Seminario de Lenguajes opción Go

Raúl Champredonde

Seminario de Lenguajes opción Go

- Arrays
- Slices
- For range
- Maps

Arrays

- Secuencia indexada de elementos de un mismo tipo, de longitud fija, con primer índice en cero.

```
var x [5]int
x[4] = 100
fmt.Println(x) // [0 0 0 0 100]
```

Arrays

```
var x [5]float64
x[0] = 98
x[1] = 93
x[2] = 77
x[3] = 82
x[4] = 83
```

```
x := [5]float64{98, 93, 77, 82, 83}
```

```
x := [...]float64{98, 93, 77, 82, 83}
```

```
x := [5]float64 {
    98,
    93,
    77,
    82,
    83,
}
```

```
for i := range x {
    fmt.Println(i)
}
```

Arrays

- Inicialización posicional vs nombrada

```
arr := [5]int{1:10, 2:20, 3:30}

fmt.Println(arr) // [0 10 20 30 0]
```

- Longitud

```
fmt.Println(len(arr))
```

Arrays multidimensionales

- Los arrays pueden contener elementos de cualquier tipo, incluso arrays

```
var a [2][2]string
a[0][0] = "Hello"
a[0][1] = "World"
a[1][0] = "Hola"
a[1][1] = "Mundo"
```

```
a := [2][2]string{
   {"Hello", "World"},
   {"Hola", "Mundo"},
}
```

```
fmt.Println(a[0], a[1])          // [Hello World] [Hola Mundo]
fmt.Println(a)                   // [[Hello World] [Hola Mundo]]
```

Slices

- La longitud de un tipo array es parte de la definición del tipo
- Esto hace que el uso de arrays sea un poco incómodo
- Entonces es más frecuente usar Slices que Arrays

- Un slice es un "segmento" de un array
- Son indexables y tienen una longitud
- Pero esta longitud puede cambiar

Slices

- Longitud y capacidad (slices y arrays)
  - `len(), cap()`
- Hay 3 formas de crear un slice (siempre hay un array subyacente)
  -
```
a := [6]int{10, 11, 12, 13, 14, 15}
s := a[2:4]        // len(s) - 2, cap(s) - 4
```
  -
```
var s1 []int           // len(s1) - 0, cap(s1) - 0
s2 := []int{}          // len(s2) - 0, cap(s2) - 0
s3 := []int{1, 2, 3}   // len(s3) - 3, cap(s3) - 3
```
  -
```
s1 := make([]int, 5, 10)   // len(s1) - 5, cap(s1) - 10
s2 := make([]int, 5)       // len(s2) - 5, cap(s2) - 5
```

Slices

- El slice vacío es un slice `nil`

```
ar s []int

fmt.Println(s, len(s), cap(s)) // [] 0 0

if s == nil {
    fmt.Println("nil!")       // nil!
}
```

Slices

- Slices son referencias a los arrays subyacentes

```
a := [6]int{10, 11, 12, 13, 14, 15}
s := a[2:4]
fmt.Println(a, s)   // [10 11 12 13 14 15]
                    //       [12 13]

s[1] = 31
a[2] = 21
fmt.Println(a, s)   // [10 11 21 31 14 15]
                    //       [21 31]
```

Slices

- Slices son referencias a los arrays subyacentes

```
s := []int{1, 2, 3, 4, 5, 6}
fmt.Println(s, len(s), cap(s))    // [1 2 3 4 5 6] 6 6
```

```
t := s[1:4]
fmt.Println(t, len(t), cap(t))
      // [2 3 4] 3 5
t = s[:2]
fmt.Println(t, len(t), cap(t))
      // [1 2] 2 6
t = s[1:]
fmt.Println(t, len(t), cap(t))
      // [2 3 4 5 6] 5 5
t = s[0:4]
fmt.Println(t, len(t), cap(t))
      // [1 2 3 4] 4 6
```

```
s = s[1:4]
fmt.Println(s, len(s), cap(s))
      // [2 3 4] 3 5
s = s[:2]
fmt.Println(s, len(s), cap(s))
      // [2 3] 2 5
s = s[1:]
fmt.Println(s, len(s), cap(s))
      // [3] 1 4
s = s[0:4]
fmt.Println(s, len(s), cap(s))
      // [3 4 5 6] 4 4
```

