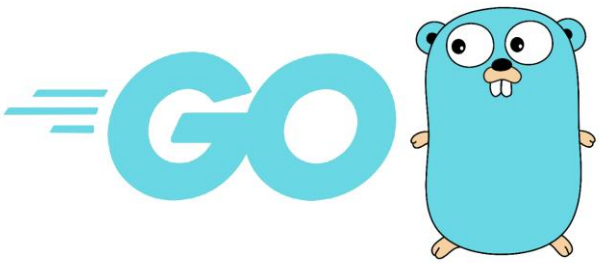
字串處理



字串處理

字串連接(組合字串)

- fmt.Sprint、fmt.Sprintln、fmt.Sprintf
  - fmt.Sprint 不會印出東西，基本上是拿來組合字串用的，組出來之後，需要一個變數去接。
  - 用fmt.Sprint組字串，比起操作字元、陣列，可讀性會較高一些。

```
func main()  {
    s1 := "I"
    s2 := "am"
    s3 := "string"

    str1 := fmt.Sprintln(s1, s2, s3)
    fmt.Println(str1)

    str2 := fmt.Sprint(s1, s2, s3)
    fmt.Println(str2)
}
/* result:
I am string

Iamstring

*/
```

https://play.golang.org/p/WgcVnCvfmBH

字串處理

字串分割

```go
package main

import (
    "fmt"
    "strings"
)

func main() {
str := "Hello, World"
str1 := strings.Split(str, ",")

fmt.Println(str1) // [Hello  World]
fmt.Println(str1[0]) // Hello
fmt.Println(str1[1]) // World
}
```

- 記得要加strings

https://go.dev/play/p/knoKhha5zaH

字串處理

字串取代

- 字串替換, -1表示全部取代

```
str := "Hello Tom"

str1 := strings.Replace(str, "Tom", "May", -1)
fmt.Println(str1) // Hello May
```

```
str := "Hello Tom Tom Tom"
str1 := strings.Replace(str, "Tom", "May", -1)
fmt.Println(str1) // Hello May May May
```

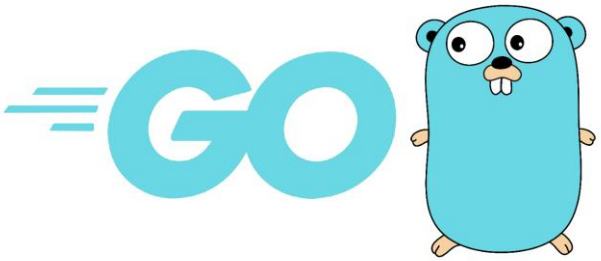
- 字串替換, 2表示替換二個，以此類推!

```
str := "Hello Tom Tom Tom"

str1 := strings.Replace(str, "Tom", "May", 2)
fmt.Println(str1) // Hello May May Tom
```

- 如果該變數沒有使用是會報錯的，RUN ERROR的時候可以確認一下

https://go.dev/play/p/ajS7MHhs21M https://go.dev/play/p/OLeo6GjrfW6 https://go.dev/play/p/Ltq0lhlorjH

流程控制



流程控制

在程式裡，每一行程式碼的執行順序就稱為流程，一般的程式流程是由程式碼的編寫順序從上而下依序執行。但是當開發較複雜的程式時，常常需要選擇性的執行某行程式碼或是重複執行某一段程式碼，這樣的行為就稱為流程控制。流程控制主要分為**分支語法**和**循環語法**兩種。

再來進到 go流程控制的部分，go 因為保留字很少，相對的邏輯控制也相當簡單。

正式來說有四種邏輯控制結構分別是：

- if
- switch
- for
- goto

流程控制

分支語法：

- 分支是指用條件來選擇執行某一段程式碼，go語言提供 **if/else** 和 switch case 兩種語法。
- if / else
  - if/else可以說是所有程式最常用的邏輯判斷語法，go的用法也差不多，只是**條件不再需要小括號( )，然後左大括號 { 必須要和 if 同一行，以及 else 和 else if 則必須和右大括號和左大括號同一行**。

```
if condition {
    // run code here.
}
```

```
if condition {
    // run code here.
} else{
    // run code here.
}
```

```
if condition {
    // run code here.
} else if condition {
    // run code here.
} else {
    // run code here.
}
```

```
myAge := 30
if myAge < 13 {
    fmt.Println("Child")
} else if myAge >= 13 && myAge < 20 {
    fmt.Println("Teen")
} else {
    fmt.Println("Adult")
}
// 執行結果: Adult
```

