CH5 函式

5-1 前言

- 函式是我們宣告來從事一項任務的一段程式碼；go語言函式與其他程式語言的不同之一，就在於可以回傳多重值

- 在以下各節，會陸續看到go語言一些與眾不同的函式特性，適用於以下不同的場合：
  - 將函式當成引數傳給其他函式
  - 將函式賦值給變數，以及當作另一個函數的回傳值
  - 將函式視為型別
  - 匿名與閉包函式

- Go語言的函式是所謂的一級函式，也就是函式可以當作其他函數的引數(傳給參數的值)或回傳值

- 可以接收其他函式為引數的函式又稱為高階函式

5-2函式

- 以下介紹幾個使用函式的理由：

1. 分解複雜的任務

2. 精簡程式碼

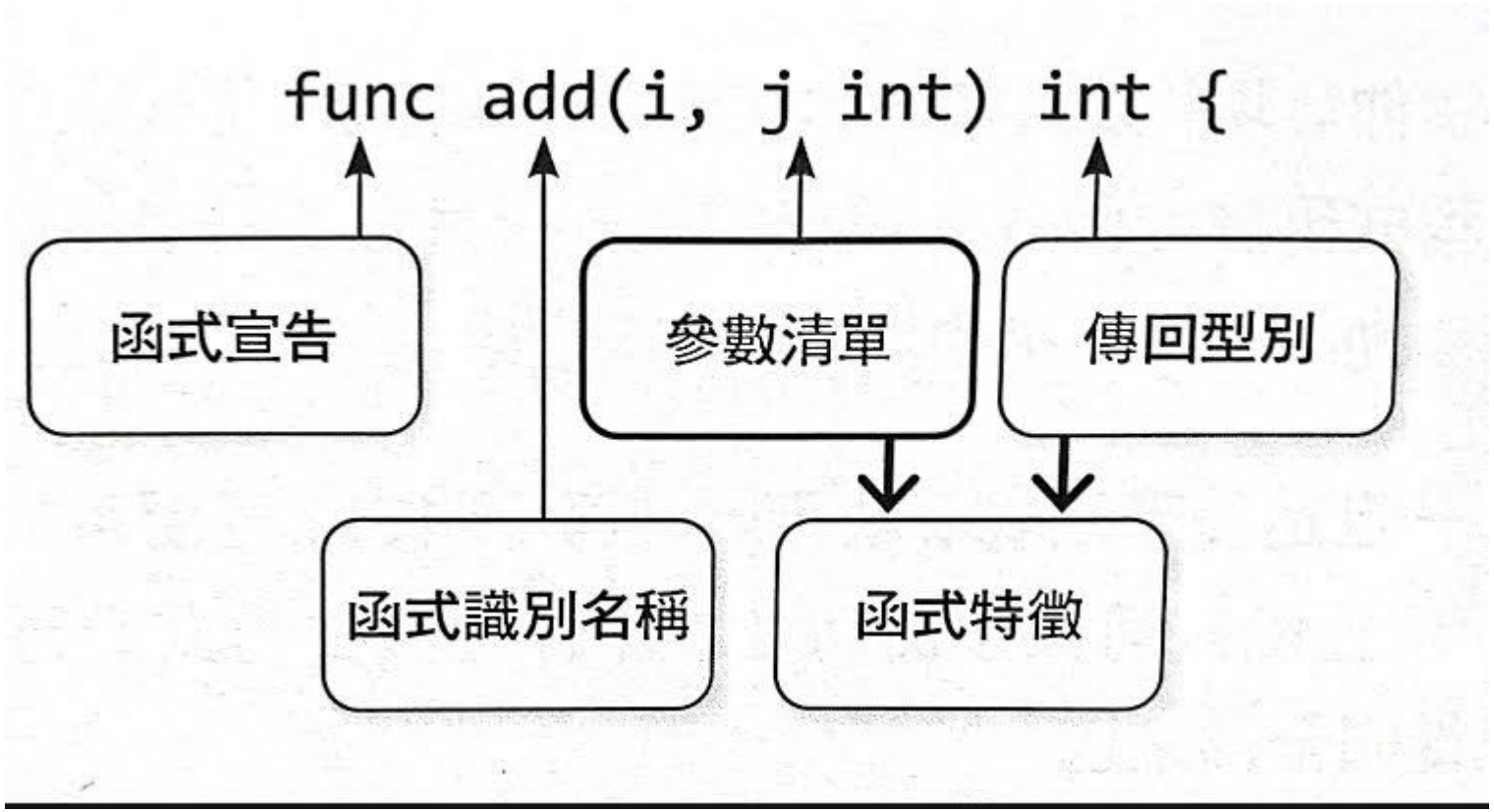
3. 重複使用性

設計函式時需要遵守的原則:

- 單一責任制 :
  - 一個函式負責一個任務，這樣測試與維護比較簡單
- 短小精幹 :
  - 函式程式碼不該動輒就超過數百行，若真的這樣，就表示程式碼需要重構，且十之八九沒有遵守上述的單一責任制
  - 保持函式簡潔可以避免大型函式除錯時面臨的複雜性，撰寫單元測試時也可以得到更好的程式碼覆蓋率

5-2-1 函式的宣告與組成

- 下圖是函式的典型結構



- 函式宣告：
  - 函式的宣告一律始於關鍵字func，你可以在任何套件層級的任何位置使用func定義函式

- 函式識別名稱：
  - 在go語言中，函式名稱的慣例是採用駝峰式大小寫，這種命名的第一個字母用小寫，但隨後每一個單字都以大寫開頭，比如 calculateTax
  - 函式名稱應該要具有意義，讓程式碼簡明易懂，不過函式名稱非絕對必要，沒有名稱的函式就是所謂的匿名函式

- 參數清單：
  - 參數是函式的輸入值，為函式之內的區域變數
  - 參數在函式的定義格式如下：
    名稱1 型別1, 名稱2 型別2, ...
  - 一個函式可以完全沒有參數，也可以有多個參數
  - 當多個參數型別一致時可以簡寫成: 名稱1 名稱2 型別1

- 傳回型別：
  - 傳回型別是一系列資料型別，而諸如布林值/字串/map或是另一個函式都可以被傳回
  - 以宣告函式的角度來說，這些型別稱為回傳型別，然而在呼叫函式時就稱作回傳值
  - 回傳值式函數的輸出，go語言可以一次回傳多個值

- 函式特徵:
  - 函式特徵其實是一個術語，它是輸入參數和輸出型別的合稱
  - 當定義一個函式後，必須盡量避免修改函式特徵，以免日後造成問題

- 函式本體：
  - 本體包含在一對大括號{}之中，這些程式碼決定了函式會做的事
  - 若函式有定義回傳型別，本體就必須要有return敘述。 return敘述會令函式立即停止，並回傳列在其後的值
  - 另外需要注意的是，go語言定義函式時左大括號{要跟函式定義在同一行，不然會出現錯誤

呼叫函式

- 執行函式的方法就是呼叫它：
  - 函式名稱(參數1, 參數2.....)

- 函式可以呼叫包含自己在內的任何函式，一旦出現呼叫動作就代表控制權轉換給被呼叫的函式

- 當被呼叫的函式傳回值或直行到其又括號}時，控制權才回到先前的呼叫方

練習：印出銷售績效

- 這個練習要建立一個函式，它不需要輸入參數和輸出型別，但內部會走訪一個map，印出當中的業務員名字和其銷售數量，同時也根據業務員的表現印出他們的績效

```go
package main

import (
    "fmt"
)

func main() {
    itemsSold()
}

func itemsSold() {
    items := make(map[string]int)
    items["John"] = 41
    items["Celina"] = 109
    items["Micah"] = 24

    for k, v := range items {
        fmt.Printf("%s 賣出 %d 件商品, 表現", k, v)
        if v < 40 {
            fmt.Println("低於預期.")
        } else if v > 40 && v <= 100 {
            fmt.Println("符合預期")
        } else if v > 100 {
            fmt.Println("超乎預期")
        }
    }
}
```

執行結果

```
John 賣出 41 件商品，表現符合預期
Celina 賣出 109 件商品，表現超乎預期
Micah 賣出 24 件商品，表現低於預期.
```

5-2-2 函式參數

- 參數決定了你能把哪些引數或值傳給函式
- 函式可以沒有參數，也可以有多個參數
- 函式參數就是它的區域變數，作用範圍僅限於函數內
- 呼叫函式時引數傳入的型別必須與參數的定義一致

練習：對應特定標頭的索引值

- 現在要定義一個函式，它接收的參數是一份CSV資料的標頭所構成的切片，我們要尋找特定標頭和他們所在的索引值，並以map的形式印出

```go
package main

import (
    "fmt"
    "strings"
)

func main() {
    hdr := []string{"empid", "employee", "address", "hours worked", "hourly rate", "manager"}
    csvHdrCol(hdr)
    hdr2 := []string{"Employee", "Empid", "Hours Worked", "Address", "Manager", "Hourly Rate"}
    csvHdrCol(hdr2)
}

```

```go
func csvHdrCol(header []string) {
    csvHeadersToColumnIndex := make(map[int]string)
    for i, v := range header {
        //用TrimSpace()把標頭去掉空白和用ToLower()轉成小寫
        //然後比對我們要找的標頭 , 以其索引為鍵放進map
        switch v := strings.ToLower(strings.TrimSpace(v)); v {
        case "employee":
            csvHeadersToColumnIndex[i] = v
        case "hours worked":
            csvHeadersToColumnIndex[i] = v
        case "hourly rate":
            csvHeadersToColumnIndex[i] = v
        }
    }
    fmt.Println(csvHeadersToColumnIndex)
}
```

