**GO un lenguaje de programación**

El Go Programming Language, también conocido como Golang, es un lenguaje de programación de propósito general, desarrollado por un equipo de Google. El lenguaje Go fue concebido en septiembre de 2007 por Robert Griesemer, Rob Pike y Ken Thompson en Google. Go apareció por primera vez en noviembre de 2009, y la primera versión del idioma se publicó en diciembre de 2012. El sitio web oficial del proyecto Go es http://golang.org/. Go tiene su propia elegancia y lenguajes de programación que hacen que el lenguaje sea productivo y divertido de codificar. Go también proporciona una biblioteca estándar completa. La biblioteca estándar proporciona todos los paquetes principales que necesitan los programadores para construir programas basados en red y en la vida real. Go es un lenguaje de programación concurrente, compilado de forma nativa y recolectado de forma estática que pertenece principalmente a la familia de lenguajes C en términos de sintaxis básica.

Go Programming Language es un proyecto de código abierto que se distribuye bajo una licencia de estilo BSD para que los programadores sean más productivos. Go es un lenguaje de programación expresivo, conciso, limpio y eficiente. Go compila rápidamente al código de máquina, pero tiene la comodidad de la recolección de basura y el poder de la reflexión en tiempo de ejecución. Es un lenguaje más rápido, estático y compilado que se siente como un lenguaje interpretado de forma dinámica.

Go es un lenguaje compilado, y como muchos lenguajes, hace un uso intensivo de la línea de comandos. Go es tanto el nombre del lenguaje de programación como el nombre del conjunto de herramientas utilizado para crear e interactuar con los programas Go.

Go es un lenguaje de programación estático. Lo que significa es que el compilador siempre quiere saber cuál es el tipo para cada valor en el programa. Cuando el compilador conoce la información de tipo de antemano, puede ayudar a asegurarse de que el programa esté trabajando con valores de forma segura. Esto ayuda a reducir posibles errores de memoria y errores, y le brinda al compilador la oportunidad de producir un código más eficiente. Go struct le permite crear sus propios tipos combinando uno o más tipos, incluidos los tipos integrados y definidos por el usuario. Las estructuras son la única forma de crear tipos concretos definidos por el usuario en Go. Cuando crea sus propios tipos utilizando struct, es importante comprender que Go no proporciona soporte para la herencia en su sistema de tipos, pero favorece la composición de tipos que le permite crear tipos más grandes combinando tipos más pequeños. La filosofía de diseño de Go es crear componentes más grandes combinando componentes más pequeños y modulares. Si usted es un programador pragmático, apreciará la filosofía de diseño de Go que favorece la composición sobre la herencia debido a sus beneficios prácticos. La herencia de tipos a veces presenta desafíos prácticos con respecto a la mantenibilidad.

En la última década, el hardware de la computadora ha evolucionado hasta tener muchos núcleos de CPU y más potencia. Hoy en día, aprovechamos en gran medida las plataformas en la nube para crear y ejecutar aplicaciones donde los servidores en la nube tienen más poder. Si bien las computadoras modernas y las instancias de máquinas virtuales en la nube tienen más potencia y muchos núcleos de CPU, aún no podemos aprovechar la potencia de las computadoras modernas que utilizan la mayoría de los lenguajes de programación y herramientas existentes. La concurrencia en Go es la capacidad para que las funciones se ejecuten de forma independiente. Sus mecanismos de concurrencia facilitan la escritura de programas que aprovechan al máximo las máquinas multinúcleo y en red, mientras que su novedoso sistema permite la construcción de programas flexibles y modulares. Cuando una función se crea como una goroutina, se trata como una unidad de trabajo independiente que se programa y luego se ejecuta en un procesador lógico disponible. Los goroutines se crean llamando a la instrucción Go seguida de la función o el método que desea ejecutar como una actividad autónoma. El programador de tiempo de ejecución de Go es un software sofisticado que administra todos los goroutines que se crean y necesitan tiempo de procesador. El programador se encuentra en la parte superior del sistema operativo, vinculando los hilos del sistema operativo a los procesadores lógicos que, a su vez, ejecutan goroutines. El programador controla todo lo relacionado con qué goroutines se ejecutan en qué procesadores lógicos en un momento dado.

**Tipo de datos**

Go es un lenguaje de programación estático. Esto significa que las variables siempre tienen un tipo específico y ese tipo no puede cambiar. La palabra clave **var** se utiliza para declarar variables de un tipo de datos en particular. Aquí está la sintaxis para declarar variables:

**var** nombre **tipo** = valor

A la izquierda usamos la palabra clave **var** para declarar una variable y luego asignarle un valor. Podemos declarar variables múltiples del mismo tipo en una sola declaración como se muestra aquí:

**var** nombre, apellido **string**(cadenas de caracteres)

Múltiples variables del mismo tipo también se pueden declarar en una sola línea: **var** x, y **int** hace a ambas variable x e y **int**(enteras). También puede hacer uso de la asignación(inicialización) paralela: a, b := 20, 16. Si está usando una expresión de inicialización para declarar variables, puede omitir el tipo usando la declaración de variable corta como se muestra aquí:

país, estado := "Alemania", "Berlín"

Usamos el operador := para declarar e inicializar variables con una breve declaración de variable. Cuando declara variables con este método, no puede especificar el tipo porque el tipo está determinado por los valores utilizados.

