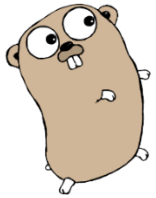
Introducción a Go



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:
  - 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)

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
- tipos genéricos (templates)
- excepciones

La herramienta go

- > go *comando*
  - build
    - compilar
  - clean
    - limpiar objetos
  - doc
    - llamar a ***godoc***
  - env
    - muestra el entorno de Go
  - fix
    - repara código fuente
  - fmt
    - formatea código fuente
  - get
    - obtiene paquetes (git, hg)
  - install
    - compilar e instalar paquetes
  - list
    - mostrar paquetes disponibles
  - run
    - compilar y lanzar programa
  - test
    - lanzar tests
  - tool
    - ejecutar herramientas extra
  - version
    - mostrar la versión actual
  - vet
    - comprobar código

Construcciones Básicas

parte I

Ficheros e identificadores

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

Palabras clave

```
break
case
chan
const
continue
default
defer
else
fallthrough
for
func
go
goto
if
interface
import
map
package
range
return
select
struct
switch
type
var
```

Identificadores reservados

```
append
bool
byte
cap
close
complex
complex64
complex128
copy
false
float32
float64
imag
int
int8
int16
int32
int64
iota
len
make
new
nil
panic
print
println
real
recover
string
true
uint
uint8
uint16
uint32
uint64
uintptr
```

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

Plantilla de programa

```
package main

import (
    “fmt”
)

const c = “C”

var v int = 5

type T struct{}

func init() { // initialization of package
}

func main() {
        var a int
        Func1()
        // ...
        fmt.Println(a)
}

func (t T) Method1() {
        //...
}

func Func1() { // exported function Func1
        //...
}
```

Asignación

Dos operadores de asignación:

= asignación normal

:= asignación con declaración corta

```
package main
import (
    “fmt”
    “os”
)

func main() {
    var goos string = os.Getenv(“GOOS”)
    fmt.Printf(“The operating system is: %s\n”, goos)
    path := os.Getenv(“PATH”)
    fmt.Printf(“Path is %s\n”, path)
}
```

Casting

El casting es obligatorio o el compilador emitirá un error

```
package main
import “fmt”

func main() {
        var n int16 = 34
        var m int32

// compiler error: cannot use n (type int16) as type int32 in assignment
        //m = n
        m = int32(n)

        fmt.Printf(“32 bit int is:  %d\n”, m)
        fmt.Printf(“16 bit int is:  %d\n”, n)
}
// the output is:
32 bit int is:  34
16 bit int is:  34
```

Cadenas

El soporte de cadenas es parecido a C++ o Java. Se soporta Unicode directamente.

Las cadenas son inmutables

```
package main
import (
“fmt”
“strings”
)
func main() {
var orig string = “Hey, how are you George?”
var lower string
var upper string
fmt.Printf(“The original string is: %s\n”, orig)
lower = strings.ToLower(orig)
fmt.Printf(“The lowercase string is: %s\n”, lower)
upper = strings.ToUpper(orig)
fmt.Printf(“The uppercase string is: %s\n”, upper)
}
```

Tiempo y fechas

El soporte de tiempo y fechas es excelente

```
package main
import (
        “fmt”
        “time”
)
```

```
var week time.Duration

func main() {
        t := time.Now()
        fmt.Println(t)     // e.g. Wed Dec 21 09:52:14 +0100 RST 2011
        fmt.Printf(“%02d.%02d.%4d\n”, t.Day(), t.Month(), t.Year())
        // 21.12.2011
        t = time.Now().UTC()
        fmt.Println(t)     // Wed Dec 21 08:52:14 +0000 UTC 2011
        fmt.Println(time.Now()) // Wed Dec 21 09:52:14 +0100 RST 2011
  // calculating times:
        week = 60 * 60 * 24 * 7 * 1e9   // must be in nanosec
        week_from_now := t.Add(week)
        fmt.Println(week_from_now)   // Wed Dec 28 08:52:14 +0000 UTC 2011
// formatting times:
        fmt.Println(t.Format(time.RFC822))     // 21 Dec 11 0852 UTC
        fmt.Println(t.Format(time.ANSIC))      // Wed Dec 21 08:56:34 2011
        fmt.Println(t.Format(“02 Jan 2006 15:04”)) // 21 Dec 2011 08:52
        s := t.Format(“20060102”)
        fmt.Println(t, “=>”, s)
                // Wed Dec 21 08:52:14 +0000 UTC 2011 => 20111221
}
```

