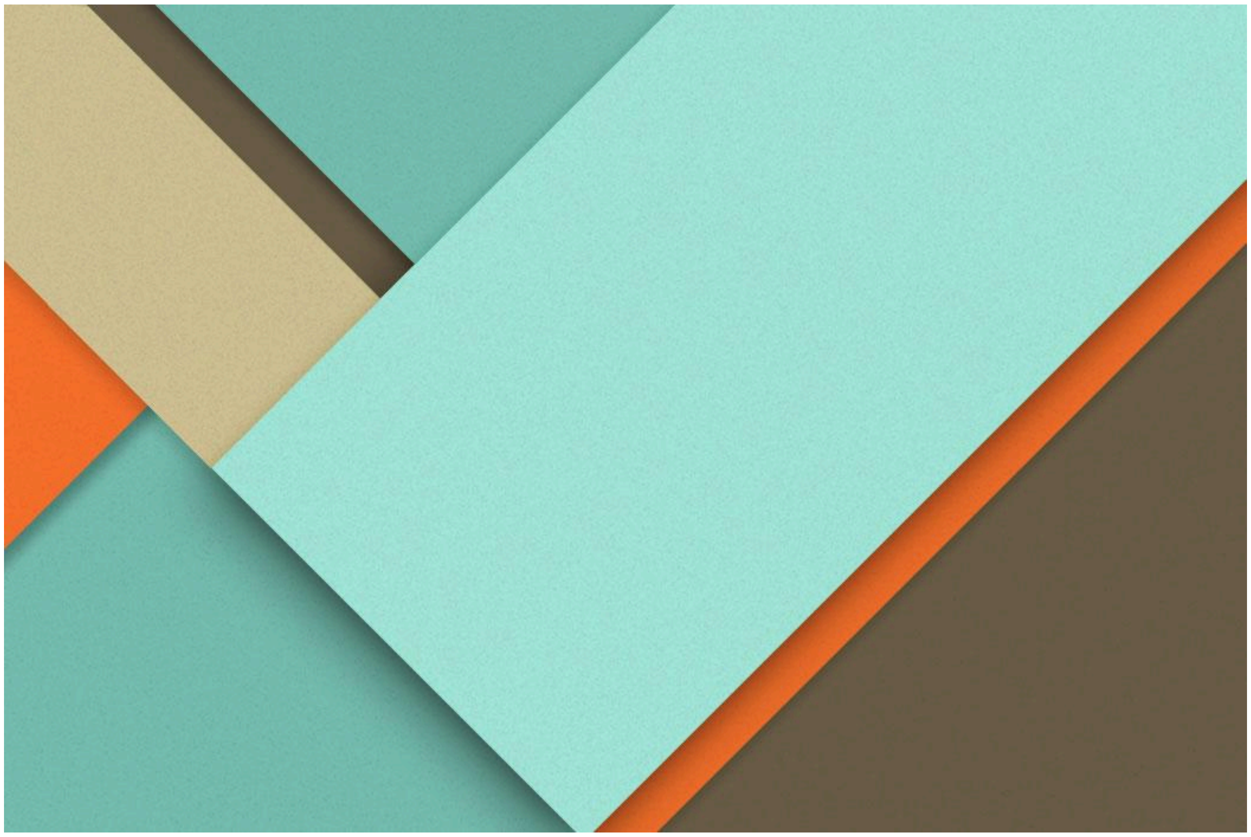


Golang CookBook

15.12.2023

v1.0.0

Chinmay Hegde.

Encora Innovation Labs

Index

Golang Basics:-

Why Go?

Choosing golang for a project is a strategic decision based on several factors that align with the language's strengths. Here are a few reasons why we should consider golang.

- Concurrency and Scalability:-
    - Go was designed with concurrency in mind. It offers goroutines, which are lightweight threads and channels to communicate between them. This makes it well suitable for development of scalable and concurrent systems. It's a good choice for web servers, network applications, and services that need to handle large numbers of concurrent connections.
- Performance:-
    - It's a statically typed, compiled language that produces a single binary. This can result in better performance compared to interpreted languages.
    - It also has a garbage collector for automatic memory management, which results in efficient use of resources.
- Simplicity and Readability:-
    - Golang offers simple and clean syntax.
- Built In Tooling:-
    - Offers built in formatting tools go-fmt
    - Offers go get for package management
- Cross Platform Compatibility
- Strong Standard Library
- Community Support
- Google's Backing
- Static Typing:
    - Go's static typing helps to catch errors at compile time
    - Essential in large code base where early error detection is essential
- Microservices and Cloud Native Development:-

