Ninja Go 1

Programacion de 0 a avanzado
por Park33r

Indice

Introduccion

Este es un manual de Programacion en Golang creado por mi a modo de guia, explicara casi todo sobre Golang, empezara con la ompresion por pantalla, se veran estructuras de datos, interfaces, canales, condicionales y mucho mas. Yo lo he escrito para mi a modo de guia a la hora de crear herramientas de hacking etico.

Github: http://park33r.github.io

twitter: @_Park33r

CONCEPTOS BASICOS DE GO

Comandos	descripcion[i]
go run <archivo>.go	corremos un archivo go
go build <archivo>.go	compilamos la aplicación

Reduccion de tamaño

A la hora de compilar un archivo este puede llegar a pesar mas ya que contendra informacion de depuracion y tabla de simbolos. Para reducir este tamaño tenemos indicadores el cual podemos invocar en el proceso de compilacion (go build):

go build -ldflags “-w -s”

-s desabilitar tabla de simbolos
-w desabilitar generacion DWARF

Compilacion para otras plataformas (compilacion cruzada)

Go permite compilar un archivo para diferentes plataformas como windows / linux / android / mac y otras mas, para hacer esto debemos declarar ciertas restrincciones.una forma de declarar es por la linea de comando que la veremos aquí.

GOOS=”linux” GOARCH=”amd64” go build <archivo.go> -o Programa

con este comando en la terminal indicamos el sistema operativo GOOS y la arquitectura GOARCH seguido del comando para compilar y el nombre del archivo junto al parametro -o el cual indica como se llamara el binario final.

Consultar documentacion de paquete – funcion – metodo o variable

Con el comando go doc en la terminal, podemos consultar la documentacion de cualquier paquete o metodo.

Ejemplo:

obtendremos informacion sobre fmt.Ptintln

go doc fmt.Println

paquetes

Tenemos paquetes que vienen incluidos el cual se importan llamando a ese paquete y tenemos paquetes de terceros. Supongamos que queremos importar paquetes de terceros, para esto necesitamos colocar el comando go get a la hora de importar los paquetes, supongamos que queremos importar el paquete stacktitan/ldapauth:

```
import (“fmt”
        “io/ioutil”
        “github.com/stacktitan/ldapauth”
)
```

como se ve lo hemos importado pero no vamos a poder acceder a el ya que necesitaremos ejecutar el comando go get y la url del paquete, con este comando descargara el paquete y lo pondra dentro del directorio $GOPATH/src

fmt

El paquete fmt viene por defecto en Go y su funcion principal es la de imprimir por pantalla.

fmt.Println(“mi texto”) imprime el texto que le pasamos

x:=”Diego”

fmt.Printf(“hola %s”, x) imprime con opcion de darle un formato

- %s formato string
- %d formato numeros enteros
- %q formato utf8

SINTAXIS

Tipos de datos

Go frece una variedad de tipos de datos como:

int enteros

float numero flotante

string cadenas de texto

bool booleanos

Tambien tenemos mas tipos de datos como int8, 18, 24, 32

Definicion de variables

A la hora de definir una variable tenemos varios metodos.

Metodo 1:

var usuarios string

En este caso indicamos var para decirle que es una variable, el nombre de la variable y el tipo de datos que es esta variable, de tipo string.

Metodo 2

usuarios:=“este es el segundo metodo”

En este caso indicamos el nombre de la variable (usuario) y con := le indicamos que va ha iniciarse y ser de tipo, seguido del valor de la variable y Go automaticamente sabra de que tipo es la variable.

Este metodo solo se puede utilizar dentro de una funcion ya sea main u otra, para declarar una variable fuera de cualquier funcion utiliamos la forma larga:

var miVariable string

constantes

Las constantes en Go son variables de un valor fijas, que no cambiaran su valor y se generan de la suigiente forma:

const miConstante int

En el codigo anterior tenemos una constante llamada miConstante de tipo entero

Array

Los array son una matriz o formacion de datos del mismo tipo, son elementos ordenados en filas. Tienen que tener declarado los espacios de valores que va a tener, es decir si creamos un array de 6 espacios solo tendremos esos 6 espacios pero si no sabemos cuantos espacios tendra ese array podemos ponerle[…] . A la hora de inicializar un array si no le ponemos valor, Go lo inicilizara con el valor de 0 en todos sus espacios.

Inicializacion de array:

var a [3]int	array de tipo entero con 3 espacios sin datos
var a [4]int{10,4,6,2}	array de tipo entero con 4 espacion y con 4 datos creados
var a […] int{1,2,3}	array de tipo entero con espacios según los valores que tenga

Para este ultimo codigo tenemos que inicializar el array con algun valor, es decir este codigo daria error: var a[...]int

```
a :=[…] int{
        1,
        2,
        3,
}
```

este ultimo codigo es un buen ejemplo para que puede servir […], podemos comentar cualquier linea con // para que no se imprima, y agregando […] evitamos tener que cambiar el valor de espacio ya que el array se acomodara al espacio que tiene de valores, si tiene cuatro valores el array sera de cuatro espacios, si comentamos un valor quedando tres valores el array se modificara automaticamente a tres valores de espacios.

Multidimencional

Tambien tenemos array de dos dimenciones el cual podemos crear tantos array de tantos espacios cada uno:

En este ejemplo tenemos 10 array de 5 espacios cada uno inicializados a 0

```
var matriz [10][5] int
```

agregando datos:

en este codigo le agregamos un valor de 2 al array llamado matriz en el 5º array en su pocision 2º

```
var matriz[5][3]= 2
```

en el array (a) en la posicion 1º le agregamos un valor de 20

```
a[1]=20
```

acceder al indice:

```
fmt.Println(a[1])
```

imprimir el valor del array (a) del indice 1.

fmt.Println(len(a)) nos devuelve la longitud, el 0 cuenta como 1

Podemos hacer comparaciones de array con el simbolo de comparacion == comparando mismo tamaño mismo tipo de dato y mismo elementos dentro, dandonos true o false.

Los array tiene como contra que no son dinamicos y que a la hora de pasarlo como parametro a una funcion, estaremos pasando una copia y no el original.

Slice

Los slice son estructuras que representa una porsion o parte de un array.

Su inicializacion es parecida al del array:

var x []int	slice de tipo entero sin valores
z := []int{1,5,3}	silce de tipo entero con 3 valores
s:=make([]int, 5)	slice creado con make sin valor de tipo entero y de longitud del slice de 5
fmt.Println(len(z))	imprime longitud del slice
fmt.Println(cap(s))	imprime la capacidad de s

el slice es una porcion de un array, asi que si tenemos un array de 10 espacion y creamos un slice de 5 espacios la longitud del slice sera de 5 pero su capacidad es la del array, 10 espacios.

r:=make([]int, 5, 20)	slice vacio de enteros con 5 de longitud y 20 de capacidad

Dejare un pequeño codigo el cual combinamos los array con los slice.

[CODIGO SILCE Y ARRAY]

Append

Con append podremos agregar datos al final de un slice, su sintaxis es la siguiente:

miSlice:=make([]int, 5, 11)	inicializamos slice de 5 de espacio y 11 de capacidad
miSlice=[]int{0,1,2,3,4,5}	le agregamos datos al slice
miSlice=append(miSlice, 6)	agregamos el 6 al slice miSlice

Explicacion:

miSlice=append(miSlice, 6)

cuando agregamos (append) un dato mas a un slice y este slice tiene la longitud de 5 y al agregar supera ese 5 de longitud,lo que hace append es crear un array subyasente con su capacidad igual al doble de la longitud del slice anterior, teniendo de esta forma mas espacio para agregar y asi es que tenemos un slice el cual podemos indicar la longitud y capacidad pero agregar mas de lo indicado sin problemas.

Copy

La funcion copy nos permite copiar los datos de un slice a otro slice,su sintaxis es la siguiente:

origen:=[]int{9,10,11}

destino:=[]int{12,13,14}

copy(destino, origen)	copia los datos de origen y los pone en destino quedando destino igual que origen

Si el slice de origen tenemos 10 datos y el slice de destino tiene longitud de 5, se copiaran solo la cantidad de datos que pueden entrar en el slice de destino, (5)

Si tenemos un caso diferente que el destino tiene 10 de longitud y el origen tenemos 2 numeros nada mas, copiara los 2 numeros al destino dejando los otros 8 numeros de destino sin modificar.

Mapas

los mapas en otros lenguajes equivalen a diccionarios o tablas hash, la sintaxis para crear un mapa es la siguiente:

Ejemplo 1 mapa:

cuidades:= make(map[string]string, 5)

Iniciamos una variable llamada ciudades,con make indicamos que vamos a crear,un mapa y tendra datos de tipo string [string] como clave y datos de tipo string como valor, y tendra reservadado 5 elementos de espacio.

Ejemplo 2 mapa:

cuidades:=make(map[string]string)

En este codigo tenemos la diferencia que no le inidcamos la capacidad de elementos que tendra como inicio el mapa,la diferencia de este ejemplo con el ejemplo 1 es su optimizacion a la hora de ejecutar el binario,ya que en el metodo 1 se iniciara el mapa vacio pero con 5 espacios creado en memoria, pudiendo sobrepasar esos espacios sin problema ya que los mapas son de espacio de almacenamiento dinamicos,en el ejemplo 2, como no tenemos indicado que tanto espacio guarde en memoria al comienzo, cada vez que agregemos datos al mapa, este tendra que iniciarlizarse cada vez que le agregemos datos para generar ese espacio y esto consume recursos.

Ejemplo 3 mapa:

```
puntuacion:=map[string]int{
        “España”:10,
        “Francia”:20,
        “Alemania:50}
fmt.Println(puntuacion[“España”])
```

En esta ocacion, inicializamos una variable la cual almacenara datos de tipo mapa y tendra como clave string y enteros como valor.

Agregando datos al mapa del ejemplo 1:

cuidades[“Europa”]=”España”

a la variable ciudades le agregamos la clave Europa con su valor España.

A la hora de llamar a esos valores podemos llamarlo imprimiendo la variable y haciendo referencia a su clave el cual llamara al valor entre corchetes: fmt.Print(ciudades[“Europa”])

eliminando elementos

tenemos una funcion para eliminar elementos de un mapa, esta funcion es la siguiente:

delete(ciudades, “Francia”)

eliminar del mapa ciudades la clave:valor que tiene como clave Francia

Esta funcion no reporta errores, es decir si eliminamos un clave:valor que no existe no nos reportara error ni nos dira que esa clave:valor no existe o ya fue eliminado, por tanto si llamamos a un elemento que no existe el mapa devolvera el valor 0 de int que seria vacio porque no existe.

