CH4 複合型別

4-1 前言

- 前一章介紹了核心型別，但如果要建構更複雜的資料模型，光靠核心型別是不夠的

- 在本章中，會學到go語言中讓我們整理資料，以便因應更複雜挑戰的進階功能，並看看要如何替型別賦予行為

4-2 型別集合

- 處理大量性質相似的資料時，我們會把它們放到一個集合中
- Go語言的集合有陣列，切片和map三種，這些同樣屬於強型別，且容易用迴圈走訪
- 不過這些集合有不同的性質，因此有其個自適用的場合

4-3 陣列

4-3-1 定義一個陣列

- 陣列是go語言最基本的集合形式，定義陣列時，必須指定陣列所含的資料形別，以及陣列的大小：
  - [長度]型別，例如[10]int就是含有10個整數原素的陣列
- 陣列元素可以式任意型別，包括指標，甚至是其他陣列，但只能有一種型別
- 另外若宣告陣列時忘記加上長度，宣告依舊會成立，但會得到一個切片而不是陣列

- 若要在宣告時就賦予初始值，可以這樣寫：
  - [長度]型別{值1, 值2, ..., 值N}
- 不過在賦予陣列初始值時，可以讓go語言根據提供的初始值數量來設定長度，辦法是將陣列長度的地方換成 ... (三個點)，例如：
  - [...]string{9,9,9,9,9}會建立一個長度為5的陣列

練習：定義空陣列

```go
package main

import "fmt"

func defineArray() [10]int { //函式的回傳值微陣列
    var arr [10]int
    return arr
}

func main() {
    fmt.Printf("%#v\n", defineArray()) //印出陣列的型別與值
}

```

- 執行結果：

```
[10]int{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

- 練習中，我們定義了一個陣列，但沒有給任何起始資料，由於陣列大小是固定的，因此當我們列印內容時，會顯示10個元素的值，只不過這些值都是陣列元素型別的零值 (0)

4-3-2 陣列的比較

- 若有兩個陣列都接受相同型別,但長度不一樣,那這兩者就不拿來相比對
- 長度不一樣,元素型別也不一樣的陣列當然更不能拿來相比了

練習：比較陣列是否相同

- 這次練習要來做陣列的比較
- 首先要定義若干個陣列，有些可以相互比較，其他則否，然後我們要修正陣列無法比較的問題

```go
package main

import "fmt"

func compArrays() (bool, bool, bool) {
    var arr1 [5]int
    arr2 := [5]int{0}
    arr3 := [...]int{0, 0, 0, 0, 0}
    arr4 := [9]int{0, 0, 0, 0, 9}
    //回傳陣列的比較結果
    return arr1 == arr2, arr1 == arr3, arr1 == arr4
}

func main() {
    comp1, comp2, comp3 := compArrays()
    fmt.Println("[5]int == [5]int{0}                :", comp1)
    fmt.Println("[5]int == [...]int{0, 0, 0, 0, 0}:", comp2)
    fmt.Println("[5]int == [5]int{0, 0, 0, 0, 9}  :", comp3)
}
```

- 顯示結果：

```
# command-line-arguments
ch4\4-3-2.go:11:45: invalid operation: arr1 == arr4 (mismatched types [5]int and [9]int)
```

- 這個錯誤告訴你arr1([5]int) 和 arr4([9]int)被視為不同型別，修正方式如下：

```go
package main

import "fmt"

func compArrays() (bool, bool, bool) {
    var arr1 [5]int
    arr2 := [5]int{0}
    arr3 := [...]int{0, 0, 0, 0, 0}
    arr4 := [5]int{0, 0, 0, 0, 9}
    //回傳陣列的比較結果
    return arr1 == arr2, arr1 == arr3, arr1 == arr4
}

func main() {
    comp1, comp2, comp3 := compArrays()
    fmt.Println("[5]int == [5]int{0}              :", comp1)
    fmt.Println("[5]int == [...]int{0, 0, 0, 0, 0}:", comp2)
    fmt.Println("[5]int == [5]int{0, 0, 0, 0, 9}  :", comp3)
}
```

- 再次執行的結果如下：

