CH2條件判斷與迴圈

2-1前言

- 條件判斷功能可以用來驗證使用者輸入的資料，是一種邏輯判斷的功能
- 迴圈則是另一種邏輯功能，可以用迴圈多次執行相同的程式碼
- 有了條件判斷與迴圈，軟體就能針對資料的變動表現出複雜的行為

2-2-1 if敘述

- If敘述會根據布林運算式的回傳值決定是否要執行某一區段的程式碼，語法如下：

```
If  布林運算式 {
        程式碼
}
```

- 若布林運算式的結果為true，便會執行程式碼
- if敘述只能在函式的範圍中使用

- 練習：檢查奇偶數

```go
package main

import (
    "fmt"
)

func main() {
    input := 5 //定義一個型別為int的變數

    if input%2 == 0 {  //檢查除以2的餘數是否為0
        fmt.Println(input, "是偶數")
    }
    if input%2 == 1 {
        fmt.Println(input, "是奇數")
    }
}
```

2-2-2 else敘述

else語法:

```
If  布林運算式 {
        程式碼1
} else {
        程式碼2
}
```

當if的布林運算式不成立時，else會執行

練習 : 使用else敘述

- 在這個練習中，我們把前面練習的內容改一下，換成 if … else 版本

```go
package main

import "fmt"

func main() {
    input := 4

    if input%2 == 0 {
        fmt.Println(input, "是偶數")
    } else {
        fmt.Println(input, "是奇數")
    }
}
```

2-2-3 else if 敘述

- 如果需要檢查兩個以上的布林運算式，但仍只會執行其中一段程式，else if 敘述就派上用場了

- 寫法：

```
If  布林運算式 {
        程式碼1
} else if  布林運算式 {
        程式碼2
} else {
        程式碼3
}
```

- 在開頭的第一個if敘述之後，你可以加入任何數量的else if 敘述，go語言會由上往下依序檢視含有布林運算式的敘述，直到找到第一個ture為止

- 要是沒有else區段，前面的布林運算式都為假，那go語言就不會執任何程式區段

練習：使用else if 敘述

- 再次拿前面的練習來修改，這次要加入富庶的檢查，且這個動作要在奇數和偶數之前進行

```go
package main

import (
    "fmt"
)

func main() {
    input := 10

    if input < 0 {
        fmt.Println("輸入值不得為負")
    } else if input%2 == 0 {
        fmt.Println(input, "是偶數")
    } else {
        fmt.Println(input, "是奇數")
    }
}
```

2-2-4 if 敘述的起始賦值

- 我們常拿函式的回傳值檢查是否執行正確 . 然後就再也不會用到該回傳值 , 但回傳值一旦接收了仍會存在其作用範圍內 , 等於多占了一份記憶體

- 為了避免這種浪費 , 我們可以把函數回傳的變數之作用範圍限制在if敘述範圍 , 這樣只要離開if敘述 , 該變數就會消滅 , 為了做到這點 , 方式就是在if敘述中加上”起始賦值敘述” :

```
If 起始賦值敘述; 布林運算式 {
        程式碼
}
```

- 起始賦值敘述用分號與布林運算式格開，布林運算式可以直接使用起始賦值敘述內宣告的變數來做判斷

- 注意：go語言只允許在起始賦值敘述中使用以下的簡單敘述:

1. 變數賦值和短變數宣告賦值，例如 I := 0
2. 算術或邏輯運算式，例如I := (j*10) == 40
3. 遞增或遞減，例如 I++
4. 在並行性運算中傳值給通道的敘述 (第16章介紹)

- 常見錯誤就是試圖在起始賦值敘述用var定義變數，這是不允許的，你只能用短變數宣告

練習:使用起始賦值敘述的if敘述

- 我們要擴充前面的練習，要加上更多的檢視規則：

1. 不能為負數
2. 不能大於100
3. 不能為7的倍數

- 有那麼多條件，若使用if … else if將會顯得不易閱讀，因此，我們把奇偶數檢查之外的條件全部都移進一個函式中，而函式檢查失敗時會回傳一個error值(第6章會詳談)