Agregando datos con incrementacion

Como vimos si imprimimos un elementos clave:valor que no existe, Go devolera 0 que significa vacio, nil o que no existe, por tanto si llamamos a un elemento que no existe y le aplicamos un incremento este elemento pasara a tener un valor distinto a 0 por lo tanto pasara a existir con ese valor, supongamos que en un mapa llamado nombres NO tenemos el elemento clave:valor juan:80

nombres:=make(map[string]int)	//Creamos un mapa de tipo clave string, valor entero
fmt.Println(nombres[juan]	//imprimimos llamando a la clave juan
0	//devuelve 0 ya que juan no existe en el mapa
nombres[“juan”]++	// le agregamos 1 (++) como valor de la clave de juan
fmt.Println(nombres)	//imprimimos todos los elementos de nombres juan:1 // nos devuelve juan 1 ya que le agregamos una unidada.

No podemos obtener la direccion de memoria de los mapas con punteros ya que los mapas son dinamicos y no guardan los valores en una lista y siempre se imprime diferente posicion.

Range

Los range es un tipo de bucle que nos permite iterar en varios datos, nos devuelve dos valores, el valor de lo iterado y el indice de lo iterado, su sintaxis es la siguiente:

```
nombres:= []string{
“ana”,
“fernanda”,
“julieta”,
“sabrina”,
for indice, datos:= range nombres {
        fmt.Printf(“La pocision es %d y su nombre es %s”, indice, datos)
```

En este codigo creamos un slice y lo iteramos en un bucle for utilizando range.

En el codigo anterior nos imprimira la lista con su respectivo indice del 1 al final de los datos.

Cuando utilizamos range nos devuelve siempre dos valores, el cual lo almacenamos en esta ocacion en una variable indice ya que ese dato sera un numero iterado del 1 hasta el final de los datos y el siguiente valor que nos devuelve es el dato del slice que le hemos pasado

Si en alguna ocacion no queremos utilizar uno de los dos valores que nos devuelve range, debemos inidcarselo al programa con un _ (guion bajo) en donde iria la variable de ese valor ya que Go nos obliga a poner siempre dos variables aunque utilizemos una.

```
for _, datos:=range nombres {
        fmt.Printf(“El nombre es: %s”, datos)
```

en el codigo anterior utilizamos solo la variable datos y no la variable indice.

Sin embargo si solo queremos el indice no hace falta poner _ (guion bajo) en donde van los datos:

```
for indice:= range nombres{
        fmt.Printf(el indice es %d”,indice)
```

Punteros

A la hora de pasar una variable a una funcion, Go lo que hace es que pasa una copia de la variable a la funcion por lo tanto estaremos trabajando en la funcion con una copia sin tener la posibilidad de modificar el valor real de esa variable, esto es un ejemplo:

```
func incrementar(x int) {
         x++
}
func main() {
        x := 10
        incrementar(x)
        fmt.Printf("El valor de x fuera de la funcion es %d", x)
}
```

En este codigo enviamos la variable x (10) a la funcion incrementar el cual recibe una variable que la llamara x y es de tipo entera, la funcion lo que hace es aumentar en 1 la variable x, despues la funcion main imprime el valor de x,el resultado es 10, es decir que 10 no incremento en 1 como hace la funcion incrementar ya que a la hora de pasar x a la funcion pasamos una copia y incrementa esa copia, dejando la variable original en su estado original. 10

Es aquí donde entran los punteros, & este simbolo indica la direccion de memoria donde esta guardado el valor de la variable que le pasemos, el simbolo * indica que convierta de la direccion de memoria a el valor que es, un codigo de ejemplo el mismo que anterior pero con puntreros

```
func incrementar(x *int) {
```

```
        *x++
}
func main() {
        x := 10
        incrementar(&x)
        fmt.Printf("El valor de x fuera de la funcion es %d", x)
}
```

explicacion: pasamos la direccion de memoria de x a la funcion incrementar,(la cual si la editamos o cambiamos el valor, cambiaremos el valor de la variable.)

la funcion incrementar recive un valor que es el la direccion de memoria de x que le pasamos, ese valor lo almacenara en una variable llamada x y sera entero, pero antes, pide que convierta esa direccion de memoria que pertenece a x al valor que es, en este caso 10

incrementarmos x refiriendonos al valor (*)

imprimimos x y nos dara 11, ya que lo que le pasamos fue la direccion de memoria de x (&x) y no una copia (x).

Estructura de control

tenemos varias estructuras de control:

- if – else if – else
- switch
- for

Dentro de for podemos iterar mapas o array de la siguiente forma:

```
numeros:=[] int{3,7,4,8}
for index, valor := range numeros {
        fmt.println(index,valor
```

en esta ocacion iteramos con for un mapa que tiene como valor 3,7,4,8

aplicaremos range que da dos valores que lo guardaremos en dos variables que estan a la derecha. Index la cual indexara de 0 hacia arriba y valor el cual imprime el valor.

Manejo datos estructurados XML

Muchas veces no tocara tratar con estos tipos de datos el cual son datos con cierta codificacion, Go contiene paquetes para procesar estos datos: encoding/json & encoding/xml

Los dos paquetes pueden armar y desarmar ese tipo de dato.

XML

```
type Estructura struct{
    Juan string `xml:"id,attr"`
    Segundo Pedro `xml:"padre>hijo"`
}
```

los valores entre comillas simples inversas (`) son etiquetas de campo, las etiquetas de campo siempre comienzan con el nombre de la etiqueta (xml seguido de : y directivas entre comillas dobles). En este caso la directiva indica que Juan debe tratarse como un atributo (attr) de nombre id, no un elemento y Pedro debe encontrarse en un sub elemento de padre nombrado hijo

JSON

Para trabajar con datos de tipo JSON tenemos el paquete encoding/json el cual es parecido a encodig/xml, primero cramos una estructura con los datos que pasaremos a formato JSON

```
type Banco struct {
    Cuenta int      `json:”Numero de cuenta”`
    Titular string  `json:”Titular"`
    Vencimiento string    `json:”vencimiento,omitempty”`
}
```

Como podemos ver tenemos una estructura con tres tipos de datos, a su derecha tenemos las etiquetas para indicar a JSON en que parte del documento y que rol tendran, el campo omitempty significa que ese campo puede ser declarado vacio omitiendo el valor sin ningun problema.

Le agregaremos valor a los campos de la estructura banco:

```
maria:=Banco{001,”Maria”,””}
json,error := json.Marshal(maria)
if error != nil {
    fmt.Println(“Error al crear la estructura JSON”)
}
fmt.Println(string(json))
```

En el codigo anterior le agregamos los datos a la estructura banco creando una variable maria, despues con json.Marshal lo pasamos a formato JSON quedando el resultado de ejecucion del programa de esta manera:

{“Numero de cuenta”:1,”Titular”:”maria”,”Vencimiento”:””}

Funciones

Es una rutina que devuelve o no devuelve un valor, podemos tener varias funciones que dentro tendra un bloque de codigo el cual estara entre llaves {} con su logica. Se llamara a esas funciones desde la funcion principal main() u otra funcion y le podemos pasar datos o parametros de funcion a funcion, su sintaxis es la siguiente:

```
func cajafuerte(modelo string) {
        fmt.Printf("modelo %s", modelo)
```

esta es una funcion que no devuelve nada, lo unico que hace es imprimir un parametro que se ha pasado, el parametro es de tipo string y se guardara en una variable dentro de la funcion llamada modelo y solo funcionara dentro de esa funcion esta variable

```
func main() {
        cajafuerte("Marchall V5")
{
```

main() es la funcion principal de Go, es por la cual va a empezar a circular el flujo o programa, en esta ocacion Go empieza en main, entra en el y llama a la funcion cajafuerte(), pasandole un string como parametro, el cual al llegar a la funcion cajafuerte esta guardara ese string en una variable llamada modelo.

```
func total(valor1 int, valor2 int) int{
        return valor1 + valor2
}
```

En esta funcion tenemos lo mismo que la funciones anterior solo que en esta recibe dos valores de tipo entero (int) que se almacenaran en variables llamadas valor1 y valor2, despues devolvera un entero, ese entero es el resultado de la suma de valor1 + valor2.

No basta con solo crear la funcion, despues hay que llamarla o invocarla para que se ejecute:

```
fmt.Println(total(10, 25))
```

llamamos a la funcion total pasandole los dos parametros que seran de tipo entero, el resultado se imprimira por pantalla.

Tipo 2:

```
func descuento(valor1, valor2 int) (resultado int) {
        resultado = valor1 – valor2
        return
```

}

En esta funcion como tenemos dos parametros que le llegan y son del mismo tipo, podemos poner los nombres de las variables y al final el tipo de dato que es mientras las variables sean del mismo tipo. Despues indicamo que tendra una variable llamada resultado que sera de tipo entera, esa variable resultado sera,el resultado de valor1 – valor2 y al final tenemos un return. El return lo que le indica a la funcion es que el codigo termino entonces la funcion devolvera la variable resultado.

Es casi lo mismo que la anterior solo creada de diferente forma.

Llamamos a la funcion:

fmt.Println(descuento(100, 50))

Tipo 3

multiplicar:=func(numero1, numero2 int) int {

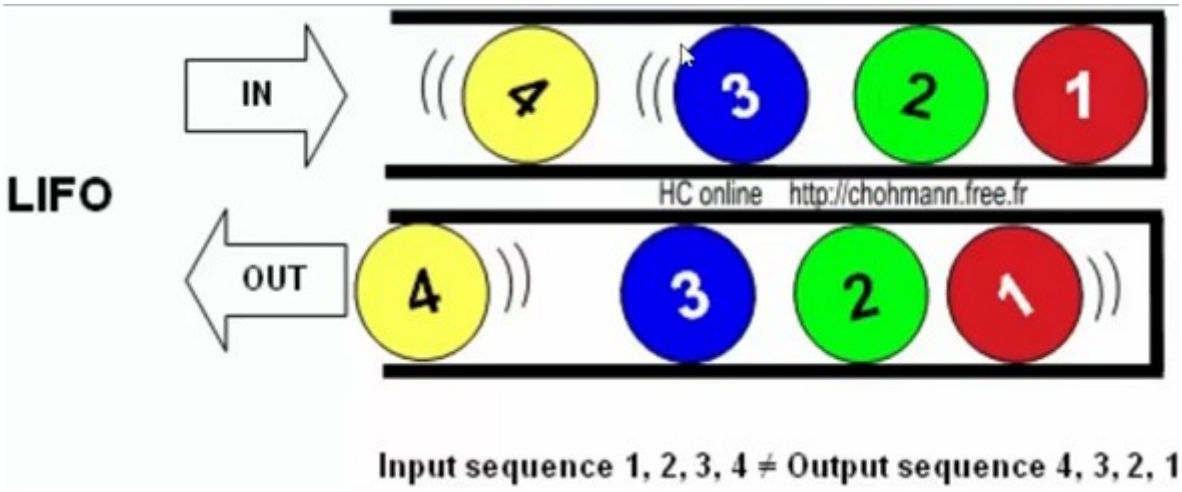
return numero1 * numero2

multiplicar(10,10)

Esta ultima es una funcion anonima dentro de una variable llamada multiplicar.

