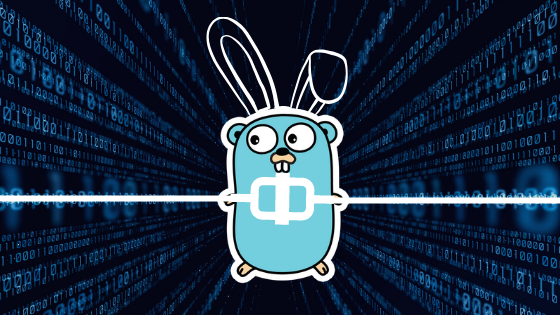
Important links:

<https://madeddu.xyz/posts/go-py-benchmark/>



*Codu is an ML product, built by us, that reviews code how a developer would, for parameters that we’ve seen mostly matter in the real world, which is the ability to****write clean code****— code that others can read and understand.*

*You can read about*[*why*](https://medium.com/@krishnannair/automated-code-reviews-for-clean-code-b9d3c5e20b22)*and*[*how*](https://medium.com/@krishnannair/how-we-automated-code-evaluation-for-clean-code-a601806b0245)*we built it.*

At [Geektrust](https://geektrust.in/), we use [RabbitMQ](https://www.rabbitmq.com/) as our message broker for communicating between distributed applications.

Recently when we introduced RabbitMQ in [Codu.ai](https://codu.ai/), we started seeing this issue of the connection getting dropped when the publisher/consumer is inactive for a long period of time.

Codu is built using [Go](http://golang.org/), and uses the [Go client](https://godoc.org/github.com/streadway/amqp) supported by [RabbitMQ](https://www.rabbitmq.com/tutorials/tutorial-one-go.html). It is currently in [private beta](https://medium.com/@krishnannair/automated-reviews-for-clean-code-private-beta-launch-a4952c4cb4d1), and there are only a few regular users. Once in a while, when someone sends a request, they don’t get any response because the [Go client has lost its connection](https://github.com/streadway/amqp/issues/91) with RabbitMQ server. As the [client library doesn’t give](https://github.com/streadway/amqp/issues/91#issuecomment-262338154) an option for automatic connection recovery, we had to implement it ourselves.

We have only one publisher process and one consumer process that will publish and consume the messages to and from multiple queues in the same exchange. The consumer listens to messages from multiple queues and handles these messages differently based on the message type. Most of the sample code we saw, showed the example of one consumer for one queue, which was slightly different from our implementation. In this post, we show how we handled the auto-reconnect of a consumer and publisher even when connected to multiple queues.

# Setting things up

package comms

import (

"errors"

"fmt"

"github.com/streadway/amqp"

)

//MessageBody is the struct for the body passed in the AMQP message. The type will be set on the Request header

type MessageBody struct {

Data []byte

Type string

}

//Message is the amqp request to publish

type Message struct {

Queue string

ReplyTo string

ContentType string

CorrelationID string

Priority uint8

Body MessageBody

}

//Connection is the connection created

type Connection struct {

name string

conn \*amqp.Connection

channel \*amqp.Channel

exchange string

queues []string

err chan error

}

var (

connectionPool = make(map[string]\*Connection)

)

//NewConnection returns the new connection object

func NewConnection(name, exchange string, queues []string) \*Connection {

if c, ok := connectionPool[name]; ok {

return c

}

c := &Connection{

exchange: exchange,

queues: queues,

err: make(chan error),

}

connectionPool[name] = c

return c

}

//GetConnection returns the connection which was instantiated

func GetConnection(name string) \*Connection {

return connectionPool[name]

}

func (c \*Connection) Connect() error {

var err error

c.conn, err = amqp.Dial("amqp://guest:guest@localhost:5672/")

if err != nil {

return fmt.Errorf("Error in creating rabbitmq connection with %s : %s", amqpURI, err.Error())

}

go func() {

<-c.conn.NotifyClose(make(chan \*amqp.Error)) //Listen to NotifyClose

c.err <- errors.New("Connection Closed")

}()

c.channel, err = c.conn.Channel()

if err != nil {

return fmt.Errorf("Channel: %s", err)

}

if err := c.channel.ExchangeDeclare(

c.exchange, // name

"direct", // type

true, // durable

false, // auto-deleted

false, // internal

false, // noWait

nil, // arguments

); err != nil {

return fmt.Errorf("Error in Exchange Declare: %s", err)

}

return nil

}

func (c \*Connection) BindQueue() error {

for \_, q := range c.queues {

if \_, err := c.channel.QueueDeclare(q, true, false, false, false, nil); err != nil {

return fmt.Errorf("error in declaring the queue %s", err)

}

if err := c.channel.QueueBind(q, "my\_routing\_key", c.exchange, false, nil); err != nil {

return fmt.Errorf("Queue Bind error: %s", err)

}

}

return nil

}

//Reconnect reconnects the connection

func (c \*Connection) Reconnect() error {

if err := c.Connect(); err != nil {

return err

}

if err := c.BindQueue(); err != nil {

return err

}

return nil

}

The above gist shows how a RabbitMQ connection is established. I use a custom Connection struct to encapsulate all the necessary information. An object of this struct is returned by using the function NewConnection on the comms package. All the connections can be managed in the pool against a name, if you have multiple consumers and publishers. After creating a Connection object we need to call the Connect and BindQueue methods, so that the connection is established and queues are bound to the exchange. There is also a Reconnect method which is called when the connection is lost.

Now coming to the most important part. In the Connect method, a Goroutine is called, which will wait on after registering a listener on the NotifyClose method on the amqp.Connection object.

