Manual de Golang

Este manual está diseñado para introducirlo a los conceptos fundamentales de Go, un lenguaje que equilibra simplicidad y potencia. A lo largo del contenido, exploraremos desde lo más básico, como el control de flujo y la asignación de variables, hasta aspectos más complejos como el manejo de punteros, memoria dinámica y estructuras definidas por el usuario. Usted se familiarizará con la sintaxis libre de Go, la declaración de variables y funciones, y el uso eficiente de tipos de datos y operadores. Con esta guía, podrá adquirir las herramientas necesarias para escribir programas claros y eficientes en Go.

Conceptos Fundamentales

1. Uso de comandos para controlar el flujo del programa.

En el lenguaje de programación Golang se usan estructuras ya definidas para controlar el flujo del programa. Lo más probable es que ya esté familiarizado con ellas, ya que son las típicas que suelen haber en todo lenguaje de programación imperativo.

Entre ellas se encuentran estructuras que controlan el programa por medio de condicionales, iteración y "saltos". Más adelante se profundizará sobre como definirlas, en la sección de estructuras de control de flujo.

2. Empleo de asignaciones a variables para manipular el estado del sistema.

En Golang se emplea el uso de variables de manera similar a muchos de los lenguajes del parádigma imperativo. En ellas se puede guardar el estado actual del programa, en contadores, flags, etc.

Además, las variables en Go no solo guardan valores literales, también son capaces de almacenar funciones, lo cual hace que el lenguajes sea sumamente flexible y permita nativamente usar patrones como dependency injection.

En cuanto a detalles más específicos sobre como declarar y asignar variables refiérase a la sección de más adelante sobre aspecto del léxico y sintaxis.

3. Forma de definir una ejecución secuencial de instrucciones

Lo primero que hay que hacer es definir el paquete que se va a ejecutar como main, ya que go funciona con paquetes.

```
package main

import "fmt"

func main() {
   fmt.Println("Hola mundo!")
}
```

Ahora bien, dentro del main está el código que será ejecutado línea por línea por go. También se pueden hacer llamadas a funciones lo cual hará que la ejecución se pase a dichas rutinas y al finalizarlas regresará a la ejecución secuencial del main.

Aspectos de Léxico y Sintaxis

1. Sintaxis de forma libre

Tokens

En go existen los tokens, hacen referencia a palabras que tienen un significado para el compilador. Pueden ser palabras reservadas, identificadores, una constante, una literal o un símbolo.

```
fmt.Println("Hola mundo!")
```

En dicho bloque hay distintos tokens que go detecta: fmt,el punto, Println, los paréntesis, y la string.

Salto de línea

En go a diferencia de lenguajes como C o Java, no se ocupa poner un ; al final de una oración, solo basta con usar saltos de línea para cambiar a una nueva instrucción.

Comentarios

Se pueden usar estos dos tipos

```
// Hola go!
/*
    Hola
    go!
*/
```

Espacios en Blanco

En go tanto los comentarios o líneas sin contenido son tomados como espacios en blanco. También le da información al compilador sobre donde está ubicados los elementos en una declaración.

2. Definiciones

Funciones

Todos los programas en go mínimo tienen la función main, pero se recomienda dividir el código en distintas. Cuando se declara una función se le informa al compilador el tipo de dato que retorna, el identificador y sus parámetros.

Usualmente esta es la estructura básica:

```
func nombre( [lista de parametros] ) [tipos de retorno]
{
   cuerpo de la función
}
```

Para más información sobre funciones refiersae a su sección

Variables

Las variables en go se pueden inicializar en dos lugares dentro de funciones (locales) o bien fuera de ellas (package-level/global).