Pila de llamada

una funcion puede llamar a otro, y esta otra a otra y esta otra a otra. Las funciones en este caso se almacenan en una pila de llamada, aquí un ejemplo de como entra la funion 1,2,3,4 y como salen 4,3,2,1



Parametros multiples

Cuando creamos una funcion y le indicamos que tendra como entrada dos datos, al llamar a la funcion tenemos que enviarle como argumento dos datos, existen casos que aveces no sabremos cuantos datos recibira si uno dos tres o los que sea, para esto podemos crear la funcion de esta manera:

```
func banco(cuentas ...int) int{
        for _, datos:= range cuentas{
        fmt.Println(datos)
}
```

Tenemos una funcion llamada banco que recibira argumentos de tipo entero que se guardaran en la variable cuentas, los tres puntos, significa que vamos a recibir parametros pero no sabemos cuantos parametros vamos a recibir asi que podemos crear otra funcion y enviarle tres, cuantro, diez paramentros sin problema

```
func main() {
        banco(1,2,3,4,5)
        banco(1,2)
        banco(1,2,3,4,5.6.7.8)
}
```

En este codigo tenemos tres llamadas a la funcion banco, una le enviamos cinco parametros en la otra dos y en la otra ocho parametros, esto se hace sin problemas gracias a que la funcion banco tiene (…) que significa que acepta un numero indeterminado de parametros de tipo enteros.

Podemos avanzarlo mas, y aprovechar que la funcion banco puede recibir una cantidad de parametros indefinidos podriamos crear un slice y enviarle este slice, la forma correcta de hacer esto es asi:

```
cuentas:=[]int{
        20,
        40,
        13,
}
banco(cuentas…)
```

como se puede ver es agregandole tres puntos (…) despues de indicarle el nombre del silce a enviar, esto se debe porque la funcion banco recibe numeros enteros y no numeros enteros de un slice, por lo tanto agregandole estos tres puntos le indicamos a Go que no tome en cuenta que es un slice.

Retorno multiple

Existe la posibilidad de que una funcion acepte un parametro como entrada, haga su logica con ese parametro y devuelva mas de un valor como se puede apreciar en el siguente codigo:

```
func multiplicar ( dinero int) (n1, n2, n3 int){
        n1=dinero * 2
        n2=dinero*3
        n3=dinero*4
        return
```

o

```
func multiplicar (dinero int) (n1, n2, n3 int) {
        return dinero*2, dinero*3, dinero*4
```

llamando a esta funcion y enviandole un parametro como argumento tendremos tres parametros como retorno ya que con solo un parametro que le hemos enviado, hara tres acciones que las devolvera.

Tambien podemos asignarle variables a los datos retornados por la funcion mutliplciar:

```
n1, n2, n3:= multiplicar(10)
fmt.Println(n1, n2, n3)
```

imprimimos n1 que es de tipo funcion multiplicar *2, imprimimos n2 que es de tipo funcion multiplicar *3 y imprimimos n3 que es de tipo funcion multiplicar *4

Podemos crear la funcion y no nombrar las variables de retorno en un principio, solo decirle de que tipo son, crear esas variables y despues en retorno indicarle que variables retornara

```
func multiplicar( dinero int) (int, int, int) {
        b1:= dinero *2
        b2:= dinero *3
        b3:= dinero *4
        return b1, b2, b3
}
```

Funciones anonimas

Una funcion anonima es una funcion normal pero que no tiene nombre:

func ()

y se la invoca de esta manera:

()

funcion dentro de funcion

En Go no podemos declarar una funcion dentro de otra funcion de esta manera:

```
func main() {
        func barra() {
                fmt.Println(“funcion dentro de main declarada”)
        }
}
```

Esto que es la funcion barra declarada dentro de la funcion main nos dara error, para hacer esto existen varias formas ya que las funciones en Go son tambien un tipo de dato como un string un entero o un booleano.

Tenemos una funcion llamada barra la cual imprime algo:

```
func barra(cadena string) {
        fmt.Println(cadena)
{
```

Despues tenemos la funcion main que es la principal la cual vamos a inicializar una variable de tipo funcion:

```
func main() {
        miFuncion := barra
}
```

Ahora tenemos la variable miFuncion de tipo funcion como lo es la funcion barra, pudiendo pasarle un parametro para que haga su funcion de imprimir:

```
miFuncion(“ok”)
```

La funcion miVariable tiene acceso a todas las variables dentro de la funcion main ya que esta dentro de ella.

Funcion que devuelve funcion

Una funcion puede devolver una funcion diferente dependiendo de su logica:

```
retornarFuncion() func() int {
        y:=0
        return func() int {
            y++
            return i*i
        }
    }
```

En este caso la funcion retornar funcion devolvera una funcion anonima que le sumara 1 a la variable y, despues la multiplicara.

```
miFuncion:=retornarFuncion()
miFuncion2:=retornarFuncion()
```

miFuncion y miFuncion2 son variables derivadas de funcion retornarFuncion no son iguales, es decir si llamamos a miFuncion, miFuncion2 no se modificara.

```
fmt.Println(miFuncion())
```

Defer

la palabra defer en Go es una palabra reservada para una funcion interna de Go, cuando le agregamos la palabra defer a una funcion o algun metodo lo que hace esta palabra es hacer que sin importar en que sitio del codigo este esa funcion invocada se ejecutara por ultimo, asi de simple, defer sirve para indicar que eso se ejecutara por ultimo, y se ejecutar si o si,aunque el programa falle, la unica forma que no se ejecute defer es un fallo del ordenador que no tenga que ver con el codigo.

Panic & recover

Todo codigo muchas veces puede tener errores a la hora de su ejecucion, gracias a panic podemos controlar esos error, su funcionamiento y sintaxis son simples,al agregar panic() el programa se cierra y imprime un texto que le pasamos como parametro:

```
panic(“error en ejecucion)
```

con esto controlamos los errores sabiendo donde fallo y cerrando el programa.

Con recover lo que hacemos es imprimir el error sin cerrar el programa si es que el programa puede seguir:

```
f, err:=os.Open(“texto.txt”)
    if err != nil {
```

```
        panic(err)
}
```

Abrimos un archivo y almacenamos su informacion en la variable f y su error si es que existe en la variable err, si las variable err es distina a nil, entonces ejecutar recover con el contenido de err que indicara el error.

Tipos

Los tipos son tipos de datos, pueden aver datos de tipo enteros (numeros) datos de tipo string (cadenas) y otros tipos mas de datos,pero tambien podemos asignar un tipo de dato especial con esta sintaxis:

type Dinero int	Creamos un tipo de dato que se llama Dinero (es global porque empieza con mayuscula, ese tipo de dato llamado Dinero es de tipo entero
var credito Dinero	inicializamos una variable y le asignamos un tipo de dato,ese tipo de datos sera de tipo Dinero el cual es un tipo de dato entero.

Gracias a TYPE podemos crear nuestro tipo de dato personalizado.

Estructuras

Las estructuras en Go pueden compararse con las clases de otro lenguaje de programacion, si sintaxis y creacion son la siguiente:

```
type Persona struct {
        Nombre string
        Cuenta int
        }
```

En este codigo podemos ver que creamos una estructura llamada Persona que contiene dentro campos llamados Nombre que es de tipo string y Cuenta de tipo entero

Como vimos en la seccion anterior estructura es un tipo de dato que contiene Nombre y Cuenta dentro, asi que podemos inicializar una variable que se a de tipo estructura:

tipo 1:

var jefe Persona

tipo 2:

jefe:=new(Persona) //En este caso tenemos un puntero a estructura por tanto al llamarlo es *jefe

inicializamos una variable de tipo estructura llamada jefe y tendra dentro las variables de la estructura Persona Nombre y Cuenta, para llamar a esas variables internas y asignarle un valor se utiliza el punto (.) seguido del nombre de la variable:

tipo 1:

jefe.Nombre = “Park33r”

jefe.Cuenta =3333

fmt.Println(jefe) el cual imprimira {park33r 3333}

tambien podemos indicarle que solo imprima el nombre:

fmt.Println(jefe.Nombre)

tipo 2:

jefe:= Persona{nombre: “David”, Cuenta: 4444}

esta es otra forma de inicializar una variable de tipo Persona asignandole los valores a sus variables.

Si tenemos una estructura pequeña podemos inicializar las variables de esta forma:

jefe:=Persona{“David”, 3333}

Sin hacer falta indicarle el nombre de la variable pero tiene que tener la misma posicion, si la estructura persona tiene primero Nombre y despues Cuenta, los valores irian en esa posicion.

Podemos declarar una estructura con sus campos anonimos, solo indicanole el tipo de dato que es, esto nos sirve para tener una estructura dentro de otra:

```
type Estudiante struct {
        Persona
        Privilegios bool
}
```

en este pedazo de codigo tenemos delcarado una estructura llamada Estudiante que tendra dentro una variable llamada privilegios que es de tipo booleana y tambien como vemos tiene un tipo de dato llamado Persona, Persona es el tipo de dato no el nombre de la variable, al tener ese tipo de dato Persona dentro de la estructura Estudiante, se le asigna automaticamente las variables de Persona quedando la estructura Estudiante con las variables:

Nombre,Cuenta,Privilegios

Ya teniendo la estructura vamos a crear a una persona,

var marta Estudiante

marta.Nombre=”Marta”

marta.Cuenta=6666

marta.Privilegios=True

O tambien podemos declarar de esta manera

```
marta:=Estudiante{
        Persona{
```

```
        Nombre:”marta”,
        Cuenta:6666,
    }
    true,
}
```

Como se puede ver separamos y asignamos mediante llaves, los valores de las variables de Persona separadas con llaves de las variables de Estudiante.

Si tenemos una estructura principal llamada UNO y tiene como campos nombre y edad, y tenemos otra estructura llamada DOS que tiene como campo estructura UNO, nombre y apellido al hacer:

```
var david DOS
david.nombre
```

a quien llamara? a nombre de la estructura UNO o DOS? La respuesta es que llamara a la estructura DOS por ser la estructura padre, para acceder a nombre de la estructura UNO y no nombre de la estrucutra DOS tenemos que indicarla:

```
var david DOS
david.UNO.nombre        //accedemos a nombre de la estructura UNO
david.nombre            //accedemos a nombre de la estructura DOS
```

o podemos declararlo asi:

```
david:=DOS{UNO{“david”, 33}”jose”,”fernandez”}
estructura DOS
estructura UNO dentro de estructura DOS
```

Metodos

Una estructura es una plantilla, no hace nada por si solo, es por eso que podemos agregarme un metodo para que haga alguna accion, la forma de declarar un metodo es la siguiente, primero necesitamos tener una estructura:

```
type Persona {
    Nombre string
    Edad int
}
```

y despues le declararemos un metodo a esa estructura:

```
func (p Persona) diez() int {
    return p.Edad + 10
```

}

en este codigo tenemos un metodo declarado, que tiene una variable p de tipo persona (tendra sus variables de persona tambien) tendra una funcion llamada diez() que devolvera un entero (int) ese entero seria el resultado de la edad + 10

asi que creamos una persona:

var juan Persona

juan.Nombre= “juan”

juan.Edad=20

fmt.Print(“la edad de juan dentro de 10 años sera de: ”, juan.diez())

Como juan tiene una estructura de Persona y Persona tiene un metodo llamado diez, entonces juan puede llamar a ese metodo.

