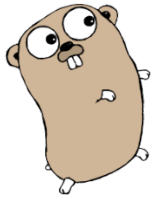
Introducción a Go



Motivación

- Go es un lenguaje nuevo:
  - Propósito general
  - Sintaxis concisa
  - Sistema de tipos expresivo
  - Concurrencia
  - Recolector de basura
  - Compilación rápida
  - Ejecución eficiente
- Objetivo:
  - Combinar ambos mundos:
    - Lenguajes compilados:
      - Tipos estáticos
      - Seguridad
      - Rendimiento
    - Lenguajes interpretados:
      - Tipos dinámicos
      - Expresividad
      - Conveniencia
  - Útil en programación de sistemas modernos a gran escala

Rápido, divertido y productivo

Creadores (Google)

Bell Labs

- Ken Thompson
  - Unix (con Dennis Ritchie)
  - B y C
  - “Reflections on trusting trust” (premio Turing)
  - UTF-8 (con Rob Pike)
- **Rob Pike**
  - Blit (X Windows…)
  - Plan9, Inferno
  - Limbo
  - UTF-8 (con Ken Thompson)

ETH Zurich

- Robert Griesemer
  - V8 (JavaScript)
  - Sawzall
  - Java HotSpot VM
  - Doctorando de Niklaus Wirth (Pascal)

Contexto

- Influencias:
  - C del siglo XXI
  - familia de C++, Java, C#
  - Pascal, Modula, Oberon (declaraciones, paquetes)
  - Limbo, Newsqueak, CSP (concurrencia)
  - Python, Ruby (características dinámicas)
- Elementos destacables:
  - Énfasis en la simplicidad
  - Memoria gestionada
  - Sintaxis ligera
  - Compilación rápida
  - Elevado rendimiento
  - Soporta concurrencia
  - Tipos estáticos
  - Librería estándar consistente
  - Facilidad de instalación
  - Autodocumentado (y bien documentado)
  - Código abierto (BSD)

Cosas que “faltan”

- sobrecarga de funciones y operadores
- conversiones implícitas
- clases o herencia de tipos
- carga dinámica de código
- librerías dinámicas
- tipos variantes (variant)
- tipos genéricos (templates)
- excepciones

La herramienta go

- > go *comando*

- *bug*
  enviar parte de error
- ***build***
  compilar paquetes y dependencias
- *clean*
  eliminar objetos y cache
- *doc*
  mostrar documentación
- ***env***
  mostrar variables de entorno de go
- *fix*
  actualizar paquetes a nuevas APIs
- *fmt*
  formato automático de fuentes
- *generate*
  generar código automáticamente
- ***get***
  obtener dependencias e instalarlas
- ***install***
  compilar e instalar paquetes y dependencias
- *list*
  mostrar paquetes o módulos
- ***mod***
  gestión de módulos
- ***run***
  compilar y ejecutar
- *test*
  realizar tests
- *tool*
  ejecutar herramientas específicas
- ***version***
  mostrar la versión actual de Go
- *vet*
  comprobación de errores (linting)

Web oficial

Web oficial: go.dev

Secciones a destacar:

- Descargas de Go.
- Inicio rápido.
- Documentación.
- Librería estándar.
- Buscador de paquetes.
- Blog.

Nociones iniciales

• • •

(ejemplos de gobyexample.com)

Ficheros e identificadores

- Fuentes de Go:
  - Se guardan en ficheros .go
  - Nombre en minúsculas
    - scanner.go
  - Se separan con subrayado
    - scanner_test.go
  - Existe un formato estándar. Se aplica automáticamente en muchos editores.
- Identificadores:
  - Empiezan por letra (UTF-8) o subrayado (similar a C)
  - El subrayado ‘ _ ’ es un identificador especial que se descarta