  - **如果大括號位置放不對，編譯就會錯誤，go語言在撰寫風格上規定的很嚴格，而且是強制性的。**

流程控制

分支語法：

- 分支是指用條件來選擇執行某一段程式碼，go語言提供 if/else 和 switch case 兩種語法。
- if / else
  - 條件也可以直接放一個布林變數或是呼叫一個回傳布林的函式：

```
isHuman := true
if isHuman {
    fmt.Println("walk")
} else {
    fmt.Println("fly")
}
// 執行結果: walk
```

```
myAge := 30
if myAge = myAge - 15; myAge < 20 {
    fmt.Println("Teen")
} else {
    fmt.Println("Adult")
}
// 執行結果: Teen
```

```
if myAge := 30; myAge < 20 {
    fmt.Println("Teen")
} else {
    fmt.Println("Adult")
}
// 執行結果: Adult
```

  - go語言的 if 還有一個特殊寫法，可以在條件之前加上一行表達式：
  - 這個條件前的表達式還可以透過 := 宣告變數，後面的條件可以拿來使用。

流程控制

分支語法：

- 分支是指用條件來選擇執行某一段程式碼，go語言提供 if/else 和 switch case 兩種語法。
- if / else
  - if 有一個很方便的方法，可以呼叫func直接在一行裡面取得值，並對值做邏輯判斷，蠻多情境下，可以讓code更清爽。

```go
package main

import (
    "fmt"
)

func Test() int {
    return 5
}

func main() {
    if a := Test(); a >= 5 {
        fmt.Println(a)
    } else {
        fmt.Println("no hit")
    }
}
```

https://play.golang.org/p/rxHltWNF5oZ

流程控制

```
switch v {
    case a, b:
        // run 1_block code here.
    case c:
        // run 2_block code here.
        fallthrough
    case d:
        // run 3_block here.
    default:
        // run 4_block here.
    }
```

分支語法：

- 分支是指用條件來選擇執行某一段程式碼，go語言提供 if/else 和 **switch case** 兩種語法。
- switch case
  - switch case也是一種很常見的語法，實際上它能做的事換成用 if/else 一樣也做得到，而會用switch case的目的也就是為了美觀，排版整齊、條件可以寫的比較少。
  - 試著想想看當 if 和 else if 越來越多時，如下：

```
myAge := 30
if myAge < 13 {
    fmt.Println("Child")
} else if myAge >= 13 && myAge < 20 {
    fmt.Println("Teen")
} else if myAge >= 20 && myAge < 50 {
    fmt.Println("Adult")
} else {
    fmt.Println("Elder")
}
// 執行結果: Adult
```

```
myAge := 30
switch {
case myAge < 13:
    fmt.Println("Child")
case myAge >= 13 && myAge < 20:
    fmt.Println("Teen")
case myAge >= 20 && myAge < 50:
    fmt.Println("Adult")
default:
    fmt.Println("Elder")
}
// 執行結果: Adult
```

  - 可以改成用 switch case 就會變得簡潔許多：

流程控制

分支語法：

- 分支是指用條件來選擇執行某一段程式碼，go語言提供 if/else 和 switch case 兩種語法。
- switch case
  - 上面的範例可以看出go的switch case比較特殊，允許case帶條件式。一般來說，在C語言與其他大部份語言中的switch case是只能帶一個值，go也有保留這樣的寫法：

```go
flag := 3
switch flag {
case 1:
    fmt.Println("First")
case 2:
    fmt.Println("Second")
case 3:
    fmt.Println("Third")
default:
    fmt.Println("Other")
}
// 執行結果: Third
```

```go
flag := 1
switch flag {
case 0, 1:
    fmt.Println("Zero - First")
case 2, 3, 4:
    fmt.Println("Second - Four")
default:
    fmt.Println("Other")
}
// 執行結果: Zero - First"
```

  - 傳統上，case只能帶值的寫法，讓switch case少了很多出場機會。而go語言改良了許多，除了現在看到的case帶條件以外，**還可以帶多個值：**
  - 最後，那就是每個**case結束後不用再寫 break 了**，go語言的case執行完就會直接離開switch block，這真是一個非常棒的改良。因為大部分的情境裡， case完成後都是要直接離開的，在C語言裡就要用 break 才能跳出switch。