執行結果

```
map[1:employee 3:hours worked 4:hourly rate]
map[0:employee 2:hours worked 5:hourly rate]
```

5-2-3 函式回傳值

- 到目前為止，我們建立的函數都沒有回傳值，但函式通常會接收輸入值做若干處理後回傳結果：

  值1, 值2, …. := 函式名稱()

練習：有傳回值的fizzBuzz()函式

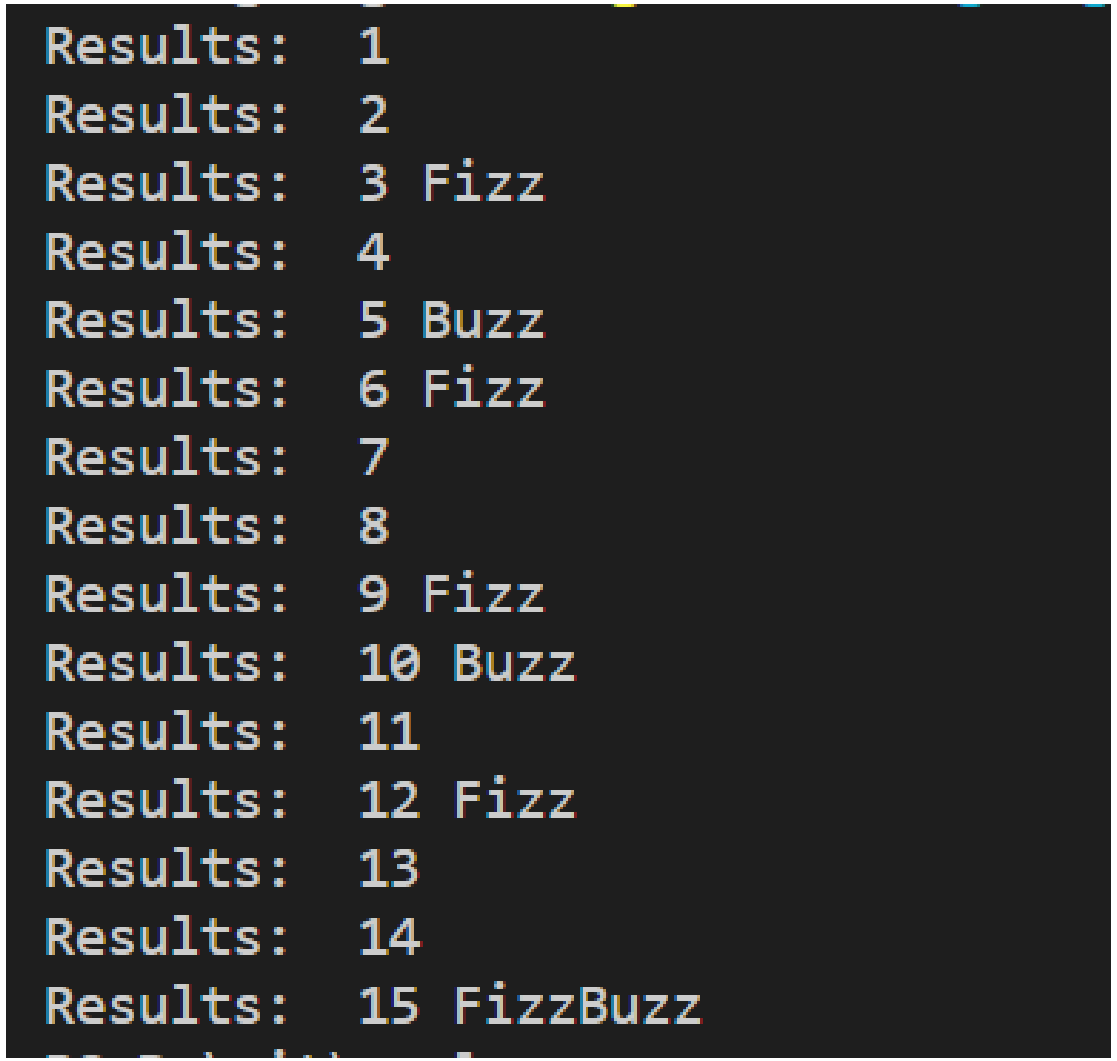
- 這裡要建立一個函式，其規則如下:

1. 用迴圈讓一個數字從1累加到某值，最後印出加總的結果，但有例外條件如下：
2. 若是3的倍數，改為顯示文字 Fizz
3. 若是5的倍數，改為顯示文字 Buzz
4. 若是3和5的公倍數，改為顯示FizzBuzz
5. 不是3也不是5的倍數顯示空字串