Existen dos tipos de declaración la standard y la short

Standard

Las variables en go se inicializan con la palabra *var* seguido de un identificador (solo un tipo de dato por línea), además usa = como signo :

```
var nombre string = "Maria"
var dia, noche bool = true, false
var edad int = 20
```

También se pueden declarar sin poner el tipo y el compilador va a inferir el tipo de dato (permite muchos tipos por línea):

```
var nombre, dia = "Maria", true
var edad, noche = 20, false
```

Además se pueden declarar sin linkear un valor (se ocupa especificar tipo de dato):

```

```

```
var nombre, apellido string
var edad, mes int
```

Finalmente se pueden declarar muchas variables con solo una palabra clave var:

```
var (
    nombre, edad = "Maria", 20
    dia, noche bool = true, false
)
```

Short

También se usa para asignar variables, pero solo dentro de funciones, además se usa el *:=* como signo y no se usa *var*:

```
nombre, apellido := "Maria", "Estrada"
edad, dia := 20, false
```

cabe recalcar que si se quiere crear una redeclaración se ocupa usar el *=*, mínimo debe haber una declaración nueva para usar *:=* :

```
// se puede
nombre := "Maria"

// se puede
edad, nombre := 20, "Maria"

// no se puede
nombre := "Maria"

// se puede
nombre = "Maria"
```

También si se declaran variables que no se usan dentro del scope, en tiempo de compilación se arrojará un error.

Tipos de Datos

En go existen los siguientes tipos de datos:

- **Boolean:** consisten en los valores binarios (bool) true y false.
- **Numeric:** son tipos númericos, se subdivide en enteros y flotantes.
- **String:** Es una cadena de bits que representa un set de valores de tipo string. Son inmutables.

- **Derived:** Incluye a los punteros, arreglos, estructuras, uniones, funciones, slices, interfaces, maps, y canales.

3. Tipos de datos básicos

Enteros

Los enteros son una subdivición de los tipos de datos numéricos. Golang ofrece divisiones por si el entero tiene signo o no, o bien para diferentes tamaños:

Numero	Tipos	Descripción
1	uint8	enteros sin signo de 8 bits (0 to 255)
2	uint16	enteros sin signo de 16 bits (0 to 65535)
3	uint32	enteros sin signo de 32 bits (0 to 4294967295)
4	uint64	enteros sin signo de 64 bits (0 to 18446744073709551615)
5	int8	enteros con signo de 8 bits (-128 to 127)
6	int16	enteros con signo de 16 bits (-32768 to 32767)
7	int32	enteros con signo de 32 bits (-2147483648 to 2147483647)
8	int64	enteros con signo de 64 bits (-9223372036854775808 to 9223372036854775807)

Dependiendo de la arquitectura en la que se esté se usará por defecto el de 32 o 64 bits.

Punto Flotantes

Los enteros son una subdivición de los tipos de datos numéricos. Golang ofrece divisiones por si el entero tiene signo o no, o bien para diferentes tamaños:

Numero	Tipos	Descripción
1	float32	Números punto flotante de IEEE-754 32-bit
2	float64	Números punto flotante de IEEE-754 64-bit
3	complex64	Números complejos con float32 real y para imaginaria
4	complex128	Números complejos con float64 real y para imaginaria

Void

En go no existe la palabra reservada void, en caso no desear un retorno simplemente se omite el tipo de dato.

4. Declaraciones e identificadores

Uso de palabras clave, como static, extern, typedef

Static

En go no se usa static, y si se desea que una función sea accesible desde otros paquetes se usan los nombres de la función en mayúscula en su primera letra.

extern

Para importar funciones o paquetes externos se usa la palabra *import*.

Typedef

En go si se le quiere dar un alias a un tipo de dato se puede usar *type*, esto permite hacer lo mismo que el typedef en C.

Identificadores

Se usan para darle nombres únicos a funciones, variables, constantes o tipos. En Go los identificadores son case sensitive, por ello VAR != var. Además, estas son las reglas básicas:

- Solo pueden contener letras, digitos, o barras bajas.
- Deben iniciar en letras o barras bajas.
- No pueden contener espacios o carácteres especiales.
- No pueden ser iguales a las palabras reservadas de Go.

Variables Globales y Locales

Ya previamente se comentó sobre ellas y sus usos con *:=* o *=*. Ahora bien, con respecto a sus diferencias, como en muchos lenguajes como C, si hay diferencias, las principales son:

- Las globales se declaran fuera de todas las funciones, mientras las locales son específicas a un scope.
- Las globales se declaran en el heap, y las locales en el stack.
- Las locales solo existen durante la ejecución del scope, y las globales durante todo el programa
- Los scopes superiores son agnósticos a las declaraciones locales de sus scopes "hijos".

