**Tutoriales de Golang**

[Naveen Ramanathan](https://golangbot.com/about/)

Traducción apoyada con el traductor [www.DeepL.com/Translator](http://www.DeepL.com/Translator)

Adaptación Rudy Gómez

**Introducción**

1 - Introducción e instalación

2 - Hola Mundo

**Variables, tipos y constantes**

3 - Variables

4 - Tipos

5 - Constantes

**Funciones y Paquetes**

6 - Funciones

7 - Paquetes

**Declaraciones y ciclos condicionales**

8 - Declaración if else

9 - Lazos (for)

10 - Declaración Switch

**Arreglos, slice y funciones variadic**

11 - Arreglos y slice

12 - Funciones variadic

**Más tipos**

13 - Mapas

14 - Cadenas(Strings)

**Punteros, Estructuras y Métodos**

15 - Punteros

16 - Estructuras

17 - Métodos

**Interfaces**

18 - Interfaces - I

19 - Interfaces - II

**Concurrencia**

20 - Introducción a la Concurrencia

21 - Goroutinas

22 - Canales (channels)

23 - Canales con memoria intermedia(buffer) y grupos de trabajo

24 - Seleccionar (Select)

25 - Mutex

**Programación orientada a objetos**

26 - Estructuras en lugar de clases

27 - Composición en lugar de herencia

28 - Polimorfismo

**Tratamiento(handling) de aplazamientos(Defer) y errores**

29 - Defer(Aplazamiento)

30 - Tratamiento de errores

31 - Errores personalizados

32 - Pánico y recuperación

**Funciones de primera clase**

33 - Funciones de primera clase (first class)

**Reflexión**

34 - Reflexión

**Tratamiento de archivos**

35 - Lectura de archivos

36 - Escribir archivos

**Parte 1: Introducción e instalación**

**Qué es Golang**

Go también conocido como Golang es un lenguaje de programación de código abierto, compilado y escrito estáticamente creado por google.

El objetivo principal de Golang es hacer que el desarrollo de aplicaciones web altamente disponibles y escalables sea simple y fácil.

**Por qué Golang**

¿Por qué elegir Golang como su lenguaje de programación de servicio cuando hay toneladas de otros lenguajes como python, ruby, nodejs.... que hacen el mismo trabajo?

Estos son algunos de los pros que encontré al elegir Go

La capacidad para el desarrollo es una parte inherente del lenguaje. Como resultado, escribir programas multihilo es pan comido. Esto se logra mediante Goroutinas y canales que discutiremos más adelante en los próximos capítulos.

Golang es un lenguaje compilado. El código fuente se compila en binario nativo. Esto falta en lenguajes interpretados como JavaScript usado en nodejs.

La especificación del lenguaje es bastante simple. Toda la especificación cabe en una página e incluso puedes usarla para escribir tu propio compilador.

El compilador go soporta enlaces estáticos. Todo el código go puede ser enlazado estáticamente en un gran binario y puede ser desplegado en servidores en nube fácilmente sin preocuparse por las dependencias.

**Instalación.** Golang es compatible con las tres plataformas Mac, Windows y Linux. Puede descargar el binario para la plataforma correspondiente desde **https://golang.org/dl/**

Mac OS

Descargue el instalador de OS X desde https://golang.org/dl/. Doble toque(click) para iniciar la instalación. Siga las indicaciones y esto debería tener Golang instalado en /usr/local/go y también habría añadido la carpeta /usr/local/go/bin a su variable de entorno PATH(ruta).

Windows

Descargue el instalador MSI desde https://golang.org/dl/. Pulse dos veces para iniciar la instalación y siga las instrucciones. Esto instalará Golang en la ubicación c:\Go y también añadirá el directorio c:\Go\bin a su variable de entorno de ruta(PATH).

Linux

Descargue el archivo tar de https://golang.org/dl/ y descomprímalo en /usr/local.

Añada /usr/local/go/bin a la variable de entorno PATH. Esto debería instalarse en linux.

**Parte 2: Hola Mundo**

No hay mejor manera de aprender un lenguaje de programación que ensuciarse las manos con el código. Sigamos adelante y escribamos nuestro primer programa. Personalmente recomendaría usar Visual Studio Code con la extensión go como IDE. Tiene autocompletar, estilo de código y una gran cantidad de otras características.

Configuración del espacio de trabajo( workspacepath) Go

Antes de empezar a escribir código, tenemos que configurar el espacio de trabajo de go. En el caso de Mac o Linux, el espacio de trabajo de go debe estar ubicado en $HOME. Así que vamos a crear un directorio dentro de $HOME.: $HOME/go

En el caso de Windows, el espacio de trabajo debe estar ubicado en C:/User/TuNombre/. Así que vamos a crear el directorio GO dentro de [C:/User/TuNombre](file:///C:/User/TuNombre): C:/User/TuNombre/go

Es posible utilizar un directorio diferente como espacio de trabajo configurando la variable de entorno GOPATH. Pero por ahora usemos la ubicación anterior para simplificar.

Todos los archivos fuente para go deben estar ubicados en un directorio llamado **src** dentro del espacio de trabajo. Así que vamos a crear un directorio **src** dentro del directorio go que creamos arriba.

Cada proyecto go debería a su vez tener su propio subdirectorio dentro de **src**. Vamos a crear un directorio **hola** dentro de **src** para llevar a cabo el proyecto **hola mundo.**

La estructura de directorios debe ser similar a la que se muestra a continuación después de crear los directorios anteriores:

go

src

hola

Guarde el siguiente programa como holamundo.go en el directorio hola que acabamos de crear.

package main

import "fmt"

func main() {

fmt.Println("Hola mundo")

}

He aquí cómo se verá la estructura de directorios después de crear el programa anterior

go

src

hola

holamundo.go

**Ejecución de un programa Go**

Hay un par de maneras diferentes de ejecutar un programa. Veamos uno por uno.