- Lightweight, Fast compiling, Efficient Concurrency model

Go Vs Java

Golang:-

- Go is a concurrent and procedural programming language.
- Go does not have exception handling, because exception thread is an invisible second control flow through the program. It makes the flow less readable and harder to reason about. Go code does what it says. It supports built-in error interface, which needs to be explicitly handled.
- Golang is faster than java because of below reasons:-
    - Concurrency Model:-
        - Go is designed with concurrency in mind. Golang offers goroutines which are lightweight in nature and support channels for communication between two go-routines, which makes it efficient for scalable solutions.
    - Efficient Compilation and Execution Speed:-
        - Go is a statically typed,compiled language and its compilation process is fast. Statically typed languages generate machine code that can directly operate on known types. Interpretation overhead is reduced due to this

Java:-

- Java is object oriented programming language
- Supports Exception Handling
- Java is slow and heavier due to JVM and bytecode conversion.
- Java supports multi threading. Compared to Go Java threads are less efficient as they are OS threads.
- Java has good support of standard libraries
- Java has a lot of abstraction involved.

Object Oriented Programming in Go

Is golang an Object Oriented Programming Language?

The answer is yes and no as per GO FAQ.

- Go has types and methods and allows object oriented style of programming
- There is no type hierarchy in golang.
- Interfaces in golang provided a different approach, easy to use and more general.
- Methods in Go are more general. They are supported for any sort of data including built-in types such as plain, unboxed integers.

Encapsulation in Go:-

- Encapsulation keeps data safe from external interference. It's a protective shield over the data which prevents data from being accessed by the code outside this shield. In golang it is done at the package level.
- Go does not have access modifiers like public,private/protected. Instead there are exported and unexported fields.

```go
type Customer struct {
    id   int
    name string
}

func (c *Customer) GetID() int {
    return c.id
}

func (c *Customer) GetName() string {
    return c.name
}
```

Inheritance In Go:-

In oop programs are designed in such a way where everything is an object that interacts with one another. Inheritance lets properties of one class to be inherited by the other. It helps to reuse the code and establish a relationship between classes.

In go we can use **composition** to get similar results. This is done by struct embedding in another struct anonymously.

```go
func main() {
   car := &Car{
       Vehicle{
           Seats: 4,
           Color: "blue",
       },
   }


   fmt.Println(car.Seats)
   fmt.Println(car.Color)
}
```

Constructors:-

Constructors are special functions for reliably creating multiple instances of similar objects.

There is no specific definition for initialization in GO. We can achieve the same by writing a function.

```go
type Customer struct {

   Name string

   Age  int

}


func (c *Customer) NewCustomer(name string, age int) {

   c.Name = name

   c.Age = age

}


func main() {

   customer := new(Customer)

   customer.NewCustomer("John", 30)

   fmt.Printf("%s: %d\n", customer.Name, customer.Age)

}
```

There is no concept of default or empty value in go. Everything is initialized to something.

Polymorphism:-

Golang supports polymorphism through interfaces only. A type is said to implements an interface, if it provides definition for all the methods declared by the interface.

```go
type Instrument interface {

   Play()

}


func PlayInstrument(i Instrument) {

   i.Play()

}


type Guitar struct {

   Type string

}


func (g Guitar) Play() {

   fmt.Println("Guitar Sounds")

}


func main() {

   g := Guitar{"Classical"}

   PlayInstrument(g)

}
```

Enum:-

Enum is a powerful data type for Enumerated data in oop languages. In Go we can use only constants and assign incrementing values by using iota.

```
const (

   Guitar = iota // 0

   Violin        // 1

   Piano         // 2

   Drums         // 3

)
```

Variables In Go:-

A variable is a storage location or place holder used for holding the value. It allows us to manipulate and retrieve the information. Variables in golang can be declared in two ways as below.

```
 var varName string = "shsh"

func main() {

   varName := "hello"

}
```

Difference between var and short declaration operator:-

var keyword	short declaration operator
var is a lexical keyword present in Golang.	:= is known as the short declaration operator.
It is used to declare and initialize the variables inside and outside the functions.	It is used to declare and initialize the variables only inside the functions.
Using this, variables have generally package level or global level scope. It can also have local scope.	Here, variables has only local scope as they are only declared inside functions.
Declaration and initialization of the variables can be done separately.	Declaration and initialization of the variables must be done at the same time.