Estructuras de Control de Flujo

1. if-else y switch-case.

if-else

Es la típica condicional de dos vías. Tiene una variante que puede generar un valor si se entra en alguna de las dos vías, de manera similar a como ocurre en los fors cuando se define la variable de control. A continuación ambas variantes

```
if  a < b {
    return a
} else {
    return b
}
```

```
if v := math.Pow(x, n); v < lim {
    return v
} else {
    return lim
}
```

Si pone atención verá que no hace falta incluir paréntesis en la condición a diferencia de otros lenguajes como C.

Ahora bien, si antes de entrar al else desea reevaluar una condición se puede hacer ya que Go también permite el uso de else if.

```
if( a == 10 ) {
    fmt.Printf("el valor es 10")
} else if( a == 20 ) {
    fmt.Printf("el valor es 20" )
}
```

Switch

Condicional con más de 2 vías (se puede usar también con dos pero no es lo usual).Además, no hace falta poner un break al final del case para que los otros no ocurra a diferencia de en otros lenguajes, y en caso de querer forzar la caída a un nivel de más abajo se usa fallthrough. También, permite indicar una entrada default por si no se cumple con ninguna de las condiciones.

Existen dos tipos fundamentales de switch, el de expresión y el de tipo,

Expresión

Es un switch similar al de los lenguajes tradicionales, y permite ser declarado de 2 formas:

- Declarar la variable a comparar, y dentro de cada case los valores a comparar:

```
switch dia := semana[3]; dia {
    case "viernes":
        fmt.Println("Ya fin de semana!")
    case "lunes":
        fmt.Println("Apenas Iniciando")
    case "miercoles":
            fmt.Println("Mitad de semana")
    default:
        fmt.Printf("cualquier otro dia")
}
```

- No declarar nada y dejar la condición a comparar dentro del case:

```
switch {
    case nota == 100:
        fmt.Println("Excelente nota")
    case nota == 80:
        fmt.Println("Buena nota")
        case nota == 60:
            fmt.Println("debe mejorar")
}
```

Tipo

Los switch de tipo, permiten identificar a que tipo de una interfáz pertenece una variable:

```
var x interface{}

switch i := x.(type) {
    case int:
        fmt.Printf("x es entero")
    case float64:
        fmt.Printf("x es float64")
    case bool, string:
        fmt.Printf("x es un bool o un string")
    default:
        fmt.Printf("no se sabe")
}
```

2. Bucles

Curiosamente en Golang solo existe una estructura para realizar ciclos, el for. No existe while pero puede ser emulado por medio del for. usualmente esta es la estructura que lleva

```
for [condición |  ( inicialización; condición; incremento ) | rango] {
   //contenido del for
}
```

Como se puede ver todos los "parámetros" en el for son opcionales, de hecho, si se omitiesen todos se crearía el bucle conocido como "while true". A continuación la explicación de cada uno:

- Condición: revisa si realizar otra iteración o si ya termino la ejecución
- Inicialización: es para crear una variable de control del loop dentro de dicho scope, (también se podría solo asignar valor).
- Incremento: es la taza de incremento del índice creado.
- Rango: Permite iterar encima de una colección de objetos.

Un for equivalente al de c++:

```
for i := 0; i < 10; i++ {
        sum += i
}
```

For usado como si fuera un while:

```
for  i > 10 {
    sum += i
    i--
}
```

for para iterar encima de objetos (permite mucha flexibilidad):

```
colores := map[string]string{
    "rojo":  "#FF0000",
    "verde": "#00FF00",
}

for clave, valor := range colores {
    fmt.Printf("Clave: %s, Valor: %s\n", clave, valor)
}
```

3. Control de flujo

En go para controlar el flujo se usan las siguientes estructuras:

- **break**: Termina la ejecución de un ciclo.
- **continue**: continúa a siguiente iteración de un ciclo.
- **goto**: Es similar al jump de ensamblador, transfiere la ejecución de un programa hacia un label. El ejemplo a continuación solo imprime los números mayores a 20 en un arreglo

```
LABEL: for i := 0; i < len(arr); i++ {

    if arr[i] <= 20 {
        goto LABEL:
    }
    fmt.Printf("menor que 20: "arr[i])
}
```