*From the AMQP library docs:*

*NotifyClose method on the amqp.Channel object registers a listener for when the server sends a channel or connection exception in the form of a Connection.Close or Channel.Close method*

When the ‘connection close’ event is received, we push a custom error to the err channel of the Connection object.

go func() {  
 //Listen to NotifyClose   
 <-c.conn.NotifyClose(make(chan \*amqp.Error))   
 c.err <- errors.New("Connection Closed")   
}()

You need to be waiting on this channel and if there is an error, we call the Reconnect method, where we reestablish the connection. We will see how this is used in the consumer and publisher.

# **The Consumer side of things**

package comms

//Consume consumes the messages from the queues and passes it as map of chan of amqp.Delivery

func (c \*Connection) Consume() (map[string]<-chan amqp.Delivery, error) {

m := make(map[string]<-chan amqp.Delivery)

for \_, q := range c.queues {

deliveries, err := c.channel.Consume(q, "", false, false, false, false, nil)

if err != nil {

return nil ,err

}

m[q] = deliveries

}

return m, nil

}

//HandleConsumedDeliveries handles the consumed deliveries from the queues. Should be called only for a consumer connection

func (c \*Connection) HandleConsumedDeliveries(q string, delivery <-chan amqp.Delivery, fn func(Connection, string, <-chan amqp.Delivery)) {

for {

go fn(\*c, q, delivery)

if err := <-c.err; err != nil {

c.Reconnect()

deliveries, err := c.Consume()

if err != nil {

panic(err) //raising panic if consume fails even after reconnecting

}

delivery = deliveries[q]

}

}

}

In this gist you can see theConsume method which will loop through the queues and consume the messages from the amqp.Delivery channel. Once consumed, it is stored in map (this is not thread safe) and returned.

The HandleConsumedDeliveries method will handle these messages that are consumed from the channel for each queue in an infinite loop. It also takes in a function of definition func(Connection, string, <-chan amqp.Delivery) which is the actual function that handles your message. This will be called in a separate go routine so that the messages are handled without any blocking. In this function we also wait for the err channel on the Connection object.

if err := <-c.err; err != nil {   
 c.Reconnect()   
 deliveries, err := c.Consume()   
 if err != nil {   
 panic(err)  
 }  
 delivery = deliveries[q]   
}

On the NotifyClose listener defined while initialising the connection an error object is pushed to this channel, whenever the RabbitMQ channel/connection is closed. Here, when the error obtained is not nil, we reconnect the connection, and start consuming the deliveries again. The delivery channel for the specific object is set again so that the messages are picked up again and the fn object is called again. The consumer section below shows how all this is wired together.

# The Publisher side of things

package comms

//Publish publishes a request to the amqp queue

func (c \*Connection) Publish(m Message) error {

select { //non blocking channel - if there is no error will go to default where we do nothing

case err := <-c.err:

if err != nil {

c.Reconnect()

}

default:

}

p := amqp.Publishing{

Headers: amqp.Table{"type": m.Body.Type},

ContentType: m.ContentType,

CorrelationId: m.CorrelationID,

Body: m.Body.Data,

ReplyTo: m.ReplyTo,

}

if err := c.channel.Publish(c.exchange, m.Queue, false, false, p); err != nil {

return fmt.Errorf("Error in Publishing: %s", err)

}

return nil

}

The publisher side of things is relatively easy.

Here, we use a non-blocking channel setup with the select keyword, as there are no Goroutines calling this method. So, if there is no error on the err channel, it just goes and publishes the message. Else if the connection is lost, it will reconnect and then publish the message.

# The Consumer application

package main

import (

"comms"

"github.com/streadway/amqp"

"log"

)

func main() {

forever := make(chan bool)

conn := comms.NewConnection("my-consumer-1", "my-exchange", []string{"queue-1", "queue-2"})

if err := conn.Connect(); err != nil {

panic(err)

}

if err := conn.BindQueue(); err != nil {

panic(err)

}

deliveries, err := conn.Consume()

if err != nil {

panic(err)

}

for q, d := range deliveries {

go conn.HandleConsumedDeliveries(q, d, messageHandler)

}

<-forever

}

func messageHandler(c comms.Connection, q string, deliveries <-chan amqp.Delivery) {

for d := range deliveries {

m := comms.Message{

Queue: q,

Body: comms.MessageBody{Data: d.Body, Type: d.Headers["type"].(string)},

ContentType: d.ContentType,

Priority: d.Priority,

CorrelationID: d.CorrelationId,

}

//handle the custom message

log.Println("Got message from queue ", m.Queue)

d.Ack(false)

}

}

The consumer main program is waiting forever as it has to wait for the messages arriving in the queues. We create a new Connection object, and call the methods Connect and BindQueue on it. Then, we consume the queued messages using the Consume method. Once this is done, we loop through all the queues, and call the HandleConsumedDeliveries method on each of them in a separate Goroutine. Notice that we pass a function messageHandler of type func(Connection, string, <-chan amqp.Delivery), which actually deals with the raw message obtained from the queue. Here I am only printing a message using it.

# The Publisher

package main

func main() {

conn := comms.NewConnection("my-producer", "my-exchange", []string{"queue-1", "queue-2"})

if err := conn.Connect(); err != nil {

panic(err)

}

if err := conn.BindQueue(); err != nil {

panic(err)

}

for \_, q := range c.queues {

m := comms.Message{

Queue: q,

//set the necessary fields

}

if err := conn.Publish(m); err != nil {

panic(err)

}

}

}

In this application, we create a new Connection object and then call the methods Connect and BindQueue on it. We loop through the queues and publish the message into each of this.

This program will be executed one time, so it won’t have the ‘connection close’ issue. But if you are running a web application and pushing messages to RabbitMQ, there is a chance that the connection will be closed on too much idling. At that time, the auto reconnect feature will come in handy.

Hope this was useful for some of you folks, and let me know if you got any questions in the comments section.