Constants In Go:-

Constants are used to represent fixed values; their values cannot be changed at a later stage in the program. Any attempt to change the value of a constant will cause a runtime panic.

```
const myValue = "Hello World"
```

```
const country, code = "Pakistan", 92
```

Note: constants cannot be created using the “:=” operator. The reason behind this is ” :=” is used for declaration and initialization of the variable. Also the variables declared using “:=”operator can be resigned with desired/computed values/expression using “=” operator. It is also necessary that constants needs to be diffrentiated from variables so that it helps in compiler optimization such as constant folding(process of recognizing constants at compile time rather than computing them at runtime)

Data Types In Go

Primary Types: Include Integers, Boolean, Float, Strings.

Numeric Data Types:-

```
uint8       the set of all unsigned  8-bit integers (0 to 255)

uint16      the set of all unsigned 16-bit integers (0 to 65535)

uint32      the set of all unsigned 32-bit integers (0 to 4294967295)

uint64      the set of all unsigned 64-bit integers (0 to 18446744073709551615)


int8        the set of all signed  8-bit integers (-128 to 127)

int16       the set of all signed 16-bit integers (-32768 to 32767)

int32       the set of all signed 32-bit integers (-2147483648 to 2147483647)

int64       the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)


float32     the set of all IEEE-754 32-bit floating-point numbers

float64     the set of all IEEE-754 64-bit floating-point numbers


complex64   the set of all complex numbers with float32 real and imaginary parts

complex128  the set of all complex numbers with float64 real and imaginary parts


byte        alias for uint8

rune        alias for int32


uint     either 32 or 64 bits

int      same size as uint

uintptr  an unsigned integer large enough to store the uninterpreted bits of a pointer value
```

Booleans In Golang:-

Boolean values are those which can be assigned true or false and have the type bool with it.

Strings In Go:-

Strings in golang are sequences of bytes. Length of the string can be discovered using built-in function len. A string's bytes can be accessed by integer indices 0 to len(s)-1.

Runes In Go:-

In the past we generally used ASCII(American Standard for Information Interchange). There, we used 7 bits to represent 128 characters including upper and lowercase English letters, numbers and a variety of punctuations and device-control characters. Due to these character limitations, a large number of the population is not able to use their custom writing system. To solve this problem unicode was invented. Unicode is a superset of ASCII. To represent a unicode code point in Golang, it supports rune. Rune is a data type and it is an alias for int32.

```
rune1 := 'B'
rune2 := 'g'
rune3 := '\a'

// Displaying rune and its type
fmt.Printf("Rune 1: %c; Unicode: %U; Type: %s", rune1,
                         rune1, reflect.TypeOf(rune1))

fmt.Printf("\nRune 2: %c; Unicode: %U; Type: %s", rune2,
                           rune2, reflect.TypeOf(rune2))

fmt.Printf("\nRune 3: Unicode: %U; Type: %s", rune3,
                             reflect.TypeOf(rune3))
```

```
msg := "a"

rune1 := 'B'

fmt.Printf("%v,%T", rune1, rune1) //output is 66,int32

for i, _ := range msg {

fmt.Printf("%v,%T", msg[i], msg[i]) //output is 97,uint8

break
}
```

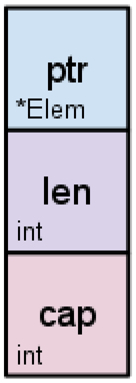
Arrays and Slice:-

- Arrays and slices in golang are both ordered sequences of elements.

Array:-

- Array in golang is a data structure that consists of an ordered sequence of elements that has its capacity defined at creation time. Once an array has allocated its size , the size can no longer be changed. A slice on the other hand is a variable length version of the array.
- Arrays do not need to be initialized explicitly. It provides a zeroed value of array for a specific data type.
- Go's array are values . An array variable denotes the entire array.

Slice:-



- Slice in go is nothing but reference to the array.
- Zeroed value of slice is nil
- Slice can be created by []T where T is data type and length is not mentioned.
- Slice can also be created by make function where make function accepts type,length and capacity. Capacity is optional, if capacity of the slice is not mentioned, it defaults capacity to length of the slice.
- A slice is growable in nature, a slice can be grown by using copy or append function.
- When append is used and if it exceeds the capacity, a new array is created and its reference is given to the slice.

Maps in Golang:-

Maps in golang are used to represent key value pairs.

Map is a reference type. It can be represented in the below structure.

len(map)
lg(#bucket)(number of buckets)
buckets
Hash seed

When we do a look up on the map below operations happen in the backend.

