CH3 核心型别

3-1 前言

- Go語言是強型別(strongly typed)語言，意即所有的資料都必須屬於某個型別，而這個型別是固定的
  - 像是python / javascript 的變數是可以任意更換型別的，屬於弱型別

- 你能或不能對此資料做的動作取決於該資料的型別

- 若要精通go語言，正確的了解每一種核心型別就是關鍵，稍後的章節還會介紹他複雜的型別，但他們全都建構在本章要介紹的核心型別之上

型別的定義包括：

- 其中能儲存何種資料
- 你能對他進行甚麼操作
- 這些操作會對資料做甚麼
- 會占用多少記憶體

3-2 布林值 true / false

- 真與偽這兩個邏輯值都屬於布林型別，在go語言寫成bool

- 當你使用 == 或 > 這類算符結果一定是bool值：

```go
package main

import "fmt"

func main() {
    fmt.Println(10 > 5)
    fmt.Println(10 == 5)
}
```

練習：利用程式判斷密碼複雜度

- 練習寫出一個程式，能顯示輸入的密碼是否符合以下要求；
  - 必須有小寫字母
  - 必須有大寫字母
  - 必須有數字
  - 必須有符號
  - 長度至少為8個字元

```go
package main

import (
    "fmt"
    "unicode" //使用unicode函式庫
)

func passwordChecker(pw string) bool {
    pwR := []rune(pw) //把密碼轉乘rune型別 , 以便接收 UTF-8 字串
    if len(pwR) < 8 { //若密碼長度不足8 , 等於檢查失敗
        return false
    }
    hasUpper := false
    hasLower := false
    hasNumber := false
    hasSymbol := false

```

```
18      for _, v := range pwR { //用for range 走訪字串的每個字元, 忽略其索引
19          if unicode.IsUpper(v) { //檢查是否有大寫字元
20              hasUpper = true
21          }
22          if unicode.IsLower(v) { //檢查是否有小寫字元
23              hasLower = true
24          }
25          // 是否有數字
26          if unicode.IsNumber(v) {
27              hasNumber = true
28          }
29
30          // 是否有符號
31          if unicode.IsPunct(v) || unicode.IsSymbol(v) {
32              hasSymbol = true
33          }
34      }
35      return hasUpper && hasLower && hasNumber && hasSymbol
36  }
37
```

```
38  func main() {
39      if passwordChecker("") {
40          fmt.Println("密碼格式良好")
41      } else {
42          fmt.Println("密碼格式不正確")
43      }
44
45      if passwordChecker("This!I5A") {
46          fmt.Println("密碼格式良好")
47      } else {
48          fmt.Println("密碼格式不正確")
49      }
50  }
```

- 在上面的程式碼中，先將sting型轉換為rune型別，這是一種可以安全接收多位元組字元(UTF -8)，本章稍後會談到

- 這裡也使用unicode套件中的幾個函式來檢查字元，它們都會回傳 true 或 false

- 在passwordChecker函式的最後，我們用 && 將所有的檢查的布林值串起來，也就是說，當所有檢查成立時才會return true

3-3 數字

3-3-1 整數

- 整數型別分為兩種：可儲存負值的有號整數 / 無法存負值型別的無號整數

- 每種型別可以儲存的最大值和最小值，都取決於型別的內部儲存容量有幾個位元組

- 下列是go語言規格中的相關整數型別：
  - Unit8 : 無號8位元整數 (0~255)
  - Unit16 : 無號16為元整數(0~65535)
  - Unit32 : 無號32位元整數(0~4294967295)
  - Unit64 : 無號64位元整數 (0~18446744073709551615)
  - Int8 : 有號8位元整數 (-128~127)
  - Int16 : 有號16位元整數 (-32768~32767)

- Int32 : 有號32位元整數 (-2147483648~2147483647)
- Int64 : 有號64為元整數 (-92233720368547758 08~ 9223372036854775807)
- byte : unit8的別稱
- rune : unit32的別稱

- 此外還有特殊的整數型別 :
  - Unit : 無號32或64為元整數
  - Int : 有號32或64位元整數
- Unit 和 int 的長度取決於你針對32位元還是64位元的系統編譯程式 (現今絕大多數都是64位元)
- 在64位元的系統上 , int型別和int64型別雖然整數範圍相同 , 但go語言視為兩種不同的型別 , 這是因為若兩者混用 , 對32位元的機器編譯同一支程式就會發生錯誤
- 這種不相容性不僅限於int而已 : 事實上任何整數型別彼此都不得混用