Punteros

Similares a C o C++ pero sin aritmética de punteros (seguros)

```
package main
import “fmt”

func main() {
        var i1 = 5
        fmt.Printf(“An integer: %d, its location in memory: %p\n”, i1, &i1)

        var intP *int
        intP = &i1
        fmt.Printf(“The value at memory location %p is %d\n”, intP, *intP)
}
```

```
package main

func main() {
        var p *int = nil
        *p = 0

}
// in Windows: stops only with: <exit code=”-1073741819” msg=“process crashed”/>
// runtime error: invalid memory address or nil pointer dereference
```

Estructuras de control

parte II

if / else

```
package main

import “fmt”

func main() {
    bool1 := true
    if bool1 {
        fmt.Printf(“The value is true\n”)
    } else {
        fmt.Printf(“The value is false\n”)
    }
}
   // Output: The value is true
```

```
package main
import “fmt”
func main() {
    var first int = 10
    var cond int
            if first <= 0 {
               fmt.Printf(“first is less than or equal to 0\n”)
            } else if first > 0 && first < 5 {
               fmt.Printf(“first is between 0 and 5\n”)
            } else {
               fmt.Printf(“first is 5 or greater\n”)
            }

            if cond = 5; cond > 10 {
               fmt.Printf(“cond is greater than 10\n”)
```

errores

```
package main
import (
         “fmt”
         “strconv”
)

func main() {
         var orig string = “ABC”
         var an int
         var err error
         an, err = strconv.Atoi(orig)
         if err != nil {
            fmt.Printf(“orig %s is not an integer - exiting with error\n”, orig)
            return
         }
         fmt.Printf(“The integer is %d\n”, an)
// rest of the code
}
```

IDIOM

```
if err := file.Chmod(0664); err !=nil {
    fmt.Println(err)
    return err
}
```

switch

Soporta cualquier tipo que se pueda comparar: cadenas, números, punteros…

```
package main
import "fmt"

func main() {
        var num1 int = 100

        switch num1 {
        case 98, 99:
                fmt.Println("It's equal to 98")
        case 100:
                fmt.Println("It's equal to 100")
        default:
                fmt.Println("It's not equal to 98 or 100")
        }
}
```

```
package main
import "fmt"

func main() {
        var num1 int = 7

        switch {
        case num1 < 0:
                fmt.Println("Number is negative")
        case num1 > 0 && num1 < 10:
                fmt.Println("Number is between 0 and 10")
        default:
                fmt.Println("Number is 10 or greater")
        }
}
```

for

no hay while ni do, sólo for que permite suplantar los anteriores e incluso un for {…} infinito

```
package main
import “fmt”

func main() {
    for i := 0; i < 5; i++ {
        fmt.Printf(“This is the %d iteration\n”, i)
    }
}
```

```
package main
import “fmt”

func main() {
         var i int = 5
         for i >= 0 {
             i = i - 1
             fmt.Printf(“The variable i is now: %d\n”, i)
         }
}
```

range

```
package main
import “fmt”

func main() {
        str := “Go is a beautiful language!”
        for pos, char := range str {
                fmt.Printf(“Character on position %d is: %c \n”, pos, char)
        }
fmt.Println()
        str2 := “Chinese: 日本語”
        for pos, char := range str2 {
                fmt.Printf(“character %c starts at byte position %d\n”, char,
                pos)
        }
        fmt.Println()
        fmt.Println(“index int(rune) rune    char bytes”)
        for index, rune := range str2 {
                fmt.Printf(“%-2d      %d      %U ‘%c’ % X\n”, index, rune, rune,
                rune, []byte(string(rune)))
        }
}
```