1. It checks if given map is nil or empty
2. It computes the hash
3. Based on the hash it finds the bucket for the lookup
4. Then it loops in the bucket for fetching the value

Structs In Golang:-

- Structs in golang is a user defined type that allows to group/combine items of possibly different types into a single type.
- In Go, structs are defined by using the type keyword, followed by the name of the new type and then the keyword struct.
- Structs in Go can have any type of fields, including another struct.
- Structs in go can have methods associated with them.
- Methods can be defined on structs using the receiver syntax.
- Structs in go can be used for defining custom types that encapsulate data and behavior.
- Go does not support Inheritance, but we can achieve composition by using struct embedding.

Efficient usage of structs:-

- The first way to optimize the struct is by ordering the fields by size. This will help to reduce the amount of padding that is required.
- Padding:- Padding is the extra memory added to a struct to ensure all of the fields are aligned on a memory boundary.
- Aligned data can be accessed more efficiently by CPU, so minimizing the padding improves the performance.
    - Modern CPU hardware performs reads and writes to memory more efficiently when the data is naturally aligned. A memory address "a" is said to be n-byte aligned, when a is multiple of n. This generally means that the data's memory address is a multiple of data size.

Bad Way:-

```
type stats struct {

NumPosts uint8

Reach uint16

NumLikes uint8

}
```

Efficient Way:-

```
type stats struct {

Reach uint16

NumPosts uint8

NumLikes uint8

}
```

Interfaces in Golang:-

- The golang interface is a custom type that contains a set of one or more method signatures.
- Interface is abstract in nature, that means it cannot be instantiated.
- It is allowed to create a variable of interface type and it can be assigned with a concrete type value which contains the methods required by the interface.
- A concrete type is said to implement an interface, when it has the method implementations of all the method signatures defined by the interface.
- It helps to achieve polymorphism

```go
type Animal interface {
   Sound() string
   Info() string
}

type Dog struct {
   Age  int
   Name string
}

type Cat struct {
   Age  int
   Name string
}

func (d *Dog) Sound() string {
   return "Woof"
}

func (d *Dog) Info() string {
   return fmt.Sprintf("DOG:%s Age:%s Sound:%s", d.Name, strconv.Itoa(d.Age), d.Sound())
}

func (c *Cat) Sound() string {
   return "Meaow"
}


func (c *Cat) Info() string {
   return fmt.Sprintf("CAT:%s Age:%s Sound:%s", c.Name, strconv.Itoa(c.Age), c.Sound())
}
```

Type Conversion And Type Assertion In Golang

- Type conversion is the process of converting the value of one type to the value of another type.
- Golang is statically typed language, which means types are determined at the compile time, rather than at the runtime. Therefore explicit type conversion is required.

We can get type of a variable by following ways:-

- By using %T in fmt functions
- By using reflect.TypeOf
- By using reflect.ValueOf(variable).Kind
- If variable “a” is an interface, then we can switch over the a.(type) for performing type specific action.

Generics In Golang:-

Generics is introduced in GO 1.18. If we take a top view to understand what is changed, below are the 3 major changes that are introduced as part of go 1.18.

1. Type parameters for functions and interfaces.
2. Type sets defined by interfaces
3. Type inference.

Now let us understand what problem is addressed by generics..

```
func fmin(x,y float32) float32 {
   if x < y {
      return x
   }
   return y
}
```

Above is the function to get a minimum of 2 float32 values. This function helps for only float32 values and if we have to get the minimum of integers we have to write new logic. That's where generics come for rescue.

```
func fmin[T constraints.Ordered](x, y T) T {
   if x < y {
      return x
   }
   return y
}
```

Above code is the representation of simple generics use case. In the above code **`[T constraints.Ordered]`**

Is the type parameter list. The type parameter **constraints.Ordered** supports all the orderable value types like int,float and ~string.

Type Set:-

In golang, a type is said to implement an interface if it provides implementation for all the methods defined by the interface.

We can have a set of types in the interface, then we can say the interface defines a set of types.

```
type Ordered interface {
   int | float64 | string
}
```

Tilde operator(~):-

This is a new token introduced(“ ~ ”). This indicates a set of types , whose underlying type is T.

For example:-

```
type Point int32


func GetMax[T ~int](x, y T) T {
   if x > y {
      return x
   }
   return y
}
```

We have defined a custom type Point of type int32. Since we have used tilde operator, Get Max supports all the types whose underlying type is int32.

Type Inference:-

The way to deduce type arguments from type parameter constraint is called **type inference.**

When To use Generics:-

1. Functions that work on Slices, maps and channels of any element type.
2. General purpose data structures.(Trees,LinkedList... etc)
3. Functions having the same implementation for different types.

Concurrency In Golang:-

Concurrency and Parallelism are two related but different concepts. Concurrency is about dealing with multiple tasks at once. It is a code property, where the processor can start, run and complete the task in overlapping time periods.