1) Usando el comando go run - Escriba go run (EspacioDeTrabajo)/src/hola/holamundo.go en la línea de comandos.

la ruta del espacio de trabajo en el comando anterior debe ser reemplazada por la ruta de su espacio de trabajo (C:/Usuarios/SuNombre/go en windows y $HOME/go en linux o Mac)

Deberías ver la salida **Hola mundo** en la consola.

2) Usando el comando go install - Ejecute el comando go install hola seguido de espacio de trabajo/bin/hola para ejecutar el programa.

en el comando anterior debe ser reemplazada por la ruta de su espacio de trabajo (C:/Usuarios/SuNombre/go en windows y $HOME/go en linux o Mac). Debería ver la misma salida: **Hola Mundo** en la salida estandar.

Cuando teclea go install hola, la herramienta go busca el paquete hola (hola se llama como paquete, más adelante veremos los paquetes con más detalle) dentro del espacio de trabajo. Luego crea un binario llamado hola(hola.exe en el caso de windows) dentro del directorio bin del espacio de trabajo. La estructura de directorios debería ser la siguiente después de ejecutar go install hello

go

bin

hello

src

hola

holamundo.go

Una breve explicación del programa hola mundo

Aquí está el programa hola mundo que acabamos de escribir

package main //1

import "fmt" //2

func main() { //3

fmt.Println("Hola mundo")

}

Aquí veremos brevemente lo que hace cada línea del programa. En los próximos capítulos nos adentraremos en cada sección.

package main - Cada archivo go debe comenzar con la declaración del nombre del paquete. Los paquetes se utilizan para proporcionar compartimentación y reutilización del código. Aquí el nombre del paquete utilizado es principal(main)

import "fmt" - El paquete fmt es importado y será usado dentro de la función principal para imprimir

texto a la salida estándar.

func main() - Es la función principal especial. La ejecución del programa comienza desde la función principal. La función principal debe residir siempre en el paquete principal. Los símbolos { y } indican el inicio y el final de la función principal.

fmt.println ("Hola mundo") - La función Println del paquete fmt se utiliza para escribir texto en la salida estándar.

**Parte 3: Variables**

Qué es una variable

Variable es el nombre dado a una posición de memoria para almacenar un valor de un tipo específico que puede ser cambiado o variado. Existen varias sintaxis para declarar variables en go.

**Declaración de una sola variable**

**var** *nombre* **tipo** es la sintaxis para declarar una sola variable.

**Declaración de una sola variable**package main

import "fmt"

func main() {

var edad int // declaración de variable

fmt.Println("Mi edad es", edad)

}

La sentencia var edad int declara una variable llamada edad de tipo int(entero, o sea, sin decimales). No se ha asignado ningún valor a la variable. Si a una variable no se le asigna ningún valor, se inicializa automáticamente con el valor cero del tipo de la variable. En este caso se asigna el valor 0 a la edad. Si ejecuta este programa, puede ver los siguientes mensajes

My edad es 0

A una variable se le puede asignar cualquier valor de su tipo. En el programa anterior se puede asignar cualquier valor entero a la edad.

package main

import "fmt"

func main() {

var edad int // variable declaration

fmt.Println("Mi edad es ", edad)

edad = 29 //asignación

fmt.Println("Mi edad es ", edad)

edad = 54 //asignación

fmt.Println("Mi edad es ", edad)

}

El programa anterior producirá la siguiente salida.

Mi edad es 0

Mi edad es 29

Mi edad es 54

**Declaración de una variable con valor inicial**

A una variable también se le puede dar un valor inicial cuando se declara.

var nombre type(tipo) = valorInicial es la sintaxis para declarar una variable con valor inicializa

package main

import "fmt"

func main() {

var edad int = 29 // variable declaración de variable con valor inicial

fmt.Println("mi edad es", edad)

}

En el programa anterior, la edad es una variable de tipo int y tiene valor inicial 29. Si ejecuta el programa anterior, puede ver el siguiente mensaje. Muestra que la edad se ha inicializado con el valor 29.

mi edad es 29

**Inferencia de tipo**

Si una variable tiene un valor inicial, Go podrá inferir automáticamente el **tipo** de esa variable usando ese valor inicial. Por lo tanto, si una variable tiene un valor inicial, se puede omitir el tipo de la declaración de la variable.

Si la variable se declara usando la sintaxis var nombre = ValorInicial, Go deducirá automáticamente el tipo de esa variable a partir del valor inicial.

En el siguiente ejemplo, se puede ver que el tipo int de la edad variable se ha eliminado en la línea no. 6. Como la variable tiene un valor inicial de 29, go puede inferir que es de tipo int.

package main

import "fmt"

func main() {

var edad = 29 // el tipo sera inferido como int

fmt.Println("mi edad es", edad)

}

**Declaración de múltiples variables**

Se pueden declarar múltiples variables en una sola sentencia.

var nombre1, nombre2 type = valorinicial1, valorinicial2 es la sintaxis de la declaración de variables múltiples.

package main

import "fmt"

func main() {

var ancho, alto int = 100, 50 //declarando multiples variables

fmt.Println("El ancho es", ancho, "El alto es", alto)

}

El tipo puede omitirse si las variables tienen valor inicial. El programa a continuación declara múltiples variables usando inferencia de tipo.

package main

import "fmt"

func main() {

var ancho, alto = 100, 50 //"int" es asignado a las variables

fmt.Println("El ancho es", ancho, "El alto es", alto)

}

El programa anterior imprimirá width is 100 height is 50 como salida.