- 這裡會讓函式接收一個整數引數，並回傳兩個值，第一個是走訪到的數字，第二個是該數字應該對應的fizz/buzz/fizzbuzz或空字串

```go
package main

import (
    "fmt"
)

func main() {
    for i := 1; i <= 15; i++ {
        n, s := fizzBuzz(i)
        fmt.Printf("Results:  %d %s\n", n, s)
    }
}

func fizzBuzz(i int) (int, string) {
    switch {
    case i%15 == 0:
        return i, "FizzBuzz"
    case i%3 == 0:
        return i, "Fizz"
    case i%5 == 0:
        return i, "Buzz"
    }
    return i, ""
}
```

顯示結果：


Results: 1
Results: 2
Results: 3 Fizz
Results: 4
Results: 5 Buzz
Results: 6 Fizz
Results: 7
Results: 8
Results: 9 Fizz
Results: 10 Buzz
Results: 11
Results: 12 Fizz
Results: 13
Results: 14
Results: 15 FizzBuzz

忽略一部分的回傳值

- Go語言允許你忽略回傳的變數

- 假設我們對fizzbuzz()回傳整數不感興趣，可以用一個底線來忽略該回傳值：

```
7   func main() {
8       for i := 1; i <= 15; i++ {
9           _, s := fizzBuzz(i)
10          fmt.Printf("Results:  %d %s\n", n, s)
11      }
12  }
```

結構方法也是函式

- 在第四章中介紹了你能如何宣告函式並把它綁在指定的結構型別，成為該結構變數的方法

- 除了呼叫方式不同外，結構方法其實也式函式

5-2-4 naked returns

- 宣告函式時，也可以選擇給回傳值加上變數名稱，這樣會使程式碼更容易閱讀

- 若你給回傳值取名，這就會在函式內建立該名稱的區域變數，其作用範圍和參數一樣，這麼一來你就可以賦值給傳回值變數

```
func greeting() (name string, age int) {
    name = "John" //變數已存在，用=而不是:=賦值
    age = 31
    return name, age
}
```

- 若你沒有在return後面指定要回傳的變數，go語言會自動將回傳值清單裡的變數傳回，這就是所謂的naked return

```go
func greeting() (name string, age int) {
    name = "John" //變數已存在，用=而不是:=賦值
    age = 31
    return
}
```

- Naked return的缺點之一是，若用在比較長的函式中，可能會讓程式碼不好理解，所以稍微複雜的函式盡量避免naked return

- 此外, naked return 還可能衍生出變數遮蔽問題：

```
func message() (massage string, err error) {
    massage = "hi"
    if message == "hi"{
        err := fmt.Errorf("say bye\n")
        return
    }
    return
}       You, 1 秒前 • Uncommitted changes
```

- 這裡會有錯誤: err is shadowed during return , 原因式變數err先在函式的回傳清單宣告過, 之後又在if敘述被宣告和初始化 , 往上遮蔽了函式層級的同名變數 , 這樣會產生不知道該回傳哪個err變數的混淆

練習：對應特定標頭的索引值:naked return版

- 下面我們來修改前面的練習, 讓它用naked return 回傳一個 map

```go
package main

import (
    "fmt"
    "strings"
)

func main() {
    hdr := []string{"empid", "employee", "address", "hours worked", "hourly rate", "manager"}
    result := csvHdrCol(hdr) //接收回傳值
    fmt.Println("Result:")
    fmt.Println(result)
    fmt.Println()

    hdr2 := []string{"employee", "empid", "hours worked", "address", "manager", "hourly rate"}
    result2 := csvHdrCol(hdr2)
    fmt.Println("Result2:")
    fmt.Println(result2)
    fmt.Println()
}

```

```go
func csvHdrCol(hdr []string) (csvIdxToCol map[int]string) { //定義傳回值的名稱和型別
    csvIdxToCol = make(map[int]string) //初始化傳回變數
    for i, v := range hdr {
        switch v := strings.ToLower(strings.TrimSpace(v)); v {
        case "employee":
            csvIdxToCol[i] = v
        case "hours worked":
            csvIdxToCol[i] = v
        case "hourly rate":
            csvIdxToCol[i] = v
        }
    }
    return //用naked return 傳回 csvIdxToCol
}
```

顯示結果：

```
Result:
map[1:employee 3:hours worked 4:hourly rate]

Result2:
map[0:employee 2:hours worked 5:hourly rate]
```

5-3 參數不定函式

- 所謂參數不定函式是可以接收不確定數量參數的函式

- 當你無法確認某參數要接收的引數有多少時，就可以使用參數不定函式：

  func f(參數名稱 ...型別)

- 型別前面的三個點稱為打包算符，用途是告訴go語言把所有符合該型別的引數都放進此參數名稱，打包成一個切片

- 這種數量可變的參數可以接收任意數量的引數，甚至可以完全沒有引數

讓函式接收不定引數

- 先看一個簡單的例子，示範上面的參數不定函式：

```go
package main

import (
    "fmt"
)

func main() {
    nums(99, 100)
    nums(200)
    nums()
}

func nums(i ...int) {
    fmt.Println(i)
}
```