```
[5]int == [5]int{0}                  : true
[5]int == [...]int{0, 0, 0, 0, 0}: true
[5]int == [5]int{0, 0, 0, 0, 9}   : false
```

- 只有陣列能這樣比較，其他集合型別(切片，map)只能用迴圈走訪兩個集合，再逐一比對元素質

- 因此如果你想在程式中比較集合，使用陣列會快很多

4-3-3 透過索引鍵賦值

- 到目前為止，我們在賦予陣列初始值時都是讓go語言決定哪個元素接收這些值，不過go語言也允許在宣告時直接針對特定索引的元素賦值 ：
  - [長度]型別{索引1 : 值1, 索引2, 值2,……索引N, 值N}
- 當陣列的索引具有特殊意義或只想對某個元素賦值，這種方式就很實用

練習：以索引鍵賦予陣列初始值

- 這個練習要在建立陣列時以索引鍵對特定元素賦值，然後來必較一下陣列，最後會印出其中一個陣列並觀察其內容

```go
package main

import "fmt"

var (
    arr1 [10]int
    arr2 = [...]int{9: 0}
    arr3 = [10]int{1, 9: 10, 4: 5}
)

func compArrays() (bool, bool) {
    return arr1 == arr2, arr1 == arr3
}

func main() {
    comp1, comp2 := compArrays()
    fmt.Println("[10]int == [...]{9:0}              :", comp1)
    fmt.Println("arr2                               :", arr2)
    fmt.Println("[10]int == [10]int{1, 9: 10, 4: 5}}:", comp2)
    fmt.Println("arr3                               :", arr3)
}
```

- 執行結果：

```
[10]int == [...]{9:0}                 : true
arr2                                  : [0 0 0 0 0 0 0 0 0 0]
[10]int == [10]int{1, 9: 10, 4: 5}}: false
arr3                                  : [1 0 0 0 5 0 0 0 0 10]
```

- 可以看到，陣列變數在定義時一樣能不寫出型別，而是讓go語言根據初始值來判斷

- 也可以發現go語言索引賦值的彈性很大，使得建立陣列時分常方便

4-3-4 讀取陣列元素值

- 若要讀取陣列中的單一元素，可以這樣寫：
  - 值 = 陣列[索引]
  - 例如 arr[0] 可以讀出陣列 arr 的第一個元素
- 陣列中元素的順序都是保證固定的，這種穩定的特質表示只要元素被放在某索引，它就永遠被該索引存取

練習：從陣列讀取單一元素

- 這個練習要定義一個陣列，然後賦予一些單字作為初始值，最後將這些單字讀出來和組成訊息，並將之印出

```go
package main

import "fmt"

func message() string {
    arr := [...]string{
        "ready",
        "Get",
        "Go",
        "to",
    }
    //用fmt.Sprintln()回傳格式化自串
    return fmt.Sprintln(arr[1], arr[0], arr[3], arr[2])
}

func main() {
    fmt.Print(message()) //印出格式化字串
}
```

- 顯示結果：

- ```
  Get ready to Go
  ```
  ```

4-3-5 寫入值到陣列

- 只要陣列定義完畢，就可以用索引修改個別元素：
  - 陣列[索引] = 值
- 在現實中，我們很常會在定義好集合後，藉由資料輸入或程式運算過程來修改集合中的資料

練習：對陣列元素賦值

- 這次的練習同樣也是要定義一個陣列，然後賦予一些單字作為元素初始值，然後更動一下這些單字的讀出順序，把它們組成訊息後印出

```go
package main

import "fmt"

func message() string {
    arr := [4]string{"ready", "Get", "Go", "to"}
    arr[1] = "It's" //修改元素值
    arr[0] = "time"
    //輸出修改後的元素值
    return fmt.Sprintln(arr[1], arr[0], arr[3], arr[2])
}

func main() {
    fmt.Print(message())
}
```