Como probablemente ya habrá adivinado, si el valor inicial no se especifica para la anchura y la altura, se les asignarán 0 como su valor inicial.

package main

import "fmt"

func main() {

var ancho, alto int

fmt.Println("El ancho es", ancho, "El alto es", alto)

ancho = 100

alto = 50

fmt.Println("El nuevo ancho es", ancho, "El nuevo alto es", alto)

}

El programa anterior imprimirá:

**El ancho es 0 El alto es 0**

El nuevo ancho es 100 El nuevo alto es 50

Puede haber casos en los que queramos declarar variables pertenecientes a diferentes tipos en una sola sentencia. La sintaxis para hacer eso es

var (

nombre1 = valorinicial1

nombre2 = valorinicial2

)

El siguiente programa utiliza la sintaxis anterior para declarar variables de diferentes tipos.

package main

import "fmt"

func main() {

var (

nombre = "Rudy"

edad = 29

peso int

)

fmt.Println("mi nombre es", nombre, ", mi edad es", edad, "y mi peso es", peso)

}

Aquí declaramos un nombre de variable de tipo string, edad y altura de tipo int. (Discutiremos sobre los varios tipos disponibles en golang en el siguiente tutorial). Si se ejecuta el programa anterior se obtendrá la salida: mi nombre es Rudy, mi edad es 29 y mi peso es 0

**Declaración corta**

Go también proporciona otra forma concisa de declarar variables. Esto se conoce como declaración corta y utiliza el operador :=

**nombre := valorinicial** es la sintaxis corta para declarar una variable.

package main

import "fmt"

func main() {

nombre, edad := "Rudy", 29 //declaración corta

fmt.Println("mi nombre es", nombre, "mi edad es", edad)

}

Si ejecuta el programa anterior, puede ver como salida: mi nombre es Rudy mi edad es 29.

La declaración a mano corta requiere valores iniciales para todas las variables en el lado derecho del operador de asignación. El siguiente programa lanzará un error que no puede asignar valor a 2 variables. Esto se debe a que no se ha asignado un valor a la edad.

package main

import "fmt"

func main() {

nombre, edad := "Rudy" //error

fmt.Println("mi nombre es", nombre, "mi edad es", edad)

}

La sintaxis de mano corta sólo se puede usar cuando al menos una de las variables en el lado izquierdo de := se declara de nuevo. Considere el siguiente programa,

package main

import "fmt"

func main() {

a, b := 20, 30 // declare variables a y b

fmt.Println("a es", a, "b es", b)

b, c := 40, 50 // b ya sido declarada pero c es nueva

fmt.Println("b es", b, "c es", c)

b, c = 80, 90 //asignación de nuevos valores a variables declaradas antes b y c

fmt.Println("el cambio en b es", b, "y en c es", c)

}

en el programa anterior, en la línea no. 8, b ya ha sido declarado pero c es nuevo y por lo tanto funciona y produce

a es 20 b es 30

b es 40 c es 50

cambios b es 80 c es 90

Mientras que si ejecutamos el programa de abajo,

package main

import "fmt"

func main() {

a, b := 20, 30 //a y b declarada

fmt.Println("a es", a, "b es", b)

a, b := 40, 50 //error, no hay nuevas variables

}

Este arrojara el error: not new variables on left side of :=

Esto se debe a que las variables a y b ya han sido declaradas y no hay nuevas variables en el lado izquierdo de :=.

A las variables también se les pueden asignar valores que se calculan durante el tiempo de ejecución. Considere el siguiente programa,

package main

import (

"fmt"

"math"

)

func main() {

a, b := 145.8, 543.8

c := math.Min(a, b)

fmt.Println("el valor mínimo es", c)

}

En el programa anterior, el valor de c se calcula durante el tiempo de ejecución y es el mínimo de a y b. El programa anterior imprimirá

el valor mínimo es 145,8

Dado que Go es fuertemente tipado(no se realiza operaciones entre variables de diferentes tipos), a las variables declaradas como pertenecientes a un tipo no se les puede asignar un valor de otro tipo. El siguiente programa lanzará un error no puede usar "naveen" (tipo cadena) como tipo int en la asignación porque la edad es declarada como tipo int y estamos tratando de asignarle un valor de cadena.

package main

func main() {

edad := 29 // edad es int

edad = "naveen" // error ya que estamos tratando de asignar una cadena a una variable de tipo entero(int)

}

Part 4: Tipos

Los siguientes son los tipos de variables básicos disponibles en go

bool

Numéricos: int8, int16, int32, int64, int

uint8, uint16, uint32, uint64, uint

float32, float64

complex64, complex128

byte

rune

string

**bool**

Un tipo bool representa a las variables que solo se le asignan los valores booleanos: true(verdadero) o false(falso).

package main

import "fmt"

func main() {

u := true // a la variable use la asigna el valor booleano true

v := false // a la variable vse le asigna el valor booleano false

fmt.Println("u:", u, "v:", v)

c := u && v

fmt.Println("c:", c)

d := u || v

fmt.Println("d:", d)

}

En el programa anterior, se le asigna true a u y false a v.

c tiene asignado el valor de u && v. El operador && devuelve verdadero sólo cuando **u** y **v** son verdaderos. Así que en este caso **c** es falso.

