Go

Go es un lenguaje de programación diseñado originalmente para resolver diversos problemas en el desarrollo de software a gran escala en el mundo real, inicialmente dentro de Google. Aborda la construcción lenta de programas, la gestión de dependencias descontrolada, la complejidad del código y la dificultad de construcción entre lenguajes.

Cada lenguaje intenta abordar estos problemas de forma diferente, ya sea restringiendo al usuario o siendo lo más flexible posible. El equipo de Go optó por abordarlos recurriendo a la ingeniería moderna: eliminando la limitación de la memoria y simplificando la ejecución de fragmentos de código en paralelo.

Go fue diseñado para la concurrencia y los servidores en red, lo que explica su rápida adopción en empresas de software de todos los tamaños en los últimos años. Go se diseñó para mejorar la productividad en una época en la que las máquinas en red multinúcleo y las grandes bases de código se estaban convirtiendo en la norma.

> Go comenzó en septiembre de 2007 cuando Robert Griesemer, Ken Thompson y yo comenzamos a discutir un nuevo lenguaje para abordar los desafíos de ingeniería que nosotros y nuestros colegas de Google enfrentábamos en nuestro trabajo diario.
> —Rob Pike, coautor de Go

Go fue diseñado teniendo en mente la simplicidad y la productividad, lo que lo convierte en una opción natural para servicios de backend, API y necesidades modernas de computación en la nube.

Recursos de aprendizaje

Se irán ampliando según se vayan encontrando.

Lo que sigue es una lista de recursos donde puedes aprender Go.

- Go by example.
- Efective Go.

Organización del código

El código fuente de Go se organiza principalmente en **paquetes** y **módulos**.

Paquetes

Los programas Go se organizan en **paquetes**. Un paquete es una **colección de archivos fuente** en el **mismo directorio** que se compilan juntos. Las funciones, tipos, variables y constantes definidas en un archivo fuente son visibles para todos los demás archivos fuente dentro del mismo paquete.

> La convención de Go es nombrar el paquete con el mismo nombre del directorio que lo contiene.

Módulos

Un **módulo** es una **colección** de **paquetes** de Go relacionados que se publican juntos. Un repositorio de Go generalmente contiene un solo módulo, ubicado en la raíz del repositorio. Un archivo llamado `go.mod` declara la **ruta del módulo**. Esta ruta del módulo es el **prefijo** que necesitamos escribir para importar cualquier paquete de ese módulo. El módulo contiene los paquetes del directorio que contiene su archivo `go.mod`, así como los subdirectorios de ese directorio, hasta el siguiente subdirectorio que contiene otro archivo `go.mod` (si lo hay).

Esto se verá más claramente cuando creemos nuestro primer módulo e importemos paquetes de ese módulo.

```
mkdir hello
cd hello
go mod init example/hello
```

En el bloque anterior, creamos un directorio para organizar el código de nuestro módulo. El nombre del módulo será `example/hello`.

Dentro del directorio `hello` escribiríamos el código de nuestros paquetes. Normalmente, crearemos un directorio para cada paquete.

Ejecución del código

Si queremos ejecutar nuestro código, necesitamos un **punto de entrada**. El punto de entrada para una aplicación Go será la función `main` del paquete `main`:

```
package "main"

import "fmt"

func main() {
  // Our code will be here.
  fmt.Print("Hello world!")
}
```

El paquete `main` es un paquete especial. Define un programa ejecutable independiente, no una biblioteca. La ejecución del programa comienza llamando a la función `main` del paquete `main`. **No seguimos la convención de nombres que establece que el nombre del paquete `main` coincida con el del directorio que lo contiene.**

Una vez que tenemos un archivo con el paquete `main` y una función `main` tenemos tres opciones para ejecutar nuestro código:

1. Usar el comando `go run` con el nombre del archivo que contiene la función `main`: `go run main.go`.
2. Usar el comando `go build` para crear un archivo ejecutable y luego ejecutarlo: `go build .` creará un archivo ejecutable con el nombre del módulo (en este caso, `hello`). Luego, podemos ejecutarlo con `./hello.exe`.
3. Usar el comando `go install` para crear un archivo ejecutable en el directorio `$GOPATH/bin` y luego ejecutarlo. En nuestro caso, la ruta será `C:/Users/<User>/go/bin/hello.exe`.

```
go run main.go
```

Visibilidad

Todo lo que empieza con mayúscula queda **expuesto** y puede usarse fuera del paquete donde está definido. Todo lo que empieza con minúscula queda **no expuesto** y solo puede usarse dentro del paquete donde está definido.