- 顯示結果：

```
It's time to Go
```

4-3-6 走訪一個陣列

- 最常操作陣列的方式是透過迴圈

- 有了迴圈，就可以對陣列中的每一個元素套用相同的處理，無須再針對單一值建立變數

- 一個典型的for I 迴圈會寫成這樣：

```
for i := 0; i < len(陣列); i++ {
    //存取 陣列[i]
}
```

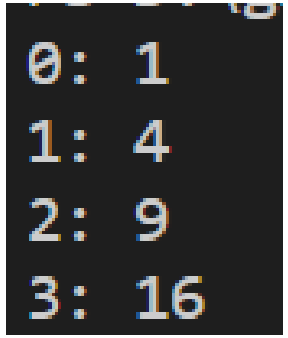
- 或許你會說直接把陣列的長度寫在條件判斷式中就好，幹嘛用len()求值？ 雖然這樣的寫法比較直覺，但把值寫死是一個壞習慣，因為這會讓程式碼日後更難維護。若你之後需要修改陣列的長度，寫死的值會變成難以找出的bug，甚至會造成執型時的錯誤

練習：用for i 迴圈走訪和處理陣列

- 這裡會定義一個陣列，並用若干數字賦予初始值
- 我們要利用迴圈逐一走訪和處理這些值，並把結果放進一個訊息然後印出

```go
package main

import "fmt"

func message() string {
    m := ""
    arr := [4]int{1, 2, 3, 4}
    for i := 0; i < len(arr); i++ {
        arr[i] = arr[i] * arr[i]                    //將原本的數字平方
        m += fmt.Sprintf("%v: %v\n", i, arr[i]) //用格式化字串將索引和值一一連起來
    }
    return m
}

func main() {
    fmt.Print(message())
}
```

- 執行結果：



練習：使用for i 迴圈走訪和處理陣列(參數版)

- 這個練習中，我們要把陣列傳給函式，而函式會對陣列作處理後回傳
- 為了能處理相同的陣列，函式的參數也必須指定同樣的陣列長度

```go
package main

import "fmt"

func fillArray(arr [10]int) [10]int { //參數和回傳值都是10個元素的int陣列
    for i := 0; i < len(arr); i++ {
        arr[i] = i + 1 //陣列元素會是1~10
    }
    return arr
}

func opArray(arr [10]int) [10]int {
    for i := 0; i < len(arr); i++ {
        arr[i] = arr[i] * arr[i] //元素平方
    }
    return arr
}

func main() {
    var arr [10]int
    arr = fillArray(arr)
    arr = opArray(arr)
    fmt.Println(arr)
}

```

- 顯示結果：

```
[1 4 9 16 25 36 49 64 81 100]
```

4-4 切片

- 陣列很方便，但是對於長度的硬性規定會帶來一些麻煩。為了提供更好的便利性，go語言提供了切片 。

- 切片 讓你能夠建立數字索引鍵的有序集合，卻不用擔心長度的問題。切片 底下的核心依舊是陣列。

- 切片 用起來跟陣列一模一樣；他只能容納單一型別的元素，可以用中括號[]讀任何一個元素，也能用迴圈走訪。

- 切片的另一個優勢是你可以用go語言的內建函式append()輕易添加元素：
  - 新切片 = append(切片，新元素)
- 在日常中切片的使用會比陣列來的多，因為它帶來長度上的彈性，除非你需要把元素限定在某個數量時才會使用陣列