Paquetes

- estructuran el código
- cada *.go* pertenece a un paquete
- un paquete puede estar formado por muchos *.go*
- todo ejecutable ha de tener paquete (y función) **main**
- el nombre es en minúsculas
- la librería estándar contiene muchos paquetes y se puede crear paquetes propios
- se importan mediante *import* y cada paquete se compila una única vez
- la visibilidad viene dada por la primera letra del identificador:
  - Mayúscula -> público
  - minúscula -> privado

Funciones

- se declaran con *func*
- *main* no recibe parámetros ni devuelve nada
- siguen el formato:

```
func función(lparam) (ldevol){
    …
}

donde lparam es (param1 tipo1,…)
y ldevol es (dev1 tipo1,…)
```

- permiten varias variables de retorno
- Es obligatorio el ‘ { ’ en la misma línea que *func*
- El ‘ } ’ se pone en una línea suelta al final del código

Hola mundo

Este es el clásico hola mundo.

Suponiendo que el código esté en el fichero *"hello-world.go"*, lo podemos ejecutar con *go run*.

Podemos obtener un ejecutable con *go build,* y ejecutar directamente ese binario.

```go
package main


import "fmt"

func main() {
    fmt.Println("hello world")
}
```

```
$ go run hello-world.go
hello world

$ go build hello-world.go
$ ls
hello-world    hello-world.go

$ ./hello-world
hello world
```

Valores

Algunos ejemplos de los distintos tipos de valores que tiene Go, entre otros: cadenas, enteros, flotantes, booleanos, etc.

```go
package main

import "fmt"

func main() {

    fmt.Println("go" + "lang")

    fmt.Println("1+1 =", 1+1)
    fmt.Println("7.0/3.0 =", 7.0/3.0)

    fmt.Println(true && false)
    fmt.Println(true || false)
    fmt.Println(!true)
}
```

```
$ go run values.go
golang
1+1 = 2
7.0/3.0 = 2.3333333333333335
false
true
false
```

Variables

En Go, las variables se declaran de forma explícita y hay comprobación de tipos por el compilador.

Se puede declara una o más variables con *var*, e incluso varias a la vez.

Go es capaz de inferir el tipo de las variables que inicialicen. Cuando no están inicializadas se inicializan automáticamente al valor cero para ese tipo.

La sintaxis := permite declarar e inicializar una variable de forma rápida (tiene algunas contrapartidas)

```go
package main

import "fmt"

func main() {

    var a = "initial"
    fmt.Println(a)

    var b, c int = 1, 2
    fmt.Println(b, c)

    var d = true
    fmt.Println(d)

    var e int
    fmt.Println(e)



    f := "apple"
    fmt.Println(f)
}
```

```
$ go run variables.go
initial
1 2
true
0
apple
```

Constantes

Go soporta constantes de carácter, cadenas, booleanos y valores numéricos que se declaran mediante *const*.

Pueden aparecer en los mismos sitios que las declaraciones con *var*.

Permiten realizar operaciones aritméticas con precisión arbitraria.

Una constante numérica adquiere el tipo cuando se utiliza o se fuerza un tipo.

```go
package main

import (
    "fmt"
    "math"
)

const s string = "constant"

func main() {
    fmt.Println(s)

    const n = 500000000


    const d = 3e20 / n
    fmt.Println(d)

    fmt.Println(int64(d))


    fmt.Println(math.Sin(n))
}
```

```
$ go run constant.go
constant
6e+11
600000000000
-0.28470407323754404
```

Bucle for

El único bucle en Go es *for*, y permite expresar otros tipos de bucles comunes:

- Con condición estilo *while*
- Tradicional con contador
- Infinito sin condición (uso de *break* para salir)
- También se puede utilizar *continue* para saltar a la siguiente iteración.

```go
package main

import "fmt"

func main() {

    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    for {
        fmt.Println("loop")
        break
    }

    for n := 0; n <= 5; n++ {
        if n%2 == 0 {
            continue
        }
        fmt.Println(n)
    }
}
```

```
$ go run for.go
1
2
3
7
8
9
loop
1
3
5
```

If/else

Su uso en Go es similar a otros lenguajes.