Slices

- Crear slices con la función `make`

```
a := make([]int, 5)
printSlice("a", a) // a len=5 cap=5 [0 0 0 0 0]

z := []int{1, 2, 3, 4, 5, 6, 7}
a = z
printSlice("a", a) // a len=7 cap=7 [1 2 3 4 5 6 7]

b := make([]int, 0, 5)
printSlice("b", b)       // b len=0 cap=5 []
printSlice("b", b[0:5]) // b len=5 cap=5 [0 0 0 0 0]

c := b[:2]
printSlice("c", c)// c len=2 cap=5 [0 0]

d := c[2:5]
printSlice("d", d) // d len=3 cap=3 [0 0 0]
```

```
func printSlice(s string, x []int) {
    fmt.Printf("%s len=%d cap=%d
%v\n",
        s, len(x), cap(x), x)
}
```

Slices multidimencionales

- Los slices pueden contener elementos de cualquier tipo, incluso slices

```
package main

import ("fmt"; "math/rand"; "strings")

func main() {
  ta_te_ti := [][]string{
    []string{"_", "_", "_"},
    []string{"_", "_", "_"},
    []string{"_", "_", "_"},
  }

  turno := 0

  signo := []string{"X", "O"}

  secuencia := make([]string, 9)
```

```
ta_te_ti := [][]string{
  {"_", "_", "_"},
  {"_", "_", "_"},
  {"_", "_", "_"},
}
```

```
  for i := 0; i < 9; i++ {
    poner(ta_te_ti, secuencia, i, signo[turno])
    turno++     // turno = (turno + 1) % 2
    turno %= 2
  }
  fmt.Println("[" + strings.Join(secuencia, " - ") + "]")
  for _, e := range ta_te_ti {
    fmt.Println("[" + strings.Join(e, "] [") + "]")
  }
}

func poner(tab [][]string, sec []string, pos int, sig string)
{
  for {
    x, y := rand.Intn(3), rand.Intn(3)
    if tab[x][y] == "_" {
      tab[x][y] = sig
      sec[pos] = fmt.Sprint("(", x, "-", y, "):", sig)
      break
    }
  }
}
```

Slices

- Agregar elementos a un slice

```
func append(slice []Type, elems ...Type) []Type

slice = append(slice, elem1, elem2)
slice = append(slice, anotherSlice...)

var s []int
printSlice("s", s)  // s len=0 cap=0 []

s = append(s, 0)
printSlice("s", s)  // s len=1 cap=1 [0]

s = append(s, 1)
printSlice("s", s)  // s len=2 cap=2 [0 1]

s = append(s, 2, 3, 4)
printSlice("s", s)  // s len=5 cap=6 [0 1 2 3 4]

s = append(s, s...)
printSlice("s", s)  // s len=10 cap=12 [0 1 2 3 4 0 1 2 3 4]
```

Slices

- Copiar slices

Las tamaños no importan

```
func copy(dst, src []Type) int

numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
printSlice("numbers", numbers)
      // numbers len=20 cap=20 [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20]
neededNumbers := numbers[:5]
printSlice("neededNumbers", neededNumbers)
      // neededNumbers len=5 cap=20 [1 2 3 4 5]
numbersCopy := make([]int, len(neededNumbers))
printSlice("numbersCopy", numbersCopy)
      // numbersCopy len=5 cap=5 [0 0 0 0 0]
copy(numbersCopy, neededNumbers)
printSlice("numbersCopy", numbersCopy)
      // numbersCopy len=5 cap=5 [1 2 3 4 5]
```

```
numbers[2] = 100

printSlice("numbers", numbers)
// numbers len=20 cap=20 [1 2 100 …]

printSlice("neededNumbers", neededNumbers)
// neededNumbers len=5 cap=20 [1 2 100 4
5]

printSlice("numbersCopy", numbersCopy)
// numbersCopy len=5 cap=5 [1 2 3 4 5]
```

Array / Slice - iteración

- Iteración - range

```
arr := [10]int{9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
```

```
for index, elem := range arr {
  fmt.Printf("%d:%d - ", index, elem)
}
fmt.Println()

for _, elem := range arr {
  fmt.Printf("%d - ", elem)
}
fmt.Println()
```

```
for i, _ := range arr {
  arr[i] = 1 << uint(i) // == 2**i
}
fmt.Println(arr)

for i := range arr {
  arr[i] = 1 << uint(i) // == 2**i
}
fmt.Println(arr)
```

```
// 0:9 - 1:8 - 2:7 - 3:6 - 4:5 - 5:4 - 6:3 - 7:2 - 8:1 - 9:0 -
// 9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1 - 0 -
// [1 2 4 8 16 32 64 128 256 512]
// [1 2 4 8 16 32 64 128 256 512]
```