- 至於要選擇哪種整數型別？直接用int即可，因為int可以完成大部分的工作，只有當int會造成問題時(例如記憶體用量)才會考慮其他型別

- 譬如，假設你的某程是把記憶體耗光了，因為這支程式宣告了大量整數，但這些數字恆正，也沒有超過255，於是可能的解法是將他們的型別由int改為unit8，如此就可以將每個數字占用的記憶體從64位元將降低到8位元

- 下面來證明這一點：

- 用兩種整數型別建立一個切片集合(int或unit8)，然後在集合中放進一千萬個數字，最後go語言會用runtime套件取得整支程式所使用的堆積記憶體量(位元組)，轉換成MB(百萬位元組)單位後印出：

```go
package main

import (
    "fmt"
    "runtime"
)

func main() {
    var list []int // 換成 var list []int8 試試
    for i := 0; i < 10000000; i++ {
        list = append(list, 100)
    }
    //印出記憶體用量
    var m runtime.MemStats
    runtime.ReadMemStats(&m)
    fmt.Printf("TotalAlloc (Heap) = %v MiB\n", m.TotalAlloc/1024/1024)
}
```

- 以下是list切片 變數宣告為int時的輸出結果：

```
TotalAlloc (Heap) = 469 MiB
```

- 以下是list切片 變數宣告為int8時的輸出結果：

```
TotalAlloc (Heap) = 49 MiB
```

3-3-2 浮點數

- go語言有兩種浮點型別 : float32 與 float64 , float64容量較大 , 精確度也較高

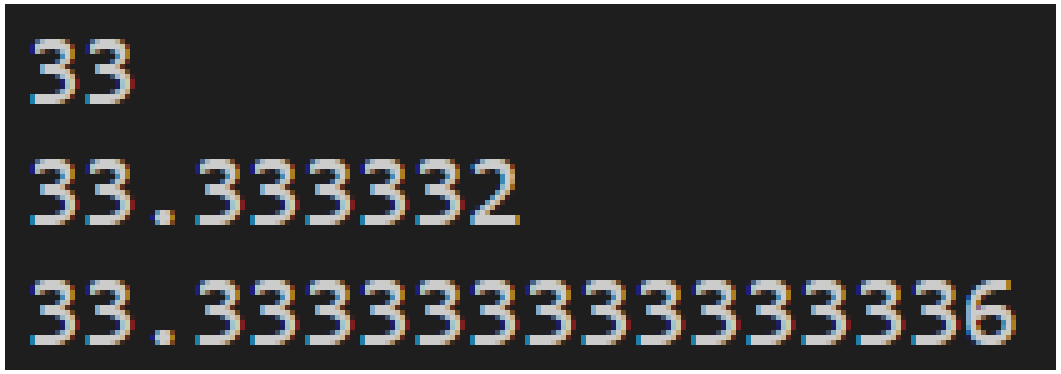
- 以下有一個練習 : 浮點數的精確度

- 將100除以3，電腦會用3.333.....無限循環小數來呈現，然而若真的這樣計算，記憶體遲早會被耗光

- 還好，浮點數型別有儲存的上限，所以我們不用擔心上述問題，但缺點是，有限的儲存方式會無法反映實際數值，所以必須在精確度和儲存空間做出取捨

```go
package main

import "fmt"

func main() {
    var a int = 100
    var b float32 = 100
    var c float64 = 100
    fmt.Println(a / 3)
    fmt.Println(b / 3)
    fmt.Println(c / 3)
}
```

顯示結果：



註:
為何小數點後會有3以外的數字？因為電腦用2進為儲存數字，因此多少會有誤差

浮點數的實用性

- 雖然儲存時可能有誤差，但大部分時間浮點數仍運作得相當理想
- 我們來測試看看，把上頁的結果乘以3是否會等於100？

```go
package main

import "fmt"

func main() {
    var a int = 100
    var b float32 = 100
    var c float64 = 100
    fmt.Println((a / 3) * 3)
    fmt.Println((b / 3) * 3)
    fmt.Println((c / 3) * 3)
}
```