On the other hand Parallelism is about running multiple tasks simultaneously, often on different cores or processors. It is the property of the running program.

Concurrency provides a mechanism to structure a solution to solve multiple tasks that may run in parallel.

Golang is designed with concurrency in mind. The Golang concurrency model is influenced by **Communicating Sequential Processes.** The motto of golang concurrency is "***Don't communicate by sharing memory; Share memory by communicating.***"

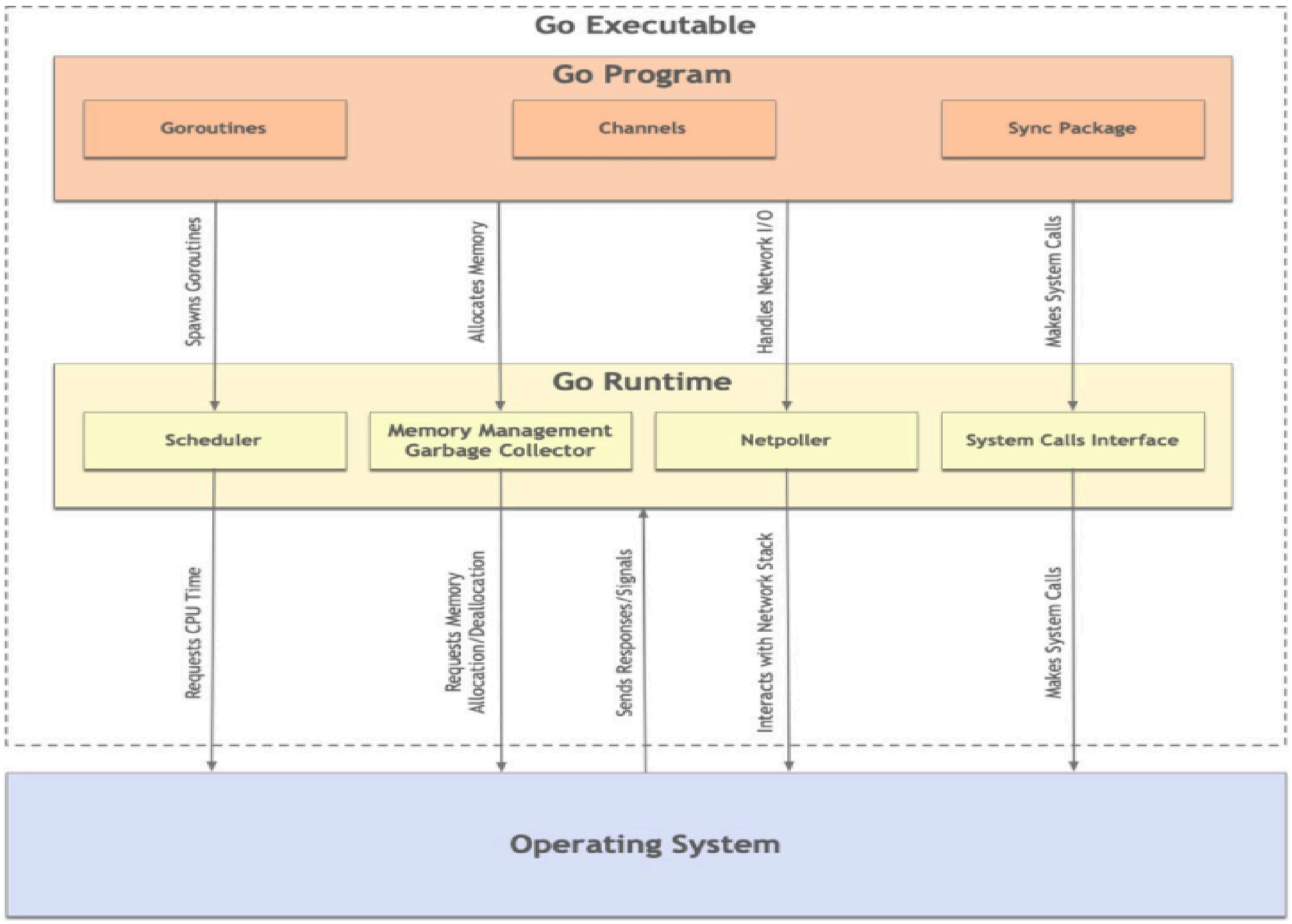
Golang Concurrency Primitives:-

1. **Go Routines:-** Goroutines are light weight threads of execution managed by go runtime.
2. **Channels**:- Goroutines can communicate with each other using channels. It allows synchronization of execution by exchanging values.
3. **Select**:- "**select**" keyword helps to efficiently handle the multiple channel operations. When a program uses the select statement it stops the execution until one of its cases is satisfied.
4. **Mutex and RWMutex**:- These primitives provide the mechanism for goroutines to lock shared resources. RWMutex is similar but provides multiple readers and exclusive writers.
5. **WaitGroups:-** WaitGroup is a primitive that allows the main goroutine to wait for a group of other goroutines to finish before proceeding further.
6. **Once and Pool:-** Once type provides a way to do something once and only once. Pool is a set of temporary objects that can be individually saved and retrieved.
7. **Cond: Cond** implements a conditional variable. A synchronization primitive that allows go routines to wait until code meets the particular condition.
8. **Sync Map:-** Concurrent Safe version of map.

Critics On Golang Concurrency:-

1. Lack of control over low level details like how to control goroutines scheduling.
2. Potential Goroutine leaks.
3. Difficulty handling errors.
4. Error Propagation for nested goroutines.

<u>Internal working of Golang Concurrency Model:-</u>



- Above diagram represents a go executable running on an Operating System.
- Go program is an application code. It contains goroutines, channels and sync packages etc.
- Go Runtime manages the execution of the Go program. It contains a scheduler, Memory Management Garbage collector, netpoller etc.
- Key responsibility of the go runtime is to manage the execution of goroutines. A Go program begins in a single goroutine initially.
- Go scheduler is part of go runtime and is responsible for managing goroutines. Unlike operating system schedulers, go scheduler works at the go routine level.
- Go scheduler employs m:n scheduling, where n os threads schedule m goroutines. This helps the runtime manage more goroutines than available threads.

