CH1變數與算符

1-1前言

1-1-1 GO語言簡介

- Go語言由google團隊開發，創造出感覺神似Javascript 或 PHP 的動態特性，卻又具備C++和Java的性能及效率，在大專案團隊裡依然具備實用性

- Go語言具有獨特的特質，例如編譯時會納入記憶體安全考量，以及通道(channel) 為基礎的併行性(concurrency)運算，往後將會探討這需特色的實作方式

- Go語言的原始碼是以純文字撰寫，在編譯成機器碼，並封裝成單一的可執行檔，此執行檔可以獨立執行，完全不用安裝任何輔助工具，因此能大幅簡化Go軟體的部屬和發布過程

- 編譯GO時，有數種目標作業系統可供選擇，包括windows，linux，macOs，Android 等等； 也就是說，GO語言所編寫的程式只要編寫一次就可以到處使用

- 另外方面，編譯式程式語言最讓人詬病的是編譯時的等待，因此Go團隊開發出神速的編輯器，即使專案再大也絲毫不遜色

- 此外，Go語言屬於靜態型別(statically typed)語言，採用了有型別安全防護的記憶體模型，並有垃圾回收(garbage collection)機制，這樣的組合可以避免開發人員製造出太多常見的程式錯誤與安全漏洞，同時也能保持性能及效率

- Ruby和Python這類的動態型別語言之所以受歡迎，部分原因就在於程式設計師認為，若不用去管資料型別和記憶體等問題，開發時的生產力會更好，但這類語言卻犧牲了性能和記憶體效率，更容易產生型別不符的錯誤

- Go語言不但達到了動態型別語言的生產力，性能與效率方面也絲毫不打折

- 最後，如果你想增加運算速度，就得善用平行運算或併型運算，因此CPU會更注重於核心數並非單一核心的時脈，但現今檯面上知名的程式語言設計時都並未善用這點，因此在撰寫並型性或多執行緒程式碼時容易出錯

- GO語言在設計之初就充分的運用CPU的多重核心，且消除了所有可能挫折跟程式碼錯誤，這種設計能讓開發人員容易且安全的撰寫並運算程式，進一步徹底地發揮現代多核心CPU和雲端運算的優勢

1-1-2 go語言的模樣

```go
package main    //宣告套件(package),所有的Go語言檔案都必須以套件宣告起頭
                //如果想直接執行此套件,就必須將套件命名為main

import (        //接者,以下程式碼會匯入各種套件
    "errors"
    "fmt"
    "log"
    "math/rand"
    "strconv"
    "time"
)

var helloList = []string{    //這裡宣告了一個全域變數
    "Hello, world",
    "Καλημέρα κόσμε",
    "こんにちは世界",
    "ایند مالس",
    "Привет, мир",
}

func main() {      //這裡宣告的是一個函式,所謂函式就是一段程式碼,呼叫函式時就會執行這段程式
    rand.Seed(time.Now().UnixNano())    //不過go語言的main()函式是Go程式碼的進入點,當你執行go程式時,它會自動呼叫main()
    index := rand.Intn(len(helloList))
    msg, err := hello(index)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(msg)
}
```

接上頁

```go
func hello(index int) (string, error) {
    if index < 0 || index > len(helloList)-1 {
        return "", errors.New("out of range: " + strconv.Itoa(index))
    }
    return helloList[index], nil  //沒有錯誤就回傳訊息
}
```

1-2 宣告變數

1-2-1用var宣告變數

```go
package main //定義套件名稱為main

import (
    "fmt" //匯入fmt套件
)

var foo string = "bar" //宣告套件範圍變數
//變數宣告: var  變數名稱  變數型別 = 值
func main() {
    var baz string = "qux" //宣告函示範圍變數
    fmt.Println(foo, baz)  //印出變數
}
```

用var一次宣告多個變數

```
1   package main
2
3   import (
4       "fmt"
5       "time"
6   )
7
8   var ( //一次宣告多個變數時要用小括號括起來
9       Debug       bool       = false       //布林值
10      loglevel    string     = "info"      //字串
11      starUpTime time.Time = time.Now() //time.Time結構
12  )
13
14  func main() {
15      fmt.Println(Debug, loglevel, starUpTime)
16  }
```