- **fallthrough**: pasa la ejecución de un bloque en un case al siguiente menor.

```
switch number {
    case 1:
        fmt.Println("uno")
        fallthrough
    case 2:
        fmt.Println("dos")
        fallthrough
    case 3:
        fmt.Println("tres")
    }
```

Si number es 2, entonces imprimiría:

```
dos
tres
```

Operadores

1. Aritméticos

Go permite los operadores ariméticos típicos de cualquier lenguaje imperativo. A continuación una tabla con cada uno:

Operador	Descripción
`+`	Suma dos operandos
`-`	Resta entre dos operandos
`*`	Multiplica ambos operandos
`/`	Divide el numerador por el denominador
`%`	Operador módulo: devuelve el resto de una división entera
`++`	Operador de incremento: Aumenta el valor entero en uno
`--`	Operador de decremento: Disminuye el valor entero en uno

2. Relacionales

A continuación los operadores relacionales de Go:

Operador	Descripción
`==`	Comprueba si los valores de dos operandos son iguales
`!=`	Comprueba si los valores de dos operandos no son iguales
`>`	Comprueba si el valor del operando izquierdo es mayor que el valor del operando derecho

Operador	Descripción
`<`	Comprueba si el valor del operando izquierdo es menor que el valor del operando derecho
`>=`	Comprueba si el valor del operando izquierdo es mayor o igual que el valor del operando derecho
`<=`	Comprueba si el valor del operando izquierdo es menor o igual que el valor del operando derecho

3. Lógicos

Los operadores lógicos de Go son práctimente los mismos que los de la matemática booleana:

Operador	Descripción
`&&`	AND lógico, ambos deben ser verdaderos, para ser true
`||`	OR lógico, con uno solo verdadero ya es true
`!`	Negación lógica, invierte el valor

4. Manejo de bits

Tiene todos los más comunes, solo le falta el rotate:

Operador	Descripción
`&`	AND binario, si ambos son 1, copia 1
`|`	OR binario, Solo uno debe ser 1 para copiar 1
`^`	XOR binario, para copiar 1 tiene que haber mínimo y máximo un 1
`<<`	Left Shift, El valor del operando izquierdo se mueve a la izquierda según el número de bits especificado por el operando derecho.
`>>`	Right Shift, El valor del operando izquierdo se mueve a la derecha según el número de bits especificado por el operando derecho.

5. Asignaciones

Golang tiene una lista muy amplia de operadores de asignación:

Operador	Descripción
`=`	Copia el valor de la derecha por el de la derecha y se lo asigna
`+-`	Le suma al operando de la izquierda el valor a la derecha y se lo asigna
`-=`	Le resta al operando de la izquierda el valor a la derecha y se lo asigna
`*=`	Le multiplica al operando de la izquierda el valor a la derecha y se lo asigna
`/=`	Le divide al operando de la izquierda el valor a la derecha y se lo asigna

Operador	Descripción
%=	Le aplica modulo al operando de la izquierda con el valor a la derecha y se lo asigna
<<=	Le aplica un shift izquierdo de bits al operando de la izquierda
>>=	Le aplica un shift derecho de bits al operando de la izquierda
&=	Le aplica un AND de bits al operando de la izquierda
^=	Le aplica un XOR de bits al operando de la izquierda
|=	Le aplica un OR de bits al operando de la izquierda

6. Manejo de punteros

Go solo tiene dos operadores que manejan punteros:

Operador	Descripción
&	Devuelve la dirección en memoria de una variable
*	Especifica que es un puntero a una variable

Funciones

1. llamada

Una vez declarada la función, el compilador conoce su identificador por lo que solo hace falta mencionarlo para llamarla. Al hacer esto se transfiere el control de la ejecución del programa a la función hasta que esta termine y al retornar devuelva el control al main.

```
package main

import "fmt"

func main() {

    fmt.Println("Dentro del main")

    funcion()
}
func funcion()
{
  fmt.Println("Dentro de la funcion")
}
```

2. Múltiples retornos

Una función en go puede retornar múltiples valores de distintos o mismo tipo de la siguiente forma:

```
func invertir(x, y string) (string, string) {
   return y, x
}
```

3. Paso de parámetros

Existen dos formas de pasar parámetros en go (por defecto se usa por valor):

- **Por valor**: Copia el valor pasado a la función y los cambios hechos en el interior de la función no afectan al valor original.
- **Por referencia**: Se pasa una referencia (puntero) hacia el parámetro enviado y así poder modificarlo desde dentro de la función. Se usa el * en el parámetro para definir que se quiere pasar por referencia, y en la llamada se usa & acompañado del nombre de la variable a enviar.