顯示結果：

```
99
100
100
```

- 可以發現，誤差造成的影響似乎沒有想像中的大
- 不過如果反覆地做乘除，誤差可能會逐漸放大，因此，出非想要節省記憶體，一般建立浮點數的首選型別都是float64

3-3-3 溢位和越界繞回

- 若試圖給予一個變數超過型別允許上限的初始值，就會發生溢位(overflow)錯誤

- 底下以int8為例，它能容許的最大是127，但故意給予128做初始值，並觀察結果

```go
package main

import "fmt"

func main() {
    var a int8 = 128
    fmt.Println(a)
}
```

顯示結果：

```
# command-line-arguments
ch3\3-3-4.go:6:15: cannot use 128 (untyped int constant) as int8 value in variable declaration (overflows)
```

- 以上的溢位問題不難修正，但若是在建立變數後才將值設定為超過127，將會發生越界繞回(wraparound)現象，也就是超過最大值後重新由最小值開始計算

- 越界繞回是容易遇到的問題，且編譯器也無法攔截，這有可能對程式的使用者造成大問題

練習：觸發越界繞回

- 這個練習要宣告兩個較小的整數型別：int8 和 unit8，並都分別先賦予一個接近其上限的起始值，然後利用迴圈將變數遞增1，最終使之超過上限，並輸出每次回圈重複時的值來觀察

```go
package main

import "fmt"

func main() {
    var a int8 = 125
    var b uint8 = 253
    for i := 0; i < 5; i++ {
        a++
        b++
        fmt.Println(i, ")", "int8", a, "uint8", b)
    }
}
```

顯示結果：

```
0 ) int8 126 uint8 254
1 ) int8 127 uint8 255
2 ) int8 -128 uint8 0
3 ) int8 -127 uint8 1
4 ) int8 -126 uint8 2
```

←發生了越界繞回

3-3-4 大數值

- 如果要使用的值超過了int64或unit64的極限，可以向內建的 math/big 套件求助

- 與先前的整數型別相比，這個套件用起來可能會有一點怪異，但只要透過它，就可以實現大數值

練習：使用大數值

- 這次練習要建立一個超過go語言核心數字型別所容許的數值，分別用int型別和math/big 的 big,int 大整數型別來表示

- 為了證明有用，我們用加減法來測試它們是否溢位，然後印出其結果

```go
package main

import (
    "fmt"
    "math"
    "math/big"
)

func main() {
    intA := math.MaxInt64 //int整數
    intA = intA + 1

    bigA := big.NewInt(math.MaxInt64) //big.Int整數
    bigA.Add(bigA, big.NewInt(1))

    fmt.Println("MaxInt64: ", math.MaxInt64)
    fmt.Println("Int      :", intA)
    fmt.Println("Big Int : ", bigA.String())
}
```

顯示結果

```
MaxInt64:   9223372036854775807
Int      : -9223372036854775808
Big Int  :  9223372036854775808
```

←int64的最大值

← int 發生越界繞回

← Big int 正確+1

- 從上面的練習可以發現，當數值+1時，int型別發生了越界繞回，而big int則順利+1

- 其他math/big的用法可以參考官方文件：https://pkg.go.dev/math/big

3-3-5 位元組 (Byte)

- go語言的byte型別其實就是unit8型別的別稱
- 在現實世界中，byte是很重要的型別，可以在很多地方看到，包括讀寫網路或檔案資料
- 每一個位元(bit)代表一個二進位直，電腦運算採用8位元一組的“位元組” 編碼

3-4 字串

3-4-1 字串與字串常值

- Go語言只有一種文字型別，就是 string

- 當你在程式中直接寫出文字值時，它叫做字串常值(string literal)