流程控制

分支語法：

- 分支是指用條件來選擇執行某一段程式碼，go語言提供 if/else 和 switch case 兩種語法。
- switch case
  - default：如果都不在case裡的情況就會執行default

```
myfriend := "Jason"
switch myFriend {
    case "Amy":
        fmt.Println("Hi, Amy")
    case "Tony":
        fmt.Println("Hi bro")
    case "Jackey":
        fmt.Println("GO AWAY!")
    default:
        fmt.Println("Nice to meet you, but who are you?")
}

輸出結果：Nice to meet you, but who are you?
```

流程控制

分支語法：

- 分支是指用條件來選擇執行某一段程式碼，go語言提供 if/else 和 switch case 兩種語法。
- switch case
  - fallthrough：在case中加入fallthrough，會接著執行下一個case，假設看到Amy的時候通常會看到Tony

```
myfriend := "Amy"
switch myFriend {
    case "Amy":
        fmt.Println("Hi, Amy")
        fallthrough
    case "Tony":
        fmt.Println("Hi bro")
    case "Jackey":
        fmt.Println("GO AWAY!")
    default:
        fmt.Println("Nice to meet you, but who are you?")
}

輸出結果：Hi, Amy
Hi bro
```

流程控制

分支語法：

- 分支是指用條件來選擇執行某一段程式碼，go語言提供 if/else 和 switch case 兩種語法。
- switch case
  - 多重情況：如果多種case的處理方式都一樣，可以這樣寫

```
myfriend := "Paul"
switch myFriend {
    case "Amy", "Emily",:
        fmt.Println("Hi, beautiful gril")
    case "Tony", "Paul":
        fmt.Println("Hi bro")
}

輸出結果：Hi bro
```

流程控制

分支語法：

- 分支是指用條件來選擇執行某一段程式碼，go語言提供 if/else 和 switch case 兩種語法。
- switch case
  - 沒有switch的對象：這個情況就有點像是if的功能了

```
myMoney := 100
switch {
    case myMoney > 500:
        fmt.Println("buy Ferrari")
    case myMoney > 250:
        fmt.Println("buy BMW")
    default
        fmt.Println("buy Toyota")
}

輸出結果：buy Toyota
```

流程控制

循環語法

- 循環是指重複執行一段程式碼，直到滿足條件後才結束。如果要讓程式碼重複執行多次會有兩個作法，**一個是迴圈，另一個是遞迴**。差別在於迴圈是用for關鍵字實現，而遞迴是用函式呼叫來實現。
- for
  - 通常稱作for迴圈(for loop)，基本語法如下：

```
for i := 0; i < 10; i++ {
    fmt.Printf("%d", i)
}
```

  - 在迴圈之中可以透過 i++ 語法，在每次執行完成時，對變數 i 自動加 1，或是想要相反就用i-- 語法，每次減少 1。**注意一點，go語言沒有提供 ++i 之類的語法**。
  - 回想一下C語言，還有看過另一個跑迴圈關鍵字 while，但是在go語言中沒有它。但可以用for做出類似的寫法：

```
i := 0
for i < 10 {
    fmt.Printf("%d", i)
    i++
}
```

```
for {
    // 用 break 跳出迴圈
}
```

  - 或是用for做一個無窮迴圈(沒有結束條件)：

流程控制

循環語法

- 循環是指重複執行一段程式碼，直到滿足條件後才結束。如果要讓程式碼重複執行多次會有兩個作法，一個是迴圈，另一個是遞迴。差別在於迴圈是用for關鍵字實現，而遞迴是用函式呼叫來實現。
- for
  - 可以用雙重迴圈來完成更多事，下面的範例是用兩個for印出九九乘法表：

```
for i := 1; i <= 9; i++ {
    for j := 1; j <= i; j++ {
        fmt.Printf("%d*%d=%d ", i, j, i*j)
    }
    fmt.Println()
}
```

```
1*1=1
2*1=2 2*2=4
3*1=3 3*2=6 3*3=9
4*1=4 4*2=8 4*3=12 4*4=16
5*1=5 5*2=10 5*3=15 5*4=20 5*5=25
6*1=6 6*2=12 6*3=18 6*4=24 6*5=30 6*6=36
7*1=7 7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49
8*1=8 8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64
9*1=9 9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
```