En cualquier caso, esto solo importa cuando usamos varios paquetes. En un solo paquete, todo es visible.

Sintaxis básica

La sintaxis de Go es muy sencilla y similar a la de otros lenguajes de programación como C, Java o JavaScript. Algunas reglas básicas son:

- El `;` al final de la línea es opcional. El compilador de Go inserta automáticamente punto y coma al final de las líneas.
- Los bloques de código se definen con `{}`.
- Las tabulaciones se utilizan para la sangría. No se permiten espacios para la sangría.

Variables

Go es un lenguaje de tipado estático, lo que significa que el tipo de una variable se conoce en tiempo de compilación y no se puede modificar. Podemos declarar una variable usando la palabra clave `var` seguida de su nombre y tipo:

```
var name string
name = "Manuel"
```

O podemos declarar e inicializar una variable en la misma línea dejando que el compilador infiera el tipo:

```
name := "Manuel"
```

"Inferido por el compilador" significa que el compilador determinará el tipo de la variable en función del valor que le asignemos.

Go permite múltiples declaraciones de variables en la misma línea:

```
var x, y, z int = 1, 2, 3
```

O mediante inferencia:

```
x, y, z := 1, 2, 3
```

Tipos de datos

Tipos básicos

Tiene tres tipos básicos diferentes: booleano, numérico y cadena.

Los tipos de datos booleanos pueden ser «verdaderos» o «falsos»:

```
var bigger bool

bigger = 3 > 2 // true
```

Los tipos numéricos se dividen además en tipos enteros y de punto flotante:

- `int`: tipo entero.
- `float32` and `float64`: para números reales en coma flotante de simple y doble precisión, respectivamente.

```
var i int

i = 10

var f float64

f = 3.14
```

Las cadenas haren referencia a una secuencia de caracteres Unicode:

```
var str string

str = "Hello, World!"
```

En Go normalmente no tenemos que especificar el tipo de una variable, el compilador lo infiere del valor que asignamos a la variable:

```
bigger := 3 > 2 // bool

x := 5 + 5 // int

pi := 3.14 // float
```

constantes

Podemos crear constantes usando la palabra clave `const`. Las constantes son valores que no cambian durante la ejecución del programa.

```
const Pi = 3.14
```

Data structures

No es nuestra intención hacer un repaso exhaustivo de las estructuras de datos en Go. Solo mencionaremos las más comunes y veremos más detalles cuando las necesitemos.

Arrays

Los arrays son colecciones ordenadas de elementos del mismo tipo. El tamaño de un array es fijo y se define en el momento de su creación.

```
var a [5]int // Array of 5 integers

a[0] = 1
```

También podemos declarar e inicializar un array en la misma línea:

```
var a = [5]int{1, 2, 3, 4, 5}
```

O haciendo que el compilador infiera el tamaño:

```
var a = [...]int{1, 2, 3, 4, 5}
```

Slices

Los slices son colecciones dinámicas de elementos del mismo tipo. A diferencia de los arrays, los slices pueden crecer y reducirse en tamaño.

```
var s []int // Slice of integers

s = append(s, 1, 2, 3) // Add an element to the slice
```

Objetos en Go

Go no tiene objetos en el sentido tradicional. En cambio, tiene estructuras e interfaces y, mediante la declaración de tipos logra un comportamiento similar.

Veremos estos conceptos con más detalle más adelante, según los necesitemos.

Control de flujo

Condicionales

Go solo tiene dos tipos de declaraciones condicionales: `if` y `switch`.

`if`

Las sentencias condicionales en Go son similares a las de otros lenguajes de programación. Las sentencias condicionales principales son `if`, `else if` y `else`.

```
// Code before...

x = 10

if x > 0 {
    fmt.Println("x is positive")
} else if x < 0 {
    fmt.Println("x is negative")
} else {
    fmt.Println("x is zero")
}

// Code after...
```

`switch`

La segunda declaración condicional es `switch`, que se utiliza para seleccionar uno de los muchos bloques de código que se ejecutarán.

```
import (
    "fmt"
    "runtime"
)
// Code before...

l := "linux"

switch os := runtime.GOOS; os { // We can initialize a variable in the switch statement.
case "darwin":
    fmt.Println("OS X.")
    // break is implicit.
case l: // We can use variables as case values.
    fmt.Println("Linux.")
    // break is implicit.
```

```
default:
    // freebsd, openbsd,
    // plan9, windows...
    fmt.Printf("%s.\n", os)
}

// Code after...
```

Bucles

Go solo tiene una construcción de bucle: el bucle `for`. Este bucle `for` puede usarse de varias maneras.