El operador || devuelve true cuando u o v es true. En este caso, **d** se le asigna true ya que **u** es true. Obtendremos la siguiente salida para este programa.

u: true v: false

c: false

d: true

**Tipos de números enteros**

int8: representa números enteros construidos con 8 bits

tamaño: 8 bits

rango: -128 a 127

int16: representa números enteros construidos con 16 bits

tamaño: 16 bits

rango: -32768 a 32767

int32: representa números enteros construidos con 32 bits

tamaño: 32 bits

rango: -2147483648 a 2147483647

int64: representa números enteros construidos con 64 bits

tamaño: 64 bits

rango: -922337203685477575808 a 9223372036854775807

int: representa números enteros de 32 ó 64 bits dependiendo de la plataforma subyacente. Generalmente se debe utilizar int para representar números enteros, a menos que sea necesario utilizar un número entero de tamaño específico. tamaño: 32 bits en sistemas de 32 bits y 64 bits en sistemas de 64 bits. rango: -2147483648 a 2147483647 en sistemas de 32 bits y -9223372036854775808 a 9223372036854775807 en sistemas de 64 bits

package main

import "fmt"

func main() {

var a int = 89

b := 95

fmt.Println("valor de a es", a, "y b es", b)

}

El programa anterior dará el valor de salida: a es 89 y b es 95

En el programa anterior a es del tipo int y el tipo b se deduce del valor que se le ha asignado (95). Como hemos dicho anteriormente, el tamaño de int es de 32 bits en sistemas de 32 bits y 64 bits en sistemas de 64 bits. Sigamos adelante y verifiquemos esta afirmación.

El tipo de una variable se puede imprimir utilizando el ***especificador de formato*** %T en el método **Printf**(*al final de capitulo se ampliará algo este aspecto)*. Go tiene un paquete(unsafe) inseguro que tiene una función Sizeof que devuelve en bytes el tamaño de la variable que se le ha pasado. El paquete inseguro debe ser usado con cuidado ya que el código que lo usa puede tener problemas de portabilidad, pero para los propósitos de este tutorial podemos usarlo.

El siguiente programa muestra el tipo y tamaño de ambas variables a y b. %T es el especificador de formato para imprimir el tipo y %d se utiliza para imprimir el tamaño.

package main

import (

"fmt"

"unsafe"

)

func main() {

var a int = 89

b := 95

fmt.Println("valor de a es", a, "y b es", b)

fmt.Printf("El tipo de a is %T, el tamaño de a is %d bytes", a, unsafe.Sizeof(a)) //tipo y tamaño de a

fmt.Printf("\nEl tipo de b is %T, el tamaño de b is %d bytes", b, unsafe.Sizeof(b)) //tipo y tamaño de b

}

El programa arriba producirá la salida siguiente:

valor de a es 89 y b is 95

El tipo de a es int, el tamaño de a es 4 bytes

El tipo de b es int, el tamaño de b es 4 bytes

Podemos inferir de la salida anterior que a y b son de tipo int y tienen un tamaño de 32 bits (4 bytes).

La salida variará si ejecuta el programa anterior en un sistema de 64 bits. En un sistema de 64 bits, a y b ocupan 64 bits (8 bytes).

**Números enteros sin signo**

uint8: representa números enteros sin signo de 8 bits

tamaño: 8 bits

rango: 0 a 255

uint16: representa enteros sin signo de 16 bits

tamaño: 16 bits

rango: 0 a 65535

uint32: representa números enteros sin signo de 32 bits

tamaño: 32 bits

rango: 0 a 4294967295

uint64: representa números enteros sin signo de 64 bits

tamaño: 64 bits

rango: 0 a 18446744073709551615

uint : representa números enteros sin signo de 32 ó 64 bits dependiendo de la plataforma subyacente.

tamaño: 32 bits en sistemas de 32 bits y 64 bits en sistemas de 64 bits.

rango : 0 a 4294967295 en sistemas de 32 bits y 0 a 18446744073709551615 en sistemas de 64 bits

**Tipos de coma flotante**

float32: números de 32 bits en coma flotante

float64: Números de coma flotante de 64 bits

package main

import (

"fmt"

)

func main() {

a, b := 5.67, 8.97

fmt.Printf("tipo de las variables a: %T y b: %T\n", a, b)

suma := a + b

difer := a - b

fmt.Println("Su suma es:", suma, "y su diferencia:", difer)

no1, no2 := 56, 89

fmt.Println("La suma de",no1," + ",no2,":", no1+no2, "y su diferencia:", no1-no2)

}

Los tipos a y b se infieren a partir del valor que se les ha asignado. En este caso a y b son del tipo float64 (float64 es el tipo por defecto para los valores de coma flotante). Sumamos a y b y lo asignamos a la variable suma. Restamos b de a y lo asignamos a difer. A continuación, se imprime la suma y la diferencia. Se realiza un cálculo similar con no1 y no2. El programa anterior imprimirá

tipo de las variables a: float64 y b: float64

Su suma es: 14.64 y su diferencia: -3.3000000000000007

La suma de 56 + 89 : 145 y su diferencia: -33

**Tipos complejos**

complex64: números complejos que tienen partes flotantes32 reales e imaginarias

complex128: números complejos con partes reales e imaginarias float64

La función compleja(complex) incorporada a GO se utiliza para construir un número complejo con partes reales e imaginarias. La función compleja tiene la siguiente definición

func complex(r, i FloatType) ComplexType

Toma una parte real e imaginaria como parámetro y devuelve un tipo complejo. Tanto las partes reales como las imaginarias deben ser del mismo tipo, es decir, float32 o float64. Si tanto la parte real como la imaginaria son float32, esta función devuelve un valor complejo de tipo complex64. Si tanto la parte real como la imaginaria son de tipo float64, esta función devuelve un valor complejo de tipo complex128