執行結果：

```
[99 100]
[200]
[]
```

- 參數不定函式也可以有其他參數，但數量不定的參數必須放在所有數量固定參數的最後面
- 此外，一個函式只容許一組數量可變參數
- 以下是錯誤寫法，會導致編譯錯誤

```go
package main

import (
    "fmt"
)

func main() {
    nums(99, 100, "james")
}

func nums(i ...int, person string) {
    fmt.Println(i)
}
```

執行後的錯誤訊息

```
can only use ... with final parameter in list
```

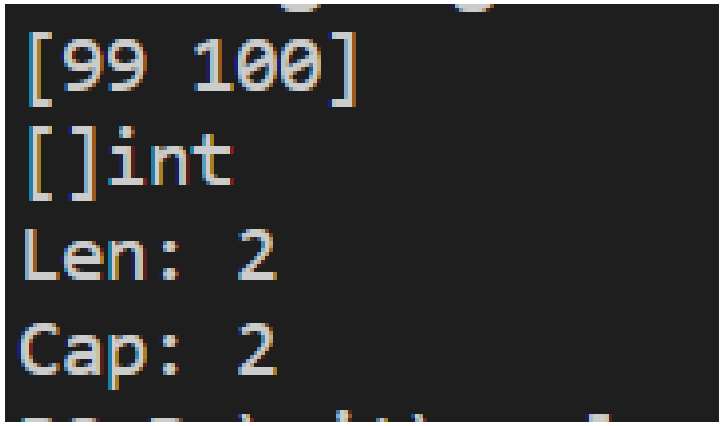
- 應該要這樣寫：

```go
package main

import (
    "fmt"
)

func main() {
    nums("james", 99, 100)
}

func nums(person string, i ...int) {
    fmt.Println(i)
}
```

- 我們可以來驗證一下參數 I 在函式內是否為切片，他的長度和容量分別是多少：

```go
package main

import (
    "fmt"
)

func main() {
    nums(99, 100)
}

func nums(i ...int) {
    fmt.Println(i)
    fmt.Printf("%T\n", i)
    fmt.Printf("Len: %d\n", len(i))
    fmt.Printf("Cap: %d\n", cap(i))
}
```

輸出結果:



將切片元素傳給參數不定函式

```go
package main

import (
    "fmt"
)

func main() {
    i := []int{5, 10, 15}
    nums(i)
}

func nums(i ...int) {
    fmt.Println(i)
}
```

- 會產生以下錯誤：

```
cannot use i (variable of type []int) as type int in argument to nums
```

- 修正方法：必須加上解包算符：

```go
package main

import (
    "fmt"
)

func main() {
    i := []int{5, 10, 15}
    nums(i...)
}

func nums(i ...int) {
    fmt.Println(i)
}
```

練習：數值加總函式

- 這次的練習要將數量不等的引數加總
- 我們會把兩種引數傳給函式：一種是引數清單，另一種則是切片
- 函式的回傳值會是整數

```go
package main

import (
    "fmt"
)

func main() {
    i := []int{5, 10, 15}
    fmt.Println(sum(5, 4))
    fmt.Println(sum(i...))
}

func sum(nums ...int) int {
    total := 0
    for _, num := range nums { //走訪數量不定參數的切片和加總
        total += num
    }
    return total
}
```

顯示結果：

```
9
30
```

5-4 匿名函數與閉包

- 到目前為止，我們使用的都是具名函式，也就是自帶識別名稱的函式，且必須在套件層級宣告
- 但其實有一種函式是沒有名稱的，且必須在其他函式內宣告，稱為匿名函式
- 匿名函式沒有名稱，也只能使用一次，除非你在建立它後指派給一個變數

- 匿名函式的宣告方法和具名函式幾乎一樣，就差在不會寫函式名稱

- 匿名函式可以搭配以下目的或功能：

1. 定義只使用一次的函式
2. 定義要回傳給另一個函式的函式
3. 定義Goroutine的程式碼區塊 (第16章)
4. 實作閉包
5. 搭配defer敘述延遲後執行程式碼

5-4-1 宣告匿名函式

- 下面式宣告匿名函式的最基本方式

```
 func main()  {
//宣告匿名函式(沒有名稱)
func() {
      fmt.Println(“Greeting”)
}()  //用()立即呼叫它
}
```

- 我們是在一個函式中宣告另一個函式，該函式沒有回傳值

- 注意到在匿名函式的右大括號後面有一對小括號()，稱為執行小括號

- 執行小括號會當場呼叫匿名函式並執行它

- 要傳給函式的引數必須寫在執行小括號：

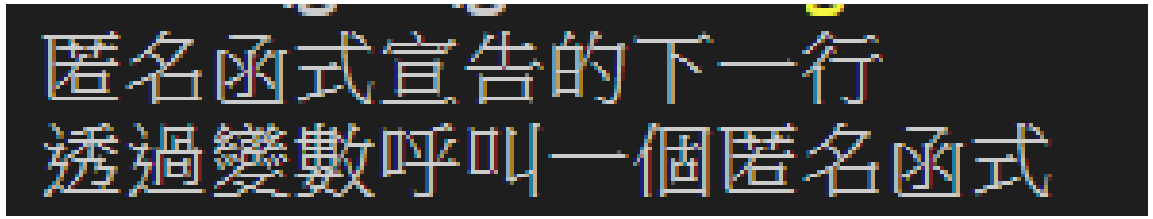
```
func  main() {
    message  :=  “Greeting”


    func(str  string)  {
            fmt.Println(str)
    }(message)
}
```

