Published in Level Up Coding

Yair Fernando Follow

May 24 · 12 min read · Listen

Concurrency in Golang, Goroutines, and Channels Explained

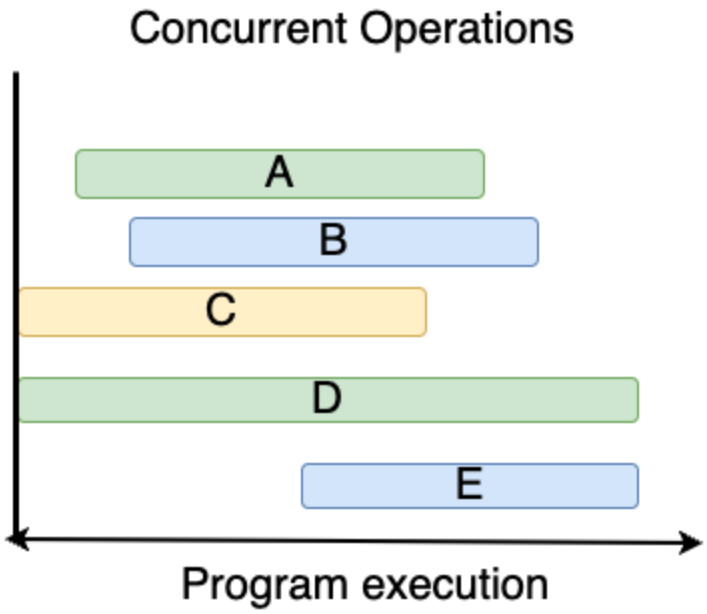
Master concurrency in Go

Go is a powerful language because it handles several things effectively and efficiently, and one of those things that make it fascinating and powerful is how it handles concurrency.

In this article, I'll aim to explain fully the core concepts of concurrency, and the approach that Golang follows to accomplish concurrent programs.

Let's start by defining what is concurrency:

A program is considered concurrent if it can deal with multiple tasks at the same time. The concept of concurrency involves the ability of a program to run several operations concurrently, even when this does not necessarily mean that these operations are running explicitly at the same time, each task can start at a different point in time. Let's see a diagram to better grasp what concurrent operations would look like.



These operations might not start at the same time, but they are running concurrently with each other. This means that a task does not have to wait until another task finish before running.

On the other hand, if your program runs multiple operations at the same time, meaning they start exactly at the same point in time, that would be considered parallelism. When using concurrency your program can also achieve parallelism, depending on the use cases.

. . .

Golang approaches concurrency using **goroutines.** A goroutine is managed by the Go runtime and is pretty similar to a thread, but with several advantages. Goroutines allow you to run multiple operations concurrently. In a multithreaded environment, to run various operations concurrently a new thread has to be created by the OS, which involves a considerable amount of resources, memory, and time, so running multiple operations concurrently using threads is more expensive for the OS. On the other hand, a goroutine is lightweight, efficient and it does not cost too many resources to be created, spinning up several hundreds of goroutines is not a problem in Go.

Shared resources with goroutines

When running multiple goroutines to complete different tasks, very often than not you'll find that goroutines need to access and modify shared resources, if multiple goroutines are accessing and modifying the same data at the same time this will lead to several problems, unexpected results and what is called race conditions.

Let's define what is a race condition in the context of a concurrent program.

A race condition occurs when multiple operations running concurrently attempt to read/write the same data at the same time.

since a key component when implementing concurrency is ensuring that your program will not end up with unexpected results.

For now, let's write our first concurrent program and see how to create a goroutine using the go keyword:

```go
package main

import "fmt"

func main() {
  fmt.Println("Main function")

  go countNumbers(20)

  fmt.Println("End main function")
}

func countNumbers(limit int) {
   num := 0
   for i := 1; i < limit; i++ {
     num+=i
   }
  fmt.Println("Num: ", num)
}
```

main.go hosted with ❤ by GitHub view raw

all your code is executed, but if you want to run another operation in a separate goroutine, we can use the **go** keyword before the function we want to run concurrently and that will effectively create a new goroutine and run that function.