# <https://github.com/EdmundMartin/crackingCodingInterview/tree/master/chapter1>

<https://github.com/EdmundMartin/goAlgoDataS>

# GoLang Requirements:

Technical Skills:

•Strong knowledge of Go programming language, paradigms, constructs, and idioms

•Knowledge of common Go routine and channel patterns

•Experience with the full site of Go frameworks and tools, including:

•Dependency management tools such as Godep, Sltr, etc.

•Go’s templating language

•Go’s code generation tools, such as Stringer

•Popular Go web frameworks, such as Revel

•Router packages, such as Gorilla Mux

•Ability to write clean and effective Godoc comments

•Familiarity with code versioning tools such as Git

•Experience in implementing web-sockets in the server side

•Experience working with NATS

•Experience using Google protocol buffers for communication

# GOLANG:

<https://callistaenterprise.se/blogg/teknik/2017/02/17/go-blog-series-part1/>

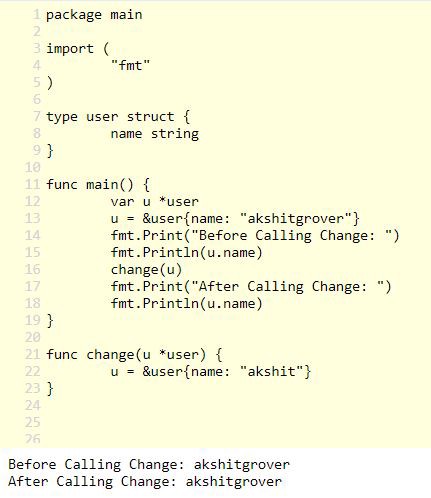
### In how many ways we can pass arguments to the function?

### Why it is not possible pass by reference in golang?

<https://dave.cheney.net/2017/04/29/there-is-no-pass-by-reference-in-go>

### For the code below what will be the output and why so? And what is the possible solution for that?

<https://codeburst.io/pass-by-reference-in-go-demystified-81e0e8dfa2ad>



### Golang Escape Analysis?

<https://medium.com/faun/golang-escape-analysis-reduce-pressure-on-gc-6bde1891d625>

## Function and Methods

### Is it possible a Function Returning Multiple Values in Go Language?

### Function vs methods in golang?

### Can it possible to have multiple functions with same name and signature?

### Can it possible to have multiple methods with same name and signature?

### Anonymous function in golang and how to pass to a function or return from a function?

### What is defer keyword ? and how it is useful ?

<https://medium.com/@edwardpie/deferring-actions-in-golang-793aca7bc2c1>

<https://blog.learngoprogramming.com/gotchas-of-defer-in-go-1-8d070894cb01>

### Function overloading in golang?

## Data Structures

### Difference between array vs slice?

### Slice functions :

|  |  |
| --- | --- |
| Compare two slice | bytes.Compare(slice\_1, slice\_2) |
| Sorting a slice in increasing order | Sort a int slice : sort.Ints()  IntsAreSorted(scl []int) bool  Sort a string slice: sort.Strings()  StringsAreSorted(scl []string) bool  Sort a float : sort.Float64s()  Float64sAreSorted |
| Copy one slice to another | Copy(dest, src[])  Here, dst represents the destination slice and src represents the source slice. It will return the number of elements copied that will be the**minimum of len(dst) or len(src)**. |
| Appending to a slice |  |
| Searching an element of int type in slice of ints | func SearchInts(s\_slice []int, i int) int  This function searches for the given element in a sorted slice of ints and returns the index of that element if present in the given slice. |
| Sorting based on some condition | // Go program to illustrate  // how to sort a slice  package main  import (  "fmt"  "sort"  )  // Main function  func main() {  // Creating and initializing  // a structure  Author := []struct {  a\_name string  a\_article int  a\_id int  }{  {"Mina", 304, 1098},  {"Cina", 634, 102},  {"Tina", 104, 105},  {"Rina", 10, 108},  {"Sina", 234, 103},  {"Vina", 237, 106},  {"Rohit", 56, 107},  {"Mohit", 300, 104},  {"Riya", 4, 101},  {"Sohit", 20, 110},  }  // Sorting Author by their name  // Using Slice() function  sort.Slice(Author, func(p, q int) bool {  return Author[p].a\_name > Author[q].a\_name })    fmt.Println("Sort Author according to their names:")  fmt.Println(Author)  // Sorting Author by their  // total number of articles  // Using Slice() function  sort.Slice(Author, func(p, q int) bool {  return Author[p].a\_article > Author[q].a\_article })    fmt.Println()  fmt.Println("Sort Author according to their"+  " total number of articles:")    fmt.Println(Author)  // Sorting Author by their ids  // Using Slice() function  sort.Slice(Author, func(p, q int) bool {  return Author[p].a\_id > Author[q].a\_id })    fmt.Println()  fmt.Println("Sort Author according to the their Ids:")  fmt.Println(Author)  } |