- 以上的寫法都是在宣告匿名函式後立刻執行它，且只用這麼一次

- 不過我們也可以把匿名函式儲存在變數裡，讓我們以截然不同的方式利用匿名函式：

```go
package main

import (
    "fmt"
)

func main() {
    f := func() { //f會變成func()型別
        fmt.Println("透過變數呼叫一個匿名函式")
    }
    fmt.Println("匿名函式宣告的下一行")
    f() //透過變數f呼叫匿名函式
}
```

- 執行結果：



練習：建立一個匿名函式來計算數值平方

- 匿名函式非常適合用來在其他函式中包裝小段的程式碼，好讓後面使用時能保持語法簡潔

- 以下我們就要建立一個匿名函式，傳遞一個引數給它來計算引數的平方值

```go
package main

import (
    "fmt"
)

func main() {
    x := 9
    sqr := func(i int) int {
        return i * i
    }
    fmt.Printf("%d 的平方為 %d\n", x, sqr(x))
}
```

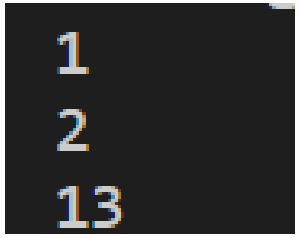
執行結果：

5-4-2 建立閉包

- 閉包式匿名函式的諸多型式之一
- 一般函式在離開某個函式範圍後，就沒辦法繼續引用父函式的區域變數，可是閉包卻能在離開後繼續引用
- 我們先看以下這個看似正常的匿名函式:

```go
package main

import (
    "fmt"
)

func main() {
    i := 0
    increment := func() int {
        i++
        return i
    }

    fmt.Println(increment())
    fmt.Println(increment())
    i += 10
    fmt.Println(increment())
}
```

- 匿名函式increment() 會把父函式的變數 i 遞增 1 並回傳，而在main()每次呼叫它時，都可以看到 i 的值改變：



- 但如果increment()是宣告在另一個函式裡面，然後被當成該函式的回傳值呢？

```go
package main

import (
    "fmt"
)

func main() {
    increment := incrementor() //接收回傳的函式

    fmt.Println(increment())
    fmt.Println(increment())
    fmt.Println(increment())
}

func incrementor() func() int {
    i := 0 //定義在匿名函式之父函式內的變數
    return func() int {
        i++
        return i
    }
}
```

執行節果：

```
1
2
3
```

- 當我們從main()呼叫incrementor()傳回匿名函式increment()時，可以發現它居然記得父函式的區域變數，儘管incrementor()已經執行完畢了！

- 在這裡，increment就是所謂的語意閉包，或簡稱閉包，因為這個函式包住了它所引用的外部變數

- 換言之，閉包能夠記住父函式的變數，即使離開了父函式的執行範圍也一樣

練習：建立一個閉包函式來製作倒數計時器

- 這個練習要來建立一個閉包函式，它有一個計數器，並會隨著每一次的呼叫，從我們指定的整數遞減到零

```go
package main

import "fmt"

func main() {
    max := 4

    counter := decrement(max) //取得閉包函式
    fmt.Println(counter())    //呼叫閉包函式
    fmt.Println(counter())
    fmt.Println(counter())
    fmt.Println(counter())
}

func decrement(i int) func() int {

    return func() int {
        //閉包函式會記住父函式的的參數i
        if i > 0 {
            i--
        }
        return i
    }
}
```

執行結果：

5-5 以函式為型別的參數

5-5-1 自訂函式型別

- 函式在go語言裡也算是一種型別，這表示我們可以將函式當成引數，傳遞給其他函式；函式也可以回函式，甚至可以拿函式賦值給變數(如前面的閉包)

- 若想把函式當作引數，需要指明該接收參數的型別，這時可以利用第四章學過的自訂函式型別，以便記憶

- 不僅如此，任何函式的參數與回傳值只要完全符合該自訂型別，就可被視為該自訂型別，這代表你能傳入多個不同的函式做為引數，只要他們都符合參數定義即可

- 下面來看看幾個自訂函數型別的例子 :

```
type message func()
```

- 這段程式定義了一個名為message的新函式型別 , 特徵為func() , 不具備輸入參數 , 也不提供回傳值

- 接著:

```
type clac func(int, int) string
```

- 這裡定義了一個名為calc的函式型別 , 他接受兩個整數型別參數 , 並回傳一個字串型別的值 (這個型別和前面的func()會是兩個不同的型別)