Funciones

parte III

múltiples valores de retorno

```
package main
import “fmt”

func main() {
        var i1 int
        var f1 float32
        i1, _, f1 = ThreeValues()
        fmt.Printf(“The int: %d, the float; %f\n”, i1, f1)
}

func ThreeValues() (int, int, float32) {
        return 5, 6, 7.5
}
```

parámetros de entrada y salida

```
package main
import (
        “fmt”
)

// this function changes reply:
func Multiply(a, b int, reply *int) {
        *reply = a * b
}

func main() {
        n := 0
        reply := &n
        Multiply(10, 5, reply)
        fmt.Println(“Multiply:”, *reply) // Multiply: 50
}
```

número variable de parámetros

```
package main
import “fmt”

func main() {
        x := Min(1, 3, 2, 0)
        fmt.Printf(“The minimum is: %d\n”, x)
        arr := []int{7,9,3,5,1}
        x = Min(arr...)
        fmt.Printf(“The minimum in the array arr is: %d”, x)
}

func Min(a ...int) int {
  if len(a)==0 {
    return 0
  }
  min := a[0]
  for _, v := range a {
    if v < min {
      min = v
    }
  }
  return min
}
```

defer

```
package main
import “fmt”

func main() {
    doDBOperations()
}

func connectToDB () {
    fmt.Println( “ok, connected to db” )
}

func disconnectFromDB () {
    fmt.Println( “ok, disconnected from db” )
}

func doDBOperations() {
     connectToDB()
     fmt.Println(“Defering the database disconnect.”)
     defer disconnectFromDB() //function called here with defer
     fmt.Println(“Doing some DB operations ...”)
     fmt.Println(“Oops! some crash or network error ...”)
     fmt.Println(“Returning from function here!”)
     return //terminate the program
     // deferred function executed here just before actually returning, even if
         there is a return or abnormal termination before
}
```

funciones como parámetros

```
package main
import (
	“fmt”
)

func main() {
	callback(1, Add)
}

func Add(a,b int) {
	fmt.Printf(“The sum of %d and %d is: %d\n”, a, b, a + b)
}

func callback(y int, f func(int, int)) {
	f(y, 2)  // this becomes Add(1, 2)
}
```

cierres (lambda)

```
package main
import “fmt”

func main() {
        f()
}

func f() {
        for i := 0; i < 4; i++ {
                g := func(i int) { fmt.Printf(“%d “, i) }
                g(i)
                fmt.Printf(“ - g is of type %T and has value %v\n”, g, g)
        }
}
```

devolviendo funciones

```
package main
import “fmt”

func main() {
        // make an Add2 function, give it a name p2, and call it:
        p2 := Add2()
        fmt.Printf(“Call Add2 for 3 gives: %v\n”, p2(3))
        // make a special Adder function, a gets value 3:
        TwoAdder := Adder(2)
        fmt.Printf(“The result is: %v\n”, TwoAdder(3))
}

func Add32() (func(b int) int) {
        return func(b int) int {
        return b + 2
        }
}
```

```
func Adder(a int) (func(b int) int) {
        return func(b int) int {
                return a + b
        }
}
```

Arrays y slices

parte IV

arrays

Similares a otros lenguajes.

Se pasan siempre por valor y no por referencia

```
package main
import “fmt”

func main() {
        var arr1 [5]int

        for i:=0; i < len(arr1); i++ {
                arr1[i] = i * 2
        }

        for i:=0; i < len(arr1); i++ {
                fmt.Printf(“Array at index %d is %d\n”, i, arr1[i])
        }
}
```

```
package main
import “fmt”
func f(a [3]int)   { fmt.Println(a) }
func fp(a *[3]int) { fmt.Println(a) }

func main() {
         var ar [3]int
         f(ar)   // passes a copy of ar
         fp(&ar) // passes a pointer to ar
}
```

arrays: literales

Se pueden declarar de diversas formas.