4-4-1 使用切片

練習：建立與使用切片

- 在這個練習中，會展示出切片的彈性，能輕易的從切片讀取資料/傳給其他函式/走訪和附加新的元素

- 這支程式需要在執行時輸入三個參數，然後它會判斷長度最長的是誰

- 為了讀取使用者在執行時輸入的參數，會使用os套件的Arg變數
- 它在os套件的定義如下：
  - var Args []string
- 可以知道os.Args是字串切片型別，它的內容是使用者用go run執行程式時, 附加在後面的參數
  - go run <.go 程式檔> 參數1 參數2 .....

```go
package main

import (
    "fmt"
    "os"
)

//將os.Args 參數的每一個元素放進一個切片後回傳
func getPassedArgs(minArgs int) []string {
    if len(os.Args) < minArgs { //如果使用者提供的參數不足就結束
        fmt.Printf("至少需要輸入 %v 個參數\n", minArgs)
        os.Exit(1) //強制結束程式
    }
    var args []string //建立空切片
    //os.Args的第一個參數是執行的程式名稱，不是參數
    //所以要從索引1開始走訪:
    for i := 1; i < len(os.Args); i++ {
        args = append(args, os.Args[i])
    }
    return args //回傳切片
}
```

```go

func findLongest(args []string) string {
    var longest string
    for i := 0; i < len(args); i++ {
        if len(args[i]) > len(longest) {
            longest = args[i]
        }
    }
    return longest
}

func main() {
    if longest := findLongest(getPassedArgs(3)); len(longest) > 0 {
        fmt.Println("傳入的最長單字:", longest)
    } else {
        fmt.Println("發生錯誤")
        os.Exit(1)
    }
}
```

- 執行結果：

```
PS D:\git\go lan> go run .\ch4\4-4-1.go  get ready to go
傳人的最長單字: ready
```

4-4-2 為切片附加多重元素

- append()其實可以附加多個值到切片裡：
  - 新切片 = append(切片，新元素1，新元素2, …)
- 這也意味著你可以在append()傳入一個切片，並在後面加上解包算符(unpack operator), 也就是三個點 … 來解開它，使得切片元素會被拆成獨立的值傳入append()

練習：一次為切片加入多個元素

- 這個練習要用append()的數量不定參數和解包算符，將多筆資料放進切片中，首先是利用是先定義好的資料，接著則加入使用使用者動態提供的資料

```go
package main

import (
    "fmt"
    "os"
)

func getPassedArgs() []string {
    //回傳包含使用者參數的切片
    //由於得去掉地一個參數 , 這裡仍得用迴圈
    var args []string
    for i := 1; i < len(os.Args); i++ {
        args = append(args, os.Args[i])
    }
    return args
}

```

```go
func getLocals(extraLocals []string) []string {
    //回傳包含語系名稱切片


    var locales []string
    //加入兩個預設"語系元素"
    locales = append(locales, "en_US", "fr_FR")
    //加入使用者提供的數量不定參數 (extraLocals 可能為空切片)
    locales = append(locales, extraLocals...)
    return locales
}

func main() {
    locales := getLocals(getPassedArgs())
    fmt.Println("要使用的語系:", locales)
}

```

- 顯示結果：

```
PS D:\git\go lan> go run .\ch4\4-4-2.go  fr_CN, en_AU
要使用的語系: [en_US fr_FR fr_CN en_AU]
```

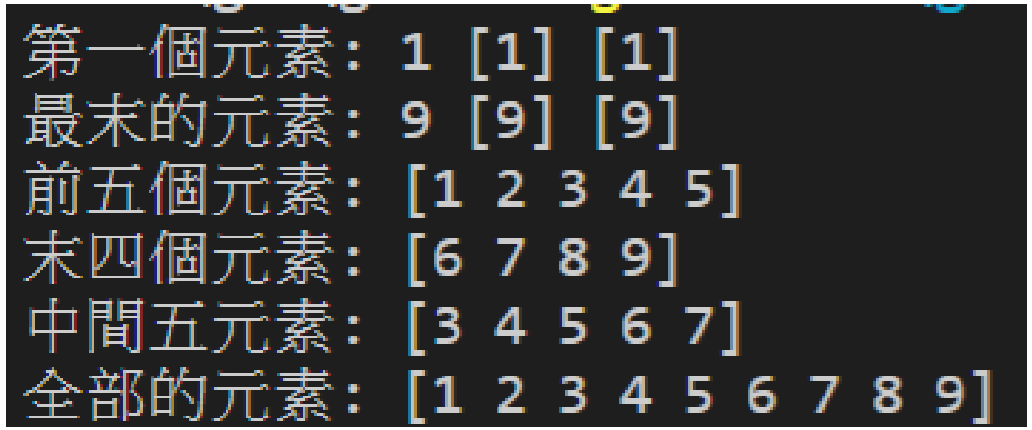
4-4-3 從切片和陣列建立新的切片

- 你不只可以用中括號取出陣列的單一元素，也可以用類似的方式取出一段新的切片，常見的語法為：
  - 新切片 = 陣列或切片[起始索引:結束索引(不含)]
- 這會讓go語言參考來源陣列或切片，將指定的範圍放入新切片中
- 起始索引和結束索引都可以省略不寫；若省略起點，會從索引0開始取值；若省略結束索引，go語言會取到最後一個值 (假如兩個都省略，就等於取出原集合的所有元素)