Se puede tener un *if* sin *else*.

No son necesarios los paréntesis para las condiciones.

Se puede tener una sentencia antes de las condiciones. Todas las variables declaradas están disponibles en todas las ramas.

No existe el operador ternario *(c ? a : b)*, por lo que siempre es necesario emplear un *if.*

```go
package main

import "fmt"

func main() {

    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }

    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }

    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}
```

```
$ go run if-else.go
7 is odd
8 is divisible by 4
9 has 1 digit
```

Switch

En Go, *switch* permite expresiones de muchos tipos y no sólo enteros o constantes.

Se pueden separar expresiones con comas y utilizar *default* para indicar todos los demás casos.

No se utiliza *break*, se termina cada *case* de forma automática.

```go
package main

import (
    "fmt"
    "time"
)

func main() {

    i := 2
    fmt.Print("Write ", i, " as ")
    switch i {
    case 1:
        fmt.Println("one")
    case 2:
        fmt.Println("two")
    case 3:
        fmt.Println("three")
    }

    switch time.Now().Weekday() {
    case time.Saturday, time.Sunday:
        fmt.Println("It's the weekend")
    default:
        fmt.Println("It's a weekday")
    }

    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("It's before noon")
    default:
        fmt.Println("It's after noon")
    }
```

Switch (II)

Existe un *switch* de tipos que permite comparar tipos en lugar de valores.

Se puede utilizar para obtener el tipo de una variable de tipo *interface{}*.

```
    whatAmI := func(i interface{}) {
        switch t := i.(type) {
        case bool:
            fmt.Println("I'm a bool")
        case int:
            fmt.Println("I'm an int")
        default:
            fmt.Printf("Don't know type %T\n", t)
        }
    }
    whatAmI(true)
    whatAmI(1)
    whatAmI("hey")
}
```

```
$ go run switch.go
Write 2 as two
It's a weekday
It's after noon
I'm a bool
I'm an int
Don't know type string
```

Arrays

En Go, un *array* es una secuencia de valores del mismo tipo de longitud determinada. El tamaño es parte del tipo.

Existe la función propia *len()* que nos devuelve la longitud del *array*.

Siempre se pasan por valor (no copia).

Es mucho más común ver *slices* que *arrays* en Go.

```go
package main

import "fmt"

func main() {

    var a [5]int
    fmt.Println("emp:", a)

    a[4] = 100
    fmt.Println("set:", a)
    fmt.Println("get:", a[4])

    fmt.Println("len:", len(a))

    b := [5]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)

    var twoD [2][3]int
    for i := 0; i < 2; i++ {
        for j := 0; j < 3; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

```
$ go run arrays.go
emp: [0 0 0 0 0]
set: [0 0 0 0 100]
get: 100
len: 5
dcl: [1 2 3 4 5]
2d:  [[0 1 2] [1 2 3]]
```

Slices

Los *slices* son clave en Go, permiten mayor funcionalidad que los *arrays* tradicionales.

El tipo de un *slice* es únicamente el de los elementos que contiene (no la longitud). Se utilizan como *arrays*.

Para crearlos se puede utilizar la función propia *make()*. También se puede utilizar *len()* para obtener su longitud.

Además soportan otra funcionalidad como *append()* que permite añadir elementos devolviendo un nuevo *slice*.