Los números complejos también se pueden crear usando la sintaxis abreviada

c := 6 + 7i

Escribamos un pequeño programa para entender los números complejos.

package main

import (

"fmt"

)

func main() {

c1 := complex(5, 7)

c2 := 8 + 27i

cadd := c1 + c2

fmt.Println("suma de ",c1,"+",c2," es:", cadd)

cmul := c1 \* c2

fmt.Println("su producto es:", cmul)

}

En el programa anterior, c1 y c2 son dos números complejos. c1 tiene 5 como parte real y 7 como parte imaginaria. c2 tiene parte real 8 y parte imaginaria 27. a cadd se le asigna la suma de c1 y c2 y a cmul se le asigna el producto de c1 y c2. Este programa emitirá

suma de (5+7i) + (8+27i) es: (13+34i)

su producto es: (-149+191i)

**Otros tipos numéricos**

byte es un alias de uint8

rune es un alias de int32

Discutiremos bytes y runas con más detalle cuando aprendamos sobre las cadenas.

**Tipo cadena**

Las cadenas son una colección de bytes en golang. Está bien si esta definición no tiene ningún sentido. Por ahora podemos asumir que una cadena es una colección de caracteres. Aprenderemos sobre las cadenas en detalle en un capitulo separado.

Permite escribir un programa usando cadenas.

package main

import (

"fmt"

)

func main() {

nombre := "Naveen"

apellido := "Ramanathan"

name := nombre +" "+ apellido

fmt.Println("Mi nombre es",name)

}

En el programa anterior, primero se asigna la cadena "Naveen" a la variable nombre, y a la variable apellido la cadena "Ramanathan". Las cadenas pueden concatenarse utilizando el operador +. Se asigna la variable name primero el valor de la variable nombre concatenado a un espacio y seguido concatenado al valor de la variable apellido de último. El programa anterior imprimirá: Mi nombre es Naveen Ramanathan como salida.

Hay algunas otras operaciones que se pueden realizar en las cadenas. Los veremos en un tutorial separado.

**Tipo Conversión**

Go es muy estricto en cuanto a la tipificación explícita. No hay ningún tipo de conversión automática. Veamos qué significa esto con un ejemplo.

package main

import (

"fmt"

)

func main() {

i := 55 //int

j := 67.8 //float64

sum := i + j //int + float64 no esta permitido

fmt.Println(sum)

}

El código anterior es perfectamente legal en el lenguaje C. Pero en el caso de go, esto no funcionará. i es de tipo int y j es de tipo float64. Estamos tratando de añadir 2 números de diferentes tipos, lo que no está permitido. Cuando ejecute el programa, obtendrá: ./(programa).go:8:14: error: incompatible types in binary expression

sum := i + j //int + float64 not allowed

Para corregir el error, tanto i como j deben ser del mismo tipo. Vamos a convertir j a int. T(v) es la sintaxis para convertir un valor v a tipo T: float64(v)

package main

import (

"fmt"

)

func main() {

i := 55 //int

j := 67.8 //float64

sum := i + int(j) //j es convertido a int

fmt.Println(sum)

}

Ahora, cuando ejecute el programa anterior, podrá ver 122 como salida.

Lo mismo ocurre con la asignación. La conversión de tipo explícito es necesaria para asignar una variable de un tipo a otro. Esto se explica en el siguiente programa.

package main

import (

"fmt"

)

func main() {

i := 10

var j float64 = float64(i) //Esta instrucción no funcionaria sin una conversión de tipo explicita

fmt.Println("j", j)

}

En la línea no. 7, i se convierte a float64 y luego se asigna a j. Cuando intente asignar i a j sin ninguna conversión de tipo, el compilador lanzará un error.

Go ofrece un excelente soporte para el formateo de cadenas en las salidas con el uso del método printf del paquete fmt.. A continuación se muestran algunos ejemplos de tareas de formato de cadena: comunes.

Para imprimir el tipo de una variable p, utilice %T

fmt.Printf("%T", p)

El formato para los valores booleanos

fmt.Printf("%t", true)

Hay muchas opciones para formatear enteros. Utilice %d para formato estándar, base-10.

fmt.Printf("%d", 123)

Esto imprime una representación binaria.

fmt.Printf("%b", 14)

Esto imprime el carácter correspondiente al entero dado.

fmt.Printf("%c", 33)

%x proporciona codificación hexadecimal.

fmt.Printf("%x", 456)

También hay varias opciones de formato para los flotxx. Para el uso de formato decimal básico %f.

fmt.Printf("%f", 78.9)

%e y %E formatean los flotxx en (versiones ligeramente diferentes de) notación científica.

fmt.Printf("%e", 123400000.0)

fmt.Printf("%E", 123400000.0)

Para la impresión básica de cadenas Utilice %s.

fmt.Printf("%s", "\"string\"")

Para doble comillas en cadenas como en el codigo de Go, utilice %q.

fmt.Printf("%q", "\"string\"")

Al igual que con los enteros vistos anteriormente, %x representa la cadena en la base-16, con dos caracteres de salida por byte de entrada.

fmt.Printf("%x", "hex this")

Para imprimir una representación de un puntero, utilice %p.

fmt.Printf("%pn", &p)

Al formatear números, a menudo querrá controlar el ancho y la precisión de la expresión resultante. Para especificar el ancho de un entero, utilice un número después de % en el verbo. Por defecto el resultado será justificado a la derecha y rellenado con espacios.

fmt.Printf("|%6d|%6d|", 12,345)