- 我們將在if敘述的起始賦值敘述呼叫此函式，並接收error然後檢查是否有錯誤發生，若沒有錯誤才會進行奇偶數檢查

```go
package main

import (
    "errors"
    "fmt"
)

func validate(input int) error {  //會回傳error的檢查函式
    if input < 0 {
        return errors.New("輸入值不得為負")
    } else if input > 100 {
        return errors.New("輸入值不得超過 100")
    } else if input%7 == 0 {
        return errors.New("輸入值不得為 7 的倍數")
    } else {
        return nil //檢查都通過時回傳nil回傳nil
    }
}
```

```
19
20  func main() {
21      input := 21
22      if err := validate(input); err != nil { //接收error並檢查是否有錯誤
23          fmt.Println(err)
24      } else if input%2 == 0 {
25          fmt.Println(input, "是偶數")
26      } else {
27          fmt.Println(input, "是奇數")
28      }
29  }
30
```

顯示結果：

```
輸入值不得為 7 的倍數
```

- 在以上練習中，運用了啟式賦值敘述來定義了一個變數err且對它賦值，然後用在if敘述的布林運算式中

- 一旦main()的if … else if …else 完成任務，err變數就會離開作用範圍並被go語言記憶體管理系統回收

2-3 switch 敘述

2-3-1 switch 敘述基礎

- 如果面臨需要一堆if才能處理的狀況，這時可以使用switch 來精簡程式碼

- 基本語法如下：

```
switch 起始賦值敘述; 運算式 {
case 運算式 :
    程式碼
case 運算式, 運算式:
    程式碼
    fallthrough
....
default :
    程式碼
}
```

- 在上述的基本語法中，起始賦值敘述和運算式都是非必要的，若沒有寫運算式，效果等於 switch true

- Switch 底下的case 是用來檢查要執行的條件，case 的運算式有兩種寫法：
  1. 布林運算式
  2. 直接寫一個值，這個值會跟swich自身運算式的值做比較，若兩值相等則進行底下的程式碼

- 當檢查到符合條件的case，go語言只會進行對應的程式碼並離開switch，若想要所有符合條件的case都執行，那這些case所屬的敘事結尾就要加上fallthrough敘述，讓go語言執行到那行時繼續往下看，並檢查下一個case

- default的用法則和if的else一樣，在所有case都不成立時，就會執行default

- 以上switch 的形式俗稱 ”運算式switch”，還有另一種形式的switch稱為 “型別switch”，留至第四章再介紹

- 練習: 使用switch 敘述

- 要撰寫一支程式，可以根據今天是星期幾印出一段訊息，我們要利用time套件來取得一周七天名稱的常數，然後用switch 敘寫程式

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    day := time.Monday //定義變數並設定為星期的某一日
    switch day { //比對變數幾是星期
    case time.Monday:
        fmt.Println("星期一，猴子穿新衣")
    case time.Tuesday:
        fmt.Println("星期二，猴子肚子餓")
    case time.Wednesday:
        fmt.Println("星期三，猴子去爬山")
    case time.Thursday:
        fmt.Println("星期四，猴子去考試")
    case time.Friday:
        fmt.Println("星期五，猴子去跳舞")
```

```
21      case time.Saturday:
22          fmt.Println("星期六，猴子去斗六")
23      case time.Sunday:
24          fmt.Println("星期日，猴子過生日")
25      default:
26          fmt.Println("日期不正確")
27      }
28  }
```

顯示結果:

星期一，猴子穿新衣

2-3-2 switch 的不同用法

練習 : switch 敘述與多重case配對值

- case後面的值或運算式 , 其實想要寫幾條都可以 , 只要用逗號分開即可 , go語言會由左到右檢查這些值或算式

- 下面的練習要來判斷某人的生日是工作天還是周末 , 並依此印出訊息 (只需要2個case , 因為一個case可以檢查多個值)

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    dayBorn := time.Sunday
    switch dayBorn {
    case time.Monday, time.Tuesday, time.Wednesday, time.Thursday, time.Friday:
        fmt.Println("生日為平日")
    case time.Saturday, time.Sunday:
        fmt.Println("生日為周末")
    default:
        fmt.Println("生日錯誤")
    }
}
```