[...] -> array

[] -> slice

Es seguro tomar la dirección de un literal para pasarlo a una función (recolector de basura)

```
package main
import "fmt"

func main() {
        var arrAge = [5]int{18, 20, 15, 22, 16}
        var arrLazy = [...]int{5, 6, 7, 8, 22}
        // var arrLazy = []int{5, 6, 7, 8, 22}
        var arrKeyValue = [5]string{3: "Chris", 4: "Ron"}
        //var arrKeyValue = []string{3: "Chris", 4: "Ron"}

        for i := 0; i < len(arrKeyValue); i++ {
                fmt.Printf("Person at %d is %s\n", i, arrKeyValue[i])
        }
}
```

```
package main
import "fmt"

func fp(a *[3]int) { fmt.Println(a) }

func main() {
        for i := 0; i < 3; i++ {
                fp(&[3]int{i, i * i, i * i * i})
        }
}
```

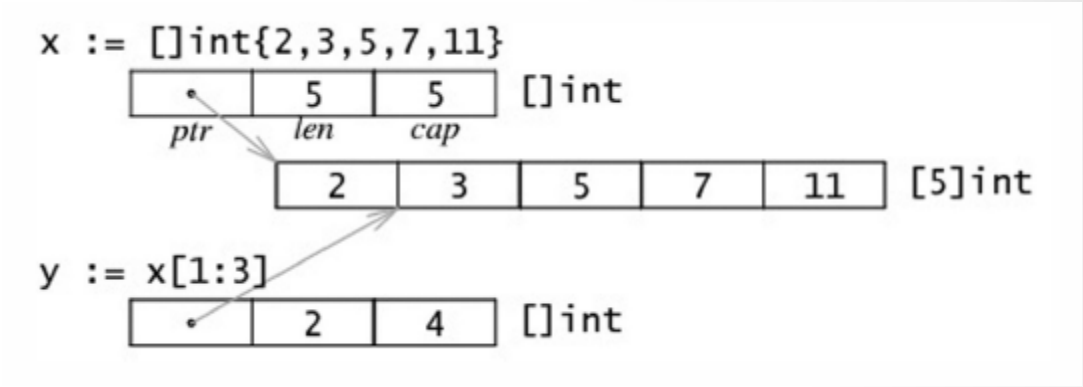
arrays multi-dimensionales

```
package main

const (
        WIDTH =1920
        HEIGHT = 1080
)

type pixel int
var screen [WIDTH][HEIGHT]pixel

func main() {
        for y := 0; y < HEIGHT; y++ {
                for x := 0; x < WIDTH; x++ {
                        screen[x][y] = 0
                }
        }
}
```

Slices (rodajas)

- Slice:
  - referencia a un trozo contiguo de un array
- Siempre se pasan por referencia
- El trozo puede ser el array entero o un subconjunto
- El índice final no está incluído
- Funcionan como arrays
  - indexables
  - len()
- Capacidad (cap())
  - Medida del tamaño máximo que puede tomar el slice
- Varios slices pueden compartir el mismo array subyacente

slices: concepto



slice: uso

```
package main
import “fmt”

func main() {
        var arr1 [6]int
        var slice1 []int = arr1[2:5] // item at index 5 not included!

        // load the array with integers: 0,1,2,3,4,5
        for i := 0; i < len(arr1); i++ {
                arr1[i] = i
        }

        // print the slice:
        for i := 0; i < len(slice1); i++ {
                fmt.Printf(“Slice at %d is %d\n”, i, slice1[i])
        }

        fmt.Printf(“The length of arr1 is %d\n”, len(arr1))
        fmt.Printf(“The length of slice1 is %d\n”, len(slice1))
        fmt.Printf(“The capacity of slice1 is %d\n”, cap(slice1))

// grow the slice:

        slice1 = slice1[0:4]
        for i := 0; i < len(slice1); i++ {
                fmt.Printf(“Slice at %d is %d\n”, i, slice1[i])
        }
        fmt.Printf(“The length of slice1 is %d\n”, len(slice1))
        fmt.Printf(“The capacity of slice1 is %d\n”, cap(slice1))

// grow the slice beyond capacity:
// slice1 = slice1[0:7 ] // panic: runtime error: slice bounds out of range
}
```

slice: make

```
package main
import “fmt”

func main() {
        var slice1 []int = make([]int, 10)
        // load the array/slice:
        for i := 0; i < len(slice1); i++ {
        slice1[i] = 5 * i
        }
        // print the slice:
        for i := 0; i < len(slice1); i++ {
        fmt.Printf(“Slice at %d is %d\n”, i, slice1[i])
        }
        fmt.Printf(“\nThe length of slice1 is %d\n”, len(slice1))
        fmt.Printf(“The capacity of slice1 is %d\n”, cap(slice1))
}
```