Difference between OS Threads and Goroutines:-

GoRoutines	Threads
Goroutines are managed by go runtime	OS threads are managed by kernal
Goroutines are not hardware dependent	OS threads are hardware dependent
Goroutines have easy communication medium known as channels	OS threads does not have easy medium of communication
Communication with Low latency due to channels	Communication with high latency
As goroutines do not have local storage, they do not have ids.	Threads have their unique ids due to local storage.
Cooperatively scheduled.	Preemptive scheduling
Go routines have growable segmented stacks	Fixed size

Channels In Golang:-

Channel is built in concurrency mechanism for communication and synchronization between goroutines.

Channels are of two types:-

1. Unbuffered channels:-
   a. When you create a channel without specifying buffer size i.e ch := make(chan int), you create an unbuffered channel.
   b. In the unbuffered channel each send operation(ch <- value) will block until there is a corresponding receive operation(v:= <-ch).
   c. This means in the unbuffered channel sender and receiver must be ready and available for communication.

2. Buffered Channels:-
   a. Buffered channels are created with specified buffer size. For example
      i. ch:= make(chan int,3)
   b. In the buffer channel, the sender can send values without blocking until the buffer is not full. Once the buffer is full, subsequent send operations will block until space is available in the buffer.
   c. Similarly the receiver can receive without blocking as long as the buffer is not empty. Once the buffer is empty, receive operations will block until values are sent to the buffer.
   d. Buffer channels allow asynchronous communication between the sender and receiver.

Sync Package In Golang:-

WaitGroups In Golang:-

Wait groups in golang allow you to block a specific code block to allow a specific set of goroutines to complete execution.

Methods of Wait Groups in Golang

- **Add**-The Waitgroup acts as a counter holding the number of functions or go routines to wait for. When the counter becomes 0 the Waitgroup releases the goroutines.
- **Wait**-The wait method blocks the execution of the application until the Waitgroup counter becomes 0.
- **Done**-Decreases the Waitgroup counter by a value of 1

Mutex:-

Mutex provides a locking mechanism to ensure only one goroutine is accessing the critical section of code/ shared resource at any given point of time. It provides below methods,

1. Lock 2. Unlock

RwMutex:-

ReadWrite Mutex allows all readers to access shared resources at a time , but the writer will lock all the other goroutines.

Once:-

Sync.once ensures a particular function is called only once, regardless of how many goroutines call it.