流程控制

循環語法

- 循環是指重複執行一段程式碼，直到滿足條件後才結束。如果要讓程式碼重複執行多次會有兩個作法，一個是迴圈，另一個是遞迴。差別在於迴圈是用for關鍵字實現，而遞迴是用函式呼叫來實現。
- for迭代
  - 能夠遍歷整個物件變數，另外用for迭代的話，就不能在裡頭修改(新增、刪除)迭代物件，會容易出問題，若真要修改物件，請用前面for迴圈的方式。

```
for a, b := range Iterable{
...
}
```

```go
nums := []int{100, 99, 98}
for index, num := range nums {
    fmt.Println(index, num)
}
for index := range nums {
    fmt.Println(index)
}

//--------

fruits := map[string]string{"a": "apple", "b": "banana"}
for index, fruit := range fruits {
    fmt.Println(index, fruit)
}
for index := range fruits {
    fmt.Println(index)
}
```

https://play.golang.org/p/2N6FrtbpB6f

流程控制

循環語法

- break搭配迴圈
  - 在迴圈中，常常會搭配 break 做使用，可依條件需求讓迴圈提早跳出結束，看一下以下程式碼:

```go
package main

import "fmt"

func main() {
   sum := 0

   for i := 1; i <= 100; i++ {
       sum += i
       if sum >200 {
           fmt.Println(i)
           break
       }
   }
}
```

```go
i := 0
for {
    do something
    if i >= 100 {
        break;
    }
    i++
}
```

無條件無窮for迴圈，
當然最後最好有個終止條件break 。

雖然計數器的條件為小於等於100，但依迴圈裡的條件，
sum 的值在大於200時，會自動 break，其結果輸出為20。

流程控制

循環語法

- continue搭配迴圈
  - 上面提到了 break，大家一定也會想到 continue，其使用方法，為若符合條件，便會跳過到此迴圈，直接進入下個迭代：

```go
package main

import "fmt"

func main() {
   for i := 1; i <= 10; i++ {
       if i%2 == 0 {
           continue
       }

       fmt.Println(i)
   }
}
```

```go
for {
    i++
    if i%2 == 0 {
        continue
    }
    fmt.Print(i)
    if i > 50 {
        break
    }
}
```

這個範例是為了印出小於10正積數，當計數器為偶數時，
判斷後會直接跳到下一個迴圈，不會執行印出的動作。

流程控制

跳躍語法：goto (跳到同函式中某個位置)

- 除了這三種控制語法，go語言還有提供一種邪惡的語法，叫做 goto。goto是一種來自低階語言的流程控制語法，並且已經被**一些高階語言(java)移除或禁止使用**。
- goto這個使用方式看起來是頗具爭議的，有些人認為使用goto會讓程式的結構變得複雜(因為會直接跳離)，但也有人主張適時的使用goto反而可以讓程式碼更簡潔，這部分就見仁見智!

```
package main

import "fmt"

func main() {

   a := 10


   LOOP: for a < 20 {
      if a == 15 {
         a = a + 1
         goto LOOP
      }
      fmt.Println(a)
      a++
   }
}
```

流程控制

產生隨機整數

- Go可使用標準函式庫math/rand package的Intn(n int)函式產生0到n的隨機整數。
- 不過直接使用rand.Intn(n int)會發現每次執行時產生的亂數都一樣，例如下面總是印出1。
- 這是因為偽隨機生成器(pseudo-random-generator)的種子(Seed)都是同一個，預設為種子值為1。

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println(rand.Intn(10))
}
```

流程控制

產生隨機整數

- 而程式執行中持續用rand.Intn(n int)產生的隨機數就會不同，但每次重新執行會發現產生的亂數序列是一樣的，原因也是使用同一個種子。例如下面每次執行產生的亂數序列都是 [1 7 7 9 1]。

```go
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    for i := 0; i < 5; i++ {
        fmt.Println(rand.Intn(10))
    }
}
```

流程控制

產生隨機整數

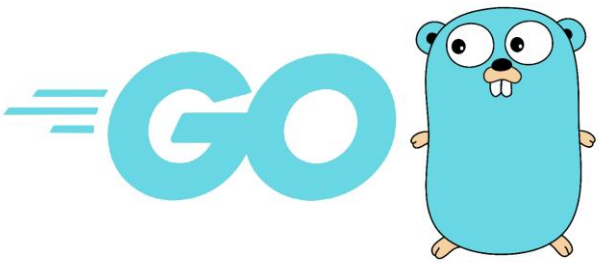
- 因此要讓每次執行程式時產生不同的隨機數列，就要餵給不同的種子值。使用rand.Seed(seed int64)設定全域種子，傳入time.Now().UnixNano()以現在時間的Unix時間作為種子值。例如：下面每次執行時產生的隨機數列都不同。