In line 8 we are telling the program to create a new goroutine for the **countNumbers** function. Then going back to the main goroutine, in line 10 there is another print statement.

So, why don't we see the print statement that is inside the **countNumbers** function?. Well, it is because the main goroutine does not wait for other goroutines to finish their work, the main goroutine will continue the execution of the main program and it will terminate without waiting to see if other goroutines have finished.

To allow the **countNumbers** function to finishing we can sleep 1 second in the main go routine in line 9.

```
time.Sleep(1 * time.Second)
```

If you run the program again you'll see the print statement now, this solution is

. . .

What is a channel?

A channel is a way of communication between different goroutines. It is a safe way of sending values of one data type to another goroutine. A channel is passed by reference meaning that when creating a channel and then passing it to other functions, these functions will have the same reference pointing to the same channel. If you understand how pointers work, this might be simple to understand with channels.

We can compare channels only if they have the same type and as I previously mentioned since they are passed by reference, a comparison between two channels will evaluate true if both are pointing to the same reference in memory. We can also compare a channel with nil.

The purpose of a channel is to allow goroutines to send and receive information, but frequently they are also used to inform other goroutines
that a process has finished and not necessarily sending any information through the channel.

A channel can also be closed, meaning it will no longer accept any more messages to be sent or received and if a goroutine tries to send or receive a

Type of channels

Unbuffered Channels: This type of channel only allows to send one piece of data and blocks the current goroutine until another one performs a receive operation on the channel. The same thing will happen if a receive operation on a channel is performed before a send operation, the goroutine where the receive operation was made will be blocked until another goroutine sends a message through the same channel.

To demonstrate this blocking concept when using unbuffered channels, let's see the following example:

```
1  package main
2
3  import (
4          "fmt"
5          "time"
6  )
7
8  var (
9          defaultTags = []string{"SystemUser", "User", "NewUser", "System"}
10 )
11
12 type Tag struct {
13         Name, Type string
14 }
15
```




```
type Post struct {
        Title  string
        Status string
        UserId string
}

func main() {
        blocking()
}

/*
Main goroutine will be blocked until second goroutine
sends a message letting the main goroutine know that has finished its work
and so the main go routine can continue
*/
func blocking() {
        user := &User{}
        done := make(chan bool) // unbuffered channel

        go func() {
                fmt.Println("[Second-GoRoutine] Start Building User")
                buildingUser(user)
                fmt.Println("[Second-GoRoutine] Finished Building User")
                done <- true

                fmt.Println("[Second-GoRoutine] Set default user tags")
                setDefaultTags(user)
        }()

        fmt.Println("[Main-Goroutine] Start importing Posts")
        posts := importingPosts()
        fmt.Println("[Main-Goroutine] Finish importing Posts")
```

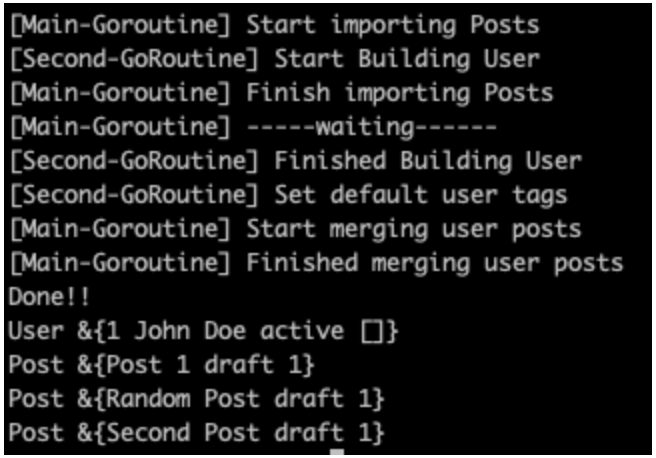
```
58          fmt.Printf("User %v\n", user)
59          for _, post := range posts {
60                  fmt.Printf("Post %v\n", post)
61          }
62  }
63
64  func mergeUserPosts(user *User, posts []*Post) {
65          fmt.Println("[Main-Goroutine] Start merging user posts")
66          for _, post := range posts {
67                  post.UserId = user.Id
68          }
69          fmt.Println("[Main-Goroutine] Finished merging user posts")
70  }
71
72  func importingPosts() []*Post {
73          time.Sleep(1 * time.Second)
74          titles := []string{"Post 1", "Random Post", "Second Post"}
75          posts := []*Post{}
76          for _, title := range titles {
77                  posts = append(posts, &Post{Title: title, Status: "draft"})
78          }
79
80          return posts
81  }
82
83  func buildingUser(user *User) {
84          time.Sleep(2 * time.Second)
85          user.Name = "John"
86          user.LastName = "Doe"
87          user.Status = "active"
88          user.Id = "1"
89  }
```




```
95          }
96  }
```