for ... range

```
package main
import “fmt”

func main() {
        slice1 := make([]int, 4)

        slice1[0] = 1
        slice1[1] = 2
        slice1[2] = 3
        slice1[3] = 4

        for ix, value := range slice1 {
                fmt.Printf(“Slice at %d is: %d\n”, ix, value)
        }
}
```

```
seasons := []string{“Spring”,“Summer”,“Autumn”,“Winter”}

        for ix, season := range seasons {
                fmt.Printf(“Season %d is: %s\n”, ix, season)
        }

        var season string
        for _, season = range seasons {
                fmt.Printf(“%s\n”, season)
}
```

slice: copy, append

```
package main
import “fmt”

func main() {
        sl_from := []int{1,2,3}
        sl_to := make([]int,10)

        n := copy(sl_to, sl_from)
        fmt.Println(sl_to)  // output: [1 2 3 0 0 0 0 0 0 0]
fmt.Printf(“Copied %d elements\n”, n) // n == 3

        sl3 := []int{1,2,3}
        sl3 = append(sl3, 4, 5, 6)
        fmt.Println(sl3)  // output: [1 2 3 4 5 6]
}
```

Mapas

parte V

map: creación

siempre se usa *make*

```
package main
import “fmt”

func main() {
        var mapLit map[string]int
        //var mapCreated map[string]float32
        var mapAssigned map[string]int

        mapLit = map[string]int{“one”: 1, “two”: 2}
        mapCreated := make(map[string]float32)
        mapAssigned = mapLit

        mapCreated[“key1”] = 4.5
        mapCreated[“key2”] = 3.14159
        mapAssigned[“two”] = 3

        fmt.Printf(“Map literal at \“one\” is: %d\n”, mapLit[“one”])
        fmt.Printf(“Map created at \“key2\” is: %f\n”, mapCreated[“key2”])
        fmt.Printf(“Map assigned at \“two\” is: %d\n”, mapLit[“two”])
        fmt.Printf(“Map literal at \“ten\” is: %d\n”, mapLit[“ten”])
}
```

map: existencia

val, ok := map[key]

```
package main
import “fmt”

func main() {
        var value int
        var isPresent bool

        map1 := make(map[string]int)
        map1[“New Delhi”] = 55
        map1[“Bejing”] = 20
        map1[“Washington”] = 25
        value, isPresent = map1[“Bejing”]
        if isPresent {
               fmt.Printf(“The value of \“Bejing\” in map1 is: %d\n”, value)
        } else {
               fmt.Println(“map1 does not contain Bejing”)
        }

        value, isPresent = map1[“Paris”]
        fmt.Printf(“Is \“Paris\” in map1 ?: %t\n”, isPresent)
        fmt.Printf(“Value is: %d\n”, value)

        // delete an item:
        delete(map1, “Washington”)
        value, isPresent = map1[“Washington”]
        if isPresent {
               fmt.Printf(“The value of \“Washington\” in map1 is: %d\n”, value)
        } else {
        fmt.Println(“map1 does not contain Washington”)
        }
}
```

for... range

```
package main
import “fmt”

func main() {
        map1 := make(map[int]float32)
        map1[1] = 1.0
        map1[2] = 2.0
        map1[3] = 3.0
        map1[4] = 4.0
for key, value := range map1 {
fmt.Printf(“key is: %d - value is: %f\n”, key, value)
        }
}
```

slice de maps

Versión B no hace *make* sobre los *maps* del *slice*

```
package main
import (
        “fmt”
)

func main() {
// Version A:
        items := make([]map[int]int, 5)
        for i := range items {
                items[i] = make(map[int]int, 1)
                items[i][1] = 2
        }
        fmt.Printf(“Version A: Value of items: %v\n”, items)
// Version B: NOT GOOD!
        items2 := make([]map[int]int, 5)
        for _, item := range items2 {
        item = make(map[int]int, 1)
                // item is only a copy of the slice element.
        item[1] = 2
                // This ‘item’ will be lost on the next iteration.
        }
        fmt.Printf(“Version B: Value of items: %v\n”, items2)
}
```

map: ordenación

No están ordenados.