Ejemplo1

package main

import

"fmt"

var (m int // Declaración de variable global

n int)

func main () {

var x int = 1 // Tipo de datos entero inicializado en 1

var y int // Tipo de datos entero

y=2

fmt.Println (x)

fmt.Println (y)

var a, b, c = 5.25,25.25,14.15 // Declaración de variable float32 múltiple

fmt.Println (a, b, c)

ciudad := "Berlin" // Declaración de variable de cadena

Pais := "Alemania" // Los nombres de las variables distinguen entre mayúsculas y minúsculas

fmt.Println (ciudad)

fmt.Println (Pais) // Los nombres de las variables distinguen entre mayúsculas y minúsculas

comida, bebida, precio := "Pizza", "Pepsi", 125 // Tipo múltiple de declaración e inicialización de variable en la misma línea

fmt.Println (comida, bebida, precio)

m, n = 1,2

fmt.Println (m, n)

}

Salida:

1

2

5.25 25.25 14.15

Berlin

Alemania

Pizza Pepsi 125

1 2

**Integer Float String Boolean**

Go proporciona aritmética de enteros con y sin signo. Hay cuatro tamaños distintos de enteros con signo: 8, 16, 32 y 64 Bits, representados por los tipos int8, int16, int32 e int64 y las versiones sin signo correspondientes uint8, uint16, uint32 y uint64.

Ejemplo4

package main

import "fmt"

func main () {

var n1 uint8 // Enteros de 8 bits positivos (0 a 255)

n1 = 200

fmt.Println (n1)

var n2 uint16 // Enteros de 16 bits positivos (0 a 65535)

n2 = 54200

fmt.Println (n2)

var n3 uint32 // Enteros de 32 bits positivos (0 a 4294967295)

n3 = 98765214

fmt.Println (n3)

var n4 uint64 // Enteros de 64 bits positivos (0 a 18446744073709551615)

n4 = 1844674073709551615

fmt.Println (n4)

var n5 int8 // Enteros de 8 bits (-128 a 127)

n5 = -52

fmt.Println (n5)

fmt.Println (n5 \* -1)

var n6 int16 // Enteros de 16 bits (-32768 a 32767)

n6 = -32552

fmt.Println (n6)

fmt.Println (n6 \* -1)

var n7 int32 // Enteros de 32 bits (-2147483648 a 2147483647)

n7 = -98658754

fmt.Println (n7)

fmt.Println (n7 \* -1)

var n8 int64 // Enteros de 64 bits (-9223372036854775808 a 9223372036854775807)

n8 = -92211111111111111

fmt.Println (n8)

fmt.Println (n8 \* -1)

}

Salida

200

54200

98765214

1844674073709551615

-52

52

-32552

32552

-98658754

98658754

-92211111111111111

92211111111111111

**Variables**

Nombrar una variable correctamente es una parte importante del desarrollo de software. Los nombres deben comenzar con una letra y pueden contener letras, números o el símbolo de subrayado (\_). Al compilador Go no le importa lo que nombre de una variable, pero debes elegir nombres que describan claramente el propósito de la variable.

package main

import "fmt"

func main () {

var x int

x = 10

fmt.Println (x)

var y int = 20

fmt.Println (y)

z := 30

fmt.Println (z)

var Rapidez, distanCia, \_tiempo, CalCulo = 10,20,30,40 // Convenciones de nombres de variables

fmt.Println (Rapidez, distanCia, \_tiempo, CalCulo)

}

En el momento de la declaración de la variable podemos omitir el tipo o el valor para la inicialización, pero debemos especificar al menos uno: el tipo o el valor. Si se omite el tipo de variable en la declaración, se determina a partir de valor utilizado para la inicialización. Si se omite el valor, el sistema asigna que el valor inicial es 0 para los tipos de variables numéricas, falso para el tipo de variable booleana y "" para el tipo de variable de cadena. Dentro de una función, una forma alternativa, llamada declaración de variable corta, puede ser usada para declarar e inicializar variables locales. Esta toma la forma: variables := valores y el tipo de las variables está determinado por el tipo de valores asignados respectivamente.

package main

import "fmt"

func main () {

nombre, apellido, edad, Casado: = "John", "Doe", 25, false

fmt.Println (nombre, apellido, edad, Casado)

}

Las variables declaradas fuera de la función principal(func main) son accesibles desde otras funciones.

Package main

import "fmt"

var x int = 10 //declaración fuera de la función principal

func main () {

fmt.Println (x)

prueba()

}

func test () {

fmt.Println (x)

}

Go también tiene otra forma corta cuando necesitas definir múltiples variables.

Package main

import "fmt"

var (

a = 5

b = 10

c = 15

)

func main () {

fmt.Println (a, b, c)

}

**Ejemplos de uso de las variables**

Ejemplo 1) Declarar un entero y una variable de cadena

La asignación de un valor en línea con la declaración de la variable.

package main

import (

"fmt"

)

func main() {

var i int = 10

var s string = "Japon"

fmt.Println(i)

fmt.Println(s)

}

Salida:

10

Japom

Una variable en Go es inicializada por la palabra clave var.

En el ejemplo anterior, las variables i y s,se le asignan el tipo entero y cadena respectivamente.

El operador de asignación = significa que a la variable a su izquierda se le asignar un valor que está a la derecha de =.

El paquete estándar fmt de biblioteca usa el nombre de las variables i y s como referencia de los valores de i y s.