1-2-3用var宣告變數時省略型別或賦值

- 宣告變數時可以不用同時宣告型別或賦予值，只要提及其中之一即可

- 如果只有型別，go語言會自動賦予該型別特有的零值(zero value)，本章後面會在介紹甚麼是零值

- 若只有給初始值卻無指定型別，go語言自有一套規則，可以根據給定的值來判斷應該採用哪種型別

```go
package main

import (
    "fmt"
    "time"
)

var (
    debug       bool      //省略初始值
    longlevel   = "info"  //省略型别
    startUpTime = time.Now() //省略型别
)

func main() {
    fmt.Println(debug, longlevel, startUpTime)
}
```

1-2-4 推斷型別發生問題的時候

- 有時候必須在宣告變數時明確寫出每個環節，例如go語言就是無法猜對你需要的正確型別時，例如以下例子:

```go
package main

import "math/rand"

func main() {
    var seed = 1234456789
    rand.Seed(seed)
}
```

輸出結果為:

```
cannot use seed (variable of type int) as type int64 in argument to rand.Seed
```

這裡的問題在於, rand.Seed()需要的參數型別必須是int64,但go語言對一個整數會將其型別判斷為init , 本章後面會再詳述init與init64的差異

- 為了排除以上錯誤，就必須在宣告seed時補上int64型別：

```
1    package main
2
3    import "math/rand"
4
5    func main() {
6        var seed int64 = 1234456789
7        rand.Seed(seed)
8    }
```

1-2-5短變數宣告

- 用 名稱 := 值 簡化變數宣告，省略了關鍵字var，且必須給予初始值
- 注意：只有在函式裡才能這樣做

```
1   package main
2
3   import (
4       "fmt"
5       "time"
6   )
7
8   func main() {
9       //以簡式寫法逐一宣告變數
10      Debug := false
11      logLevel := "info"
12      startUpTime := time.Now()
13      fmt.Println(Debug, logLevel, startUpTime)
14  }
```

- 這種簡潔的寫法深受開發者喜愛，也是最常見的變數定義方式
- :=能讓程式碼變得簡潔有力，又能明確表明其意圖
- 接下來介紹另一個捷徑：只用一行程式就搞定所有變數宣告

1-2-6:以短變數宣告建立多重變數

- 寫法:
  - 變數1, 變數2, …. ,變數n :=值1, 值2,…. , 值n

```
1   package main
2
3   import (
4       "fmt"
5       "time"
6   )
7
8   func main() {
9       debug, loglevel, starUpTime := false, "info", time.Now()
10      fmt.Println(debug, loglevel, starUpTime)
11  }
```

- 這種寫法的缺點是閱讀性較差，但還是有它的用處，因為當你呼叫會傳回多個值的函式時，這種寫法就派上用場了:

```
1   package main
2
3   import (
4       "fmt"
5       "time"
6   )
7
8   func getConfig() (bool, string, time.Time) {
9       return false, "info", time.Now()
10  }
11
12  func main() {
13      Debug, loglevel, starUpTime := getConfig()
14      fmt.Println(Debug, loglevel, starUpTime)
15  }
```

1-2-7在單行程式內用var宣告多重變數

- 寫法:
  1. 變數1, 變數2, …, 變數n　型別
  2. 變數1,變數2, …, 變數n = 值

- 這麼做的限制是宣告變數時，若只提供型別，所有變數都只能是同一型別；如果用初始值來宣告，每個變數就能根據初始值推斷出不同型別

- 以下是一個例子:

```go
package main

import (
    "fmt"
    "time"
)

func getConfig() (bool, string, time.Time) {
    return false, "info", time.Now()
}
func main() {
    //只提供型別
    var start, middle, end float32
    fmt.Println(start, middle, end)
    //初始值的型別各異
    var name, left, right, top, bottom = "one", 1, 1.5, 2, 2.5
    fmt.Println(name, left, right, top, bottom)
    //對函式一樣適用
    var Debug, logLevel, startUpTime = getConfig()
    fmt.Println(Debug, logLevel, startUpTime)
}
```