Tambien podemos crear metodos que recibira un parametro o mas de uno:

```
func (p Persona) diez(i int) int{
        return i + 10
}
```

en este ejemplo vemos que el metodo recibira un parametro de tipo entero que se guardara en la variable i, despues retornara un entero.

Si lo que queremos es aumentar el valor de una variable por medio de un metodo, tenemos que utilizar punteros a esas variables de la estructura, porque el metodo trabaja con la copia de esa estructura.

Para esto haremos un puntero de la estructura a la que le indicamos que seran los datos

```
func (r *Persona) diez(i int) int{
        r.Edad = i + 10
```

con este codigo entre llaves modificaremos cualquier variable que tenga como estructura Persona y llama a este metodo.

Interfaces

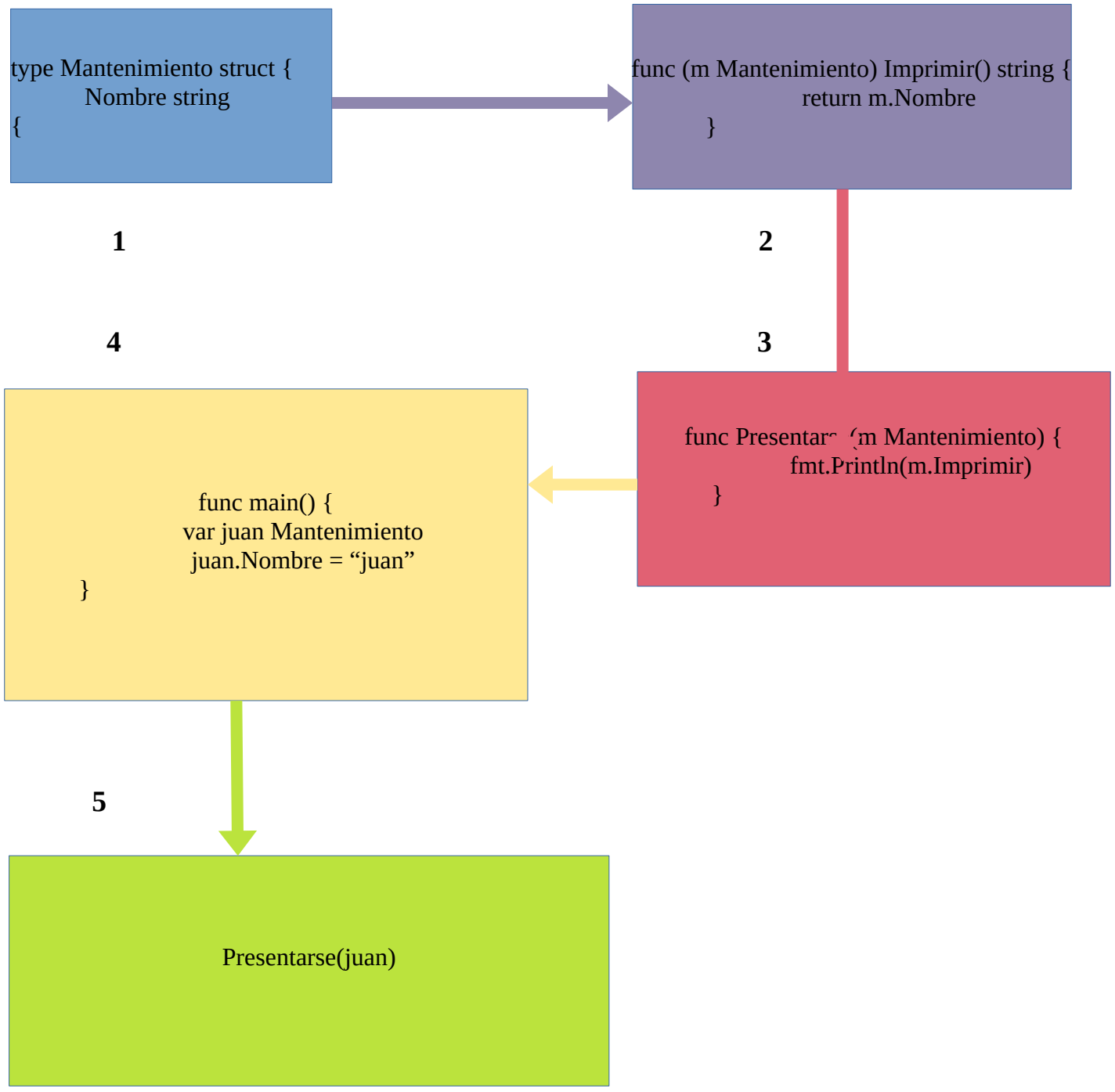
Tenemos una estructura con campos de Nombre de tipo string, esta estructura tiene un metodo el cual retorna el valor de Nombre, tambien cuenta con una funcion, que imprimira por pantalla el resultado del metodo imprimir():

```
type Mantenimiento struct {
        Nombre string
}

func (m Mantenimiento) imprimir() string {
```

```
        return m.Nombre
}
func Presentarse() {
        fmt.Println("su nombre es", imprimir())
func main() {
        var juan Persona
        juan.Nombre="juan"
        Presentarse(juan)
```

Pondre un diagrama para que sea mas claro el flujo de los datos:



Func main() llama a la funcion Presentarse enviandole a juan que es un dato de tipo Mantenimiento (estructura)

func presentarse() recibe el tipo de dato juan y lo imprime con fmt.Println llamando a m.imprimir el cual este metodo retornara Nombre que Nombre es juan.

En este ejemplo tenemos una estructura un metodo una funcion y una funcion principal, ahora pasaremos a explicar para que srive un interfaz.

Imaginemos que tenemos un codigo como el datagrama anterior, pero tendremos una estructura mas la cual se llamara Sistemas y tendra en sus campos la estructura Mantenimiento y otro campo Rango tipo string, asi que la estructura al tener Persona tendra Nombre ya que “Hereda” los campos de la estructura Mantenimiento.

```
type Sistemas struct {
        Mantenimiento
        Rango int
}
```

Aunque la estructura Sistemas tiene como “herencia” los campos de Mantenimiento, Sistemas no puede usar sus metodos ni funciones ya que a ellas le llega la estructura Mantenimiento, para esto debemos crear una interfaz.

Creamos un interfaz:

```
type Central intreface {
        imprimir() string
{
```

expliacion mirando el diagrama:

Toda ESTRUCTURA (cuadro 1) que tenga como METODO (cuadro 2) Imprimir() pertenecera a la INTERFAZ Central.

Ahora en vez de pasarle la estructura MANTENIMIENTO a la funcion presentarse, le pasaremos una interfaz CENTRAL que es un tipo de datos, aparte la interfaz esta compuesta de estructuras que tengas esos metodos, y la estructura SISTEMAS tiene los metodos de la estructura MANTENIMIENTO (imprimir()) ya que tiene la estructura mantenimiento dentro “heredando” sus variables y sus y metodos.

Asi que ahora la funcion Presentarse recibira una Interfaz que tiene dos estructuras

```
func Presentarse(c Central) {
        fmt.Printf(“Su nombre es %s”, c.Nombre())
```

Podemos agregar estructuras que cumplan los requisitos de la interfaz de esta forma

```
var Todo Central
```

Todo = Sistemas

fmt.Printlf(“Todo: “, Todo)

fmt.Println(“nombre: ”, Todo.Nombre)

Podemos inicialiar una variable de tipo intrefaz y asignarle cualquier estructura que cumpla lo que indica la interfaz con respecto a los metodos:

var Todo Centrak

Todo=Sistemas

fmt.Println(Todo)

Todo=Mantenimiento

fmt.Println(Todo)

ACLARACION

solo tendremos acceso a los metodo que tiene la interfaz sin importar cuantos metodos mas tengan las estructuras.

Manejo de errores

En Go tenemos que manejar los errores nosotros mismos, pongamos un ejemplo que tenemos una funcion con un switch el cual analizara un dato que le pasemos pero antes, crearemos unas variables con el paquete error para manejar cuyos errores:

```
var (
        ErrorCodigoIncorrecto = errors.New(“El codigo es incorrecto”)
        ErrorSinCodigo = errors.New(“No se ha ingresado ningun codigo”)
)
func codigo(pin int) (err error) {
        correcto:= 1234
        switch pin {
                case 1111:
                        return ErrorCodigoIncorrecto
                case 1234:
                        return fmt.Printf(“contraseña correcta: %d”, correcto)
                case :
                        return ErrorSinCodigo
                default:
```

return fmt.Errorf(“contraseña incorrecta”)

Como vemos tenemos una funcion con un switch que recibe un entero que la almacena en una variable llamada pin y retornara una variable de tipo error

FALTA TERMINAR

Concurrencia Gorutinas

la concurrencia permite ejecutar varios procesos a la vez utlilizando gorutinas que son funciones o metodo que se pueden ejecutar simultaneamente.

Se puede describir como hilos ligeros ya que consume menos que los hilos reales o comunes, para crear gorutinas usamos la palabra clave go antes de llamar a un metodo o funcion que desea ejecutar al mismo tiempo.

Gracias a gorutinas podemos ejecutar un proceso mientras otro proceso se este ejecutando sin tener que esperar a que termine.

Al ejecutar una gorutina se crear un trabajo aparte el cual estara esa funcion separada de lo que viene siendo la ejecucion del progama, el problema que al crear una gorutina y esta se esta ejecutando en un proceso separado de la ejecucion principal del progama, cuando el proceso principial termina, el programa se termina sin importar si la gorutina se termino o no, para esto tenemos un paquete llamado sync el cual podemos utilizar para controlar esto y indicarle al programa cuando las gorutinas termino para que el programa principal pueda terminar, esta es la sintaxis del paquete sync:

Add()	Le indicamos cuantas gorutinas se van a ejecutar
Wait()	Bloquea el programa hasta que las gorutinas terminen
Done()	disminuye el contador de 1 en 1, se usa para determinar la finalizacion de una gorutina

Entonces si tenemos tres gorutinas para ejecutar, inicializamos una variable llamada esperar de tipo sync.WaitGroup, despues con con espera.Add(3) le estamos indicando que se ejecutaran tres gorutinas, ejecutamos las tres gorutinas y antes de finalizar el codigo, en lo ultimo de todo agregamos esperar.Wait() para que el programa espere a que esperar.Add(3) se quede en esperar.Add(0) y esto lo hacemos con esperar.Done() el cual lo pondremos dentro de la gorutina para disminuir esperar.Add(3) de 1 en 1.

codigo de ejemplo:

```
var esperar sync.WaitGroup

func main() {
    esperar.Add(3)
    fmt.Println(“iniciamos el programa…”)
    go Impresion(“La primera”)
```

```
        go Impresion("La segunda")
        go Impresion("La tercera")
        fmt.Println("esperando a que terminen las go rutinas")
        esperar.Wait()
        fmt.Println("gorutinas terminadas")
func Imprimir(valor string) {
        defer esperar.Done()
        for i:= 1; i <= 10; i++ {
                fmt.Printf("ejecutando s%, %d veces", valor, i)
        }
}
```

ejecutar el programa y poner en la variable esperar un valor de 3, ejecutar las gorutinas y al llegar a esperar.Wait bloquear el progrmama ahi, hasta que esperar.Add valga 0.