Ejemplo 2) Asignación después de la declaración

Primero se declara una variable y luego se le asigna un valor más adelante en el programa.

package main

import (

"fmt"

)

func main() {

var intVar int

var strVar string

intVar = 10

strVar = "Australia"

fmt.Println(intVar)

fmt.Println(strVar)

}

Salida:

10

Australia

**Constantes**

Go también tiene soporte para constantes. Las constantes son esencialmente variables cuyos valores no pueden cambiarse más adelante. Se crean de la misma manera que crean variables, pero en lugar de usar la palabra clave **var** usamos la palabra clave **const**:

const x string = "Hello, World"

Una declaración **const** define valores con nombre que se parecen sintácticamente a las variables pero cuyo valor es constante, lo que evita cambios accidentales (o infames) durante la ejecución del programa.

package main

import "fmt"

const (

x=10

y=20

z=30

)

func main(){

const nombre string ="Simón Bolívar" // Constante con tipo de dato

fmt.Println(nombre)

const edad = 35 // Constant sin tipo de dato

fmt.Println(edad)

fmt.Println(x,y,z)

}

Salida:

Simón Bolívar

35

10 20 30

El tipo subyacente de cada constante es un tipo básico: booleano, cadena o número.

Operators

Los operadores combinan operandos en expresiones. Go Language es rico en operadores integrados y proporciona los siguientes tipos de operadores:

**Operadores aritméticos**

Los operadores aritméticos se aplican a valores numéricos y producen un resultado del mismo tipo que el primer operando. Los cuatro operadores aritméticos estándar (+, -, \*, /) se aplican a tipos enteros, de punto flotante y complejos; + también se aplica a las cadenas.

package main

import "fmt"

func main () {

var m, n int = 100,25

var a, b string = "Pedro", "Picapiedra"

var suma, diferencia, producto, cociente, resto int

var sumac string

sumac = a + "es" + b // concatena los contenidos de a,”es”,y b

sum = m + n // + suma enteros, flotantes, valores complejos, cadenas

diferencia = m-n // - diferencia de enteros, flotantes, valores complejos

producto = m \* n // \* producto de enteros, flotantes, valores complejos

cociente = m/n // / cociente de enteros, flotantes, valores complejos

resto = m%n // % resto de una división de enteros

fmt.Println (sumc)

fmt.Println (sum)

fmt.Println (diferencia)

fmt.Println (producto)

fmt.Println (cociente)

fmt.Println (resto)

}

Salida

Pedro es Picapiedra

125

75

2500

25

0

**Operadores de Bitwise**

Los operadores lógicos y de desplazamiento a nivel de bits solo se aplican a los enteros. El operador bitwise trabaja en bits y realiza operaciones bit a bit.

Package main

import "fmt"

func main () {

var x, y, z int

x = 75

y = 25

z = x & y // & bitwise AND enteros

fmt.Println (z)

z = x | y // | bitwise OR enteros

fmt.Println (z)

z = x ^ y // ^ bitwise XOR enteros

fmt.Println (z)

z = x &^ y // &^ bit clear (AND NOT) enteros

fmt.Println (z)

}

**Operadores de comparación**

Los operadores de comparación como == y < también se pueden usar para comparar un valor de una variable de un tipo declarado con otro del mismo tipo.

package main

import "fmt"

func main() {

var num1 int = 50

var num2 int = 60

fmt.Println(num1==num2) // == equal

fmt.Println(num1!=num2) // != not equal

fmt.Println(num1<num2) // < less

fmt.Println(num1>num2) // > greater

fmt.Println(num1>=num2) // > greater or equal

fmt.Println(num1<=num2) // <= less or equal

}

Salida:

false

true

true

false

false

true

Operadores lógicos

Los operadores lógicos operan sobres valores booleanos(operandos) y producen un resultado del mismo tipo que los operandos.

package main

import "fmt"

func main () {

var num1 int = 50

var num2 int = 60

if (num1 != num2 && num1 <= num2) {// && Operador lógico llamado AND(y).

fmt.Println ("True")

}

if (num1 != num2 || num1 <= num2) {// || operador lógico llamado OR(o)

fmt.Println ("True")

}

if (!(num1 == num2)) { // ! Llamado operador lógico NO. Se utiliza para invertir el estado lógico de su operando.

fmt.Println ("True")

}

}

Salida:

True

True

True

**Operadores de Asignación**

Declaración de asignación que concatena el antiguo valor de X con Y y lo asigna de nuevo a X.

X = X + Y

X += Y

El operador += es un operador de asignación. Cada operador aritmético y lógico como + o \* tiene un operador de asignación correspondiente.

package main

import "fmt"

func main () {

var X int = 50

var Y int = 60

X += Y // += suma X e Y y asigna resultado a X

fmt.Println (X)

X = 50

Y = 60

X -= Y // -= Resta Y de X y asigna resultado a X

fmt.Println (X)

X = 50

Y = 60

X \*= Y // \*= Multiplicar X por Y asigna resultado a X

fmt.Println (X)

X = 4

Y = 44

X %= Y // %= Divide X entre Y y asigna resto a X

fmt.Println (X)

X = 50

Y = 200

X /= Y // /= Dividir X entre Y y asigna resultado a X

fmt.Println (X)

}

Salida:

110

-10

3000

4

0

**Funciones**

Cada línea de código que creamos para realizar una tarea específica, y combinamos estas líneas de código para llevar a cabo el resultado deseado. A veces deseamos repetir las líneas de código con datos diferentes, y en algunas acciones nuestro código se vuelve tan largo que es difícil hacer un seguimiento de lo que hace cada parte. Aquí utilizamos, Funciones que sirven como herramientas del sistema que mantienen su código impecable y ordenado.

Una función es un grupo de declaraciones que existen dentro de un programa con el propósito de ejecutar una tarea específica. En Golang, una función se puede llamar varias veces dentro del paquete, y también desde otros paquetes si las funciones se exportan a otros paquetes.