Array / Slice - iteración

```
func nonempty1(strings []string) []string {
   i := 0
   for _, s := range strings {
      if s != "" {
         strings[i] = s
         i++
      }
   }
   return strings[:i]
}
```

```
func nonempty2(strings []string) []string {
   out := strings[:0]
   for _, s := range strings {
      if s != "" {
         out = append(out, s)
      }
   }
   return out
}
```

```
func main() {
  strs := []string{"a", "", "b", "", "c"}
  fmt.Println(nonempty1(strs))          // [a b
c]
}
```

```
func main() {
  strs := []string{"a", "", "b", "", "c"}
  fmt.Println(nonempty2(strs))          // [a b
c]
}
```

Parámetros por valor / por copia

- Hablando de “tipos” y “parámetros por copia” …

```
type Array600Int [600]int

func sumPrimes(arr Array600Int) (res int) {
   for _, e := range arr {
      res += e
   }
   arr[0] = 17
   fmt.Println(arr)    // [2 3 5 7 11 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...]
   return
}

func main() {
   primes := Array600Int{0:2, 1:3, 2:5, 3:7, 4:11, 5:13}
   fmt.Println(primes) // [2 3 5 7 11 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...]
   fmt.Println(sumPrimes(primes)) // 41
   fmt.Println(primes) // [2 3 5 7 11 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...]
}
```

Maps

- Colección no ordenada de pares clave-valor
- También llamados arreglos asociativos, tablas hash o diccionarios
- No permite claves duplicadas
- El valor por defecto es `nil`
- Se puede agregar, modificar y eliminar elementos, excepto que el map sea `nil`

Maps

- Tipos de clave permitidos:
  - Los que tienen definida la comparación por igualdad (==)
  - Se puede: booleans, numbers, strings, arrays, ...
  - No se puede: slices, maps, functions
- Tipos de valor permitidos:
  - Cualquiera

Maps

```
var a map[string]string    // a es nil
// a["clave"] = "Valor"        ERROR EN TIEMPO DE EJECUCIÓN !!!

var b = make(map[string]string)
b["clave"] = "Valor"

var c = map[string]string{"brand": "Ford", "model": "Mustang", "year": "1964"}
d := map[string]int{"Oslo": 1, "Bergen": 2, "Trondheim": 3, "Stavanger": 4}

fmt.Println(a); // map[]
fmt.Println(b); // map[clave:Valor]
fmt.Println(c); // map[brand:Ford model:Mustang year:1964]
fmt.Println(d); // map[Bergen:2 Oslo:1 Stavanger:4 Trondheim:3]
```

Maps

- Agregar o modificar
  - `m[key] = value`
- Eliminar
  - `delete(m, key)`
- Recuperar
  - `elem = m[key]` // si key no está en m, elem es el
  - `elem, ok = m[key]` // "zero value" del tipo correspondiente

  - `elem, ok := m[key]` // si elem y ok no están declaradas

Maps

- Los maps son referencias

```
var a = map[string]string{"brand": "Ford",
                          "model": "Mustang",
                          "year": "1964",
}
b := a

fmt.Println(a)  // map[brand:Ford model:Mustang year:1964]
fmt.Println(b)  // map[brand:Ford model:Mustang year:1964]

b["year"] = "1970"

fmt.Println("After change to b:")

fmt.Println(a)  // map[brand:Ford model:Mustang year:1970]
fmt.Println(b)  // map[brand:Ford model:Mustang year:1970]
```

Maps

- Maps range

```
func createMap(a []string) map[string]string {
   result := make(map[string]string)
   for _, e := range a {
      result[string(e[0])] = e
   }
   return result
}

func main() {
   arr := []string{"Hydrogen", "Helium", "Lithium", "Beryllium", "Boron",
                   "Carbon", "Nitrogen", "Oxygen", "Fluorine", "Neon"}
   m := createMap(arr)
   fmt.Println(m) // map[B:Boron C:Carbon F:Fluorine H:Helium L:Lithium N:Neon O:Oxygen]
   for k, v := range m {
      fmt.Printf("(%s:%s) ", k, v)
   }
         // (C:Carbon) (N:Neon) (O:Oxygen) (F:Fluorine) (H:Helium) (L:Lithium) (B:Boron)
}
```