Es necesario usar el paquete *sort* para ordenar *slices*

```
// the telephone alphabet:
package main
import (
        "fmt"
        "sort"
)

var (
        barVal = map[string]int{"alpha": 34, "bravo": 56, "charlie": 23,
        "delta": 87, "echo": 56, "foxtrot": 12, "golf": 34, "hotel": 16,
        "indio": 87, "juliet": 65, "kilo": 43, "lima": 98}
)

func main() {
        fmt.Println("unsorted:")
        for k, v := range barVal {
                fmt.Printf("Key: %v, Value: %v / ", k, v)
        }
        keys := make([]string, len(barVal))
        i := 0
        for k, _ := range barVal {
                keys[i] = k
                i++
        }
        sort.Strings(keys)
        fmt.Println()
        fmt.Println("sorted:")
        for _, k := range keys {
                fmt.Printf("Key: %v, Value: %v / ", k, barVal[k])
        }
}
```

Structs y Métodos

parte VI

structs

Declaración similar a C/C++

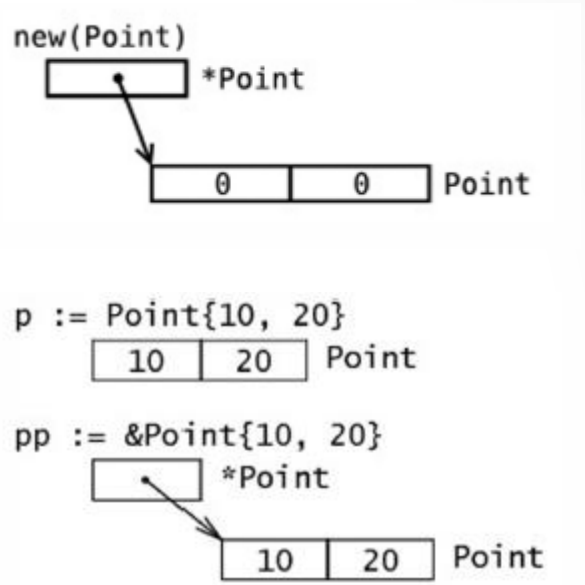
No hay clases, sólo structs

Ojo con la visibilidad (capitalización)

```
package main
import "fmt"

type struct1 struct {
        i1   int
        f1   float32
        str  string
}

func main() {
        ms := new(struct1)
        ms.i1 = 10
        ms.f1 = 15.5
        ms.str = "Chris"

        fmt.Printf("The int is: %d\n", ms.i1)
        fmt.Printf("The float is: %f\n", ms.f1)
        fmt.Printf("The string is: %s\n", ms.str)
        fmt.Println(ms)
}
```

struct: creación

```
type Point struct { x, y int }
```



structs anónimas

Parecido al concepto de herencia

Se denomina composición de tipos

```
package main
import “fmt”

type innerS struct {
        in1  int
        in2  int
}

type outerS struct {
        b   int
        c   float32
        int    // anonymous field
        innerS // anonymous field
}

func main() {
        outer := new(outerS)
        outer.b = 6
        outer.c = 7.5
        outer.int = 60
        outer.in1 = 5
        outer.in2 = 10

        fmt.Printf(“outer.b is: %d\n”, outer.b)
        fmt.Printf(“outer.c is: %f\n”, outer.c)
        fmt.Printf(“outer.int is: %d\n”, outer.int)
        fmt.Printf(“outer.in1 is: %d\n”, outer.in1)
        fmt.Printf(“outer.in2 is: %d\n”, outer.in2)

// with a struct-literal:
        outer2 := outerS{6, 7.5, 60, innerS{5, 10}}
        fmt.Println(“outer2 is: “, outer2)

}
```

métodos

Un método es una función con un *receptor* que la asocia a un tipo

```
package main
import “fmt”

type TwoInts struct {
        a int
        b int
}

func main() {
        two1 := new(TwoInts)
        two1.a = 12
        two1.b = 10

        fmt.Printf(“The sum is: %d\n”, two1.AddThem())
        fmt.Printf(“Add them to the param: %d\n”, two1.AddToParam(20))

        two2 := TwoInts{3, 4}
        fmt.Printf(“The sum is: %d\n”, two2.AddThem())
}

func (tn *TwoInts) AddThem() int {
        return tn.a + tn.b
}

func (tn *TwoInts) AddToParam(param int) int {
        return tn.a + tn.b + param
}
```

métodos: no local

No se pueden definir métodos sobre tipos no locales (definidos en otros paquetes)