- 顯示結果:

```
PS D:\git\go lan> go run "d:\git\go lan\myproject\1-2-7.go"
0 0 0
one 1 1.5 2 2.5
false info 2022-06-29 21:47:01.5017964 +0800 CST m=+0.004410701
```

- 上面的練習用短變數宣告會更精簡，但仍然有它的用途，例如需要多個型別相同的變數，又需要嚴格控制變數型別時

1-2-8 非英語的變數名稱

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    デバッグ := false
    日誌等級 := "info"
    ᑭᑌᒪᓀ := time.Now()
    _A1_Μείγμα := "?"
    fmt.Println(デバッグ, 日誌等級, ᑭᑌᒪᓀ, _A1_Μείγμα)
}
```

- Go語言支援UTF-8 , 因此可以用非英文字來為變數命名
- 不過也不是說變數的命名毫無限制 , 例如開頭必須是字元或底線
- 並不是所有語言都有這個特點, 這或許能解釋為何go語言在亞洲很受歡迎

1-3更改變數值

1-3-1更改單一變數值

```go
package main

import (
    "fmt"
)

func main() {
    offset := 5 //注意:這裡不能直接寫offset = 5
    fmt.Println(offset)

    offset = 10 //將offset從5改成10 (只有在更改變數值才可以只用等號)
    fmt.Println(offset)
}
```

- 練習:用其他變數來賦值:

```go
package main

import "fmt"

var defaultOffset = 10

func main() {
    offset := defaultOffset
    fmt.Println(offset)

    //把offset的值加上defaultOffset的值 , 重新存入offset
    offset = offset + defaultOffset
    fmt.Println(offset)

}
```

1-3-2一次更改多個變數值

- 就像在單行敘述中同時宣告多重變數時一樣，也可以用類似的寫法更改多個變數值

- 寫法:
  - 變數1, 變數2, ..., 變數n = 值1, 值2, ...,值n

- 底下是一個例子:

```go
package main

import "fmt"

func main() {
    //宣告多重變數並賦予值:
    query, limit, offset := "bat", 10, 0
    //以單行敘述一次更改全部的變數值
    query, limit, offset = "ball", offset, 20

    fmt.Println(query, limit, offset)

}
```

- 在這個例子中，我們以單行敘述更改了個變數值，這種方式在呼叫函式時一樣有用，但程式碼的首要之務就是確保其簡明易讀，如果像這樣單行敘述有可能會讓人讀得一頭霧水，那麼還是多寫幾行，讓人看了一下就懂

1-4算符

1-4-1算符基礎

- 算符是你能用來處理軟體資料的工具，例如霸資料做比較；算符也可以修改資料本身，譬如為資料加上某一數值

- 底下介紹算符的分類:

1. 算術算符(arithmetic operators):用在算術向關任務 , 例如四則運算

2. 比較算符(comparison operators): 用來比較兩個值 , 例如大於或小於

3. 邏輯算符(logical operators) : 搭配布林值的使用 , 例如判斷何者為真

4. 定址算符(address operators): 當我們談到指標時會再介紹

5. 受理算符(receive operators): 用來對go語言特有的通道寫入或讀取值 , 第16章會介紹

- 底下來看一個簡單的例子 : 用算符處理數字

```go
package main

import "fmt"

func main() {
    var total float64 = 2 * 13 //主餐 (2人, 每份 $13)
    fmt.Println("+ 主餐:", total)

    total = total + (4 * 2.25) //飲料(每杯 $2.25, 4杯)
    fmt.Println("+ 飲料:", total)

    total = total - 5 //折抵$5
    fmt.Println("- 折抵:", total)

    tip := total * 0.1 //小費10%
    total = total + tip
    fmt.Println("+ 小費:", total)

    split := total / 2 //分攤額
    fmt.Println("分攤額:", split)
```

- 接上頁

```
21
22      visitCount := 24 //來店次數(之前來過24次)
23      visitCount = visitCount + 1
24      remainder := visitCount % 5 //用餘數算符檢查除以5餘數是否為0(是5的倍數), 傳回布林值
25      if remainder == 0 {
26          fmt.Println("您已獲得來店滿 5 次折價券!")
27      }
28  }
```

字串的串接:

- 上面我們用了算數算符和比較算符來處理數字，來模擬了個計算用餐金額的帳單

- 不過，算符種類繁多，而取決於資料型別的不同，能用的算符也不同
  - 譬如+號不只可以加數字，也可以用來串接字串

- 底下是一個例子:

```go
package main

import "fmt"

func main() {
    givenName := "John"
    familyName := "Smith"
    fullName := givenName + " " + familyName //串接字串
    fmt.Println("Hello,", fullName)
}
```

1-4-2算符簡寫法

- 簡寫方式:
  - += 值 (直接加上值)
  - -= 值 (直接減去值)
  - ++ (遞增1)
  - -- (遞減1)

- 底下看一個例子:

```go
package main

import "fmt"

func main() {
    count := 5 //宣告一個變數並賦予值

    count += 5 //變數加上5,再重新賦值回給同一個變數
    fmt.Println(count)

    count++ //變數增加1
    fmt.Println(count)

    count-- //變數減去1
    fmt.Println(count)
```

- 接上頁

```
16
17      count -= 5 //變數減去5
18      fmt.Println(count)
19
20      name := "John"
21      name += " Smith" //也可以自串串接
22      fmt.Println("Hello,", name)
23  }
```

1-4-3值的比較

- 比較算符與邏輯算符大部分都在處理兩個值之間的關係，而其結果一定是一個布林值:

比較算符	
==	當前後兩個值**相等**時為真
!=	當前後兩個值**不相等**時為真
<	當左側的值**小於**右側的值時為真
<=	當左側的值**小於或等於**右側的值時為真
>	當左側的值**大於**右側的值時為真
>=	當左側的值**大於或等於右側**的值時為真
邏輯算符	
&&	如果左側與右側的布林值**均為真**時, 結果為真
||	如果左側與右側的布林值**任一為真**時, 結果為真
!	只處理單一布林值, 如果該值為偽、運算結果便**反轉**為真, 值為真時結果為偽

- 來看一個例子：

我們想要依據顧客的來店次數給予會員等級，劃分如下:

  - 銀牌 : 來店次數介於11到20
  - 金牌 : 來店次數介於21到30
  - 白金 : 來店超過30次

- 底下是程式碼

```go
package main

import "fmt"

func main() {
    visits := 15 //顧客目前的來店次數

    fmt.Println("新顧客  :", visits == 1)
    fmt.Println("熟客    :", visits != 1)
    fmt.Println("銀牌會員:", visits > 10 && visits <= 20)
    fmt.Println("金牌會員:", visits > 20 && visits <= 30)
    fmt.Println("白金 VIP:", visits > 30)
}
```

- 在這個練習中，我們以比較算符處理資料，以便做出決策
- 這些算符的結合方式有無限的可能，因此不管事甚麼問題都可以利用他們來解決

- 接下來看看，若定義變數時沒有給初始值，會發生甚麼事?

1-5零值

- 所謂零值(zero value)，指的是該型別具有的預設值或是空值(empty value)
- Go語言定義了一系列的規則，指明所有核心型別的零值:

型別	零值
Bool (布林值)	false
數字 (整數與浮點數)	0
String (字串)	“” (空字串)
指標/函式/介面/切片/通道/映射表	nil (後面章節會再詳談)

- 當然上面並未包含所有的型別，但其他的型別都是從這些核心型別延伸出來的，因此這些規則一體適用

練習 : 印出零值

- 在此練習中 , 我們會宣告出一些變數但故意不指定初始值 , 然後將其零值顯示出來
- 這裡會利用fmt.Printf()函式來印出值 , 因為它能讓我們更了解一個值得型別
- fmt.Printf()是一種格式化樣板語言 (template language) , 藉以轉換我們傳遞給它的值
- 我們在此使用的格式化符號是%#v ; 當你想要以某種方式顯示變數的值或型別時 , 這個符號就很有用

- 下表列出常見的格式化符號:

%v	任何值 , 若不介意印出值的型別 , 就直接用這個
%+v	印出值並加上額外的資訊 , 例如結構(struct)類別的欄位名稱
%#v	用Go語言印出值 , 等於%+v加上型別名稱
%T	印出值的型別
%t	印出布林值(true/false)
%d	印出10進位數字
%s	印出字串
%%	印出百分比符號

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    var count int //整數
    fmt.Printf("Count    : %#v \n", count) //count的值會帶入%#v

    var discount float64 //64位元浮點
    fmt.Printf("Discount : %#v \n", discount)

    var debug bool //布林值
    fmt.Printf("Debug    : %#v \n", debug)

    var message string //字串
    fmt.Printf("Message  : %#v \n", message)

    var emails []string //字串切片
    fmt.Printf("Emails   : %#v \n", emails)

    var startTime time.Time //time.Time結構
    fmt.Printf("Start    : %#v \n", startTime)
}
```

