

1. Go Routines
2. Blocking Operations
3. Channels (Buffered & Unbuffered)

Let's understand each of them with a simple example of a restaurant

> We have a restaurant, we give an order, the Waiter takes the order from us, gives it to a Chef. The Chef cooks the meal, hands over the food to the Waiter, who brings it to us

processOrder()
chefCooks()
processOrder()

Sequential Processing

A simple sequential program in Go might look like this

The output is something like this

```
Waiter A takes order 0 to chef Mark
Chef Mark cooks order 0
Waiter A brings order 0 from chef Mark
Waiter C takes order 1 to chef Mark
Chef Mark cooks order 1
Waiter C brings order 1 from chef Mark
Waiter A takes order 2 to chef Bob
Chef Bob cooks order 2
Waiter A brings order 2 from chef Bob
Waiter C takes order 3 to chef Bob
Chef Bob cooks order 3
Waiter C brings order 3 from chef Bob
Waiter A takes order 4 to chef Bob
Chef Bob cooks order 4
Waiter A brings order 4 from chef Bob
```

Do you see the issue here?

- We get the waiter and chef selected, sequentially the waiter takes the order to chef, the chef cooks it, then the waiter brings it back
- As our customers increase, the ***totalOrders*** increases, the orders are processed sequentially, **so the last order has to wait until all previous orders are processed**

Go Routines & Blocking Operations

Goroutines are the most basic way concurrency is implemented in Golang. They are basically functions which have been prefixed with the keyword `go`

Luckily the code can be converted to concurrent execution in Go using goroutines. Look at the following code.

Now we have added the keyword go to the 3 calls of operations. We notice two changes

- There's something called runtime.GOMAXPROCS(1)

```
This helps us control how many processors can be used for threading, it usually defaults t
o the number of cores available. We have set it to 1, meaning there is 2 OS thread availab
le to us for concurrency.
```

```
NOTE: This call has to be carefully considered during production, set it to the number of
 cores in your CPU. Also Go Team has said that this call will be removed as the Go Schedul
er improves
```

- There's something called <-time.After(time.Second*5)

```
This is called Blocking Operations. The main function in a Go program is also a goroutine
 (main goroutine). When we run concurrent operations in Go, the function execution begins,
and the control passes back to the main.
So when we don't include this line, nothing is printed on the screen, as the control comes
back to main and finds nothing to execute, so it simply exits the main goroutine.
```

```
When the main goroutine exits, the children routines are exited as well. Yay! As we don't
 have to worry about Orphans.
```

You can achieve this via ***time.Sleep(5), a for { } or something called WaitGroup(waits for goroutines to complete)*** as well. Since ***WaitGroup*** involves more code and signature changes, I felt this was a neater way.

```
NOTE: WaitGroups are proper way of doing things. Examples used here are for quick playing
 around! I have also included the WaitGroup example links, for reference
```

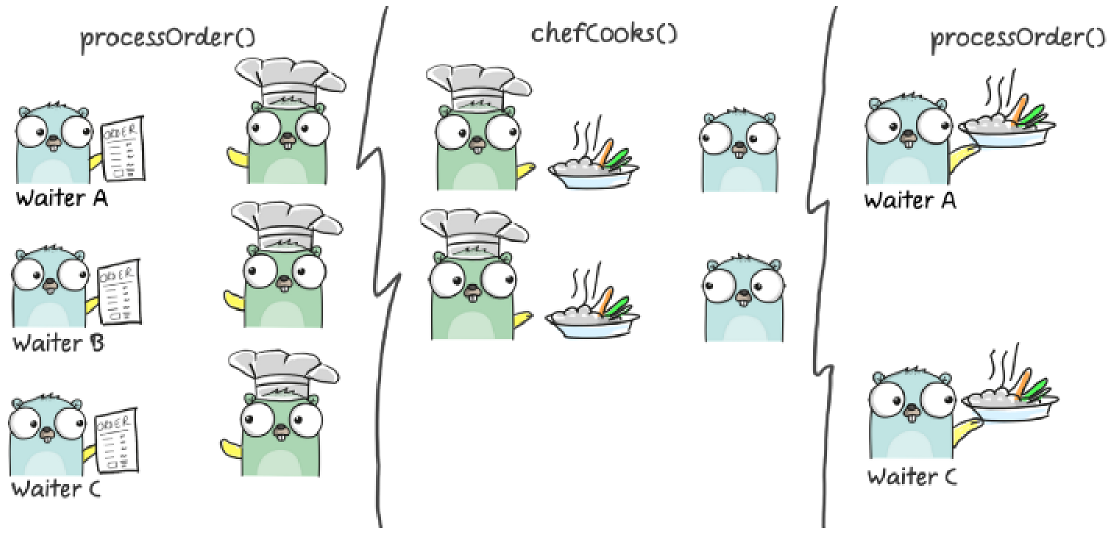
You can find the code using WaitGroup here