5-5-2 使用自訂函式型別的參數

- 現在來寫一點程式，展示一下自訂函式型別的用途：

```go
package main

import "fmt"

type calc func(int, int) string //自訂函式型別

func main() {
    calculator(add, 5, 6) //把其他函式當成引數
}


//add函式會符合自訂的 calc 型別
func add(i, j int) string {
    result := i + j
    return fmt.Sprintf("%d + %d = %d", i, j, result)
}

//接收自訂函式型別參數f
//效果等同於 f fun(int, int) string
func calculator(f calc, i, j int) {
    fmt.Println(f(i, j))
}
```

執行結果：

```
5 + 6 = 11
```

- 在以上程式碼中，函式add(I, j int)的特徵與calc型別func(int, int)定義相同，因此可以被視為calc型別

- 而函式calculator()接受一個calc型別的參數，因此我們可以將add傳給它

- 下面我們稍微修前面的例子，示範如何將幾個不同的函式傳給calculator():

```go
package main

import (
    "fmt"
)

type calc func(int, int) int

func main() {
    calculator(add, 5, 6)
    calculator(subtract, 10, 5)
}

func add(i, j int) int {
    return i + j
}

func subtract(i, j int) int {
    return i - j
}

func calculator(f calc, i, j int) {
    fmt.Println(f(i, j))
}
```

執行結果：

練習：建立各種函式來計算薪資

- 這次我們要建立幾個函式，以便計算開發人員和其經理的薪資
- 但也希望這個程式要有足夠的彈性，以便將來也可以計算不同職別人員的薪資

```go
package main

import "fmt"

type salaryFunc func(int, int) int

func main() {
    devSalary := salary(50, 2000, developerSalary)
    bossSalary := salary(150000, 25000, managerSalary)

    fmt.Printf("經理薪資    : %d\n", bossSalary)
    fmt.Printf("程式設計師薪資    : %d\n", devSalary)
}

func salary(x, y int, f salaryFunc) int {
    pay := f(x, y)
    return pay
}

func managerSalary(baseSalary, bonus int) int {
    return baseSalary + bonus
}

func developerSalary(hourlyRate, hoursWorked int) int {
    return hourlyRate * hoursWorked
}
```

- 從以上練習可以看到如何以一個salary()函式簡化程式碼

- 若將來需要額外計算其他工作人員的薪資，只需要寫一個新的函式使它符合salary()要求的輸入型別即可

5-5-3 用自訂函式型別作為回傳值

- 不只能在函式參數使用自訂函式型別，亦可將之當成回傳值
- 我們在近一步改造之前計算機程式的例子，讓它能依據情況傳回不同的函式

```go
package main

import "fmt"

func main() {
    add := calculator("+") //接收calculator()傳回的函式
    subtract := calculator("-")

    fmt.Println(add(5, 6))
    fmt.Println(subtract(10, 5))

    fmt.Printf("add()      型別: %T\n", add)
    fmt.Printf("subtract() 型別: %T\n", subtract)
}

func calculator(operator string) func(int, int) int {
    switch operator {
    case "+":
        return func(i, j int) int {
            return i + j
        }
    case "-":
        return func(i, j int) int {
            return i - j
        }
    }
    return nil
}
```

執行結果：

```
11
5
add()      型別: func(int, int) int
subtract() 型別: func(int, int) int
```

5-6 defer

5-6-1 用defer延後函式執行

- defer 可以延後函式的執行時機，使該函式等到父函式結束的前一課才會被執行

- 白話來說，當你呼叫某個函式時加上defer, 它並不會當場執行，而是變成父函式中最後被執行的部分

- 底下看一個例子來理解看看：

```go
package main

import "fmt"

func main() {
    defer done()
    fmt.Println("main()開始")
    fmt.Println("main()結束")
}

func done() {
    fmt.Println("換我結束!!")
}
```

執行結果：

```
main()開始
main()結束
換我結束!!
```

- 在main()函式裡，我們以defer延後了done()的執行時機
- 被延後的函式通常是用來“善後”的，包括釋出資源 / 關閉以開啟的檔案 / 關閉仍在連結的資料庫連線 / 移除程式先前建立的設定等
- 此外，defer也可以用來從程式的錯誤狀況復原，下一章會談到這個部分
- defer敘述不限於搭配具名函式，也可以對匿名函式使用
- 下面拿前一個程式為例：

```go
package main

import "fmt"

func main() {
    defer func() {
        fmt.Println("換我結束!!")
    }() //要記得最後的小括號!!

    fmt.Println("main()開始")
    fmt.Println("main()結束")
}
```