## Array in Golang:

<https://medium.com/rungo/the-anatomy-of-arrays-in-go-24429e4491b7>

### What is the value of an empty array?

**Example :**

**var a [3]int**  
fmt.Println(a)

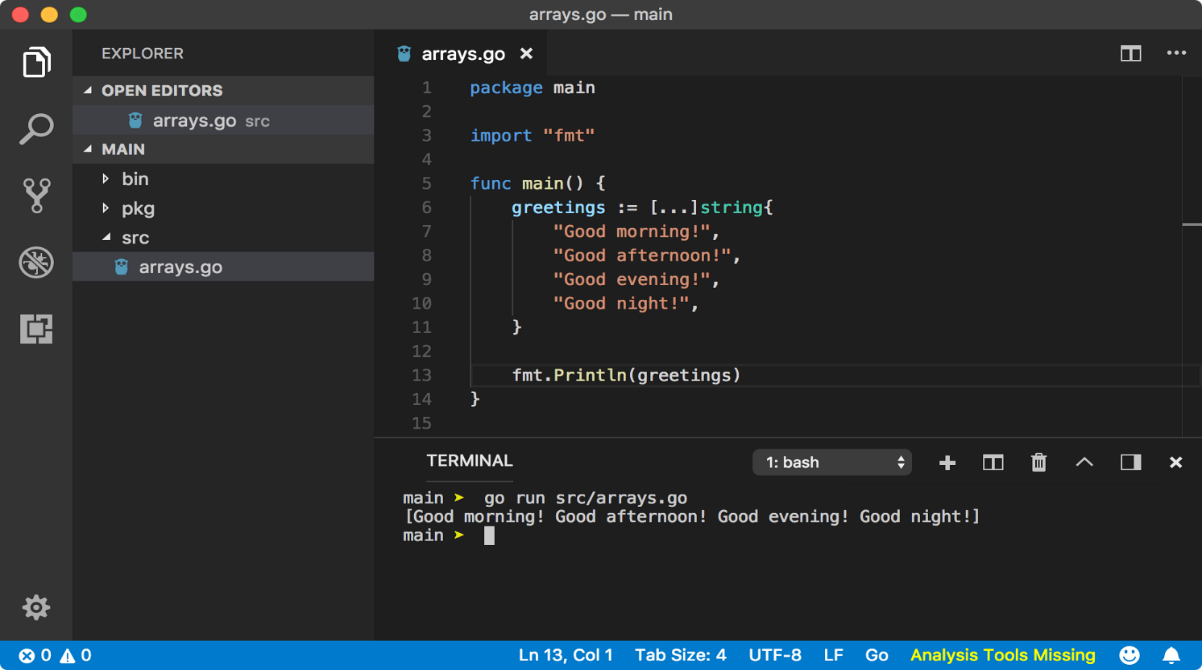
o/p: [0, 0, 0]

### How many ways to declare an array?

There are two ways to declare an array:

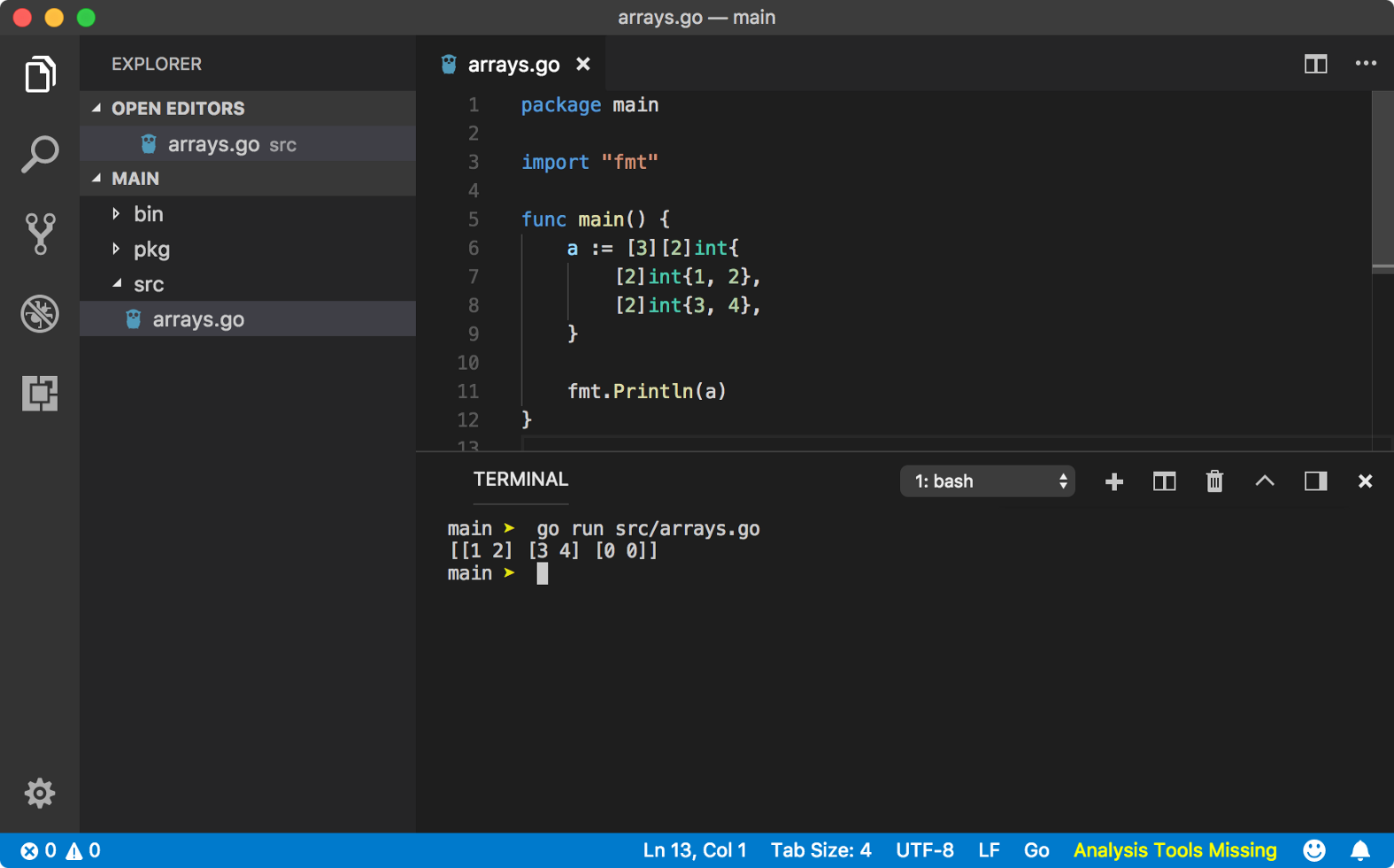
1. By specifying the size of the array.
2. When we don’t know the size of the array, we can declare by using automatic array length declaration.