unbuffered_channels.go hosted with ❤ by GitHub view raw

Running the previous example will output the following:



Let's understand the output. When the program starts, an empty user object is created and a channel of type boolean(unbuffered channel) in lines 36 and 37 respectively, then a goroutine is created in line 39 which means that the piece of code within that function will be running in a separate goroutine.

The execution of the main goroutine continues and in line 49 we have a print statement, then in the second goroutine, since it is running concurrently at this point, it reaches line 40 and also executes a print statement.

`waiting------` , this is where the blocking concept that we talked about earlier comes into play, in line 53 we see that the main goroutine is reading from the `done` channel, this basically means that the main goroutine will not continue its execution until the second goroutine sends a message to this channel.

In the second goroutine, the **buildUser** function is called and it prints `[Second-GoRoutine] Finished Building User` , then in the next line, it sends a message to the channel. At this point, the main goroutine will detect this and it will continue its execution, as well as the second goroutine.

The methods **mergeUserPosts** and **setDefaultTags** are called in the main and second goroutine respectively and we get their corresponding logs.

When we get to lines 57 to 60, the user and its posts are printed out, but if you check the tags array in the user struct is empty. The reason is that after the second goroutine sent a message to the main goroutine, both goroutines continued executing concurrently and as I previously mention the main goroutine will not wait until other goroutines finished executing, that being said, the second goroutine did not complete its work appending the user tags into the struct before the main goroutine finished and that is why the array is empty. If we remove line 91, we'll be able to see the tags array is now filled in.

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

Also how to send and receive data from a channel

```
done <- true // send
<-done // receive ignorting value
resp := <-done // receive storing value in a variable
```

Also, we saw how goroutines block execution if no other goroutine has sent/receive a message through the channel.

. . .

Channels are also used as a way of connecting multiple goroutines by using the result of one goroutine as the parameters for another one.

Let’s take a look a the next example this time using multiple goroutines.

```
1  package main
2
3  import (
```

```
        Name, Type string
}

type Settings struct {
        NotificationsEnabled bool
}

type User struct {
        Id, Name, LastName, Status string
        Tags                       []*Tag
        *Settings
}

type NotificationsService struct {
}

func main() {
        usersToUpdate := make(chan []*User)
        userToNotify := make(chan *User)
        newUsers := []*User{
                {Name: "John", Status: "active", Settings: &Settings{NotificationsEnabled: true}},
                {Name: "Carl", Status: "active", Settings: &Settings{NotificationsEnabled: false}}
                {Name: "Paul", Status: "deactive", Settings: &Settings{NotificationsEnabled: true}
                {Name: "Sam", Status: "active", Settings: &Settings{NotificationsEnabled: true}},
        }
        existingUsers := []*User{
                {Name: "Jessica", Status: "active", Settings: &Settings{NotificationsEnabled: true
                {Name: "Eric", Status: "active", Settings: &Settings{NotificationsEnabled: true}},
                {Name: "Laura", Status: "active", Settings: &Settings{NotificationsEnabled: true}}
        }

        go filterNewUsersByStatus(usersToUpdate, newUsers)
```

```
46          defer close(usersToUpdate)
47          filteredUsers := []*User{}
48          for _, user := range users {
49                  if user.Status == "active" && user.Settings.NotificationsEnabled {
50                          filteredUsers = append(filteredUsers, user)
51                  }
52          }
53
54          usersToUpdate <- filteredUsers
55  }
56
57  func updateUsers(usersToUpdate <-chan []*User, userToNotify chan<- *User, users []*User) {
58          defer close(userToNotify)
59          for _, user := range users {
60                  user.Tags = append(user.Tags, &Tag{Name: "UserNotified", Type: "Notifications"})
61          }
62
63          newUsers := <-usersToUpdate
64
65          for _, user := range newUsers {
66                  time.Sleep(1 * time.Second)
67                  user.Tags = append(user.Tags, &Tag{Name: "NewNotification", Type: "Notifications"}
68                  userToNotify <- user
69          }
70  }
71
72  func notifyUsers(userToNotify <-chan *User, users []*User) {
73          service := &NotificationsService{}
74          for _, user := range users {
75                  service.SendEmailNotification(user, "Tags", "A new tag has been added to your prof
76          }
77
```




```
83  func (n *NotificationsService) SendEmailNotification(user *User, title, message string) {
84          fmt.Printf("Email Notification Sent to %v, Hi %s, %s\n", user, user.Name, message)
85  }
```