練習 : 沒有運算式的switch 敘述

- 有時需要同時比對不同變數的值 , 或是更複雜的條件判斷 , 例如判斷某變數是否位於特定範圍等等
- 你仍可以用switch寫出精簡的條件敘述 , 因為case運算式能做到的程度就跟if的布林運算式一樣
- 這次的練習一樣要檢查生日是否在周末 , 但會簡化switch敘述 , 改用case本身來作條件判斷

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    switch dayBorn := time.Sunday; { //只有起始賦值敘述
    case dayBorn == time.Sunday || dayBorn == time.Saturday:
        fmt.Println("生日為周末")
    default:
        fmt.Println("生日非周末")
    }
}
```

2-3-3 fallthrough

- 使用 fallthrough 會强制執行後面的 case 語句，fallthrough 不會判斷下一條 case 的表達式结果是否為 true

- 底下看例子:

```go
package main

import "fmt"

func main() {
    switch {
    case false:
        fmt.Println("1、case 條件為 false")
        fallthrough
    case true:
        fmt.Println("2、case 條件為 true")
        fallthrough
    case false:
        fmt.Println("3、case 條件為 false")
        fallthrough
    case true:
        fmt.Println("4、case 條件為 true")
    case false:
        fmt.Println("5、case 條件為 false")
        fallthrough
    default:
        fmt.Println("6、默認 case")
    }
}
```

- 顯示結果：

```
2、case 條件為 true
3、case 條件為 false
4、case 條件為 true
```

2-4廻圏

2-4-1 for迴圈基礎

- Go 語言只支援for迴圈，但它非常彈性
- for迴圈的寫法分幾種，第一種常用來處理有序的集合，像是陣列跟切片等等，我們在下一章會再談到這類的資料型別
- 這種迴圈在處理有序集合時是這樣寫的：

```
for 起始賦值敘述; 條件敘述; 結束敘述 {
    程式碼
}
```

以上寫法中:

- 起始賦值敘述就跟if和switch中的一樣，同樣可以接受簡單敘述
- 條件敘述會在迴圈每一次執行時檢查，成立時繼續迴圈，同樣可以接受簡單敘述
- 結束敘述則是在迴圈跑完一輪後才執行

- For迴圈的起始賦值敘述 / 條件敘述 / 結束敘述 都可以省略 , 因此最簡單的for迴圈可以寫成如下 :

```
for {
        程式碼
}
```

- 這樣就相當於 for true , 會形成無窮迴圈 , 除非用 break 中斷

- 上述for迴圈的寫法還有另一種變形，就是從一個來源讀取資料，然後回傳一個布林值，讓迴圈判斷是否還有資料需要讀取

- 使用這種迴圈的例子包刮從資料庫 / 檔案 / 命令列輸入 / 網路socket 讀取資料等等

- 這種迴圈的格式如下：

```
for  條件敘述 {
        程式碼
}
```

其實就是第一種迴圈的簡化版，但你不用定義結束迴圈的條件，因為讀取資料來源本身就已經會回傳布林值，在讀取完畢後回傳false

- 最後一種for是用來走訪無序或長度不確定的資料集合，例如映射表(maps)，第四章會再詳細介紹map集合

- 在走訪這類資料集合時，我們會改用range敘述：