```go
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    rand.Seed(time.Now().UnixNano())
    for i := 0; i < 5; i++ {
        fmt.Println(rand.Intn(10))
    }
}
```

```go
min := 10
max := 20
n := rand.Intn(max-min) + min // n is random int between 10-20
fmt.Println(n)
```

- 產生特定區間的整數亂數寫法如下，例如產生10到20間的亂數。

常數與函式



常數與函式

```
const <常數> <型別> = <值>

const (
    <常數 1> <型別 1> = <值 1>
    <常數 2> <型別 2> = <值 2>
    ...
    <常數 N> <型別 N> = <值 N>
)
```

常數(constant)

- 在開發或編譯期間就定義好的值，在執行期間不能被修改。
- 但用法跟變數var一樣，只是把關鍵字 var 換 const，後面可以接任何型別，名稱通常為**全大寫**
- 宣告常數時，初始值是**『必要的』**，型別可有可無，不指定型別，Go語言會自行推斷

```
const PI float64 = 3.1415927
r := 10.0
```

```
// 多重宣告
const (
    pi = 3.141592
    e = 2.718281
)
```

- 常數可以用表達式指定內容，以及在陣列宣告時指定大小：

```
const pageSize = 5
const totalPage = 20
const totalSize = pageSize * totalPage

var arr [totalSize]int

fmt.Println(len(arr))
//印出: 100
```

常數與函式

常數(iota)

- iota是希臘字符，在Golang中是關鍵字之一，用在宣告常數中，**效果為數字遞增**，
  iota本身數值從0開始，不用手動打數字0、1、2、3...重複且無聊的事情。

```
const (
    A = iota        // 0
    B               // 1
    _               // 2 佔位符也會被計算
    C               // 3
    D = iota * 0.1 // 0.4 接續前面的 iota
    E               // 0.5
    F               // 0.6
    G               // 0.7
)
```

```
const (
    X = iota + 100 // 100
    Y              // 101
    Z              // 102
)
```

```
const (
    b1 = 1 << iota // 1   右側被塞入0個bit (2^0 二的零次方)
    b2             // 2   右側被塞入1個bit (2^1 二的一次方)
    b3             // 4   右側被塞入2個bit
    b4             // 8
    b5             // 16
)
```

- 起始值也不一定要從0開始；也常被拿來作左移右移(Shift Bit) 運算

https://go.dev/play/p/jVOzeLxu-mB

常數與函式

列舉常數

- go語言中**沒有定義列舉(enum)的功能**，但是可以用**常數和iota關鍵字**達到相似的效果。

```go
// 先定義一個int別名的型別Hero
type Hero int

const (
    IronMan Hero = iota
    DrStrange
    Thor
    Hulk
)

// 使用列舉常數賦值
man := IronMan

fmt.Println(IronMan, DrStrange, Thor, Hulk, man)
// 印出: 0 1 2 3 0
```

```go
type Hero int

const (
    IronMan Hero = iota*2 + 1
    DrStrange
    Thor
    Hulk
)

fmt.Println(IronMan, DrStrange, Thor, Hulk)
// 印出: 1 3 5 7
```

常數與函式

golang 不是一種 OOP(Object-oriented programming)，反而比較偏向 FP(Functional Programming)，擁有 First-class function 的特性，所以可以用一些 script language 常見的手法，使用在 golang 上面。

Go語言func命名是採用**駝峰式**大小寫，譬如：HowDoYouTurnThisOn。**字首大寫代表可對外使用，字首小寫代表此func只能在相同的Package中才能使用**。熟悉的main就是一個func，程式的進入點。

Function 宣告(func)：一般來說，函式具有4個部份，分為：函式名稱、參數、回傳值以及主體

```
func foo(name string) {
    fmt.Println("Hi " + name)
}
```

- 其中括號內分別為傳入值名稱、型態，**剛好跟常見的程式語言顛倒**。如果沒有回傳值就空著，不用填null, nil之類的

常數與函式

Function 宣告(func)：多個傳入值