練習：從切片再建立其他切片

```go
package main

import "fmt"

func message() string {
    s := []int{1, 2, 3, 4, 5, 6, 7, 8, 9}
    //用fmt.Sprintln回傳格式化字串(每一行會包含換行)
    m := fmt.Sprintln("第一個元素:", s[0], s[0:1], s[:1])
    m += fmt.Sprintln("最末的元素:", s[len(s)-1], s[len(s)-1:len(s)], s[len(s)-1:])
    m += fmt.Sprintln("前五個元素:", s[:5])
    m += fmt.Sprintln("末四個元素:", s[5:])
    m += fmt.Sprintln("中間五元素:", s[2:7])
    m += fmt.Sprintln("全部的元素:", s[:])
    return m
}

func main() {
    fmt.Print(message())
}
```

- 顯示結果：


第一個元素: 1 [1] [1]
最末的元素: 9 [9] [9]
前五個元素: [1 2 3 4 5]
末四個元素: [6 7 8 9]
中間五元素: [3 4 5 6 7]
全部的元素: [1 2 3 4 5 6 7 8 9]

4-4-4 了解切片的內部運作

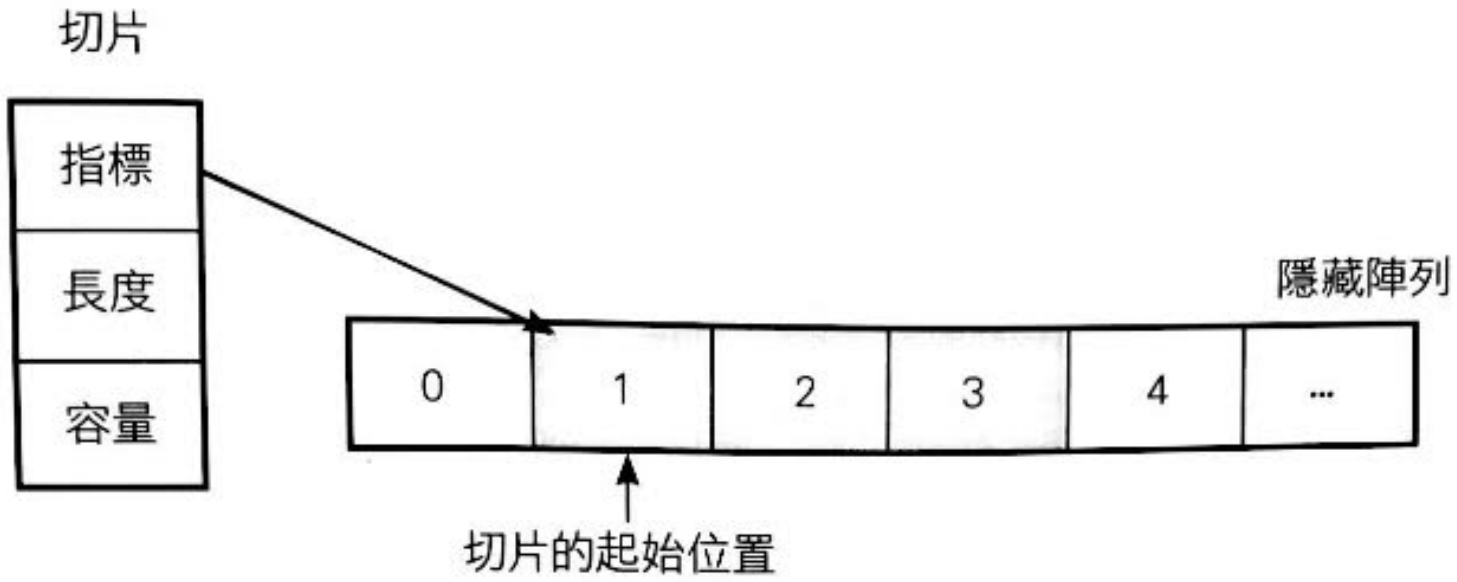
- 切片十分好用，但若不了解它的運作方式可能會造成難以察覺的錯誤

- 陣列是一種資料型別，不過有些型別是可以複製的，也可以與同型別的資料作比較。這些資料一旦複製，就與原本的資料無關了。但切片不一樣，它的運作方式比較像指標，但又不是真正的指標

- 要安全的操作切片，你必須知道切片底下其實有個隱藏陣列，用來儲存真正的資料。若你對切片加入新值，隱藏陣列有可能會換成更大的陣列 。正因為隱藏陣列都是在背景進行自動管理，一開始會較難理解切片的一些奇怪表現

- 切片擁有三項隱藏屬性：

1. 指向隱藏陣列起始位置的指標(pointer)
2. 長度(length)
3. 容量(capacity)



- 有了上面三種屬性，所以可以用len()查詢切片長度，而另一個內建函式cap()可以告訴我們切片的容量

- 長度指的是現有元素的數量，而容量是指可以放進的元素總量

- 當你用append()對切片加入一個值時，取決於切片的容量值，可能發生的情況如下:
  - 容量尚有剩餘，那麼新值就可以放入隱藏陣列，並更新切片長度值