También puede especificar el ancho de los flotxx, aunque por lo general también desea restringir la precisión de los decimales al mismo tiempo con la sintaxis de precisión de ancho.

fmt.Printf("|%6.2f|%6.2f|", 1.2, 3.45)

Para justificar la izquierda, utilice el indicador bandera -.

fmt.Printf("|%-6.2f|%-6.2f|", 1.2, 3.45)

También es posible que desee controlar el ancho al formatear cadenas, especialmente para asegurarse de que se alineen en una salida similar a una tabla. Para la anchura justificada a la derecha.

fmt.Printf("|%6s|%6s|", "foo", "b")

Par justificar a la izquierda utilice el indicador bandera - como con los números.

fmt.Printf("|%-6s|%-6s|", "foo", "b")

package main

import "fmt"

func main() {

p := 3.5

fmt.Printf("%T\n", p) // Para imprimir el tipo de un valor, utilice %T

fmt.Printf("%t\n", true) // El formato booleano es directo.

fmt.Printf("%d\n", 123) // Hay muchas opciones para formatear enteros. Utilice %d para formato estándar, base-10.

fmt.Printf("%b\n", 14) // Esto imprime una representación binaria.

fmt.Printf("%c\n", 33) // Esto imprime el carácter correspondiente al entero dado.

fmt.Printf("%x\n", 456) //% x proporciona codificación hexadecimal.

fmt.Printf("%f\n", 78.9) //También hay varias opciones de formato para los flotxx. Para el uso de formato decimal básico %f.

fmt.Printf("%e\n", 123400000.0) // %e y %E formatean los flotxx en (versiones ligeramente diferentes de) notación científica.

fmt.Printf("%E\n", 123400000.0)

fmt.Printf("%s\n", "\"string\"") // Para la impresión básica de cadenas Utilice %s.

fmt.Printf("%q\n", "\"string\"") // To double-quote strings as in Go source, use %q. Para doble comillas en cadenas como en el codigo de Go, utilice %q.

fmt.Printf("%x\n", "hex this") // Al igual que con los enteros vistos anteriormente, %x representa la cadena en la base-16, con dos caracteres de salida por byte de entrada.

fmt.Printf("%p\n", &p) //Para imprimir una representación de un puntero, utilice %p.

fmt.Printf("|%6d|%6d|\n", 12,345) // Al formatear números, a menudo querrá controlar el ancho y la precisión de la expresión resultante.

// Para especificar el ancho de un entero, utilice un número después de % en el verbo.

// Por defecto el resultado será justificado a la derecha y rellenado con espacios.

fmt.Printf("|%6.2f|%6.2f|\n", 1.2, 3.45) //También puede especificar el ancho de los flotxx, aunque por lo general también desea

//restringir la precisión de los decimales al mismo tiempo con la sintaxis de precisión de ancho.

fmt.Printf("|%-6.2f|%-6.2f|\n", 1.2, 3.45) //Para justificar la izquierda, utilice el indicador bandera -.

fmt.Printf("|%6s|%6s|\n", "foo", "b") // También es posible que desee controlar el ancho al formatear cadenas,

// especialmente para asegurarse de que se alineen en una salida similar a una tabla. Para la anchura justificada a la derecha.

fmt.Printf("|%-6s|%-6s|\n", "foo", "b") // Par justificar a la izquierda utilice el indicador bandera - como con los números.

}

Salida:

float64

true

123

1110

!

1c8

78.900000

1.234000e+08

1.234000E+08

"string"

"\"string\""

6865782074686973

0x18300118

| 12| 345|

| 1.20| 3.45|

|1.20 |3.45 |

| foo| b|

|foo |b |

Part 5: Constants

Definición

El término constante se utiliza en Go para denotar valores fijos como 5, -89, "me encanta Go", 67.89 y así sucesivamente.

Considere el siguiente código,

var a int = 50

var b string = "me encanta Go"

En el código anterior a las variable a y b se le asignan las constantes 50 y *me encanta Go* respectivamente. La palabra clave const se utiliza para indicar constantes a 50 y *me encanta Go*. Aunque no usamos explícitamente la palabra clave const en ninguna parte del código anterior, internamente son constantes en Go.

Como indica su nombre las constantes no pueden ser reasignadas con ningún otro valor y por lo tanto el siguiente programa no funcionará y fallará con un error de compilación que no se puede asignar a a.

package main

func main() {

const a = 55 //allowed

a = 89 //reasignación no permitida

}

salida:

./(programa).go:4:7: error: invalid left hand side of assignment

a = 89 //reassignment not allowed

*El valor de una constante debe conocerse en el momento de la compilación*. Por lo tanto, no se puede asignar a una variable un valor devuelto por una llamada de función, ya que la llamada de función tiene lugar en tiempo de ejecución.

package main

import (

"fmt"

"math"

)

func main() {

fmt.Println("Hola")

var a = math.Sqrt(4)//permitido

const b = math.Sqrt(4)//no permitido

fmt.Println("Suma de ",a," + ",b,"es",a+b)

}

En el programa anterior, a es una variable y por lo tanto se le puede asignar el resultado de la función math.Sqrt(4) (Discutiremos las funciones con más detalle en un capitulo separado).

b es una constante y el valor de b necesita ser conocido en tiempo de compilación. La función math.Sqrt(4) será evaluada sólo durante el tiempo de ejecución y por lo tanto const b = math.Sqrt(4) arroja error: ./(programa).go:9:20: error: expression is not constant

const b = math.Sqrt(4)//no permitido

**Constantes de cadena**

Cualquier valor entre comillas dobles es una constante de cadena en Go. Por ejemplo, cadenas como "Hola mundo" o "Sam" son todas constantes en Go.