```go
package main

import "fmt"

func main() {

    s := make([]string, 3)
    fmt.Println("emp:", s)

    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println("set:", s)
    fmt.Println("get:", s[2])

    fmt.Println("len:", len(s))

    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)
```

```
$ go run slices.go
emp: [  ]
set: [a b c]
get: c
len: 3
apd: [a b c d e f]
cpy: [a b c d e f]
sl1: [c d e]
sl2: [a b c d e]
sl3: [c d e f]
dcl: [g h i]
2d:  [[0] [1 2] [2 3 4]]
```

Slices (II)

Los *slices* también se pueden copiar con la función propia *copy()*.

Soportan el operador de *reslicing*, con el formato *slice[bajo:alto]*. Así, *s[2:5]* devuelve un nuevo *slice* con los elementos *s[2]*, *s[3]* y *s[4]* (pero no *s[5]*). Se puede omitir uno de los extremos del *reslice*.

Se pueden construir *slices* multidimensionales. Es necesario llamar a *make()* para cada dimensión.

```
    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cpy:", c)

    l := s[2:5]
    fmt.Println("sl1:", l)

    l = s[:5]
    fmt.Println("sl2:", l)

    l = s[2:]
    fmt.Println("sl3:", l)

    t := []string{"g", "h", "i"}
    fmt.Println("dcl:", t)

    twoD := make([][]int, 3)
    for i := 0; i < 3; i++ {
        innerLen := i + 1
        twoD[i] = make([]int, innerLen)
        for j := 0; j < innerLen; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

```
$ go run slices.go
emp: [  ]
set: [a b c]
get: c
len: 3
apd: [a b c d e f]
cpy: [a b c d e f]
sl1: [c d e]
sl2: [a b c d e]
sl3: [c d e f]
dcl: [g h i]
2d:  [[0] [1 2] [2 3 4]]
```

Mapas

Son el tipo asociativo (dicts, hash-maps…) en Go. Para crear un map vacío, se utilizar make():

make(map[tipo-clave]tipo-val)

Se asignan y obtienen valores como en un array o slice.

Se puede comprobar la existencia de un elemento con el segundo valor de retorno booleano (se utiliza _ para ignorar el valor que se devuelve primero).

```go
package main

import "fmt"

func main() {

    m := make(map[string]int)


    m["k1"] = 7
    m["k2"] = 13

    fmt.Println("map:", m)


    v1 := m["k1"]
    fmt.Println("v1: ", v1)

    fmt.Println("len:", len(m))


    delete(m, "k2")
    fmt.Println("map:", m)

    _, prs := m["k2"]
    fmt.Println("prs:", prs)



    n := map[string]int{"foo": 1, "bar": 2}
    fmt.Println("map:", n)
}
```

```
$ go run maps.go
map: map[k1:7 k2:13]
v1:  7
len: 2
map: map[k1:7]
prs: false
map: map[bar:2 foo:1]
```

Range

Este tipo de bucle permite iterar sobre los valores de algunas estructuras de datos como arrays, slices, maps y cadenas.

En arrays y slices devuelve índice, valor.

En mapas devuelve pares clave, valor. **No** se debe presuponer un orden concreto.

En cadenas itera sobre las distintas runas (caracteres Unicode)

```go
package main

import "fmt"

func main() {

    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)

    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }

    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }

    for k := range kvs {
        fmt.Println("key:", k)
    }

    for i, c := range "go" {
        fmt.Println(i, c)
    }
}
```

```
$ go run range.go
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b
0 103
1 111
```

Funciones

Las funciones en Go son muy similares a otros lenguajes. Se requiere *returns* explícitos.

Soportan múltiples valores de retorno y cierres (lambdas o funciones anónimas).

Las funciones son *ciudadanos de primera clase* en Go, y pueden ser variables, parámetros y tipos de devolución en sí.

```go
package main

import "fmt"

func plus(a int, b int) int {


    return a + b
}

func plusPlus(a, b, c int) int {
    return a + b + c
}



func main() {

    res := plus(1, 2)
    fmt.Println("1+2 =", res)

    res = plusPlus(1, 2, 3)
    fmt.Println("1+2+3 =", res)
}
```

```
$ go run functions.go
1+2 = 3
1+2+3 = 6
```

Funciones (II)

El uso de múltiples variables de retorno es idiomático en Go y se utiliza en la librería estándar.