4. Tipos de declaración

Por valor:

- Se pueden crear dentro de la declaración de a una variable:

```
func main(){

   raizCuadrada := func(x float64) float64 {
      return math.Sqrt(x)
   }

   fmt.Println(raizCuadrada(9))
}
```

- También se puede asignar una función nueva a una variable

```
func raizCuadrada(x float64) float64 {
      return math.Sqrt(x)
}
func main(){

   raiz := raizCuadrada

   fmt.Println(raiz(9))
}
```

- Pasarla como parámetro a otra función:

```
func calcular(x int, y int, op func(int, int) int) int {
    return op(x, y)
}
```

```
func multiplicar(x int, y int) int {
    return x * y
}

func main() {
    resultado := calcular (2, 5, multiplicar)
    fmt.Println("Resultado:", resultado)
}
```

- Retornada en una función:

```
func calcular(factor int) func(int) int {
    return func(valor int) int {
        return factor * valor
    }
}

func main() {
    multiplicarPor2 := calcular(2)
    resultado := multiplicarPor2(20)
    fmt.Println("Resultado:", resultado)
}
```

Por Closure:

Este tipo de función es muy única. Ocurre cuando una función puede acceder a los valores de una función que la envuelve. A continuación un ejemplo de un contador que al llamar la variable que almacena la función incrementa a count:

```
contador := func() func() int {
    count := 0
    return func() int {
        count++
        return count
    }
}
incrementar := counter()

//incrementa la variable count en 2
incrementar()
incrementar()
```

Por metodo:

Este tipo es muy común usarlo con structs. Lo que hace es asociar un tipo (structs en su mayoría) con una función usando el operado ".":

```
type Rectangulo struct {
    Ancho, Alto float64
}

func (r Rectangulo) Area() float64 {
    return r.Ancho * r.Alto
}


func main(){

    rect := Rectangulo{2, 3}
    // se puede llamar a la función usando el .
    fmt.Println(rect.area())

}
```

Punteros y Arreglos

1. Punteros

Declaración y Asignación

En Go (y en general), un puntero es una variable que tiene como valor la dirección de memoria de otro objeto. Se declara de la siguiente manera

```
var nombreDeVariable *tipoDeVariable
```

No importa el tipo de dato u objeto al que el puntero haga referencia, siempre lo que tendrá guardado es una dirección en memoria en hexadecimal. A continuación un ejemplo de como asignar la referencia a una variable a un puntero:

```
package main
func main() {
   var num int = 20
   var puntero *int

   puntero = &num  //así se pasa la dirección de memoria al puntero
}
```

De mismo modo, si imprimieramos el valor de &num, puntero, y *puntero se vería así respectivamente:

```
&num -> 0x7FFD3A8B4C20
puntero -> 0x7FFD3A8B4C20
*puntero -> 20
```

Puntero nulo

En Golang un puntero nulo es uno que tiene como valor 0, o bien se le puede asignar nil. Indica que el puntero o bien no ha sido inicializado o que falló su inicialización.

Desreferenciar

Ya previamente en un ejemplo se desreferenció un puntero, esto ocurre cuanod se coloca un * al principio de él. Esto lo que permite es acceder al valor al que hace referencia. Además, permite modificar dicho valor. A continuación un ejemplo:

```
package main
import "fmt"
func main() {
    var num int = 20
    var puntero *int

    fmt.Println("Valor de num ", num)
    fmt.Println("Valor de puntero ", puntero)

    puntero = &num

    fmt.Println("Valor de puntero ", puntero)
    fmt.Println("Valor de *puntero ", *puntero)

    *puntero = 30

    fmt.Println("Valor de num ", num)
}
```

Así se vería la consola luego de ejecutar el programa:

```
Valor de num  20
Valor de puntero  <nil>
Valor de puntero  0xc0000120a0
Valor de *puntero  20
Valor de num  30
```