**Declarando funciones**

Aquí está la sintaxis para escribir funciones en Go:

La primera línea de una función proporciona información sobre lo que la función aceptará como entrada y qué esperar como salida, y esto lo llamamos como la firma de la función.

func [Nombre\_función] (tipo param1, tipo param2 ...) (tipo devuelto1, tipo devuelto2 ...) {

// cuerpo de la función

}

La palabra clave **func** significa que este es el punto de inicio de una función. Luego viene el nombre de la función. Luego hay entre paréntesis la declaración de el conjunto de las variables que almacenan los valores esperados (una lista de parámetros) para esta función. Después viene entre paréntesis la declaración de una lista opcional de variable que devuelven los resultados. El llave de apertura significa el inicio del cuerpo de la función, donde escribimos la lógica de la función seguido de la llave de cierre.

**Función con valor de retorno.**

En este ejemplo, la función add () toma la entrada de dos números enteros y devuelve un valor entero con un nombre de total.

package main

import "fmt"

func main() {

fmt.Println(suma(20, 30))

}

func suma(x int, y int) int {

total := 0

total = x + y

return total

}

Salida:

50

Los tipos de entrada y valor de retorno deben coincidir con la firma de la función. Si modificamos el programa anterior y pasamos algún valor de cadena en el argumento, el programa lanzará una excepción "no se puede usar" prueba "(tipo de cadena) como tipo int en el argumento para agregar".

Función sin valor de retorno

package main

import "fmt"

func main() {

add(20, 30)

}

func add(x int, y int) {

total := 0

total = x + y

fmt.Println(total)

}

Salida:

50

Los tipos de devolución pueden tener nombres

También podemos definir las variables que tendrán los valores de retorno. Aquí se define la variable total de tipo entero para el valor que devuelve la función.

package main

import "fmt"

func main() {

fmt.Println(add(20, 30))

}

func add(x int, y int) (total int) {

total = x + y

return total

}

**Llamadas de funciones diferidas**

Go tiene una declaración especial llamada aplazamiento que programa una llamada a la función para que se ejecute después de que la función se complete. Considere el siguiente ejemplo:

package main

import "fmt"

func first() {

fmt.Println("Primero")

}

func second() {

fmt.Println("Segundo")

}

func main() {

defer second()

first()

}

Salida:

Primero

Segundo

Este programa imprime primero seguido de segundo

Una declaración diferida se usa a menudo con operaciones emparejadas como abrir y cerrar, conectar y desconectar, o bloquear y desbloquear para asegurar que los recursos se liberen en todos los casos, sin importar cuán complejo sea el flujo de control. El lugar correcto para una declaración diferida que libera un recurso es inmediatamente después de que el recurso se haya utilizado con éxito.

A continuación se muestra el ejemplo para abrir un archivo y realizar una acción de lectura / escritura en él. En este ejemplo, a menudo hay lugares donde desea regresar temprano.

**Sin diferir**

func ReadWrite() bool {

file.Open("file")

if failureX {

file.Close() //And here...

return false

}

if failureY {

file.Close() //And here...

return false

}

file.Close() //And here...

return true

}

Una gran cantidad de código se repite aquí. Para superar esto Go tiene la declaración de aplazamiento. El código anterior se puede reescribir de la siguiente manera. Esto hace que la función sea más legible, más corta y coloca el cierre justo al lado del Abrir.

**Con diferir**

func ReadWrite() bool {

file.Open("file")

defer file.Close() //file.Close() is agregado a la lista defer

// Do your thing

if failureX {

return false // Close() es ahora ejecutado automáticamente

}

if failureY {

return false // y aquí también

}

return true // y aqui

}

Esto tiene varias ventajas:

- Mantiene nuestra llamada Close cerca de nuestra llamada Open para que sea más fácil de entender.

- Si nuestra función tenía varias declaraciones de devolución (tal vez una en un if y una en otro), Close ocurrirá antes de las dos.

- Las funciones diferidas se ejecutan incluso si se produce un pánico en el tiempo de ejecución.

- Las funciones diferidas se ejecutan en orden LIFO, por lo que se imprime el código anterior: 4 3 2 1 0.

- Puedes poner múltiples funciones en la "lista diferida", como este ejemplo.

package main

import "fmt"

func main() {

for i := 0; i < 5; i++ {

defer fmt.Printf("%d ", i)

}

}

**Funciones variables**

La función Variadic es una función en la que podemos pasar un número infinito de argumentos a una función en lugar de solo un argumento a la vez.

Para declarar una función variable, el tipo del parámetro final está precedido por una elipsis, "...", que muestra que la función puede ser llamada con cualquier número de argumentos de este tipo.

Seleccione un solo argumento de todos los argumentos de la función variadic.

En el siguiente ejemplo, vamos a imprimir s [0] la primera y s [3] el cuarto, valor como argumento pasado a la función variadicExample ().

package main

import "fmt"

func main() {

variadicExample("rojo", "azul", "verde", "amarillo")

}

func variadicExample(s ...string) {

fmt.Println(s[0])

fmt.Println(s[3])

}

Salida:

rojo

amarillo

Debe ser preciso al ejecutar una llamada de función vacía, si el código dentro de la función que espera un argumento y la ausencia de un argumento generará un error "pánico: error en tiempo de ejecución: índice fuera de rango". En el ejemplo anterior tienes que pasar al menos 4 argumentos.

**Pasando múltiples argumentos de cadena a una función variadic**

El parámetro **s** acepta un número infinito de argumentos. La elipsis con puntos de árbol le dice al compilador que esta cadena aceptará, de cero a múltiples valores.

package main

import "fmt"

func main() {

variadicExample()

variadicExample("rojo", "azul")

variadicExample("rojo", "azul", "verde")

variadicExample("rojo", "azul", "verde", "amarillo")

}

func variadicExample(s ...string) {

fmt.Println(s)

}

Salida:

[]

[red blue]

[red blue green]

[red blue green yellow]

En el ejemplo anterior, hemos llamado a la función con argumentos simples y múltiples; Y sin pasar ningún argumento.