  - 容量已滿，go語言會產生一個更大的新隱藏陣列，把舊陣列的元素複製過去，並放入新值，最後更新長度值，需要的話也更新切片在陣列的起始位置

- 切片也會記住它在原始陣列中的起始位置，若切片長度和原集合一樣長，其指向的陣列起點自然是隱藏陣列的第一個元素

- 你可以在定義切片時就做初始化，控制它的長度和容量，方法是使用內建函數make():
  - 新切片 = make(切片型別, 長度, [容量])
- make()的長度參數是必填的，容量則為選擇性(不填時即為長度)
- 它實際上會建立一個內容全是零值的陣列，並回傳指向這個陣列的新切片

練習：用make控制切片 容量

- 這個練習要利用var關鍵字和make()建立若干切片，並顯示其長度和容量

```go
package main

import "fmt"

func genSlices() ([]int, []int, []int) {
    var s1 []int                 //用var建立切片
    s2 := make([]int, 10)        //用make建立切片，指定長度
    s3 := make([]int, 10, 50) //用make建立切片，指定長度和容量
    return s1, s2, s3
}

func main() {
    s1, s2, s3 := genSlices()
    fmt.Printf("s1: len = %v cap = %v\n", len(s1), cap(s1))
    fmt.Printf("s2: len = %v cap = %v\n", len(s2), cap(s2))
    fmt.Printf("s3: len = %v cap = %v\n", len(s3), cap(s3))
}
```

- 顯示結果：

```
s1: len = 0 cap = 0
s2: len = 10 cap = 10
s3: len = 10 cap = 50
```

- 如果你很肯定切片有多大，指定容量可以省去go語言做額外的隱藏陣列管理，提升運作效率

4-4-5 切片的隱藏陣列置換

- 有鑑於切片的可變型和運作方式，切片是無法互相比較的
- 唯一能做的是拿切片和nil做比較
- 前面提過，切片是一種特殊的資料結構，本身並不直接存值，而是透過隱藏陣列存放資料
- 切片本身只是指向隱藏陣列的指標，代表陣列從何開始，然後舊址紀錄切片的長度和容量，這三個屬性使切片變成隱藏陣列的窗格(window)，就像在陣列上開了一扇窗，讓你可以窺探底下陣列的一部份樣貌