C-style

Este bucle for funciona de forma similar al bucle for de Java. Tiene una sentencia de inicialización, una condición y una sentencia post.

- Sentencia de inicialización: se ejecuta antes de la primera iteración `i := 0`.
- Condición: se evalúa antes de cada iteración, `i < 10`, para determinar si el bucle debe continuar.
- Sentencia post: se ejecuta al final de cada iteración `++i`.

```
for i := 0; i < 10; ++i {
    fmt.Println(i)
}
```

While-style

Este bucle funciona como un bucle while en otros lenguajes. Solo tenemos una condición que se evalúa antes de cada iteración. El bucle continuará mientras la condición sea verdadera.

```
condition := true
i := 0

for condition {
    fmt.Println(i)
    condition = i < 10
    ++i
}
```

Iterar sobre una colección

Esta última versión del bucle for es similar al bucle **for-each** de Java y otros lenguajes. Podemos usarlo para iterar sobre arrays, segmentos, mapas, cadenas y canales. (Veremos estas estructuras de datos más adelante).

```go
// We declare a collection (a map) with pairs of key-value.
m := map[int]string{
    1: "one",
    2: "two",
    3: "three",
}

for key, value := range collection {
    fmt.Println("key:", key, "=>", "value: ", value)
}
```

Funciones

En Go, las funciones son **ciudadanas de primera clase**. Esto significa que pueden asignarse a variables, pasarse como argumentos a otras funciones y devolverse desde otras funciones.

Para definir una función en Go, usamos la palabra clave `func` seguida del nombre de la función, los parámetros y el tipo de retorno.

```go
func addition(a int, b int) int {
    return a + b
}

// We can call the function like this:
result := addition(3, 4)

fmt.Printf("The result is: %d", result)
```

También podemos asignar funciones a variables (y pasarlas como argumentos a otras funciones):

```go
// We can assign a function to a variable.
add := addition

// And we can call the function using the variable.
result := add(3, 4)

fmt.Printf("The result is: %d", result)
```

Una función en Go puede devolver múltiples valores:

```go
func swap(a, b string) (string, string) {
    return b, a
}
a, b := "world", "hello"
fmt.Println(a, b) // world hello

a, b := swap(a, b)

fmt.Println(a, b) // hello world
```

Para controlar los errores, Go utiliza un patrón común donde las funciones devuelven un valor y un error. Si la función se ejecuta correctamente, el error será `nil`. Si hay un error, este contendrá un mensaje de error.

```
value, err := strconv.ParseBool("true")

if err != nil {
    fmt.Println("Error:", err)
} else {
    fmt.Println("Value:", value)
}
```

El ejemplo anterior muestra cómo convertir una cadena a un booleano. La función `ParseBool` devuelve dos valores: el valor booleano analizado y un valor de error. Si la conversión se realiza correctamente, el valor de error será `nil`. Si la conversión falla, el valor de error contendrá un mensaje de error.

Si queremos ignorar uno de los valores devueltos, podemos utilizar el identificador en blanco `_`:

```
value, _ := strconv.ParseBool("true")
```

Argumentos por valor y por referencia

En Go, los argumentos se pasan por valor. Esto significa que, al pasar una variable a una función, pasamos una **copia de la variable**, y no la variable original.

```
// previos code...

a := 10

func modify(x int) {
    x = 20
}

modify(a)

fmt.Println(a) // 10
```

Si necesitamos modificar la variable original, podemos pasarle un **puntero**. Un puntero es una referencia a la ubicación de memoria donde se almacena la variable. Podemos crear un puntero usando el operador `&` y desreferenciarlo usando el operador `*`.

```go
// previous code...

a := 10

func modify(x *int) {
    *x = 20
}

modify(&a)

fmt.Println(a) // 20
```

Con esto ya tenemos información suficiente para empezar a programar en Go. A medida que avancemos, iremos viendo más detalles y características del lenguaje.