Se puede ignorar alguna variable de retorno mediante _ .

```go
package main

import "fmt"

func vals() (int, int) {
    return 3, 7
}

func main() {

    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)

    _, c := vals()
    fmt.Println(c)
}
```

```
$ go run multiple-return-values.go
3
7
7
```

Funciones (III)

También se soportan cierres. Son útiles para definir una función en línea sin tener que nombrarla para llamadas, devoluciones o valores de tipo función.

```go
package main

import "fmt"

func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}

func main() {

    nextInt := intSeq()



    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    newInts := intSeq()
    fmt.Println(newInts())
}
```

```
$ go run closures.go
1
2
3
1
```

Cadenas

Una cadena en Go es un slice de bytes inmutable (solo lectura).

El lenguaje y la librería estándar tratan las cadenas como contenedores de texto codificado en UTF-8.

En otros lenguajes, las cadenas contienen caracteres; en Go, las cadenas contienen runas que son enteros (a veces multibyte) que representan un code point en Unicode.

```go
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {

    const s = "สวัสดี"



    fmt.Println("Len:", len(s))


    for i := 0; i < len(s); i++ {
        fmt.Printf("%x ", s[i])
    }
    fmt.Println()

    fmt.Println("Rune count:", utf8.RuneCountInString(s))
}
```

```
$ go run strings-and-runes.go
Len: 18
e0 b8 aa e0 b8 a7 e0 b8 b1 e0 b8 aa e0 b8 94 e0 b8 b5
Rune count: 6
```

Structs

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func newPerson(name string) *person {


    p := person{name: name}
    p.age = 42
    return &p
}
```

```go
func main() {

    fmt.Println(person{"Bob", 20})

    fmt.Println(person{name: "Alice", age: 30})

    fmt.Println(person{name: "Fred"})

    fmt.Println(&person{name: "Ann", age: 40})

    fmt.Println(newPerson("Jon"))


    s := person{name: "Sean", age: 50}
    fmt.Println(s.name)

    sp := &s
    fmt.Println(sp.age)


    sp.age = 51
    fmt.Println(sp.age)
}
```

Son similares a otros lenguajes:

- Pueden tener métodos.
- Se puede devolver un puntero a una variable *struct* local.
- Se accede a los campos con punto, incluso si son punteros.

```
$ go run structs.go
{Bob 20}
{Alice 30}
{Fred 0}
&{Ann 40}
&{Jon 42}
Sean
50
51
```

Métodos

Se pueden definir métodos sobre *structs*.

El método se asocia a un *struct* mediante un *receptor* que va antes del nombre de la función.

El *receptor* puede ser por puntero o por valor. Go se encarga de hacer la conversión de forma automática, pero puede ser interesante hacer receptores por puntero para evitar que se copien todos los elementos o para que las modificaciones persistan.

Se pueden definir colecciones de métodos en *interfaces*.

```go
package main

import "fmt"

type rect struct {
    width, height int
}

func (r *rect) area() int {
    return r.width * r.height
}

func (r rect) perim() int {
    return 2*r.width + 2*r.height
}

func main() {
    r := rect{width: 10, height: 5}

    fmt.Println("area: ", r.area())
    fmt.Println("perim:", r.perim())

    rp := &r
    fmt.Println("area: ", rp.area())
    fmt.Println("perim:", rp.perim())
}
```

```
$ go run methods.go
area:  50
perim: 30
area:  50
perim: 30
```

Aprendiendo Go

Existe mucha más información tanto en los tutoriales de la web de Go como en gobyexample. Entre otros, sería muy recomendable investigar:

- La librería estándar de Go.

- Otros ejemplos de gobyexample
  - Interfaces
  - Errores
  - Gorutinas y paralelismo
  - Gestión de tiempo
  - Ficheros y redes
  - Encoding, JSON, compresión, cifrado
  - Etc.