Go **provide ... operator** to put in place of n in [n]T array type syntax. Go compiler will find the length on its own. **You can only use this operator when you are defining an array with an initial value.**



### How many ways to declare a multi-dimensional array?

Same as C++ , m\*n where m is number of rows , n is number of columns.



Another way is: By using Automatic array length declaration

a := [...][2]int{{1, 2}, {3, 4}, {5, 6}}

### What is called automatic array length declaration?

### Array is pass by value or pass by reference?

Array is pass by value. We can pass as pointer to an array but not recommended. For that reason

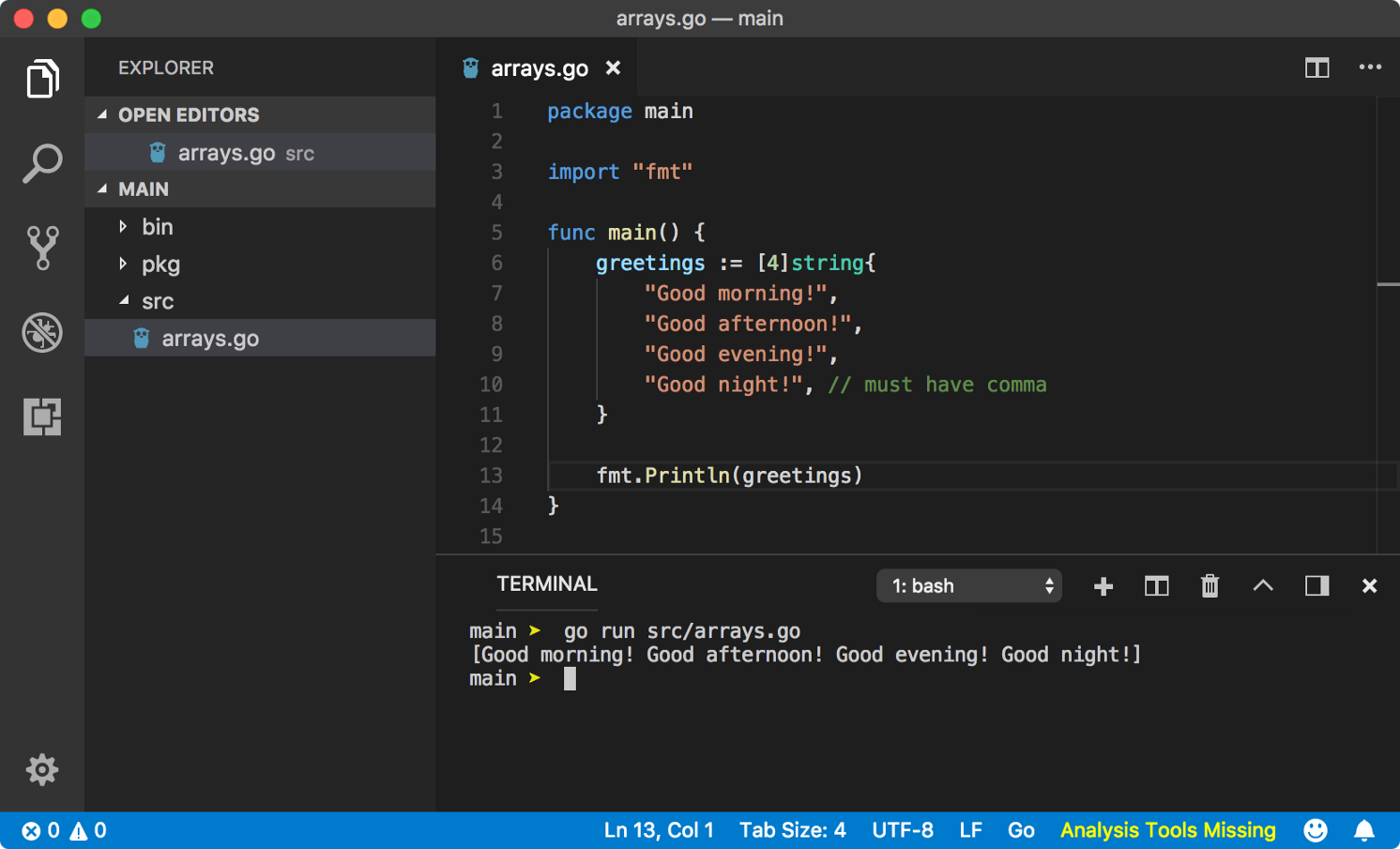
### How array comparison is done?

Arrays of same length and same type can be compared.

### How to iterate over an array?

Using the for loop or using range.

### Initialize multiline in an array?



## Slice in Golang:

<https://medium.com/rungo/the-anatomy-of-slices-in-go-6450e3bb2b94>

<https://medium.com/swlh/golang-tips-why-pointers-to-slices-are-useful-and-how-ignoring-them-can-lead-to-tricky-bugs-cac90f72e77b>

<https://medium.com/a-journey-with-go/go-slice-and-memory-management-670498bb52be>

Syntax to define a slice is pretty similar to that of an array **but without specifying the elements count**. Hence s is a slice. A slice is just like an array which is a **container to hold elements of the same data type** but slice can **vary in size**.

How to append anything to slice?

<https://yourbasic.org/golang/append-explained/>

### [Difference between []\*Users and \*[]Users golang struct](https://stackoverflow.com/questions/50659408/difference-between-users-and-users-golang-struct)

<https://stackoverflow.com/questions/50659408/difference-between-users-and-users-golang-struct/50660143>

### What is the difference between Nil vs Empty Slice?

var nilSlice []string //This is nil   
emptySlice := make([]string, 5) //This is []

What is the difference between above two?