顯示結果:

```
PS D:\git\go lan> go run "d:\git\go lan\ch1\1-5.go"
Count    : 0
Discount : 0
Debug    : false
Message  : ""
Emails   : []string(nil)
Start    : time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC)
```

- 以上練習定義了各種型別的變數，但都沒有給初始值．然後用fmt.Printf()顯示這些變數的零值為何
- 同時也能發現，%#v對數字/字串/布林值只會顯示其值，但對切片與結構這類複合型別藉會輸出更多資訊
- 只要了解型別的零值，以及go語言如何控制，就可以避免錯誤和寫出精簡的程式碼

1-6 值 VS. 指標(pointer)

1-6-1 了解指標

- 當我們把init/bool/string這類值傳給函式處理時，go語言會在函式中複製這些值，建立出新的變數
  - 這個複製動作意味著你呼叫函式時，若函式對參數做出更動時，原始值也不會受影響，能減少錯誤
- 對於這種傳值方式，go語言採用了一種記憶體管理系統:堆疊(stack),每個參數都會在堆疊中獲得自己的記憶體
  - 缺點是，有越多的值在函式之間傳遞，這樣的複製動作就會消耗越多記憶體
  - 現實中的函式都很短小，任務會分割成多個函式進行，因此多次複製行為到頭來可能消耗比實際需求多的記憶體

- 不過，還有一種在函是之間傳值的替代方案，用用的記憶體較少
  - 這種方式不會複製值，而是建立指標(pointer)來傳遞給函式
  - 指標跟值本身是兩回事，指標的唯一用途就只是拿來取得值，可以將指標想像成一個通往值的路標，若想取得值，就必須跟著路標的方向走
  - 在使用指標傳值給函式時，go語言就不會複製指標指向的值
  - 對一個值建立指標後，go語言就無法以堆疊來管理該值所使用的記憶體
    - 這是由於堆疊必須仰賴簡單的變數範圍邏輯，以便得知何時可以回收該值所使用的記憶體，以上規則並不適用於指標
  - Gp 語言會轉而把值放到堆積(heap)記憶體空間：堆積允許一個值存在，直到程式中沒有任何指標參照它為止
  - Go語言會用所謂的垃圾回收機制(garbage collection)程序來回收記憶體
    - 這種機制會在背景定期運作

- 注意:
  - 一旦為值設置了指標，就等於把值放進了堆積
  - go語言會使用逃逸分析的程序來判斷一個值是否還應該放到堆積上
  - 不過,有時候就算值沒有設定指標也會因為其他原因被放進堆積裡
  - 值究竟要放到堆疊還是堆積上，你是無法介入的
  - 記憶體管理不是go語言規格的一部份，而是被視為內部時做細節，這表示它有可能隨時變更

- 雖然以指標取代值來傳遞大量函式，對記憶體運用的好處不可言喻，可是對CPU的負擔就比較難評估
  - 複製資料時，go語言需要幾個CPU指令週期來取得記憶體和稍後釋出
  - 如果用指標傳值，可以減少CPU的用量，但需要仰賴繁瑣的垃圾回收機制，尤其是堆積內累積了大量資料時
- 所以該不該用指標其實沒有標準答案
  - 最好的方式是採取經典的效能最佳化方法:
    - 一開始不及著做最佳化，若程式效能不佳，先測試執行時間，再進行調整

- 除了改善效能，也可以用指標改變程式的設計方式

- 有時使用指標可以讓函式呼叫起來更清爽，舉例來說:
  - 若想判斷某值存在與否，判斷一般的變數就會有問題，因為它一定至少會帶有零值，而零值在程式碼內仍然是合法的
  - 相對，指標會有”未設定”的狀態，它沒有儲存值時會回傳nil，而nil在go語言裡就代表無值