Ticker In Golang:-

It's used for triggering events at regular intervals, similar to a ticking clock. Tickers are useful for tasks such as running periodic cleanup routines, updating statistics, or triggering regular checks.

```go
func Ticker() {
   ticker := time.NewTicker(1 * time.Second)
   done := make(chan bool)

   go func() {
      for {
         select {
         case <-done:
            return
         case t := <-ticker.C:
            fmt.Println(t)
         }
      }
   }()

   time.Sleep(1600 * time.Millisecond)
   ticker.Stop()
   done <- true
}
```

Context In Golang:-

- In Go (or Golang), "context" refers to a package (context) and a pattern used for handling cancellation, timeouts, and passing request-scoped values across API boundaries and between goroutines (concurrent functions).
- The context package provides a way to pass context between functions and across API boundaries. It's especially useful in cases where you have a chain of function calls or multiple goroutines working on the same task and you need to propagate cancellation signals, deadlines, or request-scoped values.
- Context provides different functions Like
    - **context.Background():** This function returns an empty Context. It's typically used as the starting point for creating contexts. All contexts derive from this background context.
    - **context.TODO():** This function returns a Context that indicates that no context is available. It's used as a placeholder when you intend to add context later but don't have it currently available.
    - **context.WithCancel**(parent Context) (ctx Context, cancel CancelFunc): This function creates a new Context derived from the provided parent context. It also returns a CancelFunc, which can be used to cancel the context and any contexts derived from it.
    - **context.WithDeadline**(parent Context, deadline time.Time) (ctx Context, cancel CancelFunc): This function creates a new Context derived from the provided parent context with a deadline. The context will be canceled after the deadline has passed.
    - **context.WithTimeout**(parent Context, timeout time.Duration) (ctx Context, cancel CancelFunc): Similar to WithDeadline, this function creates a new Context with a deadline, but the deadline is specified as a duration from the current time.
    - **context.WithValue**(parent Context, key interface{}, val interface{}) Context: This function creates a new Context derived from the provided parent context with a key-value pair associated with it. It's commonly used for passing request-scoped values through the call stack.
    - **context.Value(**key interface{}) interface{}: This method retrieves the value associated with the provided key from the context. It's used to access the request-scoped values stored within a context.

For more reference please refer https://pkg.go.dev/context@go1.22.0

Errors and Panics In Golang:-

In Go, errors are represented by the built-in error interface, which is defined as follows:

```
type error interface {
   Error() string
}
```

In Go, errors are handled using explicit checks. Functions that may produce errors usually return an error value as the last return value. Developers are expected to check this error value and handle it appropriately. For example:

```
result, err := someFunction()
if err != nil {
// Handle the error
fmt.Println("Error:", err)
return
}
// Proceed with the result
```

Custom Error Types: While Go encourages simple error handling, you can define custom error types to provide more context or additional information about errors. Custom error types are just types that implement the error interface.

```
type MyError struct {
   Msg string
}

func (e *MyError) Error() string {
   return e.Msg
}
```

Panics In Golang:-

In Go (Golang), "panic" and "recover" are mechanisms provided by the language for dealing with exceptional situations or unrecoverable errors. They allow you to gracefully handle unexpected conditions that might otherwise cause your program to terminate abruptly.

Panic:

- A panic is a built-in function in Go that halts the normal execution of a program.
- It's typically used to signal that something unexpected happened and the program cannot continue in its current state.
- When a function encounters a situation it cannot handle, it can call panic() to trigger a panic. This immediately stops the normal flow of execution and starts unwinding the stack, running deferred functions along the way.
- A panic will propagate up the call stack until it reaches the top-level function of a goroutine, at which point the program terminates and displays the panic message, including a stack trace. Example of using panic():

```
func someFunction() {
    if somethingUnexpected {
        panic("unexpected situation occurred")
    }
}
```

Recover:

- recover() is a built-in function in Go that allows you to handle panics gracefully by regaining control of the program flow. It's used in conjunction with deferred function calls.
- When a function calls recover() within a deferred function, it stops the propagation of the panic and returns the value passed to the panic() function.
- If recover() is called outside of a deferred function or if there is no panic, it returns nil.
- Example of using recover():

```
func handlePanic() {
    if r := recover(); r != nil {
        // Handle the panic gracefully
        fmt.Println("Recovered from panic:", r)
    }
}
```

```
func someFunction() {
   defer handlePanic()

   if somethingUnexpected {
      panic("unexpected situation occurred")
   }
}
```

Defer In Golang:-

In Go (Golang), the defer statement is used to schedule a function call to be executed just before the surrounding function returns. It allows you to ensure that certain cleanup or finalization tasks are performed regardless of how the function exits, whether it returns normally, panics, or encounters a runtime error.

Here are the key points about defer:

1. Syntax: The defer statement is followed by a function or method call, which will be executed when the surrounding function exits. The syntax is as follows:

```
defer someFunction()
```

2. Execution Order: Multiple defer statements within the same function are executed in the last-in-first-out (LIFO) order. The deferred functions are executed in reverse order from the order in which they were deferred.

3. Arguments Evaluation: The arguments to deferred function calls are evaluated at the time the defer statement is executed, not at the time the function is actually called.