Let's look at the output, you will see something like this

```
Waiter A brings order 4 from chef Bob
Waiter A takes order 0 to chef Mark
Chef Mark cooks order 0
Waiter A brings order 0 from chef Mark
Waiter C takes order 1 to chef Mark
Chef Mark cooks order 1
Waiter C brings order 1 from chef Mark
Waiter A takes order 2 to chef Bob
Chef Bob cooks order 2
Waiter A brings order 2 from chef Bob
Waiter C takes order 3 to chef Bob
Chef Bob cooks order 3
Waiter C brings order 3 from chef Bob
Waiter A takes order 4 to chef Bob
Chef Bob cooks order 4
```

Everything is great except the last chunk there, **the order 4 is brought to the customer before it gets cooked?** What magic is this?

As I said, the go routine is concurrent, which means whatever we have called as go routine is executed simultaneously by switching context. The second ***processOrder*** on line 46, must have finished first and printed, before the others had chance to execute.



Goroutines at work

This brings us to the important part **Communication between the processes.** This is achieved using the Channels

Channels

A Channel is simple data structure which is shared between the goroutines for communication. Think of it as a notification you get for any updates on your favourite apps. Something new has happened, you get a notification.

Similarly when a process has done something exciting, before handing off control to another process, it allows the other processes know what's up. This is summarized in the Golang motto

> Don’t communicate by sharing memory; share memory by communicating

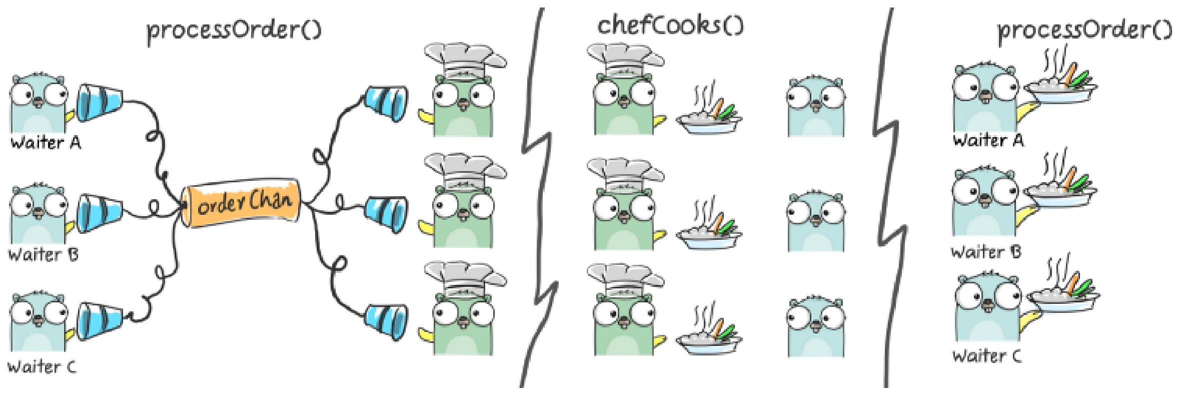
This means that the processes have to talk to each other before utilizing variables, memory or resources. Consider the following modified code

We see some changes in this code

- A struct which will tag our chef with an order, meaning we now know which chef has which order
- Notice the function signature of ***chefCooks*** and ***processOrder*** has changed. This allows us to tell the functions to talk to each other, over a channel.

```
NOTE:
Earlier we used to pass process type[takes/brings], order number, waiter and chef details.
Now we only pass process type, order and channel.
```

- Meaning the Chef will wait for someone to post order over the channel, only then will he cook the order.