¿A qué tipo pertenece una constante de cadena? La respuesta es que ellas son sin tipo

Una constante de cadena como "Hello World" no tiene ningún tipo.

const hola = "hola mundo"

En el caso anterior, hemos asignado "Hola mundo" a una constante llamada hola. ¿Ahora la constante hola tiene un tipo? La respuesta es No. La constante todavía no tiene un tipo.

Go es un lenguaje fuertemente tipado. Todas las variables requieren un tipo explícito. Entonces, ¿cómo funciona el siguiente programa que asigna a una variable name una constante sin tipo “Sam”?

package main

import (

"fmt"

)

func main() {

var name = "Sam"

fmt.Printf("type %T value %v", name, name)

}

La respuesta es que las constantes sin tipo tienen un tipo predeterminado asociado y se proporciona si y solo si una línea de código lo exige. En la declaración var name = "Sam", name necesita un tipo y lo obtiene del tipo predeterminado de la constante de cadena "Sam", que es una cadena.

¿Hay una manera de crear una constante con tipo? La respuesta es sí. El siguiente código crea una constante con tipo.

const typedhello string = "Hello World"

typedhello en el código anterior es una constante de tipo cadena.

Go es un lenguaje fuertemente tipado. No se permiten los tipos de mezcla durante la asignación. Veamos qué significa esto con la ayuda de un programa.

package main

func main() {

var defaultName = "Sam" // permitido

type myString string

var customName myString = "Sam" //permitido

customName = defaultName //no permitido

}

En el código anterior, primero creamos una variable defaultName(nombre predeterminado) y le asignamos la constante Sam. El tipo predeterminado de la constante Sam es cadena, por lo que después de la asignación, defaultName es de tipo Cadena.

En la siguiente línea creamos un nuevo tipo myString que es un alias de cadena.

Luego creamos una variable customName de tipo myString y le asignamos la constante Sam. Dado que la constante Sam no tiene tipo, se puede asignar a cualquier variable de cadena. Por lo tanto, esta asignación está permitida y customName obtiene el tipo myString.

Ahora tenemos una variable defaultName de tipo string y otra variable customName de tipo myString. Aunque sabemos que myString es un alias de cadena, la política de tipificación fuerte de Go no permite que se asignen variables de un tipo a otro. Por lo tanto, la asignación customName = defaultName no está permitida y el compilador lanza un error: ./(programa).go:7:20: error: incompatible types in assignment (cannot use type string as type myString)

customName = defaultName //no permitido

**Constantes booleanas**

Las constantes booleanas no son diferentes de las constantes de cadena. Hay dos constantes no tipadas true(verdadero) y false(falso). Las mismas reglas para las constantes de cadenas se aplican a los booleanas, por lo que no las repetiremos aquí. El siguiente es un programa simple para explicar las constantes booleanas.

package main

func main() {

const trueConst = true

type myBool bool

var defaultBool = trueConst //permitido

var customBool myBool = trueConst //permitido

defaultBool = customBool //no permitido

}

El programa anterior es auto explicativo.

**Constantes numéricas**

Las constantes numéricas incluyen enteros, flotadores y constantes complejas. Hay algunas sutilezas en las constantes numéricas.

Veamos algunos ejemplos para aclarar las cosas.

package main

import (

"fmt"

)

func main() {

const a = 5

var intVar int = a

var int32Var int32 = a

var float64Var float64 = a

var complex64Var complex64 = a

fmt.Println("intVar",intVar, "\nint32Var", int32Var, "\nfloat64Var", float64Var, "\ncomplex64Var",complex64Var)

}

Salida:

intVar 5

int32Var 5

float64Var 5

complex64Var (5+0i)

En el programa anterior, la const a no tiene tipo y tiene un valor de 5. Usted se estará preguntando cuál es el tipo por defecto de a y si tiene uno, ¿cómo lo asignamos a variables de diferentes tipos?. La respuesta está en la sintaxis de a. El siguiente programa hará las cosas más claras.

package main

import (

"fmt"

)

func main() {

var i = 5

var f = 5.6

var c = 5 + 6i

fmt.Printf("i's type %T, f's type %T, c's type %T", i, f, c)

}

Salida: i's type int, f's type float64, c's type complex128

En el programa anterior, el tipo de cada variable está determinado por la sintaxis de la constante numérica. 5 es un número entero por sintaxis, 5.6 es un flotador y 5 + 6i es un número complejo por sintaxis. Cuando se ejecuta el programa anterior, imprime i's tipo int, f's tipo float64, c's tipo complex128

Ahora espero que quede claro cómo funcionó el programa de abajo.

package main

import (

"fmt"

)

func main() {

const a = 5

var intVar int = a

var int32Var int32 = a

var float64Var float64 = a

var complex64Var complex64 = a

fmt.Println("intVar",intVar, "\nint32Var", int32Var, "\nfloat64Var", float64Var, "\ncomplex64Var",complex64Var)

}

salida:

intVar 5

int32Var 5

float64Var 5

complex64Var (5+0i)

En este programa, el valor de a es 5 y la sintaxis de a es genérica (puede representar un flotador, un entero o incluso un número complejo sin parte imaginaria) y por lo tanto se puede asignar a cualquier tipo compatible. El tipo por defecto de este tipo de constantes se puede pensar que se generan sobre la marcha dependiendo del contexto. var intVar int = a requiere a ser int para que se convierta en una constante int. var complex64Var complex64 = a requiere a ser un número complejo y por lo tanto se convierte en una constante compleja. Bastante bien:).