Sí podemos hacer un alias o componer con un tipo local, etc.

```
package main
import (
    "fmt"
    "time"
)

type myTime struct {
    time.Time //anonymous field
}

func (t myTime) first3Chars() string {
    return t.Time.String()[0:3]
}

func main() {
       m := myTime{time.Now()}
       //calling existing String method on anonymous Time field
       fmt.Println("Full time now:", m.String())
       //calling myTime.first3Chars
       fmt.Println("First 3 chars:", m.first3Chars())}
```

métodos: receptor como puntero

El receptor se pasa por valor.

Se puede usar un puntero (paso por referencia) para poder cambiarlo o por razones de rendimiento

No es necesario -> para punteros

```
package main
import (
        “fmt”
)

type B struct {
        thing  int
}

func (b *B) change() {  b.thing = 1 }

func (b B) write() string { return fmt.Sprint(b) }

func main() {
        var b1 B  // b1 is a value
        b1.change()
        fmt.Println(b1.write())

        b2 := new(B)  // b2 is a pointer
        b2.change()
        fmt.Println(b2.write())
}
```

métodos: conversión automática

Go hace una conversión automática para poder llamar a cualquier método desde el tipo o el puntero al tipo

```
package main
import (
        “fmt”
)

type List []int
func (l List) Len() int { return len(l) }
func (l *List) Append(val int) { *l = append(*l, val) }

func main() {
        // A bare value
        var lst List
        lst.Append(1)
        fmt.Printf(“%v (len: %d)\n”, lst, lst.Len()) // [1] (len: 1)

        // A pointer value
        plst := new(List)
        plst.Append(2)
        fmt.Printf(“%v (len: %d)\n”, plst, lst.Len()) // &[2] (len: 1)
}
```

métodos: "herencia"

Mediante la composición obtenemos un mecanismo muy similar a la herencia en OO clásica

```
package main
import (
        "fmt"
        "math"
)

type Point struct {
        x, y float64
}

func (p *Point) Abs() float64 {
        return math.Sqrt(p.x*p.x + p.y*p.y)
}

type NamedPoint struct {
        Point
        name string
}

func main() {
        n := &NamedPoint{Point{3, 4}, "Pythagoras"}
        fmt.Println(n.Abs()) // prints 5
}
```

métodos: "herencia múltiple"

```
package main
import "fmt"

type Camera struct { }

func (c *Camera) TakeAPicture() string {
        return "Click"
}

type Phone struct { }

func (p *Phone ) Call() string {
        return "Ring Ring"
}

// multiple inheritance
type CameraPhone struct {
        Camera
        Phone
}

func main() {
        cp := new(CameraPhone)
        fmt.Println("Our new CameraPhone exhibits multiple behaviors ...")
        fmt.Println("It exhibits behavior of a Camera: ", cp.TakeAPicture())
        fmt.Println("It works like a Phone too: ", cp.Call())
}
```

métodos: Stringer

Para que un tipo pueda ser imprimido a medida por fmt.Print... se puede definir el método *String*

```
package main
import (
        “fmt”
        “strconv”
)

type TwoInts struct {
        a int
        b int
}

func main() {
        two1 := new(TwoInts)
        two1.a = 12
        two1.b = 10
        fmt.Printf(“two1 is: %v\n”, two1)    // output: two1 is: (12 / 10)
        fmt.Println(“two1 is:”, two1)        // output: two1 is: (12 / 10)
        fmt.Printf(“two1 is: %T\n”, two1)
                // output: two1 is: *main.TwoInts
        fmt.Printf(“two1 is: %#v\n”, two1)
                // output: &main.TwoInts{a:12, b:10}
}

func (tn *TwoInts) String() string {
        return “(“+ strconv.Itoa(tn.a) +” / “+ strconv.Itoa(tn.b) + “)”
}
```