**Normal funcionamiento de las funciones variables**

En el siguiente ejemplo, la firma de función acepta un número arbitrario de argumentos de tipo slice.

package main

import (

"fmt"

"reflect"

)

func main() {

variadicExample(1, "red", true, 10.5, []string{"foo", "bar", "baz"},

map[string]int{"apple": 23, "tomato": 13})

}

func variadicExample(i ...interface{}) {

for \_, v := range i {

fmt.Println(v, "--", reflect.ValueOf(v).Kind())

}

}

Salida:

1 -- int

red -- string

true -- bool

10.5 -- float64

[foo bar baz] -- slice

map[tomato:13 apple:23] – map

**Pánico y Recuperación**

El sistema de tipo incorporado de GO Language detecta muchos errores en el momento de la compilación, pero no puede verificar errores como una matriz fuera de límites, acceso o deferencia de puntero nulo que requieren comprobaciones en el tiempo de ejecución. GO no tiene un mecanismo de excepción, no puedes lanzar excepciones. Durante la ejecución, cuando Go detecta estos errores, genera un error y detiene toda la ejecución normal, se ejecutan todas las llamadas de función diferidas en ese goroutine y finalmente el programa se bloquea con un mensaje de registro. Este mensaje de registro generalmente tiene suficiente información para analizar la causa raíz del problema sin ejecutar el programa repetidamente, por lo que siempre debe incluirse en un informe de error sobre un programa en pánico.

Panic es una función incorporada que detiene el flujo ordinario de control y comienza a entrar en pánico. Cuando la función X llama al pánico, la ejecución de X se detiene, cualquier función diferida en X se ejecuta normalmente, y luego X regresa a su interlocutor. Para la persona que llama, X se comporta como una llamada al pánico. El proceso continúa en la pila hasta que todas las funciones en el goroutine actual han regresado, momento en el que el programa falla. Los pánicos se pueden iniciar invocando el pánico directamente. También pueden ser causados por errores en tiempo de ejecución, como los accesos a matrices fuera de los límites.

No todos los pánicos vienen del tiempo de ejecución. La función de pánico incorporada puede ser llamada directamente; Acepta cualquier valor como argumento. Un pánico suele ser lo mejor que se puede hacer cuando ocurre una situación "imposible"; por ejemplo, la ejecución llega a un caso que lógicamente no puede ocurrir:

package main

import "fmt"

func main() {

var action int

fmt.Println("Ingrese 1 para Estudiante y 2 para Profesional")

fmt.Scanln(&action)

/\* Uso de Switch Case en Golang \*/

switch action {

case 1:

fmt.Printf("Yo soy una estudiante")

case 2:

fmt.Printf("Yo soy un Profesional")

default:

panic(fmt.Sprintf("Yo soy un %d",action))

}

fmt.Println("")

fmt.Println("Ingrese 1 para Tigre and 2 para Leon")

fmt.Scanln(&action)

/\* Use del Switch Case en Golang \*/

switch action {

case 1:

fmt.Printf("Yo soy un Tigre")

case 2:

fmt.Printf("Yo soy un Leon")

default:

panic(fmt.Sprintf("yo soy un %d",action))

}

}

En el programa anterior, el programa detendrá la ejecución si el usuario ingresa cualquier otro valor distinto a 1 o 2.

Recuperar es una función incorporada que recupera el control de un goroutine en pánico. Recuperar solo es útil dentro de funciones diferidas. Durante la ejecución normal, una llamada para recuperar devolverá cero y no tendrá ningún otro efecto. Si el goroutine actual está en pánico, una llamada para recuperar capturará el valor dado al pánico y reanudará la ejecución normal.

package main

import "fmt"

func pami() {

action := recover()

fmt.Println(action)

}

func main() {

var action int

fmt.Println("Ingrese 1 for Estudiante y 2 for Profesional")

fmt.Scanln(&action)

/\* Use del Switch Case en Golang \*/

switch action {

case 1:

fmt.Printf("Yo soy Estudiante")

case 2:

fmt.Printf("Yo soy Profesional")

default:

defer pami()

panic(fmt.Sprintf("Yo soy %d",action))

}

}

**Arreglos**

Un arreglo es una estructura de datos que consiste en una colección de variables de un mismo tipo y bajo un mismo nombre y que se diferencian entre ellas a través de una variable\_índice que las identifican. Un arreglo en Go es un tipo de datos de longitud fija que contiene un bloque contiguo de elementos del mismo tipo. Estos podrían ser de tipos como enteros y cadenas.

**Declaración e Inicialización**

Hay cuatro maneras comunes de declarar arreglos.

1) **Una declaración básica de un arreglo es la siguiente**:

**var** nombre[length] element\_type

La definición de tipo de un arreglo se compone de el nombre común de las variables que lo componen, longitud o número de variables, incluido entre corchetes, seguida del tipo de sus elementos almacenados.

Por ejemplo:

var Precios [5] int

El nombre de este arreglo es Precios, de 5 elementos y tipo int. En este ejemplo, el arreglo se inicializa o llena automáticamente por compilador, que asigna a cada una de las 5 elementos(variables) o posiciones de índice, con el valor predeterminado 0.

var carros[10] string

El nombre de este arreglo de cadenas es cadena con una longitud de 10 elementos con valores predeterminados “”.

Cuando declaramos una matriz, se pueden reasignar los datos que queramos tener, siempre que estos datos sean del tipo declarado. Por ejemplo:

Precios [0] = 100

Precios [1] = 110

Precios [2] = 120

Precios [3] = 130

Precios [4] = 140

Observe que los valores entre corchetes varían entre 0 y 4, cinco elementos.