執行結果：

5-6-2 多重defer的執行順序

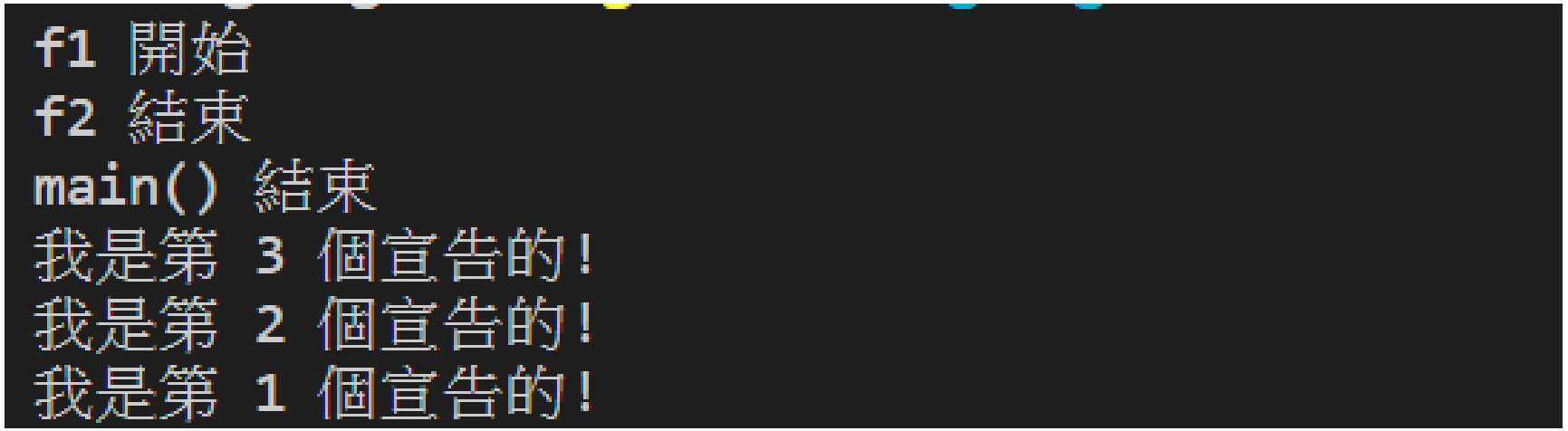
- 當我們對多個函式使用defer時，其執行順序會依循所謂的先進後出原則(First In Last Out, FILO)

- 可以把FILO的過程想像成疊盤子：最早放下去的盤子會最晚使用

- 來看一個例子：

```go
package main

import "fmt"

func main() {
    defer func() {
        fmt.Println("我是第 1 個宣告的!")
    }()
    defer func() {
        fmt.Println("我是第 2 個宣告的!")
    }()
    defer func() {
        fmt.Println("我是第 3 個宣告的!")
    }()
    f1 := func() {
        fmt.Println("f1 開始")
    }
    f2 := func() {
        fmt.Println("f2 結束")
    }

    f1()
    f2()
    fmt.Println("main() 結束")
}
```

執行結果：



5-6-3 defer對變數的副作用

- 使用defer敘述時請務必謹慎，其中一個你必須考量到的是，若defer函式有用到外部變數，執行時會發生甚麼結果

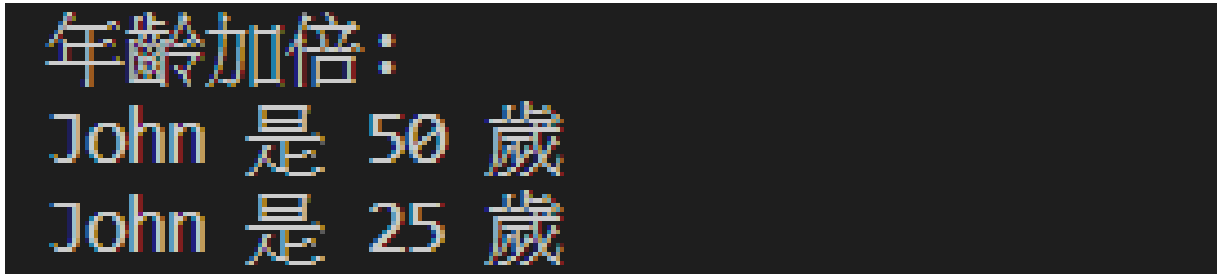
- 當變數傳給被延後的函式時，函式會取得變數在傳遞那一刻當下的值

- 就算變數值在該函式之後有所變動，等到defer函式的實際執行時，它看到的變數值也不會反映外圍函式中的變動：

```go
package main

import "fmt"

func main() {
    age := 25
    name := "John"
    defer personAge(name, age)

    age *= 2
    fmt.Println("年齡加倍:")
    personAge(name, age)
}

func personAge(name string, i int) {
    fmt.Printf("%s 是 %d 歲\n", name, i)
}

```

執行結果

- 即便age變數在呼叫personAge()後有所變動，但該函式看到的仍是變動前的值:



本章結束