- 指標本身可以是nil的這種特性，意味著就算指標沒有任何值，你還是可以透過取得指標自身的值，進而導致執行期間錯誤
  - 為了避免這種問題，可以先拿指標和nil做比較 (像是 <指標> != nil),然後才對指標賦值

- 相同型別的指標也可以互相比對，看看是否相等，但結果一定是false，因為指標在做比較時，被比較的是指標本身而非是它們指向的值，指標只有在跟自己比較時才會得到true

1-6-2 取得指標

- 要取得指標有以下幾種方式 :

1. 可以用var宣告變數 , 但把其型別設為指標 , 也就是把*放在型別前面:
   - var 變數 *型別

2. 內建函式 new()可以達到賦值的效果 , 該函式的用意在於為某種型別取得記憶體/填入該型別的零值 , 然後傳回記憶體的指標 :
   - 變數 := new(型別)

3. 想取得某個既有的變數指標 , 請利用&算符 :
   - 變數1 := &變數2

- 練習：取得指標

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    var count1 *int  //宣告一個指標變數 (值為為nil)
    count2 := new(int) //宣告第二個指標變數 (值為0)
    countTemp := 5
    count3 := &countTemp  //從別的變數建立指標
    t := &time.Time{}   //直接從結構型別建立指標

    fmt.Printf("count1: %#v\n", count1)
    fmt.Printf("count2: %#v\n", count2)
    fmt.Printf("count3: %#v\n", count3)
    fmt.Printf("time  : %#v\n", t)
}
```

- 顯示結果:

```
count1: (*int)(nil)
count2: (*int)(0xc0000aa058)
count3: (*int)(0xc0000aa070)
time  : time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC)
```

- 可以看到，用%#v印出指標時，會看到其他型別以及參照的記憶體位置
  - time.Time結構比較特別，也可以看到其值，但仍是個

1-6-3 從指標取得值

- 若要取得指標的關聯值，必須把*算符放在變數名稱前面，以便解除指標的參照，取得真正的值：
  - 值 = *指標變數

- 一個常見的錯誤是嘗試去解除一個無值指標(nil)，但是go編譯器無法事先警告你這一點，只能等到程時執行時問題才會浮現，因此最好養成好習慣：解除指標前先檢查它是否為nil

  - 你並不需要每次都自行解除指標參照；舉例來說，若要存取指標指向的一個結構屬於函式，go語言會自動替你解除參照 (詳見第四章)，並不用擔心何時才該解除參照，因為go語言會以明確的錯誤警訊告知你能否這樣做

- 練習：從指標取得值

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    var count1 *int
    count2 := new(int)
    countTemp := 5
    count3 := &countTemp
    t := &time.Time{}

    if count1 != nil {
        fmt.Printf("count1: %#v\n", *count1) //用*取得指標的值
    }
    if count2 != nil {
        fmt.Printf("count2: %#v\n", *count2)
    }
```

- 接上頁

```
21      if count3 != nil {
22          fmt.Printf("count3: %#v\n", *count3)
23      }
24      if t != nil {
25          fmt.Printf("time  : %#v\n", *t)
26          //存取t(time.Time結構)自身的函示時不需要寫成*t來解除參照
27          fmt.Printf("time  : %#v\n", t.String())
28      }
29  }
```

- 顯示結果：

```
count2: 0
count3: 5
time  : time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC)
time  : "0001-01-01 00:00:00 +0000 UTC"
```

- 在以上練習中，我們用解除參照指標取出實際值，同時也加上nil檢查，以免遭到解出參照錯誤

1-6-4採用指標的函式設計

- 到第五章會再詳談自訂函式，不過這裡可以看看指標會如何改變函數的使用方式

- 函數必須明確的改寫才可以接收指標

- 此外若你的變數是指標，或者傳遞給函式的是指標變數，那麼在函式中對該參數做的任何變動，也會影響到函式外部原始變數的值

- 練習：
  - 在這個練習中，我們要建立兩個函式: 一個可以接收數值，對值加上5再顯示到主控台，另一個函式則是接收數值的指標，同樣也是加上5然後顯示到主控台
  - 同時，我們會在每次呼叫函式過後把數值印出來，以便觀察函式對傳入值的變數有何影響

```go
package main

import "fmt"

func add5Value(count int) {  //傳入值
    count += 5
    fmt.Println("add5Value     :", count)
}

func add5Point(count *int) {  //傳入指標
    *count += 5
    fmt.Println("add5Point     :", *count)
}

func main() {
    var count int
    add5Value(count)
    fmt.Println("add5Value post:", count)
    add5Point(&count)
    fmt.Println("add5Point post:", count)
}
```