Los valores enteros entre corchetes determinan a que elemento del arreglo se le asigna el valor declarado a la derecha del signo =. El elemento de índice [0] toma el valor de 100, primer elemento del arreglo Precios. Al igual que otros lenguajes de programación, la indexación en arreglos comienza con cero.

2) **Declarar un arreglo con algunos o todos los valores en el momento de la declaración**

var nombre = tipo{lista de valores de elementos separados por comas}

Los valores para un arreglo se componen de la definición del tipo seguida de un conjunto de valores separados por comas, entre paréntesis. Por ejemplo:

var Precios = [5] int {100,110}

En el ejemplo, el elemento [0] contiene 100, el elemento [1] contiene 110 y los elementos restantes consisten 0.

También puede proporcionar valores para elementos específicos como se muestra aquí:

var Precios = [5] int {0:100,2:120,4:140}

Los elementos de índice 0,2 y 4 les fueron asignados los valores 100,120 y 140 respectivamente; los valores restantes serán 0.

3) **Declare un arreglo dentro de la función usando la declaración de variable corta**

Precios := [5] int{100,110,120,130,140}

4) **Declarando un arreglo con puntos suspensivos …**

La longitud de una matriz se puede omitir y reemplazar por puntos suspensivos durante la inicialización. La capacidad se determina en función del número de valores inicializados.

Precios := [...] int {100,110,120,130,140}

La capacidad se determina en función del número de valores inicializados.

**Ejercicio**

1) Uso del **for** para iterar sobre un arreglo.

Cree una arreglo de enteros llamada Precios, con una longitud de 5. Use un bucle for para iterar sobre la matriz, muestre el contenido de la matriz.

package main

import "fmt"

func main() {

fmt.Println("Incrementos de los precios del artículo xxx en la semana:\n")

Precios := [5]int{100,110,120,130,140}

for i := 0; i < len(Precios); i++ { // la función len arroja el tamaño o capacidad del arreglo

fmt.Println("BS",Precios[i])

}

}

Salida:

Incrementos de los precios del artículo xxx en la semana:

BS 100

BS 110

BS 120

BS 130

BS 140

2) **Asignando un arreglo por valor y referencia**

Los valores "Perú", "Argentina", "Chile" se asignan a Paises1. Paises1 se asigna a Paises2, lo que significa que se crea una copia de esos valores. La reasignación de Paises1 a Paises3 a través de una asignación por referencia.

package main

import "fmt"

func main () {

Paises1 := [3] string {"Perú", "Argentina", "Chile"}

fmt.Printf ("Paises1: %v \n", Paises1)

Paises2 := Paises1 // los datos se pasan por valor (copiados)

fmt.Printf ("Paises2: %v \n", Paises2)

Paises1 [0] = "Brasil"

fmt.Printf ("Paises1: %v \n", Paises1)

fmt.Printf ("Paises2: %v \n", Paises2)

Paises3 := &Paises1

fmt.Printf ("Paises3: %v \n",Paises3)

fmt.Printf ("&Paises3: %v \n",&Paises3)

fmt.Printf ("\*Paises3: %v \n",\*Paises3)

}

strArray3 es un arreglo de cadena de punteros, lo que significa que Paises3 es un puntero a la dirección de memoria de Paises1. Para eliminar la referencia a strArray3, debemos agregar el operador \* como prefijo \*strArray3.

3) **Un ejemplo de programa que explora el tipo de arregloo**

package main

import "fmt"

func main () {

var x [5] int // Declaración del arreglo

x [0] = 10 // Asignar los valores a un índice específico

x [4] = 20 // Asignar valor a al arreglo en cualquier orden

x [1] = 30

x [3] = 40

x [2] = 50

fmt.Println ("Valores del arreglo X:", x)

// Declaración de un arreglo e inicialización con índices específicos

y := [5] int {0: 100,1: 200,3: 500}

fmt.Println ("Valores del arreglo Y:", y)

// Declaración de Arreglo e Inicialización

País := [5] string {"Venezuela", "Nicaragua", "Panama", "México", "Brasil"}

fmt.Println ("Países de América:", País)

// Declaración de arreglo sin longitud e inicialización.

Transporte := [...] string {"Tren", "Autobús", "Avión", "Coche", "Bicicleta"}

fmt.Println ("Tipos de transporte:", Transporte)

}

Slices(Segmentos de arreglos)

Un **Slice** es una estructura de datos flexible y extensible para implementar y administrar colecciones de datos. Un **slice** es un segmento dinámico de un arreglos que puede crecer y reducirse según lo crea conveniente. Al igual que los arreglos, los **slices** son indexables y tienen una longitud. A diferencia de los arreglos, son flexibles en términos de longitud porque tienen su propia función incorporada llamada **append**, que incrementa su longitud con eficiencia. También puede recortar el tamaño de un slice cortando una parte de la memoria oculta. **Slices** nos ayuda en las optimizaciones de indexación, iteración y recolección de basura porque la memoria oculta se asigna en bloques contiguos.

**Creando slice**. Hay tres formas comunes de declarar slice:

**1)** **Una declaración básica de un slice es usando la función make** ()

var nombre = make ([] tipo, longitud, capacidad)

Un slice puede inicializarse en tiempo de ejecución utilizando la función interna make

Ejemplo de un programa que declara un slice en golang:

package main

import "fmt"

func main () {

var Slice1 = make ([] int, 10) // cuando la longitud y la capacidad son iguales

var Slice2 = make ([] int, 10,20) // cuando la longitud y la capacidad son diferentes. Aquí la longitud puede crecer hasta 20(su capacidad)

fmt.Printf ("Slice1 \tLen:%v \tCap:%v \n", len(Slice1), cap(Slice1))

fmt.Printf ("Slice1 \tLen:%v \tCap:%v \n", len(Slice2), cap(Slice2))

}

Salida:

Slice1 Len:10 Cap:10

Slice1 Len:10 Cap:20

Crea un arreglo de tipo int, con longitud y capacidad de 10. El tercer parámetro que especifica la capacidad máxima del arreglo es opcional.