En la gorutina Imprimir tenemos un defer esperar.Done() que disminuye esperar.Add a 2 y despues a 1 en su proxima ejecucion y a 0 en su proxima ejecucion, cada vez que se ejecuta disminuye la variable esperar que le asignamos un 3 con esperar.Add(3), cuando la variable esperar este en 0, esperar,Wait() se desbloqueara y seguira el flujo del codigo.

Canales

Despues de ver y trabajar con las gorutinas tendremos que tener un mecanismo para que una gorutina se comunique con otra gorutina y asi tener sincronizado los trabajos, o enviar un dato de una gorutina a otra gorutina, para esto existe el tipo de dato channel o canal, esta es su sintaxis

```
ch:=make(chan string)
```

al pasar la variable ch estamos pasando por referencia como un slice y no por valor, asi que no hace falta pasarle con un puntero.

En ch estamos declarando un canal de tipo string

Los canales envian datos y reciben, esta son sus dos formas:

ch ← varaiable	envio del varlor de la variable al channel ch
variable ← ch	recibo datos a la variable desde el channel ch

Tenemos dos gorutinas las cuales se ejecutan al mismo tiempo de forma concurrente gorutina primera y gorutina segunda, la gorutina primera imprime por pantalla un valor que la gorutina segunda le envia, y despues de imprimir hace una pausa de 5 segundos, el problema esta en que la gorutina segunda esta ejecutandose "paralelamente" y si la gorutina primera tiene una pausa de 5 segundos que pasa con la gorutina segunda que esta enviandole constantemente el dato para que lo imprima?

Para esto existen los canales, imaginemos que los canales es un recipiendo donde la gorutina segunda coloca un dato, y espera a que el recipiente este vacio para seguir colocando datos, la gorutina primera lo que hace es quitar ese dato del reciiente dejandolo vacio y asi la gorutina segunda lo vuelva a llenar o alimentar, mientras que la gorutina primera no vacie el recipiente la gorutina segunda va a quedar en pausa esperando a que la gorutina siga con su trabajo, esto se le llama sincronizacion de gorutinas.

```
func main() {
        miCanal:=make(chan string)
        go enviarDatos(miCanal)
        go recibirDatos(miCanal)

        var entrada string
        fmt.Scanln(&input)
        fmt.Println("Terminado…")

func enviarDatos(miCanal  chan string) {
        for {
                c ← "Codigos"
        }
}

func recibirDatos(recibido chan string) {
                var contador int
                for {
                        contador++
                        fmt.Println(c ←, contador)
                        time.Sleep(time.Second * 5)
        }
}
```

Como se explico anteriormente creamos una variable llamada miCanal de tipo canal que movera datos de tipo string, y llamamos a dos gorutinas, las cuales se pasaran datos de tipo string por medio de un canal.

Los canales a la hora de pasarlos por parametros se pasan de la siguiente forma, aquí tenemos un ejemplo de dos variables:

func prueba(recibo chan string, envio chan string)

variable de tipo channel que recibe datos de tipo string

variable de tipo channel que envia datos de tipo string

Para cerrar un channel llamado miCanal se cierra de esta forma:

Close(miCanal) y si el canal es de recibir datos. Quien recibe los datos no lo puede cerrar, quien lo puede cerrar es quien lo tiene como channel de envio.

Canales unidireccionales

Para crear un canal unidirecciona que seria solo de recibir o solo de enviar lo indicamos asi:

func datos(salida chan← string) canal llamado salida asignado de solo salida de la funcion

func datos(entrada ←chan string) canal llamado entrada asignado de solo entrada de datos a la funcion

Buffer channel

los buffer channel son canales que tiene una pila o un buffer el cual puede guardar tantos bytes de datos:

miCanal:=make(chan string, 4)

creamos un canal de tipo string con capacidad de 4 bytes, si se llena el buffer del canal las gorutinas se bloquearan hasta que el channel de 4 bytes quede almenos un espacio vacio.

Select

el select es parecido al switch solo que en su evaluacion de case sera evaluar canales y se ejecutara la linea de codigo que corresponde.

```
Func main() {
        done:= time.After(10 * time.Second)
        eco:= make(chan []byte)
        go leerDatos(eco)
        for {
                select {
                        case datos:=←eco:
                                os.Stdout.Write(datos)
                        case ←done:
                                fmt.Println(“Tiempo terminado”)
```

os.Exit(0)

expliacion:

Creamos dos calanes, done y eco, el cual el canal done es un canal creado con time.After que nos da un canal el cual podemos ejecutar con inicio diferido, le pasamos un tiempo y ese canal se ejecutara o inicializara pasado ese tiempo.

Y tenemos otro canal llamado eco el cual almacenara los datos que entran en la variable datos, le pasamos a la funcion leerDatos() el canal eco.

```
Func leerDatos( salida chan← []byte) {
        datos:= make([]byte, 1024)
        n, _ :=os.Stdin.Read(datos)
        if n > 0 {
                out ← datos
        }
}
```

Aqui tenemos la funcion leerDatos() la cual en func main() le habiamos pasado un canal el cual se llamara salida y sera unidireccional de salida y de tipo slice de bytes, despues creamos una variable llamada datos la cual sera un slice de bytes de 1024 de espacio. Escribimos en la variable con os.Stdin.Read la cual nos devuelve dos valores, el numero de datos (n) y los datos, si n es mayor que 0 entonces enviar los datos al canal out.

En la funcion main tenemos recibiendo por el canal eco datos,asi que se ejecutara ese codigo, hasta pasado 20 segundos que se ejecutara el canal done el cual hara cambiar el select al case donde indica que se cierre el programa.sdf sadfsdfds

Despues de trabajar con los canales podremos cerrarlos con close(“canal”)

GO AVANZADO

Log/syslog

El paquete log nos permite escribir el Stdout y el Stderr en un archivo log, la ventaja de esto es que podemos tener un registro de los fallos que se han producido, tambien podemos posteriormente tratar esos archivos con herramientas de unix (awk, grep, sed). Veremos como configurar en que archivo de log se registrara

Seteando archivos log y enviando informacion a archivos log:

```
Func main() {
        miPrograma:=filepath.Base(os.Args[0])
        sysLog, err:=syslog.New(syslog.LOG_INFO | syslog.LOG_LOCAL7 , miPrograma)
```

}

En este codigo tenemos como argumento[0] el path del programa, despues creamos un syslog el cual le pasamos tres arguementos.

1º es la prioridad del mensaje, puede ser debug, info, notice, warning, err, critical, alert y emerg

2º establece el registro en el archivo registro local7 que es cisco.log, puede ser mail ftp u otro

3º es el nombre del proceso que en este caso sera el programa

```
if err != nil{
        log.Fatal(err)
}else{
        log.SetOutput(sysLog)
}
log.Println("LOG_INFO + LOG_LOCAL7: inicio sesion el Go")
```

en este codigo verificamos si la funcion syslog.New en la variable err es diferente a nil, si es diferente entonces imprimimos con log el error (err)

Si no hubo ningun error entonces llamamos a la funcion log.SetOutput() esta establece el destino de salida del registro, que sera sysLog el cual creamos anteriormente.

```
SysLog, err= syslog.New(syslog.LOG_MAIL "Mi Programa")
if err != nil {
        log.Fatal(err)
}else{
        log.SetOutput(SysLog)
}
log.Println("LOG_MAIL: inicio de sesion en mail en go")
fmt.Println("el usuario incio sesion")
```

En este codigo podemos ver como cambiamos syslog al log de mail, y manejamos los errores, si al cambiar a LOG MAIL existe un error entonces log.Fatal(err) que imprimira el error, y sino existe error entocnes seteamos el output del log en syslog he imprimimos en en el log ya que utilizamos log.Println y no fmt.

Los logs se pueden encontrar en el archivo de configuracion en etc/rsyslog.conf el cual en su parte izquierda tendira sus "accesos directos" y a la derecha su ruta, por ejemplo el cisco.log que indicamos local7:

```
local7  /var/log/cisco.log
```

Si el servidor no esta configurado para escribir ciertos registros entonces el codigo anterior nos podria dar un error sin que Go nos avisara

Log Fatal

La funcion log.fatal() la utlilizamos para manejar los errores. Este log fatal creara un registro en el log si es que ocurre un problema y cerrara el programa, esta funcion se utiliza en caso de errores fatales.

```
Func main() {
        sysLog, err:= syslog.New(syslog.LOG_ALERT | syslog.LOG_MAIL, “mi programa”)
        if err != nil {
                log.Fatal(err)
        }esle{
                logSetOutput(sysLog)
        }
        log.Fatal(sysLog)
        fmt.Println(“esta todo ok”)
}
```

En este codigo no llega a ejecutarse “esta todo ok” ya que antes invocamos a log.Fatal y el programa se cerrara, pero quedara un registro devido al sysLog cuando empieza main()

Log Panic

El log panic es parecido a log fatal solo que este nos devuelve informacion a bajo nivel de lo que ha pasado

Manejo de errores

En Go tenemos un tipo de dato llamado error para poder manejar los errores, asi que podemos crear facilmente nuestros errores.

```
func retornarError(a, b int) error{
        if a == b {
                err:= errors.New(“Error en la funcion, numeros iguales”)
                return err
        }else{
                return nil
```

```
}
func main() {
err:=retornarError(10, 10)
if err == nil {
        fmt.Println(“La funcion se ejecuto correctamente”)
}else{
        fmt.Println(err)
{
if err.Error() == “Error en la funcion, numeros iguales”{
        fmt.Println(“!!”
```

Creamos una funcion llamada retornarError que recibira dos parametros de tipo entero y retornara un dato de tipo error, despues hace una comparacion de esos dos datos, si son iguales crea una variable err que sera de tipo error y con New crea el error indicando lo que dira ese error, en caso contrario que los dos parametros NO sean iguales entonces retornara nil que seria null o nulo, nada.

Creamos la funcion main la cual creamos una variable err que tendra el resultado de la funcion con dos parametros dados, si la devolucion de err es nil, entonces imprime un mensaje, de lo contrario imprimira el contendio que devolvio la funcion retornarError(), tambien tenemos otro if el cual con err.Error() se comparara con un string gracias que con Error podemos convertir una variable de tipo error a string.

Go casi siempre devolvera dos tipos de datos en casi todas las funciones, uno de esos tipos de datos es un error, si existe tendra el error pero si no existe ese error la variable error que devuelve sera nil, asi que muchas veces veremos este codigo para comprobar si hubo error:

```
if err != nil {
        fmt.Print(err)
        os.Exit(10)
{
```

Esto compara la variable err que nos la devolvio una funcion, y si la variable err es diferente a nil quiere decir que hubo un error.