顯示結果

```
add5Value      : 5
add5Value post: 0
add5Point      : 5
add5Point post: 5
```

- 以上練習中展示了如何以指標將值傳遞給函式，還有這種作法如何影響傳入的變數值

- 以值傳遞變數時，在函式內做的變動只有在函式內生效，不會影響原始變數；然而若是以指標的形式傳入函式，就會更動原始變數的值

1-7常數

- 常數就像變數，但你無法改變它的初始值，如果程式在執行時，有個數字不須變動,也不該變動時，常數就能派上用場

- 常數的宣告跟使用var很類似，但改用了const關鍵字
  - 宣告常數時，初始值是不可少的，型別則可有可無，若不指定go語言會自行推斷
  - 初始值可以是一段簡單的運算式，也可以直接引用其他的常數
  - 跟var一樣，可以用一個const同時宣告多個常數

- 練習：宣告常數
  - 在這裡我們要解決一個問題：資料庫伺服器太慢了，我們需要建立一個自訂的記憶體快取
  - 在此我們要使用go語言的map集合型別 (第四章詳談)，以它來擔任快取表，但這個快取表中可容納的總項目數量是有上限的
  - 此外，快取中需要宣告兩種類型的資料：書本和CD唱片，兩者都具有是別用的ID 字串與對應的名稱，因此我們需要想辦法在共用的快取表中區分這兩種資料，並有辦法讀取和寫入資料

```go
package main

import "fmt"

const GlobalLimit = 100                    //單筆資料筆上限
const MaxCacheSize int = 2 * GlobalLimit //快取最大容量 (單筆上限 x 2)

const (
    CacheKeyBook = "book_" //書本id的前綴字
    CacheKeyCD   = "cd_"   //CD id 的前綴字
)

var cache map[string]string //快去集合

func cacheGet(key string) string { //從快取取出某個鍵的值
    return cache[key]
}

```

```go
func cacheSet(key, val string) { //將某個鍵和值寫入快取
    if len(cache)+1 >= MaxCacheSize { //如果快取大小已達極限就跳出函式
        return
    }
    cache[key] = val
}

func SetBook(isbn string, name string) { //加入書本資料
    cacheSet(CacheKeyBook+isbn, name) //加入書本前綴字後呼叫cacheSet()
}

func GetBook(isbn string) string { //讀取書本資料
    return cacheGet(CacheKeyBook + isbn) //加上書本前綴字後呼叫cacheGet()
}

func SetCD(sku string, title string) { //加入CD資料
    cacheSet(CacheKeyCD+sku, title) //加上CD前綴字後呼叫cacheSet()
}
```

```
37
38  func GetCD(sku string) string { //讀取CD資料
39      return cacheGet(CacheKeyCD + sku) //加上CD前綴字後呼叫cacheGet()
40  }
41
42  func main() {
43      cache = make(map[string]string) //初始化快取
44      //在快取寫入資料
45      SetBook("1234-5678", "Get Ready To Go")
46      SetCD("1234-5678", "Get Ready To Go Audio Book")
47      //讀取和印出快取資料
48      fmt.Println("Book :", GetBook("1234-5678"))
49      fmt.Println("CD   :", GetCD("1234-5678"))
50  }
51
```

- 顯示結果

```
Book : Get Ready To Go
CD   : Get Ready To Go Audio Book
```

- 在這個練習中，我們運用常數來定義程式碼執行時不須更動，但你可以在日後修改的值

- 程式碼中的常數宣告也用了不一樣的寫法，有些加上型別，有些則無，也有的常數值是以其它的常數值計算而來

- 在這個程式中，書本會以book_1234-5678的鍵輸入快取，CD則為cd_1234-5678，這些前綴字都是由常數提供，當寫入的資料達到常數定義的200筆時，快取表就不會再接受任何資料 (若想改變快取大小，修改程式開頭的常數定義即可)