```
func foo(name1 string, name2 string) {
    fmt.Println("Hi " + name1 + ", " + name2)
}
```

- 如果兩個值型態相同可以這樣：

```
func foo(name1, name2 string) {
    fmt.Println("Hi " + name1 + ", " + name2)
```

```
package main

import "fmt"

func add(a int, b int) int {
    return a + b
}

func main() {
    fmt.Println(add(3, 2))
}
```

https://play.golang.org/p/AQ9ZaPECNYk

常數與函式

Function 宣告(func)：多個傳入值

- 如果實在不知道該傳入幾個變數也可以這樣：

```
func total(x ...int) int {
    var t int
    for _, n := range x {
        t += n
    }
    return t
}
```

- 關於for的第一個參數為底線是Blank identifier。

常數與函式

Function 宣告(func)：return

- 跟很多程式語言先寫回傳值型態又相反，**回傳型態是寫在括號後面的**。

```
func foo(name string) string {
    var str = "Hi " + name
    return str
}
```

Function 宣告(func)：return命名

- 還可以顯示的命名回傳值，直接在回傳值上宣告要命名的變數名稱，同樣括號內前者是名稱，後者是型態。最後在 return 時，就不用刻意指定要回傳哪個變數回去了。

```
package main

import "fmt"

func add(a int, b int) (c int) {
    c = a + b
    return
}

func main() {
    fmt.Println(add(3, 2))
}
```

https://play.golang.org/p/s0L2JHXYD6X

常數與函式

Function 宣告(func)：多重return

```
func foo(x, y int) (int, int) {
    return x + y, x - y
}
```

- 還可以**多重回傳值**，此手法在 **golang error handling**(後續會介紹)很常見，約定俗成來說，**如果有 error 要回傳，最右邊的變數為 error**。下面範例：回傳一個 int和一個 string。

```
package main

import "fmt"

func Info() (age int, name string) {
    age = 18
    name = "syhlion"
    return
}

func main() {
    age, name := Info()
    fmt.Printf("age:%d, name:%s\n", age, name)
}
```

```
func foo(a int, b string) (string, int) {
    return b + "2", a + 2
}

func main() {
    b, a := foo(2020, "Iron Man")
    fmt.Printf("%d %s\n", a, b)
    // 印出: 2022 Iron Man2
}
```

- 不想全部都要的話，也是可以忽略掉部份回傳值，用底線(_)替代變數：

```
_, a := foo(2020, "Iron Man")
```

https://play.golang.org/p/pVu38aHJCBE

常數與函式

Function 宣告(func)：函式變數

- 函式也是一種資料型別，可以被當作值相互傳遞。以下舉個例子來說明：

```go
func add(a, b int) int {
    return a + b
}

func minus(a, b int) int {
    return a - b
}

func addAndMinus(a, b int) (int, int) {
    return a + b, a - b
}

func main() {
    var foo func(a, b int) int

    foo = add
    foo = minus
    // foo = addAndMinus 這一行會錯誤

    a := foo(1, 2)
    fmt.Printf("%d\n", a)
    // 印出: 3
}
```

看到範例宣告一個叫做foo的函式變數，而函式型別之間是用**參數與回傳值的不同來做區分**。因此可以看到foo是一個有2個整數參數和一個整數回傳值的函式變數，只要是**符合這樣條件的函式都可以被賦值給它**，像是上面的add和minus，而若是賦值addAndMinus就會錯誤。

當foo函式變數被賦值後，就可以像一般函式那樣直接呼叫它，例如：foo(1, 2)。但是如果沒有賦值，函式變數預設值為nil，呼叫它會發生錯誤：

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x48e554]
```

這個錯誤簡單來說，就是使用的記憶體位置(變數)沒有定義預期的資料。

https://go.dev/play/p/WR5uBodlkFb

常數與函式

Function 宣告(func)：函式變數(function回傳function)

- 上個提到function可以當變數，所以當然也可以用來回傳：

```
func foo() func() string {
    return func() string {
        return "I don't know what is this mean"
    }
}
```

常數與函式

Function 宣告(func)：匿名函式(當變數宣告)