**2)** **Declaración usando la palabra *new***

package main

import "fmt"

func main() {

var Rosas = new([50]int)[0:10]

fmt.Printf("Rosas \tLen: %v \tCap: %v\n", len(Rosas), cap(Rosas)) // \t se usa para tabular la salida de títulos

//len y cap funciones internas proporcionan la longitud y capacidad del slice Rosas

}

Salida:

Rosas Len: 10 Cap: 50

En la declaración anterior el slice Rosas comienza con una longitud de 10, pasando de 0 a 50. La capacidad es de 50, por lo que puede expandirse hasta 50 sin solicitar un nuevo arreglo.

**3) Declaración literal**

var nombre = [] slice\_tipo {lista de valores de elementos separados por comas}

El siguiente fragmento de código ilustra las variables de slice inicializadas con valores literales compuestos:

package main

import "fmt"

func main () {

var Precios = [] int {10,20,30,40}

var Paises = [] string {"India", "Rusia", "China"}

fmt.Printf ("Precios \tLen:%v \tCap:% v\n", len(Precios), cap(Precios))

fmt.Println (Precios)

fmt.Printf ("Paises \tLen:%v \tCap:%v \n", len(Paises), cap(Paises))

fmt.Println (Paises)

}

Salida:

Precios Len:4 Cap: 4

[10 20 30 40]

Paises Len:3 Cap:3

[India Rusia China]

La longitud y capacidad son de 3 y 4 elementos respectivamente.

El número de elementos proporcionados en el slice no está rodeado por un tamaño fijo. Esto sugiere que el literal puede ser tan grande como sea necesario.

**Ejercicio de slice**

**1) Ampliar una slice usando la función de *append***

***append*** agrega elementos al final de un slice. Si hay suficiente capacidad en el arreglo subyacente, el elemento se coloca después del último elemento y se incrementa la longitud. Sin embargo, si no hay suficiente capacidad, se crea un nuevo arreglo, todos los elementos existentes se copian, el nuevo elemento se agrega al final y se devuelve el nuevo slice.

El siguiente fragmento de código amplía una slice utilizando la función ***append***:

package main

import "fmt"

func main () {

// Crear un slice pequeño

A := make ([] int, 2, 5)

A [0] = 10

A [1] = 20

fmt.Println ("Slice A:", A)

fmt.Printf ("La longitud es %d La capacidad es %d \n", len(A), cap(A))

// Crear un slice más grande

A = append (A, 30, 40, 50, 60, 70, 80, 90)

fmt.Println ("Slice A después de agregar datos:", A)

fmt.Printf ("Su longitud es %d La capacidad es %d \n", len(A), cap(A))

}

Salida:

Slice A: [10 20]

La longitud es 2 La capacidad es 5

Slice A después de agregar datos: [10 20 30 40 50 60 70 80 90]

Su longitud es 9 La capacidad es 10

Utilizando make(), se creó un slice de tipo int, se le asígnó el nombre A, de longitud 2 y capacidad de 5 elementos. Se utilizó el operador := en lugar de var para crear el slice dentro de la función principal. 7 elementos agregados al slice usando la función append que aumentó la longitud a 9 y la capacidad a 12.

**2) Ampliar un slice utilizando la función *copy***

La función *copy* integrada en Go se utiliza para copiar datos de un slice a otro. La copia toma dos argumentos: dst y src. Todas las entradas en src se copian en dst sobrescribiendo lo que sea que esté allí. Si las longitudes de los dos slice no son las mismas, se utilizará la menor de las dos.

El siguiente fragmento de código amplía un slice utilizando la función *copy*:

package main

import "fmt"

func main () {

// Crear una porción más pequeña

A := [] int {5, 6, 7}

fmt.Println ("[slice: A]",A," La longitud es ",len(A)," La capacidad es ",cap(A)," \n")

// Crear un slice más grande

B := make ([] int, 5, 10)

fmt.Println ("[Slice: B]",B," La longitud es ",len(B)," La capacidad es ",cap (B)," \n")

copy (B,A) // la función copy

fmt.Println ("slice B después de copiar:", B)

B [3] = 8

B [4] = 9

fmt.Println ("slice B después de agregar elementos:", B)

}

Salida:

[slice: A] [5 6 7] La longitud es 3 La capacidad es 3

[Slice: B] [0 0 0 0 0] La longitud es 5 La capacidad es 10

slice B después de copiar: [5 6 7 0 0]

slice B después de agregar elementos: [5 6 7 8 9]

Usando make(), se creó un slice de tipo [] int llamado B, de longitud de 5 y una capacidad de 10 elementos. Con la función copy se copiaron todos los elementos de A en B. Se agregaron en B dos nuevos elementos 8 y 9 en las posiciones 3,4.