Observe the output below to understand the difference between the two:

fmt.Println(nilSlice) // Output: []  
fmt.Println(len(nilSlice), cap(nilSlice)) // Output: 0 0  
fmt.Println(nilSlice == nil) // Output: truefmt.Println(emptySlice) // Output: []  
fmt.Println(len(emptySlice), cap(emptySlice)) // Output: 5 5  
fmt.Println(emptySlice == nil) // Output: false

By far you must have understood that what is the difference between nil and empty Slice.

Observe the output below :

fmt.Println("Is nilSlice and emptySlice equal?", reflect.DeepEqual(nilSlice, emptySlice)) // Output: false

fmt.Printf("Got: %+v, Want: %+v\n", nilSlice, emptySlice)

//Output: Got: [], Want: [ ]

## What is the value of empty slice?

var s []int

Because **slice is just a reference to an array**. S is not pointing to any array above output is nil.

### **What will happen to the array if I change the value of an element in the slice?**

### **Length and capacity of a slice?**

### **What will happen if I append more elements than the capacity of a slice?**

### **make function in slice?**

### make is a built-in function that helps you create an **empty slice**. The signature of make function is as below. The make function can create many empty composite types.

func make(t Type, size ...IntegerType) Type

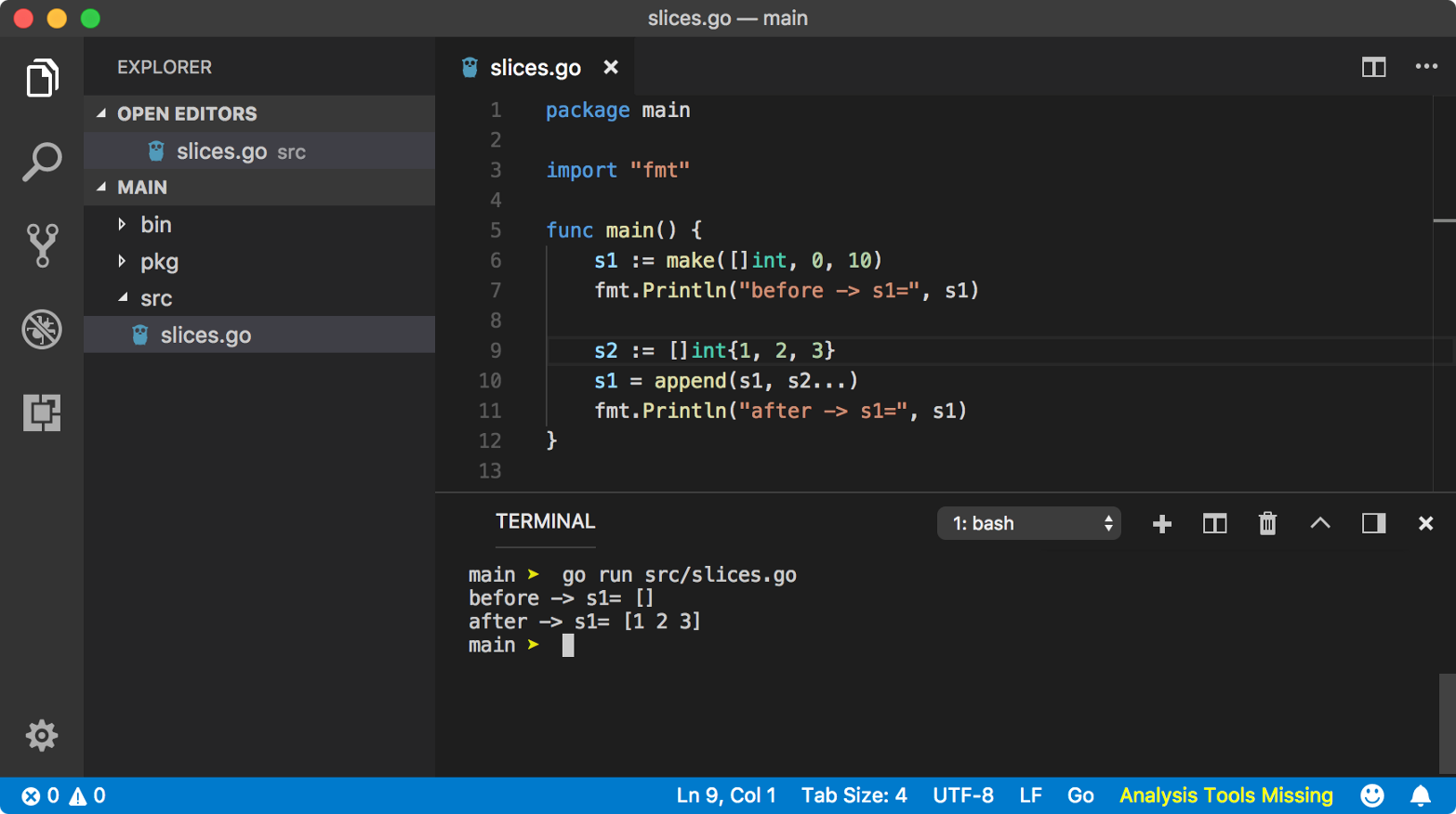
In the case of slice, make function looks like below.

**s := make([]type, len, cap)**

### Unpack operator?

What if you have a slice and you need to append values from it to another slice. In that case ... operator is useful because append does not accept slice as an argument, only the type which slice element is made of.

The … is called unpack operator.