Interfaces

• • •

parte VII

Go: Interfaces

- Go no es un lenguaje OO clásico:
  - no hay clases ni herencia
- Go posee interfaces muy flexibles
- Los interfaces no contienen código y son un tipo que define una serie de métodos
- Se puede declara una variable de tipo interfaz
- Los tipos tienen el conjunto de métodos que implementa el interfaz
- Un tipo no necesita declarar que implementa un interfaz (*duck typing*)
- Un tipo que implementa un interfaz puede tener otros métodos o implementar otros interfaces

Interfaces

ejemplo

```
package main
import “fmt”

type Shaper interface {
        Area() float32
}

type Square struct {
        side float32
}

func (sq *Square) Area() float32 {
        return sq.side * sq.side
}

type Rectangle struct {
   length, width float32
}

func (r Rectangle) Area() float32 {
   return r.length * r.width
}
```

```
func main() {
   r := Rectangle{5, 3} // Area() of Rectangle needs a value
   q := &Square{5       // Area() of Square needs a pointer
   // shapes := []Shaper{Shaper(r), Shaper(q)}
   // or shorter:
   shapes := []Shaper{r, q, c}
   fmt.Println(“Looping through shapes for area ...”)
   for n, _ := range shapes {
       fmt.Println(“Shape details: “, shapesArr[n])
       fmt.Println(“Area of this shape is: “, shapes[n].Area())
   }
}
```

Interfaz vacío

interface{} -> void *

Permite contener cualquier tipo y crear funciones genéricas

```
package main
import “fmt”

var i = 5
var str = “ABC”

type Person struct {
        name string
        age    int
}

type Any interface{}

func main() {
        var val Any
        val = 5
        fmt.Printf(“val has the value: %v\n”, val)
        val = str
        fmt.Printf(“val has the value: %v\n”, val)
        pers1 := new(Person)
        pers1.name = “Rob Pike”
        pers1.age = 55
        val = pers1
        fmt.Printf(“val has the value: %v\n”, val)
        switch t := val.(type) {
        case int:
               fmt.Printf(“Type int %T\n”, t)
        case string:
               fmt.Printf(“Type string %T\n”, t)
        case bool:
               fmt.Printf(“Type boolean %T\n”, t)
        case *Person:
               fmt.Printf(“Type pointer to Person %T\n”, *t)
        default:
               fmt.Printf(“Unexpected type %T”, t)
 }
}
```

interface{}: comprobación de tipos

Se puede hacer un switch de tipos para determinar el tipo original

```
package main
import “fmt”

type specialString string
var whatIsThis specialString = “hello”

func TypeSwitch() {
        testFunc := func(any interface{}) {
                switch v := any.(type) {
                case bool:
                fmt.Printf(“any %v is a bool type”, v)
                case int:
                fmt.Printf(“any %v is an int type”, v)
                case float32:
                fmt.Printf(“any %v is a float32 type”, v)
                case string:
                fmt.Printf(“any %v is a string type”, v)
                case specialString:
                fmt.Printf(“any %v is a special String!”, v)
                default:
                fmt.Println(“unknown type!”)
                }
        }
        testFunc(whatIsThis)
}

func main() {
        TypeSwitch()
}
```

tipos generales

Podemos usar interface{} para crear nodos que acepten tipos distintos

```
package main
import “fmt”

type Node struct {
        le    *Node
        data  interface{}
        ri    *Node
}

func NewNode(left, right *Node) *Node {
        return &Node{left, nil, right}
}
func (n *Node) SetData(data interface{}) {
        n.data = data
}

func main() {
        root := NewNode(nil,nil)
        root.SetData(“root node”)
        // make child (leaf) nodes:
        a := NewNode(nil,nil)
        a.SetData(“left node”)
        b := NewNode(nil,nil)
        b.SetData(“right node”)
        root.le = a
        root.ri = b
        fmt.Printf(“%v\n”, root) // Output: &{0x125275f0 root node 0x125275e0}
}
```

Go avanzado

parte VIII

Otros conceptos

- Gestión de paquetes
- Reflexión
- Concurrencia:
  - goroutines, channels
- Librería estándar:
  - I/O, red, web, templates…
- Errores y tests
- Go en AppEngine