**3) trucos de slice**

El siguiente fragmento de código ilustra varios trucos de slice.:

package main

import "fmt"

func main() {

var Paises = []string{"India", "Brasil", "canada", "Italia", "Iran"}

fmt.Printf("Paises: %v\n", Paises)

fmt.Printf(":2 %v\n", Paises[:2])

fmt.Printf("1:3 %v\n", Paises[1:3])

fmt.Printf("2: %v\n", Paises[2:])

fmt.Printf("2:5 %v\n", Paises[2:5])

fmt.Printf("0:3 %v\n", Paises[0:3])

fmt.Printf("Last element: %v\n", Paises[4])

fmt.Printf("Last element: %v\n", Paises[len(Paises)-1])

fmt.Printf("Last element: %v\n", Paises[4:])

fmt.Printf("All elements: %v\n", Paises[0:len(Paises)])

fmt.Printf("Last two elements: %v\n", Paises[3:len(Paises)])

fmt.Printf("Last two elements: %v\n", Paises[len(Paises)-2:len(Paises)])

fmt.Println(Paises[:])

fmt.Println(Paises[0:])

fmt.Println(Paises[0:len(Paises)])

}

Salida:

Paises: [India Brasil canada Italia Iran]

:2 [India Brasil]

1:3 [Brasil canada]

2: [canada Italia Iran]

2:5 [canada Italia Iran]

0:3 [India Brasil canada]

Last element: Iran

Last element: Iran

Last element: [Iran]

All elements: [India Brasil canada Italia Iran]

Last two elements: [Italia Iran]

Last two elements: [Italia Iran]

[India Brasil canada Italia Iran]

[India Brasil canada Italia Iran]

[India Brasil canada Italia Iran]

Declara un slice con 5 elementos. Usando corchetes podemos crear diferentes slice de un slice. Función de len utilizada para calcular la longitud del slice.

4) Asignar partes de un slice a otro slice

Esta técnica ilustra el uso de rangos para seleccionar elementos de un slice.

package main

import "fmt"

func main() {

var original = []string{"Argentina", "Chile", "Uruguay", "Paraguay", "Bolivia"}

var Temporal []string

fmt.Printf("original: %v\n", original)

fmt.Println(" se crea Gnuevo una copia de original con los tres primeros elementos. Modificaciones en Gnuevo afectaran original")

Gnuevo := original[0:3]

fmt.Println(" se crea Temporal agregandole el primero del original")

Temporal = append(Temporal, original[:1]...) // :1 los dos puntos delante del número indican que se anexa solo el elemento 1

fmt.Printf("Gnuevo: %v\n", Gnuevo)

fmt.Printf("Temporal: %v\n", Temporal)

fmt.Printf("original longitud: %v\tcapacidad: %v\n", len(original), cap(original))

fmt.Printf("Gnuevo longitud: %v\tcapacidad: %v\n", len(Gnuevo), cap(Gnuevo))

fmt.Printf("Temporal longitud: %v\tcapacidad: %v\n", len(Temporal), cap(Temporal))

fmt.Println(" Peru se le asigna al primer elemento de Gnuevo")

Gnuevo[0] = "Peru"

fmt.Printf("Gnuevo: %v\n", Gnuevo)

fmt.Printf("original: %v\n", original)

fmt.Println(" se agrega Ecuador a Gnuevo")

Gnuevo = append(Gnuevo, "Ecuador")

fmt.Printf("Gnuevo: %v\n", Gnuevo)

fmt.Printf("original: %v\n", original)

fmt.Println(" se agrega Brasil a original")

original = append(original, "Brasil")

fmt.Printf("original: %v\n", original)

fmt.Println(" se agrega Venezuela a Gnuevo")

Gnuevo = append(Gnuevo, "Venezuela")

fmt.Printf("Gnuevo: %v\n", Gnuevo)

fmt.Printf("original: %v\n", original)

}

Salida:

original: [Argentina Chile Uruguay Paraguay Bolivia]

se crea Gnuevo una copia de original con los tres primeros elementos. Modificaciones en Gnuevo afectaran original

se crea Temporal agregandole el primero del original

Gnuevo: [Argentina Chile Uruguay]

Temporal: [Argentina]

original longitud: 5 capacidad: 5

Gnuevo longitud: 3 capacidad: 5

Temporal longitud: 1 capacidad: 1

Peru se le asigna al primer elemento de Gnuevo

Gnuevo: [Peru Chile Uruguay]

original: [Peru Chile Uruguay Paraguay Bolivia]

se agrega Ecuador a Gnuevo

Gnuevo: [Peru Chile Uruguay Ecuador]

original: [Peru Chile Uruguay Ecuador Bolivia]

se agrega Brasil a original

original: [Peru Chile Uruguay Ecuador Bolivia Brasil]

se agrega Venezuela a Gnuevo

Gnuevo: [Peru Chile Uruguay Ecuador Venezuela]

original: [Peru Chile Uruguay Ecuador Bolivia Brasil]

5) Añadir un slice a un slice existente

Esta técnica ilustra el uso de puntos triples ... puntos suspensivos para anexar un sector.

package main

import "fmt"

func main() {

var slice1 = []string{"india", "japan", "canada"}

var slice2 = []string{"australia", "russia"}

slice2 = append(slice2, slice1...)

fmt.Printf("slice1: %v\n", slice1)

fmt.Printf("slice2: %v\n", slice2)

}

Salida

slice1: [india japan canada]

slice2: [australia russia india japan canada]

6) Adjuntar parte de un slice a un slice existente

Esta técnica ilustra el uso de puntos suspensivos con rango para agregar un slice.

Package main

package main

import "fmt"

func main() {

var slice1 = []string{"india", "japan", "canada", "us", "uk", "italy", "germany"}

var slice2 = []string{"australia", "russia"}

slice2 = append(slice2, slice1[3:]...)

fmt.Printf("slice1: %v\n", slice1)

fmt.Printf("slice2: %v\n", slice2)

}

Salida:

slice1: [india japan canada us uk italy germany]

slice2: [australia russia us uk italy germany]