Strings, runes y bytes en Go

Tipos de datos

Tipo de datos *byte*

Un byte en Go es sinónimo de un *uint8*, un unsigned int8.

- 8 bits que podemos asignar de manera directa a diferentes notaciones.
- Como es un tipo de dato uint8 nos permite usar cualquier número entre 0 y 255.

Tipo de datos byte

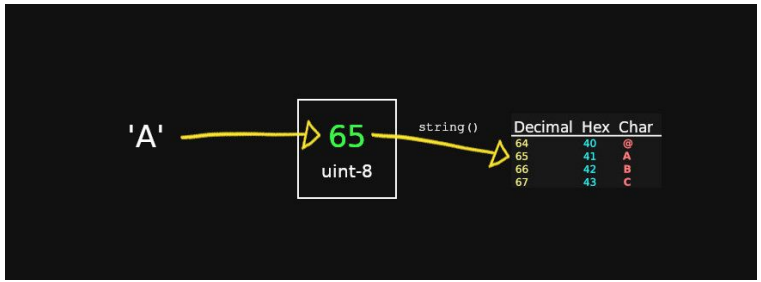
```
var ch byte = 65 // decimal

var ch1 byte = 0b1000001 // binario

var ch2 byte = 0o101 // octal

var ch3 byte = 0X41 // hexadecimal
```

```
var ch4 byte = 'A'
fmt.Println(ch4)  // 65
fmt.Println(string(ch4))  // 'A'
```

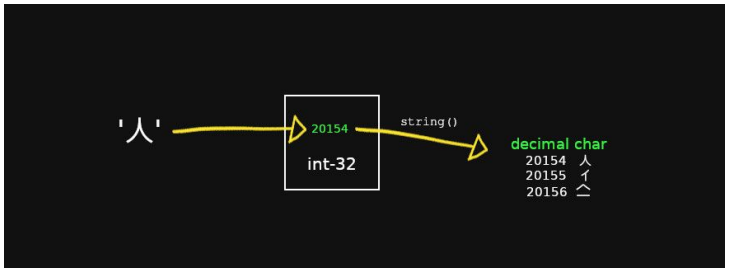


Tipo de datos *rune*

- Las runas son sinónimo de un tipo *int32*.
- Es el tipo de variable por defecto cuando defines un carácter, **utilizamos comillas simples para declararlo**.
- Si no especificas byte u otro tipo de dato, go dará por sentado que se trata de una runa.

```
var runa rune = 65
fmt.Printf("type:%T, value:%v\n", runa, runa)
// type:int32, value:65

var runa rune = '人'
fmt.Printf("type:%T, value:%v\n", runa, runa)
// type:int32, value:20154
```



[]byte a string

```
// Instanciado directamente de un string
t1 := []byte("ABCDE")

// Como si fuera un array de caracteres
t2 := []byte{'A', 'B', 'C', 'D', 'E'}

// Como si fuera un array de números ord()
t3 := []byte{65, 66, 67, 68, 69}

// Con la función copy
var t4 = make([]byte, 5)
copy(t4, "ABCDE")

// En todos los casos obtenemos:
//[65 66 67 68 69]

t3 := []byte{65, 66, 67, 68, 69}
fmt.Println(string(t1))
// ABCDE
```

[]rune a string

```
arrayRunas := []rune("Jello, ")
arrayRunas[0] = 'H'
arrayRunas = append(arrayRunas, '世', '!')
fmt.Println(arrayRunas)
// [72 101 108 108 111 44 32 19990 33]

fmt.Println(string(arrayRunas))
// Hello, 世!
```

UTF-8

- Es un estándar de codificación de caracteres de longitud variable.
- El tipo *rune* está codificado así. En la tabla los x son los bits del valor que se quiere codificar:

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	[b]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

UTF-8

- Los primeros 128 códigos son compatibles con el ASCII de 7 bits.
- Con el segundo byte se cubren 1920 codificaciones que cubren los alfabetos latinos y varios más.
- Con el tercero aparecen los caracteres chinos, japones y coreanos.
- Con el cuarto byte de pueden codificar otros símbolos.

Proceso de codificación

Los dígitos rojo, verde y azul indican cómo se distribuyen los bits a codificar entre los bytes UTF-8. Los bits adicionales agregados por el proceso de codificación UTF-8 se muestran en negro:

1. El código UTF-8 para el símbolo del euro € es U+20AC.
2. Como código se encuentra entre U+0800 y U+FFFF, necesita tres bytes para codificarse.
3. El hexadecimal 20AC es binario 0010 0000 10 10 1100 . Los dos ceros iniciales se agregan porque una codificación de 3 bytes necesita 16 bits.
4. Debido a que la codificación tendrá una longitud de tres bytes, su byte inicial comienza con tres 1, luego un 0 (1110...)
5. Los cuatro bits más significativos del código se almacenan en los cuatro bits restantes de orden bajo de este byte (1110 0010), dejando 12 bits del código aún por codificar (... 0000 10 10 1100).
6. Todos los bytes de continuación contienen 6 bits de código. Entonces, los siguientes seis bits del código se almacenan en los seis bits de orden inferior del siguiente byte y 10 se almacena en el orden superior. Son dos bits para marcarlo como un byte de continuación. Por lo que el segundo byte queda 10 000010.
7. Finalmente, los últimos seis bits del código se almacenan en los seis bits de orden inferior del byte final, y nuevamente 10 se almacena en el orden superior de dos bits (10 101100).
8. Finalmente los tres bytes codificados quedarían así 1110 0010 10 000010 10 101100.

Tipo de datos *string*

En Go un **strings** es una secuencia de caracteres de ancho variable donde todos y cada uno de los caracteres están representados por uno o más bytes usando la codificación UTF-8.

- El string es una cadena inmutable de bytes arbitrarios.
- Es string es un []bytes de solo lectura.
- Debido a la codificación UTF-8, el string Golang puede contener un texto que es una mezcla de cualquier idioma.
- Los literales del string se escriben entre comillas dobles.

Strings

- Cada índice del slice se refiere a un byte. Esto es importante, porque si iteramos sobre un string, vamos a obtener una cantidad diferente de bytes a los caracteres que forman nuestro string.

```
str := "Ahí."
fmt.Println(len(str))
// 4
// 5
```

Iteramos por los bytes

```go
package main

import (
    "fmt"
)

func main() {

    str := "Ahí."

    for i := 0; i < len(str); i++ {
        fmt.Printf("%d -> %c\n", i, str[i])
    }
}
```

```
// 0 -> A
// 1 -> h
// 2 -> Ã
// 3 ->
// 4 -> .
```

Iteramos por las runas

```go
package main

import (
    "fmt"
)

func main() {

    str := "Ahí."

    for i, c := range str {
        fmt.Printf("%d -> %c (%v)\n", i, c, c)
    }

}
```

```
0 -> A (65)
1 -> h (104)
2 -> í (237)
4 -> . (46)
```

Dimensiones

```go
package main

import (
   "fmt"
   "unsafe"
)

func main() {
  var b byte
  b = 'í'
  fmt.Printf("type: %T, size: %d\n", b, unsafe.Sizeof(b))
  // type: uint8, size: 1

  r := 'í'
  fmt.Printf("type: %T, size: %d\n", r, unsafe.Sizeof(r))
  // type: int32, size: 4

  s := "í"
  fmt.Printf("type: %T, size: %d\n", s, unsafe.Sizeof(s))
  fmt.Println("length: ", len(s))
  //type: string, size: 16
  //length: 2
}
```