unbuffered_channels2.go hosted with ❤ by GitHub view raw

The output of this example looks like this:

```
Email Notification Sent to &{ Jessica  active □ 0x1400001e375}, Hi Jessica, A new tag has been added to your profile!!
Email Notification Sent to &{ Eric  active □ 0x1400001e376}, Hi Eric, A new tag has been added to your profile!!
Email Notification Sent to &{ Laura  active □ 0x1400001e377}, Hi Laura, A new tag has been added to your profile!!
Email Notification Sent to &{ John  active [0x140000661e0] 0x1400001e371}, Hi John, You got your first tag!!
Email Notification Sent to &{ Sam  active [0x14000066200] 0x1400001e374}, Hi Sam, You got your first tag!!
```

In this example, we have two channels **usersToUpdate** and **userToNotify**, notice how the first channel accepts an array of users and the second one only one single user object. Then there are two arrays of users, one for existing users and one for new users.

In the first goroutine, we send the **usersToUpdate** channel and the slice of **newUsers**, so when the program gets to line 40 a new goroutine is created.

Notice the syntax in **filterNewUsersByStatus** function for the **usersToUpdate** param.

Channels by default are **bi-directional** meaning that you can send and receive information through them, but when passing a channel to a function you can change this behavior and tell the channel that in the context of the function it will only serve one purpose, either to receive information or to send information.

So in this case, we are telling the channel **usersToUpdate** that in the context of this function this channel will only accept sending information and not receiving it.

This function **filterNewUsersByStatus** range over the **newUsers** and only selects the ones that are active and has the setting enabled for notifications. After that in line 54, the **filtered users** are sent through the channel.

At this point, this channel will not be used anymore for sending data so it is important to close the channel. In this case, we are using the **defer** function to call the built-in **close** function and close the **usersToUpdate** channel.

In the second goroutine, we send the **usersToUpdate** channel, the **userToNotify** channel and the **existingUsers** slice. This is where the concept of using a channel's results as the input for another goroutine comes into play.

In this function we are also defining for each channel if it will be used for

In line 59 the function first updated the existing users by appending a new tag to each of them. Then in line 63, it creates a new variable assigning it to the result of the **usersToUpdate** channel. This line will block the execution of this goroutine until the channel sends a message. In other words, if the **filterNewUsersByStatus** takes a lot of time to send the **filteredUsers**, this goroutine will have to wait in this line before proceeding.

Once the data is received this goroutine ranges over the **newUsers** and also updates their tags, but also sends the user through the **userToNotify** channel in line 68.

The **userToNotify** will also need to be closed after this function completes its work, so in line 58 we have a **defer** to close the channel.

Then in line 42, there is a function that is called in the main goroutine, that will notify users, it takes the **userToNotify** channel and the **existingUsers** as parameters.

This function first initializes a service for sending notifications and then ranges over the existing users and sends an email notification to each of them.