Podemos crear una variacion si es que el error ocurrido es muy importante:

```
if err != nil {
        panic(err)
        os.Exit(10)
}
```

El panic en este caso lo que hace es que detiene el programa y nos dara la maxima infomracion que pueda sobre el error que ha ocurrido.

Recoleccion de basura

La recoleccion de basura es el proceso de liberar espacio de memoria que no se esta utlilizando, este proceso ocurre de manera simultanea, es decir que mientras un programa de Go se esta ejecutando y tenemos un objeto inalcanzable o fuera del alcance, Go liberara ese espacio de memoria que ocupaba el objeto mientras el programa se sigue ejecutando.

Podemos ingresar el siguiente comando a la hora de ejecutar un programa para ver el funcionamiento de la basura de Go:

GODEBUG=gctracec=1 go run <programa.go>

Llamar a codigo C desde Go

G es muy parecido al codigo C, y muchas veces podremos vernos que tenemos un programa en C pero nosotros estamos programando en Go. Asi que Go puede hacer llamadas al codigo C sin problemas.

Codio C en el codigo Go como argumento

package main

//#include <stdio.h>

// void callC() {

//printf(“Esta es una funcion en C\n”);

//}

import “C”

El codigo C esta incluido en los comentarios (//), gracias al paquete C, Go interpretara esos comentarios como codigo C

la importacion del paquete C debe estar separada de las importaciones de los demas paquetes.

Import “fmt”

func main() {

fmt.Println(“Este es un codigo en Go”)

C.CallC()

fmt.Println(“esta es una funcion en Go”)

Codigo C en un archivo externo al codigo Go

tendremos dos archivos: callC.h y callC.c, el archivo callC.h contiene el siguiente codigo:

#ifndef CALLC_H

```
#define CALLC_H
void cHola();
void imprimirMensaje(char* mensaje);
#endif
```

el archivo fuente callC.c contiene el siguiente codigo:

```
#include <stdio.h>
#include “callC.h”
void cHola() {
        printf(“Hola desde codigo C\n”);
}
void imprimirMensaje(char* mensaje) {
        printf(“Va a enviarme %s \n”, mensaje);
}
```

estos dos archivos estan almacenados en el directorio ./exterior

codigo GO

```
package main
//#cgo CFLAGS: -I${SRCDIR}/exterior
//#cgo LDFLAGS: ${SRCDIR}/callC.a
//#include <stdlib.h>
//#include <callC.h>
import “C”
import (“fmt”
        “unsafe”)
func main() {
        fmt.Println(“Voy a llamar a una funcion en C”)
        C.cHola()
        fmt.Println(“Se llamara a otra funcion de C”)
miMensaje:=C.Cstring(“Este es el nuevo mensaje”)
defer C.free(unsafe.Pointer(miMensaje))
C.imprimirMensaje(miMensaje)
fmt.Println(“Todo ok”)
```

tenemos dos funciones en C, una llamada cHola() que imprime hola desde C y otra que se llama imprimirMensaje la cual imprime mensajes pasados desde el codigo GO, para hacer esto utlimo tenemos que utilizar la funcion C.CString() y pasarle el mensaje como argumento para que se pueda trasladar al codigo C, y tambien tenemos C.free con un defer el cual liberara espacio de la cadena que le pasamos a C.CString cuando ya no la usemos.

Compilamos el codigo C con:

gcc /*.c

ar rs callC.o

ar rs callc.a *.o

despues de esto tendremos un archivo llamado callC.a ubicado en el directorio que tenemos a nuestro programa en Go, haremos un go build <archivo.go> y listo.

Entorno en Go

Encontraremos informacion acerca del entorno de Go utilizando las funciones y el paquete runtime utilizandolo con el paquete fmt:

rutime.Compiler, runtime.GOARCH, runtime.NumCPU, runtime.Version y muchas mas cosas combinadas con el fmt podemos imprimirlas teniendo como resultado version de Go, numeros de CPU, gorutinas y mas.

Detector de Version

Utilizaremos runtime y strings para saber la version de Go, separar la cadena y comparar entre el minimo y el mayor para saber si estamos utilizando Go en 1.8 o superior.

```
Package main

import (“fmt”
        “runtime”
        “strings”
        “strconv”)


func main() {
        miVersion:= runtime.Version()
        mayor:=strings.Split(miVersion, “.”)[0][2]
        m1,_:=strconv.Atoi(string(mayor))
        menor:=strings.Split(miVersion, “.”)[1]
m2,_:=strconv.Atoi(string(menor))
```

Con strings.Split separaremos la cadena que nos devuelve la variable miVersion, la funcion usara como separador el punto. En la variable menor nos quedaremos con la pocision numero 1 contando desde el 0, con strconv convertimos ese string a entero.

Go1.15.7 quedaria separado por los puntos y go1, pocision 0, 15 posicion 1 y 7 posicion 2.

Bucle for

En Go tenemos el bucle for el cual nos sirve para iterar sobre algun dato:

```
for i:=0; i <100;i++{
        fmt.Print(“ok”)
}
```

En este codigo tenemos una variable i en un bucle for, la variable tiene como valor inicial 0 y mientras que i sea menor que 100, i le sumaremos 1, entre las llaves tenemos un codigo el cual imprime ok, despues el bucle vuelve a arriba y empieza otra vez, esta vez con i valiendo 1, y asi sucesivamente hasta que i valga 99, ya que cuando llegue a 100, el bucle parara.

Podemos parar el bucle for sin que llegue a 100 con la palabra break la cual frenara el codigo y saldra del bucle, y con la palabra continue podemos omitir el codigo y volveremos al principio del bucle para seguir iterando.

For en lugar de while

Go no contiene la palabra clave while el cual en otros lenguajes es un bucle, podemos realizar un bucle do while utilizando un bucle for:

```
for miBucle:=true;miBucle;miBucle=true {
        fmt.Println(“bucle infinito”)
}
```

o

```
condicional:=10
for {
        if condicional < 0 {
                break
        }
        fmt.Print(condicional, “”)
        condicional - -
```

o

```
x := 0
```

```
condicional:=true

for miBucle:=verdadero;miBucle;miBucle=condicional {
        if x > 10 {
                condicional = false
        }
        fmt.Println(x)
        x++
}
```

En este codigo tenemos una variable llamada miBucle de tipo buleana verdadera, mientras que la variable sea verdadera el bucle se ejecutara infinitamente.

Range con for

Como vimos antes el range nos sirve para indexar datos, podemos utilizarlo con el bucle for, aquí un ejemplo:

```
miArray := [5] int {1,2,3,4,5}

for indice, valor:= range(miArray){
        fmt.Printf(“indice %d valor %d”. indice, valor)
```

range devuelvo dos valores,el cual el primero sera indice y el segundo sera el valor.

Fecha y Hora

veremos como configurar fecha y hora en diferentes formatos imprimibles, es bueno saber esto ya que podemos sicronizar varias tareas, tenemos que importar el paquete time.

```
func main() {
        fmt.Println(time.Now())
        t:=time.Now()
        fmt.Println(t,t.Format(time.RFC3339)
        fmt.Println(t.Weekday(),t.Day(),t.Month(),t.Year()
        time.Sleep(10 * time.Second)
        t1:= time.Now()
        fmt.Println(“diferencia horaria: “, t1.sub(t))
```

En este codigo tenemos time.New() el cual nos dara la fecha y hora, tambien tenemos una variable t con el tiempo almacenado. Tenemos t.Format(time.RFC3339) el cual nos imprime el tiempo en un cierto formato y tambien tenemos t.Weekday() t.Day() t.Month() y t.Year() en cual nos imprime dia numero mes y año, con t1.sub(t) podemos ver la diferencia entre t1 y t.

Analisis del tiempo

Para analizar tiempos tenemos una funcion en time la cual es time.Parse(), esta funcion recibe dos parametros, el primero es el formato el cual estara la fecha dada y el segundo parametro es la fecha dada, tenemos una lista de constantes que se pueden utilizar en Go ya que Go no maneja el tipico formato DDYYYYMM o %D %Y %M como lo suelen hacer otros tipos de lenguaje, asi que aquí esta la tabla la cual contiene un lista con los diferentes formatos disponibles en Go: https://golang.org/src/time/format.go

```
package main

import (“fmt”
        “os”
        “path/filepath”
        “time”
)

func main() {
        var miTiempo string
        if len(os.Args) !=2 {
                fmt.Printf(“uso: %s cadena \n”, filepath.Base(os.Args[0]))
                os.Exit(1)
        }
        miTiempo=os.Args[1]
        d, err:=time.Parse(“15:04”, miTiempo)
        if err == nil{
                fmt.Println(“Correcto”, d)
                fmt.Println(“Hora: “,d.Hour(), d.minute())
        }else{
                fmt.Println(err)
        }
```

En este codigo podemos ver la forma que nuestro programa aceptara la fecha, que seria formato hora “15:04” , tambien podemos cambiarlo por 02 january 2006 el cual acepatara ese formato de fecha como entrada.

Expreciones regulares

Las expreciones regulares permite por medio de una exprecion buscar coincidencias en un lista, el paquete encargado en Go de esto se llama regexp

Crearemos un codigo el cual leera un archivo de registro de un servidor web apache linea por linea, despues hara coincidir con expreciones regulares las fechas del archivo.

Los paquetes que utilizaremos seran:

```
package main
import (“fmt”
        “bufio”
        “io”
        “os”
        “regexp”
        “strings”
        “time”
)
```

la siguiente parte del codigo:

```
func main() {
        argumentos:=os.Args
        if len(argumentos) == 1 {
                fmt.Println(“indique un archivo de texto”)
                os.Exit(1)
        }
        nombreArchivo:=argumentos[1]
        archivo, err:=os.Open(nombreArchivo)
        if err !=nil {
                fmt.Println(“Error al abrir el archivo”,err)
                os.Exit(1)
        }
        defer archivo.Close()
        marca:=0
        lectura:=bufio.NewReader(archivo)
        for {
                linea, err:=lectura.ReadString(‘\n’)
                if err == io.EOF{
```

```
            break
        }else if err !=nil {
            fmt.Println("error al leer el archivo")
        }
```

Hasta aquí lo que hemos echo es ver si la longitud del argumento es igual a 1, si es ese caso le diremos que tiene que dar un argumento ya que la longitud de argumento si es 1 quiere decir que solo tiene el nombre del programa.

Tenemos un defer que se ejecutara al final del codigo cerrando el archvo.

Creamos una marca la cual la usaremos mas adelante en las expreciones regulares.

Leemos el archivo con bufio.NewReader(archivo) y lo almacenamos en la variable lectura.

Haremos un bucle for el cual recorrera con ReadString hasta un salto de linea (\n) y esa linea la guardara en la variable linea.

Siguiente codigo:

```
r1:=regexp.MustCompile
(`. * \ [(\ d \ d \ / \ w + / \ d \ d \ d \ d: \ d \ d: \ d \ d: \ d \ d. *) \]. *`)
if r1.MatchString(linea) {
        coincidencias:=r1.FindStringSubmatch(linea)
        d1,err:=time.Parse("02/Jan/2006:15:04:05-0700", coincidencias[1])
        }else{
            marca ++
        }
continue
}
```

En este pedazo de codigo se puede ver que si el formato de fecha no coincide el programa continuara, si entra en el bloque if omitira el codigo restante.