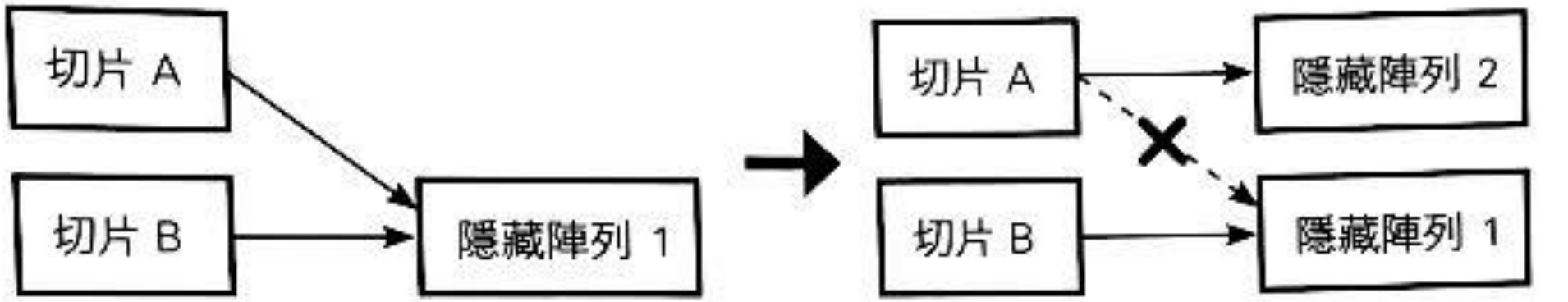
- 窗格可能和陣列一樣大，也可能只有一小塊

- 同一個隱藏陣列也能被多個切片共用，但彼此的窗格不見得一樣大，因為某些切片的值會比其他切片多

- 既然可能共用隱藏陣列，當你修改其中一個值時，實際是修改隱藏陣列，連帶會改變其他陣列的窗格 (若你不知道這個特性，開發程式時可能會產生意想不到的bug)

- 而更大的問題是，當切片需要擴充到超過其隱藏陣列的大小(窗格比隱藏陣列更大)時，go語言會建立更大的陣列，把舊內容搬過去再把切片的指標重新指向新的陣列

- 而這種陣列置換的動作，可能導致不同的切片失去連結性，在此之後更改其中一方的原素，改變就不會反映在另一個切片中



切片 A 和 B 都指向同一個隱藏陣列

對切片 A 新增元素，導致陣列置換發生

- 若想複製一份切片，又要確保該切片能指向一個新的隱藏陣列，不要和原本的切片有關聯，有以下兩種方式：

1. 用append()把來源切片附加到另一個無關的新陣列

2. 用內建函式copy()把來源切片複製到目標切片：
   copy(目標切片, 來源切片)

注意：使用copy函式時，Go 語言不會改變目標切片的大小，所以要確定目標切片有足夠的空間容納複製的元素

練習：觀察切片的連結行為

- 這次練習要探索把資料從一個切片複製到另一個切片的方式，然後觀察這些方式對內部運作的影響

```go
package main

import "fmt"

func linked() (int, int, int) {
    s1 := []int{1, 2, 3, 4, 5}
    s2 := s1     //直接複製切片 (指向同一個陣列)
    s3 := s1[:] //從切片建立相同長度的切片(指向同一個陣列)
    s1[3] = 99
    return s1[3], s2[3], s3[3]
}

func noLink() (int, int) {
    s1 := []int{1, 2, 3, 4, 5}
    s2 := s1
    s1 = append(s1, 6) //加入新值, 超過s1的容量, 使s1置換陣列
    s1[3] = 99
    return s1[3], s2[3]
}

```

```go
func capLinked() (int, int) {
    s1 := make([]int, 5, 10) //將s1容量設為10
    s1[0], s1[1], s1[2], s1[3], s1[4] = 1, 2, 3, 4, 5
    s2 := s1
    s1 = append(s1, 6) //加入新值，不超過s1容量，s1不會置換陣列
    s1[3] = 99         //更動仍會反映在s2
    return s1[3], s2[3]
}

func capNoLink() (int, int) {
    s1 := make([]int, 5, 10)
    s1[0], s1[1], s1[2], s1[3], s1[4] = 1, 2, 3, 4, 5
    s2 := s1
    s1 = append(s1, []int{10: 11}...) //加入新值(索引10)，超過s1容量
    s1[3] = 99
    return s1[3], s2[3]
}