Then in line 78, it ranges over the **userToNotify** channel, and for each user that

prevent us from reading from a closed channel, as I mentioned before this is one way of ensuring you don't read from a closed channel. The other syntax is as follows:

```
resp, ok := <-userToNofity
```

The `ok` variable will be false if we are reading from a closed channel and true otherwise, but it will not panic.

As you can see in this example the functions will run concurrently and they communicate to each other using channels to send information about the filtered users and the users to notify.

In this example, we learned how to close a channel using defer and close function.

```
defer close(done)
```

Also how to make a channel uni-directional when it is passed to a function

How to range over a channel, this is useful when we don't know how many items will be sent through a channel but we want to read all of them.

```
for user := range userToNotify {}
```

. . .

Buffered Channels: This type of channel allows you to store more than one piece of data specified by the capacity, when that capacity is reached, subsequent messages that are sent to the channel will block until at least one message is read so that the channel has capacity again.

To create a buffered channel we only need to pass an additional parameter to the make function:

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

performed. The same thing will happen if the channel is empty and a receive operation is performed, it will block until a sent operation is performed.

The data structure used to keep track of the capacity of the channel is a queue, which means that the first element that gets into the queue will be the first getting out of the queue.

Let's look into this using the following code:

```
1  package main
2
3  import (
4          "fmt"
5          "time"
6  )
7
8  func main() {
9          names := make(chan string, 3)
10         go generateName(names)
11
12   // Simulate that a different process takes 5 seconds to run before the main goroutine
13   // starts reading values from the channel.
14         time.Sleep(5 * time.Second)
15
16         for name := range names {
17                 fmt.Printf("Name received: %v\n", name)
18         }
```

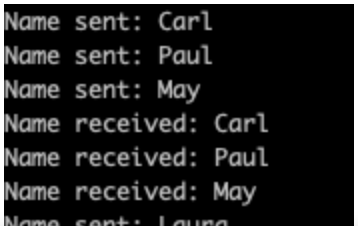
```
24                  time.Sleep(1 * time.Second) // Simulate some latency of 1sec before sending each n
25                  names <- name
26                  fmt.Printf("Name sent: %v\n", name)
27          }
28  }
```

buffered_channels.go hosted with ❤ by GitHub view raw

In the above scenario, we create a buffered channel with a capacity of 3, which means that it can hold 3 strings at a time without blocking the goroutine. In line 10 a goroutine is created and the channel is passed in. That function will send multiple names to the channel with a latency of 1sec between each send, when finishing sending the names the channel will be closed using defer.

In the main goroutine, there is a sleep function call to simulate that the main goroutine executes another operation that took 5 sec before reading values from the channel.

Let's see the output of this code:



As you can see, the first 3 names are sent to the channel without blocking since the buffered size is 3, but after that, the execution of the second goroutine blocks until at least one element is read from the channel, when the main goroutine starts reading from the channel, the second goroutine is unblocked and continues sending the remaining names.

. . .

Key takeaways

- Use goroutines to speed up your Go program.
- Use the make keyword to create an unbuffered channel.
- Use the make keyword specifying the capacity to create a buffered channel.
- Read data from a channel with this syntax `resp := <-names`.
- Send data to a channel with this syntax `numbers <- num`.
- Read all data sent to a channel using a range for loop.
- Close a channel using the defer and close built-in functions.
- Blocking concepts between different goroutines.
- Change bi-directional channels to behave as sent-only or read-only channels within function contexts.

We've seen a lot of concepts related to concurrency in Golang. I hope you enjoyed it and learned from this article!

Thank you for reading. Stay tuned.

Resources

If you are interested in learning more about Go, the following articles may be helpful.

Implementing Interfaces With Golang

Learn how to leverage the power of interfaces

betterprogramming.pub

Go API Design With Protocol Buffers and gRPC

A step-by-step guide based on a social media app

betterprogramming.pub

Get an email whenever Yair Fernando publishes.

Subscribe

Emails will be sent to jon_monts@hotmail.com.
Not you?