### **How to delete slice elements?**

### **Comparison of Slices?**

### **Memory optimization in slice?**

### **How can you print the memory address of slice?**

# [**How to get the underlying array of a slice in Go?**](https://stackoverflow.com/questions/36706843/how-to-get-the-underlying-array-of-a-slice-in-go)

|  |
| --- |
| package main  import (  "fmt"  "reflect"  "unsafe"  )  func main() {  s := []int{1, 2, 3, 4}  **hdr := (\*reflect.SliceHeader)(unsafe.Pointer(&s))**  data := \*(\*[4]int)(unsafe.Pointer(hdr.Data)) //this is to get the underlying array of a slice  fmt.Printf("slice: %T\n\t%#v\n\tlen: %d\n\tcap: %d\n", s, s, len(s), cap(s))  fmt.Printf("hdr: %#v\n", hdr) // &reflect.SliceHeader{Data:0x40e020, Len:4, Cap:4}  fmt.Printf("data: %#v\n", data) // [4]int{1, 2, 3, 4}  s = append(s, 5)  **hdr2 := (\*reflect.SliceHeader)(unsafe.Pointer(&s))**  data2 := \*(\*[8]int)(unsafe.Pointer(hdr2.Data))  fmt.Printf("slice: %T\n\t%#v\n\tlen: %d\n\tcap: %d\n", s, s, len(s), cap(s))  fmt.Printf("hdr2: %#v\n", hdr2) // &reflect.SliceHeader{Data:0x45e020, Len:5, Cap:8}  fmt.Printf("data2: %#v\n", data2) // [8]int{1, 2, 3, 4, 5, 0, 0, 0}  } |

## Struct in Golang:

<https://medium.com/rungo/structures-in-go-76377cc106a2>

### Anonymous Fields?

### Is Nested structure allowed?

### Is Nested Interface allowed?

### Promoted Fields?

### Promoted Function?

### How you can declare a function field?

### Structure comparison in Golang?

## Interface in Golang:

### What is an empty interface?

The interface type that specifies zero methods is known as the *empty interface*:An empty interface may hold values of any type. (Every type implements at least zero methods.)

Empty interfaces are used by code that handles values of unknown type. For example, fmt.Print takes any number of arguments of type interface{}

## How to create a python defaultdict() equivalent in golang?

<https://github.com/jxub/defaultdict_go/blob/master/main.go>

DS and Algorithm:

<https://github.com/floyernick/Data-Structures-and-Algorithms/blob/master/Trie/Trie.go>

## Polymorphism with Golang

<https://medium.com/technofunnel/polymorphism-with-golang-interfaces-b2f58a05b221#:~:text=Polymorphism%20with%20Golang%20Interfaces%E2%80%A6&text=Interfaces%20in%20Golang%20works,in%20other%20server%2Dside%20languages.&text=Interfaces%20in%20Golang%20provides%20a,to%20work%20with%20specified%20Interfaces.>

## <https://medium.com/@simplyianm/why-gos-structs-are-superior-to-class-based-inheritance-b661ba897c67>

## Object Oriented programming

<https://medium.com/technofunnel/golang-object-oriented-programming-f2e6448b8f24>

<https://medium.com/@prac_coder/oop-inheritance-in-golang-complete-guide-62bfe99f75e6> Very imp

## What is an anonymous function? And what is Immediately-invoked function expression (IIFE)?

What is a closure in Golang?

<https://www.calhoun.io/5-useful-ways-to-use-closures-in-go/>

Sorting custom class in golang

<https://yourbasic.org/golang/how-to-sort-in-go/>

Another Important Article:

<http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/>

<https://golangbyexample.com/>

## Goroutine

When go keyword is placed before a function call, it becomes goroutines.

goroutines behave like threads but technically; it is an abstraction over threads.

| **Thread** | **goroutine** |
| --- | --- |
| OS threads are managed by kernal and has hardware dependencies. | goroutines are managed by go runtime and has no hardware dependencies. |
| OS threads generally have fixed stack size of 1-2MB | goroutines typically have 8KB (2KB since Go 1.4) of stack size in newer versions of go |
| Stack size is determined during compile time and can not grow | Stack size of go is managed in run-time and can grow up to 1GB which is possible by allocating and freeing heap storage |
| There is no easy communication medium between threads. There is huge latency between inter-thread communication. | goroutine use channels to communicate with other goroutines with low latency ([read more](https://blog.twitch.tv/gos-march-to-low-latency-gc-a6fa96f06eb7)). |
| Threads have identity. There is TID which identifies each thread in a process. | goroutine do not have any identity. go implemented this because go does not have TLS([Thread Local Storage](https://msdn.microsoft.com/en-us/library/windows/desktop/ms686749(v=vs.85).aspx)). |
| Threads have significant setup and teardown cost as a thread has to request lots of resources from OS and return once it's done. | goroutines are created and destoryed by the go's runtime. These operations are very cheap compared to threads as go runtime already maintain pool of threads for goroutines. In this case OS is not aware of goroutines. |
| Threads are preemptively scheduled ([read here](https://stackoverflow.com/questions/4147221/preemptive-threads-vs-non-preemptive-threads)). Switching cost between threads is high as scheduler needs to save/restore more than 50 registers and states. This can be quite significant when there is rapid switching between threads. | goroutines are coopertively scheduled ([read more](https://stackoverflow.com/questions/37469995/goroutines-are-cooperatively-scheduled-does-that-mean-that-goroutines-that-don)). When a goroutine switch occurs, only 3 registers need to be saved or restored. |

## Your Program Starts

When your Go program starts up, it’s given a Logical Processor (P) for every virtual core that is identified on the host machine. If you have a processor with multiple hardware threads per physical core ([Hyper-Threading](https://en.wikipedia.org/wiki/Hyper-threading)), each hardware thread will be presented to your Go program as a virtual core.

Suppose I have a single processor with 4 physical cores. The Intel Core i7 processor has Hyper-Threading, which means there are 2 hardware threads per physical core. This will report to the Go program that 8 virtual cores are available for executing OS Threads in parallel.