```
for 鍵, 值 := range 集合 {
    程式碼
}
```

2-4-2 for i 廻圏

練習 : 使用for i 迴圈

- 這次練習要運用組成for迴圈的三個部分 , 於迴圈中建立並使用一個變數 :

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       //在迴圈建立變數i , 初始值為0 , 在i小於5時繼續並重複迴圈
7       //每次迴圈結束後i增加1
8       for i := 0; i < 5; i++ {
9           fmt.Println(i)
10      }
11  }
```

- 顯示結果：

```
0
1
2
3
4
```

- 這樣的迴圈，很常用來處裡用數字做為索引的有序集合，諸如陣列和切片都是如此

練習 : 用for迴圈走訪切片元素

- for迴圈的運作其實也適用於擁有類似內容的陣列;下一章會更深入介紹陣列和切片差異
- 我們先定義一個切片，然後寫一個迴圈，並利用這個集合本身的長度來控制迴圈何時停止
- 此外，迴圈內也會用一個變數(俗稱計數器變數)來追蹤我們處理到變數的哪個元素
- 陣列和切片的索引一定是連續遞增，且永遠從0算起，而內建函式 len()能取得任何集合的長度，我們可以利用它的回傳值來檢查迴圈是否已經走訪到集合尾端

```go
package main

import "fmt"

func main() {
    names := []string{"Jim", "Jane", "Joe", "June"}
    for i := 0; i < len(names); i++ {
        fmt.Println(names[i])
    }
}
```

顯示結果：

```
Jim
Jane
Joe
June
```

2-4-3 for range 廻圏

- 另一種型式的資料集合 map , 其鍵與值不一定照順序排列 , 這表示我們得用range來取代原本迴圈裡的條件敘述

- Range每次都會從集合裡取出一個鍵與值 , 下一輪迴圈就換下一組

- 練習：利用迴圈走訪map元素

- 在這個練習中，我們要建立一個map集合，其鍵與值皆由字串構成

- 我們會在for迴圈利用range來走訪整個map，然後把其鍵與值列印出來

```go
package main

import "fmt"

func main() {
    config := map[string]string{ //建立map, 元素由一對鍵與值構成
        "debug":    "1",
        "logLevel": "warn",
        "version":  "1.2.1",
    }

    for key, value := range config { //走訪map並逐次取出鍵/值
        fmt.Println(key, "=", value)
    }
}
```

顯示結果

```
version = 1.2.1
debug = 1
logLevel = warn
```

補充：range鍵或值的省略，以及陣列/切片的走訪順序

- 如果在迴圈中用不到key或value變數，可以在接收時寫成底線字元告知編譯器你不需要它：

```
for _, value := range config { //不要鍵，只要值
    fmt.Println(value)
}
```

- 若只想要鍵，則可以直接不寫第二個變數：

```
for key := range config { //不要值，只要鍵
    fmt.Println(key)
}
```

- Range也可以用在切片或陣列上，此時鍵就是元素索引，值就是元素值：

```
names := []string{"Jim", "Jane", "Joe", "June"}
for i, value := range names {
    fmt.Println("Index", i, "=", value)
}
```

2-4-4 break 和 continue 敘述

- 可能會遇到某些場合，需要跳過一輪迴圈，甚至停止整個迴圈，這時有兩個選擇：

1. 關鍵字continue：立即跳至下一輪的迴圈，迴圈的結束敘述仍會執行

2. 關鍵字break：立即離開整個迴圈

練習：利用break和continue來控制迴圈

- 下面會隨機產生介於0~8之間的數字，迴圈要略過任何3的倍數，若是偶數就結束迴圈，程式也會印出每一輪迴圈處理到的變數 i，以便觀察

```go
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    for {
        r := rand.Intn(8) //產生0~8整數亂數
        if r%3 == 0 {     //若亂數是三的倍數則跳過這輪迴圈
            fmt.Println("略過")
            continue
        } else if r%2 == 0 { //若亂數是偶數則跳出迴圈
            fmt.Println("跳出")
            break
        }
        fmt.Println(r)
    }
}
```

本章結束