Puntero a puntero

En go se pueden definir punteros a punteros, esto es exactamente lo que el nombre sugiere, es un puntero, que tiene como valor la dirección de memoria de otro puntero. Se declara de la siguiente manera:

```
var puntero **int;
```

3. Arreglos

Aspectos Generales

En Go los arreglos son una colección de valores de un mismo tipo de dato, que además tienen una longitud definida. La forma en la que se declara es la siguiente:

```
var nombreDeVariable [Tamaño] tipoDeVariable
```

Y si se deseara asignar un valor a alguna posición sería de esta forma:

```
nombreDeVariable[i] = nuevoValor
```

Ahora bien, si se desea inicializarlos con valores se hace de las siguientes maneras:

- Tamaño fijo, no se pueden poner más valores que los que el tamaño dice:

```
var impuesto = [5]float32{10.0, 2.0, 3.4, 7.0, 50.0}
```

- Tamaño inferido, si no se especifica el tamaño el compilador determina el tamaño

```
var impuesto = []float32{10.0, 2.0, 3.4, 7.0, 50.0}
```

Finalmente, para poder acceder a una posición se hace de la siguiente forma:

```
var impuesto float32 = impuesto[5]
```

Memoria Dinámica

1. New

Previamente se habló sobre como asignar un valor a un puntero declarado, se hace dándole una dirección de memoria. Pero en ocasiones se desea tan solo asignarle espacio en memoria y ya despues inicializarle valores directamente. Esto es útil a la hora de querer crear un objeto desde cero. Un ejemplo sería:

```
func main() {
    var p *Persona

    //Ahora le asignamos memoria con new
    p = new(Persona)
```

```
    // Inicializamos los campos
    p.Nombre = "Maria Estrada"
    p.Edad = 30
}
```

De todos modos, la práctica más común suele ser dejar que el compilador de Go infiera que queremos crear dinámicamente un struct, por lo que el new no es muy usado.

```
p := &Person{
    Nombre:   "Maria Estrada",
    Edad:    30
}
```

2. Garbage Collector

En Golang no hace falta preocuparse por fugas de memoria o errores como segfault, ya que implementa un garbage collector que se encarga de manejar toda la memoria dinámica.

Estructuras Definidas por el usuario

1. Estructuras

Go también permite crear estructuras para poder agrupar datos un conjunto de variables dentro de una. Para definir una estructura se hace de la siguiente forma:

```
type nombreDelStruct struct {
   variable tipoDeDato
   variable1 tipoDeDato1
   ...
   variableN tipoDeDatoN
}
```

Y para poder crear una variable struct se hace de las siguientes formas:

```
p := Person{"Maria Estrada", 30}

p := Person{
    Nombre: "Maria Estrada",
    Edad: 30
}

var p Person
p.Nombre = "Maria Estrada"
p.Edad = 30
```

Ambiente de Ejecución

Main

Go similar a como lo hace C, también empieza la ejecución de su programa desde el main, y luego de ahí ya el programa toma la ruta que el programador decida.

Referencias

[1] D. Atal, "Understanding the Fallthrough Keyword in Golang: A Hidden Gem", *Medium*, [Online]. Disponible en: https://durgeshatal1995.medium.com/understanding-the-fallthrough-keyword-in-golang-a-hidden-gem-285fa26f47af.

[2] S. Deshmukh, "Scope of Variables in Go", *Scaler Topics*, [Online]. Disponible en: https://www.scaler.com/topics/golang/scope-of-variables-in-go/.

[3] D. Erhabor, "The new() vs make() Functions in Go – When to Use Each One", *freeCodeCamp*, [Online]. Disponible en: https://www.freecodecamp.org/news/new-vs-make-functions-in-go/.

[4] "Go Tutorial", *tutorialspoint*, [Online]. Disponible en: https://www.tutorialspoint.com/go/index.htm.

[5] GO, "Welcome to a tour of Go", *Go wiki*, [Online]. Disponible en: https://go.dev/tour/list.

[6] GO, "Go Wiki: Switch", *Go wiki*, [Online]. Disponible en: https://go.dev/wiki/Switch.

[7] Go 101 Project, "Constants and Variables", [Online]. Disponible en: https://go101.org/article/constants-and-variables.html.