Siguiente parte del codigo:

```
r2: = regexp.MustCompile (`. * \ [(\ w + \ - \ d \ d- \ d \ d: \ d \ d: \ d \ d: \ d \ d. *) \]. *`)
if r2.MatchString (línea) {
        coincidencias: = r2.FindStringSubmatch (línea)
        d1, err: = time.Parse ("Jan-02-06: 15: 04: 05 -0700", coincidencias[1])
        if err == nil {
            newFormat: = d1.Format (time.Stamp)
            fmt.Print (strings.Replace (linea, match [1], newFormat, 1))
        demás {
```

```
                marca ++
        }
        continue
}
fmt.Println(marca.”las lineas no coinciden”)
```

Inicializamos una variable llamada r1 la cual tendra regex.MustCompile y un string el cual sera la exprecion regular,

despues tenemos un if con r1.MatchString(linea) el cual significa que si ha encontrado coincidencia de la exprecion regular r1 en la variable linea, y agregamos a la variable coincidencia, la coincidencia gracias a FindSubmatch(linea).

El siguiente codigo que contiene una variable llamada r2 tambien tenemos lo mismo pero con diferente formato la exprecion regular.

Coincidencias de direcciones IPv4

las direcciones IPv4 tienen cuatro partes numericas separadas por un punto, los numeros pueden ir de 00000000 en fomrato binario que seria equivalente a 0, hasta 255 que en binario seria 11111111.

Estos seran los paquetes a importar: bufio, io, os, regexp, fmt, net y path/filepath

la segunda parte sera:

```
func buscarIP(entrada string) string {
        parteIP:=”(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])”
        gramatica:=parteIP + “\\.”parteIP + “\\.”parteIP + “\\.”parteIP
        coincidencia:=regexp.MustCompile(gramatica)
        return coincidencia.FindString(entrada)
```

En esta parte del codigo primero creamos una exprecion regular numerica, despues construimos la IP ya que constara de cuatro partes de lo que viene siendo parteIP.

Creamos la variable de coincidencia (regexp.MustCopile) con los valores de gramatica.

Devolvemos el resutlado de la que coinciden entre la variable coincidencia que tiene como valor gramatica y la variable entrada.

Explicacion de parteIP:

las ip pueden empezar con 250, hasta 255

o puede comenzar con 2, seguido de 0, 1, 2, 3, 4 y terminar con 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Alternativamente puede empezar con 1 seguido de los digitos 0-9, 0-9 ejemplo: 1[0-9][0-9]

la ultima combinacion seria, un primer digito que puede ser opcional del 0 al 9 y el segundo que es obligatorio del 0-9

la utlima parte del codigo:

```
func main() {
	argumentos:=os.Args
	if len(argumentos)<2 {
		fmt.Println(“uso: “, filepath.Base(argumentos[0]))
		os.Exit(1)
	}
	for _, archivo:= range argumentos[1:] {
		f, err:= os.Open(archivo)
		if err != nil{
			fmt.Println(“no se puede abrir el archivo”)
			os.Exit(1)
		}
		defer f.Close()
		r:=bufio.NewReader(f)
		for{
			linea, err:=r.ReadString(“\n”)
			if err == io.EOF {
				break
			}else if err != nil {
				(“error al leer el archivo”)
				break
			}
			ip:= buscarIp(linea)
			prueba:= net.ParseIP(ip)
			if prueba.To4() == nil{
				continue
			}else{
				fmt.Print(ip)
```

En este codigo leemos los argumentos como hemos echo anteriormente, despues abrimos el archivo pasado como argumento y leemos linea por linea guardandola en la variable “linea” esa variable linea la pasamos a la funcion buscarIp(linea) la cual el resultado se guardara el la variable ip, la

funcion net.Parse comprueba (parsea) que estemos trabajando con datos en formato ip, si esto es nil continua y pasara a imprimir el resutlado de la funcion buscarIP alojada en la variable (ip).

Este codigo puede leer mas de un archivo a la vez, podemos mezclarlo con comando de unix el cual tenemos sort para ordenar, uniq para que no se imprima los repetidos y hasta controlar la salida con > guardandola en un archivo.

Strings

Strings es un paquete de Go que es muy potente y tiene varias funciones interesantes.

```
Import (“fmt”
        s “strings”
        “unicode”
)
```

Podemos ver aquí que a la hora de importar un paquete de Go, tiene la (s) esto significa que el paquete strings ahora podermos llamarlo con s, si strings tiene una funcion que se llamaria strings.FunctionName() ahora la funcion la invicariamos asi: s.FunctionName() ya que el alias de strings ahora es (s).

Tambien podemos asignar una funcion a una variable para poder llamar a la funcion llamando a la variable:

```
var f := fmt.Println
f(“Hola”)
```

en este caso llamamos a f ya que ahora la variable f tiene dentro la funcion fmt.Println().

Paquete Syscall

El paquete syscall es un paquete que trabaja a bajo nivel y se comunica con el sistema operativo, varios paquetes de Go lo utilizan como puede ser fmt, net y otros mas.

Imprimir el ID del proceso y el ID del usuario

```
pid,_,_,_:=syscall.Syscall(39,0,0,0)
```

con el paquete fmt podemos imprimir el pid que seria el ID del proceso

```
uid,_,_,_:=syscall.Syscall(24,0,0,0)   (en linux esto no funciona)
```

con fmt print uid imprimiremos el ID del usuario.

Imprimir por pantalla

```
mensaje:=[]byte{‘h’,’o’,’1’,’a’,’!’,’\n’}
pantalla:=1
```

syscall.Write(pantall,mensaje)

Ejecutar comandos

Comandos:=”/bin/ls”

env:=os.Environ()

syscall.Exec(comando,[]string{“ls”,”-la”},env)

En este caso no tenemos el control de la salida del comando, solo se imprimira por pantalla

En la variable comandos sera el archivo binario que ejecutemos (ls) y en string, seria el comando a ejecutar ya que ls tiene varios comandos como ls -la

Plantillas y HTML

Las plantillas se utilizan para separar lo que es la estructura y el formato de un documento de lo que viene siendo los datos que estan en el, una plantilla puede ser una cadena o un archivo, si la plantilla es pequeña se recomienda que la plantillas sea de string y si es una plantilla ya mas grande que se encuentre en un archvio aparte externo.

No se puede importar el paquete text/template y html/template ya que pertenecen al mismo paquete, si el caso es que se tiene que importar si o si, deberiamos agregarle un alias a uno de ellos para que no haya problemas.

Si utilizamos text/templates la salida se veria como un texto y si utiliamos html/templates la salida seria leida por un navegador web pudiendo tener un codigo HTML.

text/template

importaremos el paquete text/template

Crearemos una estructura llamada Entrada:

type Entrada struct{

    Nombre int

    Cuadrado int

}

creamos una variable que contendra la plantilla:

plantillaArchivo:=os.Args[1]

Creamos una variable de tipo slice multidimencional la cual tendra los datos que se imprimira el la plantilla:

DATOS:=[][]int{{-1,1},{-2,4},{-3,9},{4,16}}

Creamos una variable llamada entradas que sera de tipo Entrada (estructura creada)

Haremos un bucle con range el cual recorrera los DATOS leyendo solo el dato y ignorando el indice:

```
for _,i:= range DATOS {
        if len(i) == 2{
                temp:=Entrada{Numero.i[0],Cuadrado:i[1]}
                Entradas=append(Entradas,temp)
```

como vemos si la longitud de i que son los datos de DATOS({-1,1},{-2,4}) es igual a dos entonces inicializamos una variable llamada temp la cual sera una estructura Entrada y tendra Numero i[0] y Cuadrado i[1]

Despues agregamos ese valor de variable a la estructura Entradas.

```
 t:=template.Must(template.ParseGlob(plantillaArchivo))
t.Execute(os.Stdout, Entradas)
```

Por ultimo tenemos una variable inicializada que se llama t la cual tiene como funcion template.Must que devuelve un tipo de dato de plantilla, a la vez template.ParseGlob(plantillaArchivo) lee ese archivo pasandolo a template.Must que devolvera los datos en tipo plantilla.

Explicacion plantilla:

Calculando los cuadrados de algunos numeros enteros!!!

{{ range. }} El cuadrado de {{printf “%d” .Numero }} es {{ printf “%d” .Cuadrado }}

{{ end }}

La palabra clave range nos perimte iterar sobre las lineas de entrada, que se da como una porcion de estructura (Numero y Cuadrado) Las variables y texto dinamico iran entre {{ }} y usaremos el printf para darle formato a la salida, en este caso un entero (%d) al final del todo tenemos {{ end }} el cual indicara al programa que hasta ahi es la plantilla.

html/template

La diferencia entre text/teplate y html/template es su presentacion ya que html/template devolvera la informacion el formato HTML que es segura contra inyeccion de codigo.

Interfaces con switch

Ya hemos visto como funcionan y que son las interfaces, ahora procederemos a utilizarlas con el switch de esta manera, tendremos tres estructuras que seran cuadrado circulo y triangulo:

```
type cuadrado struct {
        X float64
```

```
}
type circulo struct {
	R float64
}
type rectangulo struct {
	X float64
	Y float64
}
```

Estas estructuras son de tipo de dato estructura, despues tendremos un switch el cual dependiendo del tipo de dato, si es de tipo estructura cuadrada, estructura rectangular o estructura circulo ejecutara un bloque u otro, para saber el tipo utilizaremos la funcion .(type)

```
func llamada(x interface{}) {
	switch v := x.(type) {
	case cuadrado:
		fmt.Println("Es un cuadrado")
	case circulo:
		fmt.Printf("%v es un cuadrado\n", v)
	case rectangulo:
		fmt.Println("Es un rectangulo")
	default:
		fmt.Printf("Tipo desconocido %T\n", v)
	}
}
```

Una vez teniendo las estructuras y el switch contruiremos las estructuras dandole un valor a las mismas y llamando a la funcion llamada que tiene el switch dentro.

```
func main() {
	x := cuadrado{X: 10}
	llamada(x)
	y := circulo{R: 15}
	llamada(y)
	z := rectangulo{X: 20, Y: 20}
```

```
	llamada(z)
	llamada(10)
}
```

Go en Unix

En unix nos encontramos con tres categorias de procesos, proceso de usuario, proceso demonio y proceso de kernel.

Los procesos de usuario se ejecutan en el espacio de usuario y generalmente no tienen derecho de acceso especiales, los procesos del kernel se ejecutan solo en el espacio del kernel y pueden llegar a tener acceso por completo a todos los datos. Los procesos demonio son programas que se pueden encontrar en el espacio de usuario y ejecutar en el fondo sin necesidad de terminal.

Paquete flag / comando terminal

El paquete flag vendra muy bien si estamos desarollando utilidades de linea de comando, tiene varias caracteristicas.