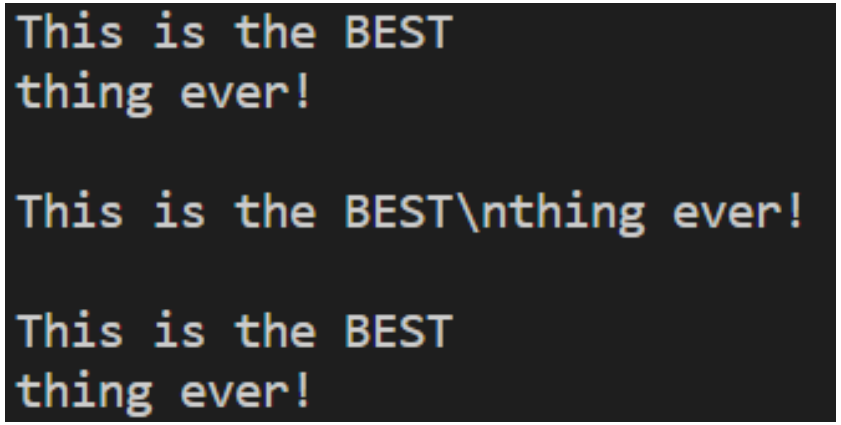
- Go語言支援兩種字串常值：
  1. 原始的(raw)：由一對反引號` 括住的字串

  2. 轉譯的(interpreted)：由一對雙引號括住的字串

- 若字串變數儲存的是原始字串時，變數內容會跟字串在螢幕上的內容完全一樣
- 若是轉譯字串，go語言會先掃描你寫的內容，並用它的規則轉換某些文字
- 底下來示範兩種字串的顯示效果

```go
package main

import "fmt"

func main() {
    comment1 := `This is the BEST
thing ever!`
    comment2 := `This is the BEST\nthing ever!` //原始字串, 換行符號不轉譯
    comment3 := "This is the BEST\nthing ever!" //轉譯字串, 換行符號會轉譯

    fmt.Print(comment1, "\n\n")
    fmt.Print(comment2, "\n\n")
    fmt.Print(comment3, "\n")
}

```

顯示結果：



- 在轉譯的字串中，\n代表換行，但\n在原始字串裡只是一般文字
- 轉譯字串有很多種用法，但最常用的是\n以及\t當作tab
- 唯一不能出現在原始字串內的字元是反引號，如果需要在在串內加上反引號就必須使用轉譯字串
- 底下再看一個範例

```go
package main

import "fmt"

func main() {
    comment1 := `In "Windows" the user directory is "C:\Users\"`
    comment2 := "In \"Windows\" the user directory is \"C:\\Users\\\""

    fmt.Println(comment1)
    fmt.Println(comment2)
}
```

顯示結果:

```
In "Windows" the user directory is "C:\Users\"
In "Windows" the user directory is "C:\Users\"
```

- 在轉譯字串中想要表達雙引號或反斜線，必須在前面多寫一次反斜線，所以此時用原始字串會比較方便

- 此外，字串常值只是用來把文字存進string型別變數的方法，文字一旦存入變數，不管你用甚麼辦法存的就都沒有差別了

- 接下來要探討如何安全的處理多位元組字串

3-4-2 rune

- Rune(符文)是一種具備充足空間，足以容納單一一個UTF-8字元(unicode編碼會占用1~4個位元組不等)的型別

- 在go語言中，字串常值都是用UTF-8來編碼，因為UTF-8是一種極受歡迎且應用廣的多位元組文字編碼標準

- 以string型別來說，它能儲存的文字並不局限於UTF-8編碼，因此在處理字串時可能需要額外的檢查才能避免錯誤

- 不同的編碼方式，會以不同數量的位元組來替文字編碼
- 舊式標準如ASCII只用一個位元組來編碼，UTF-8則最多會用到4個位元組
- 當文字以string型別儲存時，go語言會以byte集合來儲存所有的字串（string實際上便是唯讀的byte切片），這意味著有些UTF-8字元會被拆分成多個位元組
- 為了能安全的處理任何字串，不論其編碼方式是採用單一還是多位元組，最好是把字串從byte 集合轉換成rune集合

以go語言處理字串個別位元非常簡單, 請看下例 :

```go
package main

import "fmt"

func main() {
    username := "Sir_King_Über" //建立含有多重位元組字元的字串

    for i := 0; i < len(username); i++ { //走訪字串中的每一個位元
        fmt.Print(username[i], " ")  //印出一個字元加一個空格
    }
}
```