Every P is assigned an OS Thread (“M”). The ‘M’ stands for machine. This Thread is still managed by the OS and the OS is still responsible for placing the Thread on a Core for execution. This means when I run a Go program on my machine, I have 8 threads available to execute my work, each individually attached to a P.

Every Go program is also given an initial Goroutine (“G”) i:e main, which is the path of execution for a Go program. A Goroutine is essentially a [Coroutine](https://en.wikipedia.org/wiki/Coroutine) but this is Go, so we replace the letter “C” with a “G” and we get the word Goroutine.

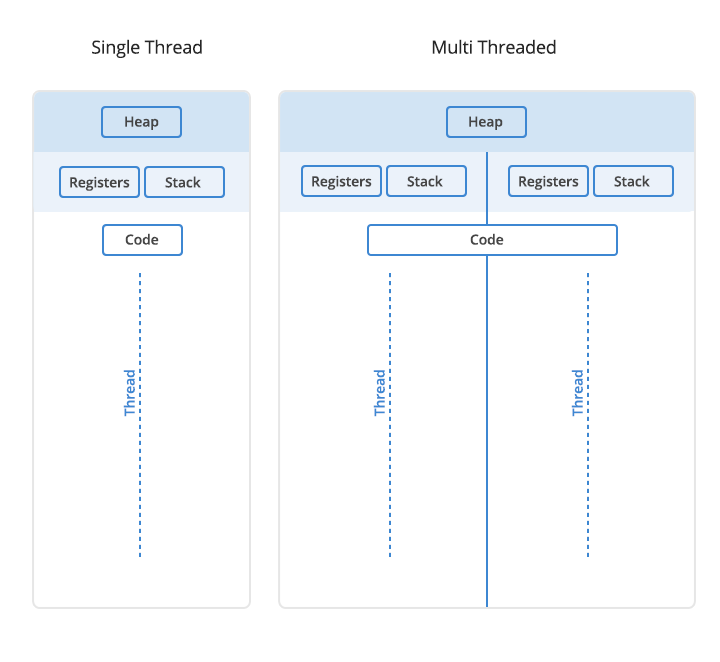
You can think of Goroutines as application-level threads and they are similar to OS Threads in many ways. Just as OS Threads are context-switched on and off a core, Goroutines are context-switched on and off an M.

Why Goroutines are application level threads?

The Go scheduler is part of the Go runtime, and the Go runtime is built into your application. This means the Go scheduler runs in [user space](https://en.wikipedia.org/wiki/User_space), above the kernel. The current implementation of the Go scheduler is not a preemptive scheduler like our OS scheduler but a [cooperating](https://en.wikipedia.org/wiki/Cooperative_multitasking) scheduler. Being a cooperating scheduler means the scheduler needs well-defined user space events that happen at safe points in the code to make scheduling decisions, meaning at some definite points in the code we tell go scheduler that current goroutine will yield and it can schedule another go-routine.

There are four classes of events that occur in your Go programs that allow the scheduler to make scheduling decisions. This doesn’t mean it will always happen on one of these events. It means the scheduler gets the opportunity.

* The use of the keyword go
* Garbage collection
* System calls
* Synchronization and Orchestration



<https://medium.com/@riteeksrivastava/a-complete-journey-with-goroutines-8472630c7f5c>

<https://www.ardanlabs.com/blog/2018/12/scheduling-in-go-part3.html>

What are channels:  
Channels are type safe message queues that have the intelligence to control the behavior of any goroutine attempting to receive or send on it. A channel acts as a conduit between two goroutines and will synchronize the exchange of any resource that is passed through it.

When a channel is created with no capacity, it is called an unbuffered channel. In turn, a channel created with capacity is called a buffered channel.

**Unbuffered Channels**  
Unbuffered channels have no capacity and therefore require both goroutines to be ready to make any exchange. When a goroutine attempts to send a resource to an unbuffered channel and there is no goroutine waiting to receive the resource, the channel will lock the sending goroutine and make it wait. When a goroutine attempts to receive from an unbuffered channel, and there is no goroutine waiting to send a resource, the channel will lock the receiving goroutine and make it wait.

**Buffered Channels**  
Buffered channels have capacity and therefore can behave a bit differently. When a goroutine attempts to send a resource to a buffered channel and the channel is full, the channel will lock the goroutine and make it wait until a buffer becomes available. If there is room in the channel, the send can take place immediately and the goroutine can move on. When a goroutine attempts to receive from a buffered channel and the buffered channel is empty, the channel will lock the goroutine and make it wait until a resource has been sent.

So the follow up question is:

Why the code produces output below:

fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:

main.main()

/home/tarrsalah/src/go/src/github.com/tarrsalah/tour.golang.org/65.go:8 +0x52

exit status 2

|  |
| --- |
| package main  import "fmt"  func main() {  c := make(chan int)  c <- 1  fmt.Println(<-c)  } |

Follow the unbuffered channel definition:

when a channel is full, the sender waits for another goroutine to make some room by receiving. you can see an unbuffered channel as an always full one, so there must be another goroutine to take what the sender sends.

This line

c <- 1

blocks because the channel is unbuffered. As there's no other goroutine to receive the value, the situation can't resolve, this is a deadlock.

You can make it not blocking by changing the channel creation to

c := make(chan int, 1)

so that there's room for one item in the channel before it blocks.

But that's not what concurrency is about. Normally, you wouldn't use a channel without other goroutines to handle what you put inside. You could define a receiving goroutine like this :

func main() {

c := make(chan int)

go func() {

fmt.Println("received:", <-c)

}()ss

c <- 1

}

## Select Statement in Golang:

Even if buffered or unbuffered, basic sends and receives on channels are blocking. Fortunately, GoLang is simply awesome and let you create explicit non-blocking channels, using the select statement: thus, you can use the select with default clause to implement non-blocking sends, receives, and even non-blocking multi-way selects.