4. Deferred Function Calls During Panics: If a panic occurs within a function with deferred calls, the deferred functions will still be executed before the function exits due to the panic. This allows you to perform cleanup operations even in the presence of panics.

5. Common Use Cases:
    a. Closing files or releasing other resources.
    b. Unlocking mutexes.
    c. Logging actions or errors.
    d. Deferred initialization or finalization tasks.

Garbage Collection In Golang:-

Garbage collection (GC) in Go is an automatic memory management process that helps manage memory allocation and deallocation for Go programs. Go's garbage collector is a concurrent, tri-color, mark-and-sweep garbage collector. Here's an overview of how it works:

Tri-color Mark-and-Sweep Algorithm:

Go's garbage collector uses a tri-color mark-and-sweep algorithm. It divides objects into three colors: white, gray, and black.

White: Represents objects that have not been visited during the marking phase.

Gray: Represents objects that have been visited but whose references have not been processed yet.

Black: Represents objects that have been visited and whose references have been processed.

Marking Phase:

The marking phase begins with the roots of the program, such as global variables and the stacks of all goroutines. These roots are scanned, and any reachable objects are marked as gray.

The garbage collector continues to process gray objects, marking their references as gray and changing their color to black.

This process continues until there are no more gray objects left to process.

Sweeping Phase:

Once the marking phase is complete, the garbage collector sweeps through the heap, reclaiming memory from any objects that remain white (unreachable).

The reclaimed memory is added back to the free list and can be reused for future allocations.

Concurrent Execution:

Go's garbage collector operates concurrently with the running Go program. This means that garbage collection happens in parallel with the execution of the program, reducing the impact on application performance.

However, there may be short pauses during garbage collection when the mutator (the running program) is briefly paused to allow the garbage collector to perform its tasks.

GOGC Environment Variable:

*Target heap memory = Live heap + (Live heap + GC roots) * GOGC / 100*

The GOGC environment variable controls the target heap size growth relative to live data size before garbage collection is triggered.

By default, GOGC is set to 100, meaning that the garbage collector will try to keep the heap size within a factor of two of the live data size.

Adjusting the value of GOGC can impact garbage collection frequency and pause times. Higher values may lead to less frequent garbage collection but longer pause times, while lower values may result in more frequent garbage collection but shorter pause times.

In summary, Go's garbage collector uses a tri-color mark-and-sweep algorithm to reclaim memory from unreachable objects. It operates concurrently with the running program and aims to balance the trade-offs between garbage collection frequency, pause times, and overall performance. The GOGC environment variable provides a mechanism for tuning garbage collection behavior based on application requirements.

Go RunTime:-

In Go, the runtime package provides functions and operations related to the Go runtime system, including low-level control over goroutines, the garbage collector, and memory management. Here's an overview of some of the functionalities provided by the runtime package:

Goroutine Management:

runtime.NumGoroutine(): Returns the number of active goroutines.

runtime.Gosched(): Yields the processor, allowing other goroutines to run.

runtime.Goexit(): Terminates the goroutine that calls it.

runtime.LockOSThread() and runtime.UnlockOSThread(): Control the association between the calling goroutine and the operating system thread.

Garbage Collection:

runtime.GC(): Requests a garbage collection to occur.

runtime.SetFinalizer(): Sets a finalizer on an object to be called before it is garbage collected.

Memory Management:

runtime.MemStats: Struct containing memory statistics such as the total allocated memory, heap memory, and garbage collector statistics. This can be retrieved using runtime.ReadMemStats().

runtime.HeapAlloc(), runtime.HeapSys(), runtime.HeapIdle(), runtime.HeapInuse(), etc.: Functions to query specific memory statistics.

runtime.ReadTrace(): Reads execution traces, useful for debugging and performance analysis.

Panic and Stack Traces:

runtime.Stack(): Retrieves a stack trace of the current goroutine.

runtime.Caller(), runtime.Callers(): Retrieve information about the calling goroutine's stack.

OS Threads and Processors:

runtime.NumCPU(): Returns the number of logical CPUs on the system.

runtime.GOMAXPROCS(): Sets the maximum number of CPUs that can be executing simultaneously in the Go runtime.

Profiling and Debugging:

runtime.SetCPUProfileRate(), runtime.SetBlockProfileRate(): Control CPU and blocking profiles.

runtime.SetMutexProfileFraction(): Control mutex profiles.

runtime.SetTraceback(): Controls stack trace printing during panics.

Operating System Information:

runtime.GOOS, runtime.GOARCH: Constants representing the operating system and architecture for which the Go binary was compiled.

runtime.Version(): Returns the Go version.