Single Channel Implementation

You will get an output like this, it might vary with execution.

```
Waiter A takes order 0 to chef Mark
Chef Mark cooks order 0
Waiter C brings order 0 from chef Mark
Waiter A takes order 1 to chef Bob
Chef Bob cooks order 1
Waiter C brings order 1 from chef Bob
Waiter A takes order 2 to chef Bob
Chef Bob cooks order 2
Waiter B brings order 2 from chef Mark
Waiter C takes order 3 to chef Jack
Chef Jack cooks order 3
Waiter C brings order 3 from chef Bob
Waiter C takes order 4 to chef Jack
Chef Jack cooks order 4
Waiter C brings order 4 from chef Mark
```

Now this is only a single channel. i.e. The Chef will keep waiting until has has an order, the waiter keeps posting new order to the channel.

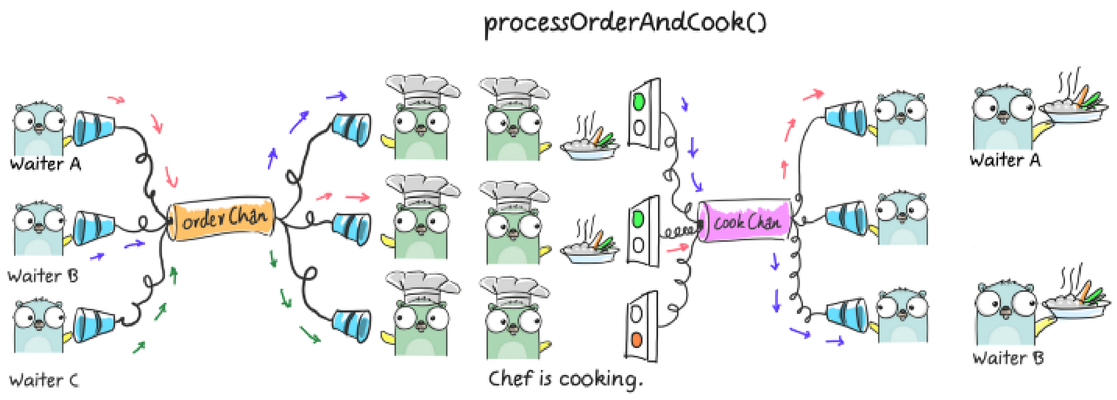
We can simplify the entire process using two channels.

1. **orderChan—**Channel where ***Waiter pushes the order, Chef listens on it***, if the Chef gets new order here, he/she will cook the meal.
2. **cookChan**—Channel where ***Chef pushes meal after he finishes cooking, the Waiter listens on it,*** if Chef gives cooked meal, the Waiter delivers it.

Check out the following code.

The output will be same as the previous case, except the code is shorter.

- We reduced the 3 concurrent function calls to one function which runs concurrently, monitoring the channels.
- We use the select { } block to monitor the channel.
- Waiters and Chefs work in tandem to deliver the delicious meal to you.



Two Channels

Congratulations! We have a restaurant up and running! Logic wise! Code wise!

A full example using WaitGroups can be found here

A Buffered Channel, is similar to the Unbuffered Channel that we implemented, but uses a blocking mechanism.

In our case, the Chef will keep cooking as long as he gets the order from the channel. This isn't an ideal case, as our Chef is Human (Or an incredibly talented Rat).

Chefs get tired, so a limit will be put on the channel, say a Chef can only cook 3 dishes at a time. So the function will block if the channel is full. Until the Chef finishes cooking the dishes in the list.

Chefs huh!

So, here we are, fully equipped with the knowledge of how concurrency is implemented in Golang. Now using this knowledge you can explore the **Concurrency Patterns** (how cool applications can be developed using it) Check out the following links

- Concurrency Patterns
- Concurrency Patterns Video
- Multiple CPU in one Board?

That's it for now! Until next time! See you all on the other side! Be Safe! Code safe! Bye!

P.S—Thanks to Nicholas Eden for some excellent tips on WaitGroups. Also here's the **link** where you can create your own Go Beaver (Gopher)