**Expresiones numéricas**

Las constantes numéricas son libres de ser mezcladas y emparejadas en expresiones y sólo se necesita un tipo cuando se asignan a variables o se utilizan en cualquier lugar del código que exija un tipo.

package main

import (

"fmt"

)

func main() {

var a = 5.9/8

fmt.Printf("a's type %T value %v",a, a)

}

salida: a's type float64 value 0.7375

En el programa anterior, 5.9 es un float por sintaxis y 8 es un entero por sintaxis. Todavía se permite 5.9/8 ya que ambas son constantes numéricas. El resultado de la división es 0.7375 es un flotador y por lo tanto la variable a es de tipo flotador. La salida del programa es un valor de tipo float64 de 0.7375.

Part 6: Functions

¿Qué es una función?

Una función es un bloque de código que realiza una tarea específica. Una función toma una entrada, realiza algunos cálculos sobre la entrada y genera una salida.

Declaración de funciones

La sintaxis general para declarar una función en go es la siguiente

func functionname(parametername type) returntype {

//function body

}

La declaración de función comienza con una palabra clave func seguida del nombre de la función. Los parámetros se especifican entre ( y ) seguidos del tipo de retorno de la función. La sintaxis para especificar un parámetro es el nombre del parámetro seguido del tipo. Se puede especificar cualquier número de parámetros como (parámetro1 tipo, parámetro2 tipo). Luego hay un bloque de código entre { y } que es el cuerpo de la función.

Los parámetros y el tipo de retorno son opcionales en una función. Por lo tanto, la siguiente sintaxis es también una declaración válida de función.

func functionname() {

}

Función de ejemplo

Escribamos una función que toma el precio de un producto y la cantidad de productos como parámetros de entrada y calcula el precio total multiplicando estos dos valores y devuelve este como salida de la función.

func calculateBill(price int, cant int) int {

var totalPrice = price \* cant

return totalPrice

}

La función anterior tiene dos parámetros de entrada price y cant de tipo int y devuelve el totalPrice que es el producto de price y cant. El valor de retorno es también de tipo int.

Si los parámetros consecutivos son del mismo tipo, podemos evitar escribir el tipo cada vez y es suficiente con que se escriba una vez al final.ie price int, no int puede escribirse como price, cant int. Por lo tanto, la función anterior puede reescribirse como,

func calculateBill(price, no int) int {

var totalPrice = price \* cant

return totalPrice

}

Ahora que tenemos la función lista, llamémosla desde algún lugar del código. La sintaxis para llamar una función es functionname(parameters). La función anterior se puede llamar utilizando el código.

calculateBill(10, 5)

Aquí está el programa completo que utiliza la función anterior e imprime el precio total.

package main

import (

"fmt"

)

func calculateBill(price, cant int) int {

var totalPrice = price \* cant

return totalPrice

}

func main() {

price, cant := 90, 6

totalPrice := calculateBill(price, cant)

fmt.Println("Precio Total es", totalPrice)

}

El programa arriba imprimira:

Precio Total es 540

**Múltiples valores de retorno**

Es posible devolver múltiples valores de una función. Escribamos una función rectProps que toma el largo y ancho de un rectángulo y devuelve tanto el área como el perímetro del rectángulo. El área del rectángulo es el producto de el largo y la anchura y el perímetro es el doble de la suma de el largo y la anchura.

package main

import (

"fmt"

)

func rectProps(largo, ancho float64)(float64, float64) {

var area = largo \* ancho

var perimetro = (largo + ancho) \* 2

return area, perimetro

}

func main() {

area, perimetro := rectProps(10.8, 5.6)

fmt.Printf("El área del rectángulo de largo %f y ancho %f es %f su perímetro %f",largo,ancho, area, perimetro)

}

Si una función devuelve múltiples valores de retorno, deben especificarse entre ( y ). func rectProps(length, width float64)(float64, float64) tiene dos parámetros float64 largo y ancho y también devuelve dos valores float64. El programa anterior produce:

Area 60.480000 Perimeter 32.800000

**Valores de retorno utilizando variables**

Es posible devolver valores en variables desde una función. Si el valor de retorno se realiza en una variable, esta se declara como una variable en la primera línea de la función.

La función rectProps de arriba pueden ser reescritos usando variables para los retornos

func rectProps(largo, ancho float64)(area, perimetro float64) {

area = largo \* ancho

perimetro = (largo + ancho) \* 2

return //no explicit return value

}

Las variables area y el perimetro contienen los valores de retorno en la función anterior. Tenga en cuenta que la declaración de retorno(return) de la función no devuelve explícitamente ningún valor. Dado que las variables area y perimetro se especifican en la declaración de función como valores de retorno, se devuelven automáticamente desde la función cuando se encuentra la declaración de retorno(return).

**Identificador en blanco**

\_ es conocido como el identificador en blanco en Go. Se puede utilizar en lugar de cualquier variable de cualquier tipo. Veamos para qué sirve este identificador en blanco.

La función rectProps devuelve el área y el perímetro del rectángulo. ¿Y si sólo necesitamos el área y queremos descartar el perímetro? Aquí es donde \_ es útil.

El programa de abajo utiliza sólo el área devuelta por la función rectProps.

package main

import (

"fmt"

)

func rectProps(largo, ancho float64) (float64, float64) {

var area = largo \* ancho

var perimetro = (largo + ancho) \* 2

return area, perimetro

}

func main() {

area, \_ := rectProps(10.8, 5.6) // perimetro se descarta

fmt.Printf("Area %f ", area)

}

En la línea no. 13 Usamos sólo el area y el identificador \_ se usa para descartar el perimetro.