- 有時候會覺得定義函式名稱是一件麻煩的事，特別是在臨時要宣告一個函式的時候。可以直接定義一個匿名函式，然後丟給一個函式變數或是一個接受函式變數的函式。
- 一個類似變數的寫法：

```
foo := func() {
    fmt.Println("Hi " + name)
}
```

- 當然這個function的名稱就是foo：

```
// foo函式接收2個整數和一個函式變數
func foo(a, b int, f func(a, b int) int) int {
    return f(a, b)
}

func main() {
    var add = func(a, b int) int {
        return a + b
    }

    a := foo(1, 2, add)
    fmt.Printf("%d\n", a)
    // 印出: 3
}
```

https://go.dev/play/p/LSh6NbmRM_9

常數與函式

函式注意事項

1. 在使用函式時，只要一觸發 return ，那麼後方的程式碼就不會繼續執行
2. 有回傳值的函式不一定要去接收回傳值
3. 函式中所宣告的變數有所謂的 scope，不會影響其他函數中所設的變數
4. 有規定回傳值型態的函式，在函數最尾一定要使用 return
5. 函式擺放的順序不重要

常數與函式

函式注意事項

- reutrn 後的程式碼並不會執行，有回傳值的函式不一定要去接收回傳值

```
package main
import "fmt"

func test(num int) int{
    if num > 10{
        fmt.Println("num > 10")
        return num                  // 第 7 行
    }
    fmt.Println("num < 10")
    return num
}

func main(){
    test(100)    // 第 14 行
}
```

執行結果：

num > 10

很明顯可以看出，當執行到第 7 行後，就會跳回第 14 行，後方的程式碼就不會再執行了。只要在函式中遇到 return ，就不會繼續將後方的程式。

另一方面，有回傳值(不管有幾個回傳值)的函數並沒有強制一定要有接收他的變數。

https://go.dev/play/p/luSHO3bFhVY

常數與函式

函式注意事項

- 函式中所宣告的變數有所謂的 scope

```go
package main
import "fmt"

func test(num int){
    fmt.Println("in test()", num)
}

func main(){
    num := 10
    test(100)
    fmt.Println("in main()", num)
}
```

執行結果：

```
in test() 100
in main() 10
```

雖然 main() 和 test() 中都是使用 num 但是兩者完全不會受到影響，這就是所謂的 scope，如此一來可以更方便使用變數，使用時只要顧慮自己的 scope 即可

https://go.dev/play/p/UNOgQ5zK2mY

常數與函式

函式注意事項

- 有規定回傳值型態的函式，**在函數最尾一定要使用 return**
- 許多程式語言不會檢查這個錯誤，但是Golang會。首先改造一下前面範例的函式：

```go
package main
import "fmt"

func test(num int) int{
    if num > 10{
        fmt.Println("num > 10")
        return num
    }
    fmt.Println("num < 10")
}

func main(){
    test(100)
}
```

```
Exec-in-cmd
# command-line-arguments
.\lesson06.go:10:1: missing return at end of function
```

```go
package main
import "fmt"

func test(num int) int{
    if num > 10{
        fmt.Println("num > 10")
        return num
    }
    fmt.Println("num < 10")
    return num
    fmt.Println("after return")
}

func main(){
    test(100)
}
```

```
Exec-in-cmd
# command-line-arguments
.\lesson06.go:12:1: missing return at end of function
```

常數與函式

函式注意事項

- **函式擺放的順序不重要**
- 在一些程式語言中(如C)，一定要先宣告才能呼叫，沒有宣告的函式是不可以呼叫的，但在 Golang 中則沒有這個限制，顛倒前面範例的 func test() 和 func main()：

```
package main
import "fmt"

func main(){
    num := 10
    test(100)                       // 先呼叫使用 func test
    fmt.Println("in main()", num)
}

func test(num int){                 // 後宣告 func test
    fmt.Println("in test()", num)
}
```

執行結果與前面範例相同無異

https://go.dev/play/p/pi8V9B08EcU

常數與函式

callback函式

- callback就是把function A當作變數傳進另一個function B內。而執行B的時候就會callback回去參考A：

```go
package main

import "fmt"

func visit(friends []string, callback func(string)) {
    for _, n := range friends {
        callback(n)
    }
}

func main() {
    visit([]string{"Tina", "James", "Mary"}, func(n string) {
        fmt.Println(n)
    })
}
```