1-8 列舉

- 列舉是一種定義一系列常數的方式，這些常數的值是整數，而且會彼此相關

- Go語言沒有內建列舉專用的型別，但提供了一種稱為iota的工具，讓你可以用常數定義出自己的列舉資料

舉例來說，我們在以下的程式碼將一周中的每一天定義為常數:

```
const (
        Sunday        =0
        Monday        = 1
        Tuesday       = 2
        Wednesday  =3
        Thursday      = 4
        Friday         = 5
        Saturday      = 6
)
```

- 但與其手動定義每個值，上面的程式十分適合用iota功能來實現
- Go語言會協助我們管理以上那樣的常數清單
- 下頁程式碼改以iota寫成，但效果一模一樣

```
const (
        Sunday = iota  //0
        Monday  //1
        Tuesday  //2
        Wednesday   //以下以此類推…
        Thursday
        Friday
        Saturday
)
```

現在 , iota會代勞指派述職的工作 , Sunday是0 , Monday則是1 , 後面以此類推

1-9變數作用範圍

- 在go語言中，所有的變數都有其運作的範圍
- 最頂層的範圍是套件(package)範圍，每個變數範圍底下又可以包括其他的子範圍
- 子範圍有幾種定義方式，最簡單的方法就是觀察{ }，在大括號之間就是一段範圍
- 變數範圍的上下層關係，是在編譯時就決定好的，並不是等到執行時才決定

- 當你的某段程式碼存取某個變數時，go語言會檢查該程式碼的運作範圍

  - 如果在範圍內找不到該名稱，就會一層層往上找，直到最頂層的套件範圍為止，途中只要找到同名的變數就會停止搜索，並使用那個變數；彈道頂端還是找不到變數的話，就會丟出錯誤訊息

  - 如果找到同名變數，型別卻不一樣，也會丟錯誤訊息

- 練習 :從子範圍存取上層變數

- 在下例中，程式有四個不同的運作範圍，但只定義一次level變數，而該變數位於最高的套件層級，這表示無論在何處存取變數，都會讀到同一個變數：

```go
package main

import "fmt"

var level = "pkg" //套件範圍變數

func main() {
    fmt.Println("Main start  :", level)  //main()層級
    if true {
        fmt.Println("Block start :", level) //main底下的if層級
        funcA()
    }
}

func funcA() {
    fmt.Println("funcA start :", level) //funcA()函式層級
}

```

顯示結果

```
Main start  : pkg
Block start : pkg
funcA start : pkg
```

變數的遮蔽

- 在第二個例子中，我們要用子範圍的同名level變數來遮蔽(shadow)套件範圍的level變數，這兩個變數彼此沒有關係

- 當我們在子範圍內印出level變數時，go語言一找到這個範圍的變數就會停止，所以印出的直會有所不同

- 你也可以看出變數的型別有所不同，而go語言變數一旦定義就不能改變型別，換句話說，子範圍的level變數遮蔽了套件範圍的level變數

```go
package main

import "fmt"

var level = "pkg" //在套件範圍定義level

func main() {
    fmt.Println("Main start  :", level)

    level := 42 //在main範圍定義level
    if true {
        fmt.Println("Block start :", level)
        funcA()
    }
    fmt.Println("Main end    :", level)
}

func funcA() {
    fmt.Println("funcA start :", level)
}
```

執行結果:

```
Main start  : pkg
Block start : 42
funcA start : pkg
Main end    : 42
```

當我們呼叫funcA()時，go語言動用了靜態範圍解析，他不會管funcA()是在何處被呼叫的，因此funcA()看見的依然是套件層級的level變數

子範圍的變數在外部無法取得

- 在某層級中，你無法取得定義在其子範圍內的變數:

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       {
7           level := "Nest 1"
8           fmt.Println("Block end   :", level)
9       }
10      // 將產生錯誤: undefined: level
11      fmt.Println("Main end    :", level)
12  }
```

本章結束