顯示結果

```
83 105 114 95 75 105 110 103 95 195 156 98 101 114
```

- 以上顯示的數值，都是字串中每個字元對應的編碼數值
- 我們定義的字串明明是13個字元，卻因為其中夾雜了一個由雙位元組編碼的文字Ü，導致列印出來有14個數值
- 底下我們再試著把這些位位元組數值轉回字串，這個動作會用到型別轉換，後面會再提到

```go
package main

import "fmt"

func main() {
    username := "Sir_King_Über"

    for i := 0; i < len(username); i++ {
        fmt.Print(string(username[i]), " ") //用string()把字元轉成文字印出來
    }
}

```

- 顯示結果：

```
S i r _ K i n g _ Ã  b e r
```

- 輸出的一開始都和原始字元一樣，直到 Ã 才有點不對勁

- 這是因為我們用函式將每個位元組轉回字元時，Ü兩位元的編碼 195 和 156 被拆開解讀，結果當然出錯

- 為了安全的處理字串，我們必須把byte型別的字串切片 轉成rune型別的切片

```go
package main

import "fmt"

func main() {
    username := "Sir_King_Über"
    runes := []rune(username) //將字串轉成rune切片

    for i := 0; i < len(runes); i++ {
        fmt.Print(string(runes[i]), " ") //將rune轉字串印出
    }
}
```

- 顯示結果：

```
S i r _ K i n g _ Ü b e r
```

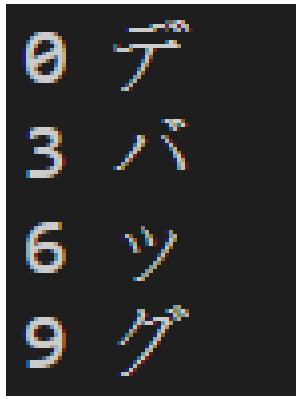
- 但是為甚麼可以用for I 走訪就能正確解讀字串呢？因為go語言編譯器發現你要走訪rune切片時，會自動轉成 for range迴圈，因此如果你真的要個別處理UTF-8字串的字元，將字串轉成rune切片，再用for range來走訪是最方便的

練習：安全的走訪一個字串

- 這裡要宣告一個以多重位元組編碼的字串，並用for range來走訪每個字元，把每個字元以及它在字串中的位元組索引印出

```go
package main

import "fmt"

func main() {
    logLevel := "デバッグ"
    for index, runeVal := range logLevel {
        fmt.Println(index, string(runeVal))
    }
}
```

- 顯示結果：



- 以上練習顯示，你能安全走訪一個使用多重位元組編碼的字串，而這種功能已經內建在go語言中

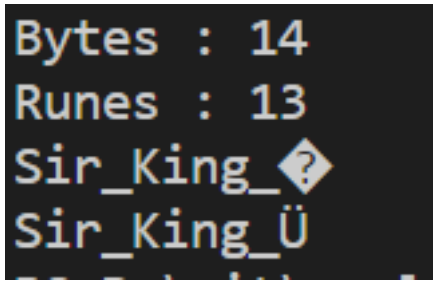
- 只要使用這種作法即可避免讀取無效的UTF-8字元

檢查字串長度

- 另一個常見的錯誤，是直接以 len()檢查字串中有幾個字元，以下就是錯誤示範：

```go
package main

import "fmt"

func main() {
    username := "Sir_King_Über"

    fmt.Println("Bytes :", len(username))          //取得字串長度 (位元組長度)
    fmt.Println("Runes :", len([]rune(username))) //取得rune集合長度
    //用切片讀取字串的前10個元素 , 立論上剛好到Ü
    fmt.Println(string(username[:10]))
    fmt.Println(string([]rune(username)[:10]))
}
```

- 顯示結果：



- 可以發現，若直接對字串(含有多位元組字元的)以len() 處理，得出的字元數顯然是錯的

- 所以每當在處理string變數，且需要計算長度或擷取特定數量的字元等等時，應該要先轉成rune切片

3-5 nil值

- 第一章提到過的nil其實不是一個型別，而是go語言的一個特殊資料值，代表一個無形別也無值的狀態，在處理指標、map及介面(interface，後面會介紹)以及error值時，都必須確認他們不是nil，若試圖拿一個nil值做運算，程式會掛掉

- 若你不確定某資料值是否為nil，可以這樣檢查：

```go
package main

import "fmt"

func main() {
    var message *string //沒有初始值的指標變數會是nil

    if message == nil {
        fmt.Println("錯誤, 非預期的 nil 值")
        return
    }
    fmt.Println(&message)
}
```

本章結束