Pondremos de ejemplo el siguiente codigo que utilizara el paquete flag:

```
import (“flag”
	“fmt”
)


func main() {
	comando1:=flag.Bool(“k”,true,”k”)
	comando2:=flag.int(“i”,0,”i”)
	flag.Parse()
```

En este segmento de codigo creamos dos variables comando1 y comando2, recibiran tres parametros, el primer seria como se invocara a ese parametro por la terminal, el segundo define el tipo de dato que es y su valor inicial, y el tercero es el que se usara para mostrar mas informacion de uso del programa.

```
valor1:=*comando1
valor2:=*comando2
fmt.Println(“-comando 1:”,valor1)
fmt.Println(“-comando 2:”,valor2)
```

En este ultimo codigo guardamos los valores recibidos en valor1 y valor2 y despues lo imprimiremos por pantalla.

De esta forma podremos invocar el programa y pasarle sus respectivos parametros de esta forma:

go run <programa.go> -k=true -i=10

go run <programa.go> -k=false -i=8

El flag.Parse() debera incertarse siempre despues de invocar un flag.

Paquete bufio / lecutra de archivos

Antes de leer o escribir archivos debemos tener en cuelta que existe la salido o entrada con o sin buffer. Podremos abrir un achivo y leerlo a medida que se procesan los bytes o abrir el archivo cargar los bytes en un buffer y leerlos de ahi o dejar que otra persona los lea.

A la hora de elegir que metodo utilizar, si se tratan de archivos que estan en constante actualizacion la entrada y salida SIN buffer es la mejor opcion.

Gracias al paquete bufio podremos leer archivos desde Go. Podremos leerlos de tres formas, palabra por palabra, linea por linea o carácter por carácter.

Con este codigo podremos leer un archivo de texto linea por linea:

importamos los paquetes: fmt, flag, bufio, os, io

```
func lecturaLineas( archivo string)error {
	var err error
	f,err:=os.Open(archivo)
	if err != nil {
		fmt.Println(“Error al abrir el archivo”,archivo)
		return
		defer f.Close()
}
	r:=bufio.NewReader(f)
	for {
		linea,err:=r.ReadString(‘\n’)
		if err ==io.EOF {
			break
	}else if err != nil{
			fmt.Println(“Error al leer el archivo”)
			break
		}
	fmt.Print(linea)
	}
```

return nil

}

En este segmento de codigo es donde manejaremos la apertura y lectura del archivo, creamos la funcion llamada lecturaLinea la cual recibira un parametro que se guardara en la variable archivo y sera de tipo string, y devolvera un error.

Dentro de la funcion creamos una variable llamada err de tipo error, abriremos con os.Open el archivo y controlaremos los errores. Coloraremos un defer de Close para cerrar el archivo al finalizar la funcion, una vez abierto el archivo lo pasaremos a bufio.NewReader el cual leera el archvio, con ReadString le diremos que lea hasta (\n) que seria hasta que haga un salto de linea, otra vez controlamos los errores y si va todo bien imprimiremos la linea.

```
func main() {
	flag.Parse()
	if len(flag.Args()) == 0  {
		fmt.Prinf(“uso: <programa.go> archivo.txt”)
	}
	for _, archivo:=range flag.Args() {
		err:=lecturaLineas(archivo)
		if err!= nil{
			fmt.Println(err)
			}
		}
}
```

Este ultimo codigo tendra un flag.Parse para podre usar el paquete flag, el paquete flag tiene flag.Args el cual se utiliza como os.Args.

Controlaremos que nos indique un archivo, creamos un bucle for con un range el cual descartaremos el indice con _ y recorreremos el flag.Args. Los datos que nos devuelva range que sera el nombre de los archivos se pasara a la variable archivo y crearemos una variable err, la cual tendra el resultado de la devolucion de la funcion lecturaLineas(archvio) con el archivo enviado, si da error lo imprimimos de lo contrario la funcion imprimira la linea.

Archivos CSV

Los archivos CSV son archivos sin formatos que contienen caracteres separados por comas.

En este codigo abriremos un archivo y sus datos los pasaremos a una grafica utilizando Glot, importaremos cinco paquetes el cual un paquete sera externo y tendremos que descargarlo con:

go get github.com/Arafatk/glot

despues procederemos a importar los paquetes: fmt, encoding/csv, github.com/Arafatk/glot, os, strconv

```
package main
func main() {
	if len(os.Args) !=2{
		fmt.Println(“Necesitas un archivo de datos”)
		return
	}
	archivo:=os.Args[1]
	_,err:=os.Stat(archivo)
	if err != nil{
		fmt.Println(“No se puede establecer”, archivo)
		return
```

Esta parte del codigo donde tenemos os.Stat() verificamos que el archivo si el archivo existe o no.

```
f,err=os.Open(archivo)
if err!=nil {
	fmt.Println(“No se puede abrir”, archivo)
	fmt.Println(err)
	return
}
defer f.Close()
lectura:=csv.NewReader(f)
lectura.FieldsPerRecord=-1
allRecords,err:=lectura.ReadAll()
if err != nil {
	fmt.Println(err)
	return
}
pA:=[]float64 {}
pB:=[]float64 {}
```

```
for _,rec:=range allRecords {
        x,_:strconv.ParseFñpat(rec[0],64)
        y,_:strconv.ParseFloat(rec[1],64)
        pA=append(pA, x)
        pB=append(pB,y)
}
puntos:=[] []float64{}
puntos=append(puntos, pA)
puntos=append(puntos, pB)
fmt.Println(puntos)

dimensiones:=2
persistir:=verdadero
depirar:=falso
plot,_:=glot.NewPlot(dimensiones,persistencia,depurar)
plot.setTitle(“Uso de Glot con CSV”)
plot.SetXlabel(“eje X”)
plot.SetYlabel(“eje Y”)
estilo:=”circulo”
plot.AddPointGroup(“Circulo:”,estilo,puntos)
plot.SavePlot(“salida.png”)
}
```

Tendremos otro archivo el cual contendra estos datos para poder leerlos con el codigo anterior escrito.

1,2
2,3
3,3
4,4
5,8
6,5

Escribir en un archivo

Veremos varias manera de escribir en un fichero utilizanso Go:

importaremos bufio,fmt,io,io/ioutil y os

Forma 1

```
func main() {
        s:=[] byte(“Datos para escribir\n”)
        fichero,err:=os.Create(“fichero.txt”)
        if err!= nil{
                fmt.Println(“Error al crear el fichero”, fichero)
                return
        }
        defer fichero.Close()
        fmt.Fprintf(fichero, string(s))
```

En este segmento de codigo lo que tenemos es una variable de tipo slice llamada s que tiene los datos que escribiremos en el fichero, despues creamos un fichero con os.Create y manejamos los errore si llegara a tener algun error, si todo esta bien con fmt.Fprintf() escribiremos en el fichero la cadena (string) de la variable s.

Forma 2

```
        fichero2,err:=os.Create(“fichero2.txt”)
        if err !=nil{
                fmt.Println(“No se puede crear el archivo”,err)
                return
        }
        defer fichero2.Close()
        n, err:=fichero2.WriteString(string(s))
        fmt.Printf(“escribio %d bytes \n”,n)
```

Forma 3

En este segmento es otra forma de escribir en un fichero.

```
fichero3,err:= os.Create(“fichero3.txt”)
if err!=nil {
```

```
        fmt.Println("error al crear archivo", fichero)
        return
}
w:=bufio.NewWriter(fichero3)
n,err:=w.WriteString(string(s))
w.Flush()
```

En este caso utilizamos bufio.NewWrite para abrir el fichero antes creado, y bufio.WriteString para escribir en el.

Forma 4

```
fichero4:="fichero4.txt"
err=ioutil.WriteFile(fichero4,s,0644)
if err != nil {
        fmt.Println(err)
        return
}
```

Este codigo necesita solo una llamada a ioutil.WeiteFile el cual crea un fichero con nombre del valor de la variable fichero4, escribe sus datos en el, los datos los extrae del valor de la variable s y le da permisos 0644.

Forma 5

```
fichero5,err:=os.Create("fichero5.txt")
if err != nil {
        fmt.Println(err)
        return
}
n,err:=io.WriteString(fichero5,string(s))
if err != nil {
        fmt.Println(err)
        return
{
```

Por utlimo este codigo utiliza io.WriteString para escribir el valor de la variable s en el fichero5.

Tuberias

La mejor forma para comunicar la salida de un programa con otro, es decir el resultado de un programa que se envie a otro programa y asi tener “dos programas” en uno es mediante tuberias, las tuberias son conectores que conectan el STDOUT del programa uno con el STDIN del programa dos y asi puede conectar mas de dos de tres y mas de cuatro programas o utilidades a la vez.

Las tuberias tienen limitaciones como por ejemplo son unidireccional, van de izquierda a derecha.

Cat

En esta parte veremos la utilidad cat en accion creada con codigo Go, importamos los paquetes fmt, bufio, io, os.

```
func imprimir(archivo string) error {
	a, err:=os.Open(archivo)
	if err != nil {
		return err
	}
	defer a.Close()
	escanear:= bufio.NewScanner(a)
	for escaner.Scan() {
		io.WriteString(os.Stdout, escanear.Text())
		io.WriteString(os.Stdout, “\n”)
	}
	return nil
}
```

En esta seccion de codigo recibimos un archivo que se guardara en la variable archivo, la funcion imprimir devolvera un error, abrimos el archivo y lo almacenamos en la variable a, tratamos su error (err) si es que da error retornamos ese error, ponemos un defer para cerrar el archivo al terminar la funcion. Escaneamos el archivo abierto para leerlo y hacemos un for del archivo cargado listo para leer y lo imprimimos en os.Stdout el archivo en formato texto, despues escribiremos un salto de linea, y al final retornamos un nil

```
func main() {
	nombreArchivo:=""
	argumentos:=os.Args
	if len(argumentos)== 1 {
		io.Copy(os.Stdin, os.Stdout)
		return
	}
	for i; 1 ; i <len(argumentos);i++{
		nombreArchivo=argumentos[i]
		err:=imprimir(nombreArchivo)
		if err != nil {
			fmt.Println(err)
		}
	}
}
```

Este codigo que sera el primero que se ejecutara ya que tiene la funcion main, creara una variable vacia llamada nombreArchivo y otra llamada argumentos con los argumentos del programa. Manejaremos los argumentos, si la longitud del argumento es 1, es decir solo tendra el nombre del programa, lo inicializamos y copiara todo lo que le escibamos (os.Stdin) a la salida (os.Stdout) gracias a la funcion io.Copy()

sin embargo si el programa detecta que el argumento no es == 1 entonces tenemos un bucle for el cual iterara con una i inicializada en 1, la variable nombreArchivo[i] por tanto iterara desde el argumento 1 para arriba, dejando el argumento 0 fuera ya que el 0 es el primero y es nuestro programa. Despues tendremos una variable err que sera el resultado de la funcion imprimir enviandole el nombreArchivo iterado, si nos devuelve un error lo imprimimos, pero si no devuelve nada no haremos nada ya que la funcion imprimir, imprimira por pantalla el archivo que le hemos pasado.

FINAL.