- 可以看到在main裡面呼叫了visit，傳入了另一個function，而在執行visit中callback的時候又跑回來參考。

https://go.dev/play/p/yKuhKsGjWaO

常數與函式

recursion(遞迴)

- 顧名思義就是function一直呼叫自己，記得要有停止條件，要不然就變無窮迴圈了。

```go
package main

import "fmt"

func double(x int) int {
    x = x * 2
    if x > 64 {
        return x
    }
    return double(x)
}

func main() {
    fmt.Println(double(5))
}
```

這個範例會回傳x一直乘以二後第一個超過64的值

https://go.dev/play/p/HVC68_jDM_v

常數與函式

defer(延遲執行)

- defer就是在整個**function結束後才會執行**，滿不錯用的

```go
package main

import "fmt"

func hello() {
    defer fmt.Println("Nice to meet you")
    fmt.Println("Hello Tina")
}

func main() {
    hello()
}
```

```
Hello Tina
Nice to meet you
```

從這邊可以看到，雖然是nice to meet you在hello的第一行，
但是加入defer後就會變成hello執行完才執行defer的內容

https://go.dev/play/p/8IsouOvwyEM

常數與函式

defer(延遲執行)

- defer前後順序

```go
func main() {
    defer print1()
    defer print2()
}

func print1() {
    fmt.Println("p1")
}

func print2() {
    fmt.Println("p2")
}
```

```go
/* result:
p2
p1
*/
```

兩個defer執行的優先順序，為了要貫徹拖延的行為，
越早分派的任務要越晚達成才行

https://play.golang.org/p/IFgv-It3Y7Z

常數與函式

defer(延遲執行)

- 修改 defer中的參數
  - defer印出的值是多少，更動前的值？還是更動後？

```go
func main() {
    assign2(50)
}

func assign2(a int) int {
    defer fmt.Println(a) // 任務交代下來的時候 a值是50，然後盡可能地拖延
    a = 100              // 老闆更動了a為100
    return a
}

/* result:
50
*/
```

```go
func main() {
    assign1(50)
}

func assign1(a int) int {
    a = 100              // 老闆更動了a為100
    defer fmt.Println(a) // 任務交代下來的時候 a值是100，然後盡可能地拖延
    return a
}

/* result:
100
*/
```

  - 全看交代時的參數

https://play.golang.org/p/MYCoWj7Womg

常數與函式

defer(延遲執行)

- 當 defer遇上 os.Exit()
  - defer總想等到程式結束時再做行動，但是如果遇到os.Exit..

```
func main() {
    defer func() {
        fmt.Println("我很懶，想等到退出func的時候再印東西")
    }()
    os.Exit(0) // func直接被砍了
}

/* result:

*/
```

https://play.golang.org/p/2ihalSoyjbd

常數與函式

defer(延遲執行)

- 匿名函式裡面的defer

```go
func main() {
    fmt.Println(func1())
}

func func1() int {
    var a int
    defer func() {
        a = 100
    }()
    return a    // defer：『喔 要回傳a了喔，可是func還沒退出所以我不想做事，反正回上司也沒有規定要回傳哪個a，所以擺爛。』
}

/* result:
0
*/
```

```go
func main() {
    fmt.Println(func2())
}

func func2() (a int) {
    defer func() {
        a = 100
    }()
    return a //defer：『蛤，要回傳了喔？雖然想擺爛，但上司一開始指名規定要回傳a，先趕一下進度好了。』
}

/* result:
100
*/
```

https://play.golang.org/p/AE9WS1XOGjo

常數與函式

自行執行function

- 宣告完後自行執行

```go
package main

import "fmt"

func main() {
    func() {
        fmt.Println("lalala")
    }()
}
```

- 記得function宣告完後要加()代表執行的意思

https://go.dev/play/p/D6vl4rNPdXr

常數與函式

init函式

- init 函式是個特殊的函式，如果一段程式碼內包含著 init 函式，則在執行整個程式碼時，**會優先執行 init裡的程式碼**，通常需要初始化一些外部資源時，會將這些程式碼寫在init函式裡，以下是 Go 官方網站的例子：

```
func init() {
    if user == "" {
        log.Fatal("$USER not set")
    }
    if home == "" {
        home = "/home/" + user
    }
    if gopath == "" {
        gopath = home + "/go"
    }
    // gopath may be overridden by --gopath flag on command line.
    flag.StringVar(&gopath, "gopath", gopath, "override default GOPATH")
}
```