```

```go
func copyNoLink() (int, int, int) {
    s1 := []int{1, 2, 3, 4, 5}
    s2 := make([]int, len(s1)) //用make建立與s1同長的切片
    copied := copy(s2, s1)     //用copy複製切片並傳回複製的數量
    s1[3] = 99
    return s1[3], s2[3], copied
}

func appendNoLink() (int, int) {
    s1 := []int{1, 2, 3, 4, 5}
    s2 := append([]int{}, s1...) //將s1的元素附加到新的切片
    s1[3] = 99
    return s1[3], s2[3]
}

```

```go
func main() {
    l1, l2, l3 := linked()
    fmt.Println("有連結                    :", l1, l2, l3)
    nl1, nl2 := noLink()
    fmt.Println("無連結                    :", nl1, nl2)
    cl1, cl2 := capLinked()
    fmt.Println("有設容量, 有連結          :", cl1, cl2)
    cnl1, cnl2 := capNoLink()
    fmt.Println("有設容量, 無連結          :", cnl1, cnl2)
    copy1, copy2, copied := copyNoLink()
    fmt.Print("使用 copy(), 無連結  : ", copy1, copy2)
    fmt.Printf(" (複製了 %v 個元素)\n", copied)
    a1, a2 := appendNoLink()
    fmt.Println("使用 append(), 無連結:", a1, a2)
}
```

- 顯示結果

```
有連結                          : 99 99 99
無連結                          : 99 4
有設容量，有連結                : 99 99
有設容量，無連結                : 99 4
使用 copy()，無連結             : 99 4 (複製了 5 個元素)
使用 append()，無連結: 99 4
```

問題 : 想複製切片而不影響原切片的作法 , copy()和append(), 哪個方式比較常用呢 ?

答 : append()較常用 , 因為copy的目標切片有長度的限制 , 需要考慮目標切片的長度是否足夠

小技巧：更有效率的append()複製切片

- 其實還有一個更快的寫法可以達到複製切片而不影響原切片的效果：

```
s1 := []int{1, 2, 3, 4, 5}
s2 := append(s1[:0:0], s1...)
```

這裡用了極少用到的擷取範圍語法：
切片[起始索引:結束索引(不含):容量]

- 雖然s1[:0:0]仍指向元切片的隱藏陣列，但既然切片的容量被設定為0，在接收資料時會因為超過容量而自動置換陣列，這樣就不用浪費記憶體空間額外多件一個空切片了

4-5 映射表(map)

4-5-1 map基礎

- 陣列和切片彼此相似，有時也可通用，但Go 語言的另一種集合映射表(map)與前兩者截然不同。這是因為Go 語言的 map 是要用在不同的用途。

- 以電腦科學術語來說，Go 語言的 map 是一種雜湊表 (hashmap)。與其他形式的集合相比，雜湊表的差別在於索引鍵：對陣列或切片來說，索引鍵只代表位置和計數器，本身並無意義，跟資料也並無直接的關係。但在 map中，索引鍵本身同時也是資料，與其對應值有真實的關聯。

- 舉例來說，你可以在map 儲存一系列使用者帳戶資料。索引鍵是員工編號，這本身也是貨真價實的資料，不只是代表資料位置而已。如果某人給你他的員工編號，你就可以調出相關帳戶資料，毋須逐一走訪整個陣列或切片才能找到。

- 有了map，你就可以快速建立、取得和刪除集合中的資料。

- 註 :雜湊表的意思是把鍵值用雜湊函式轉換過，讓它盡量以獨一無二的值儲存在雜湊表中。如此一來只要把欲查詢的鍵轉成雜湊值，通常就能馬上找到，效率極高。

- 要取得 map 內的元素，做法和存取陣列跟切片很類似：使用中括號[]

- 任何可以用來做比較的資料型別，都可以當成 map 的索引鍵型別，不論整數或字串都可以。切片不能當作索引鍵，因為切片不能被比較；至於 map 的資料值則可以是任何型別，包括指標、切片、甚至是另一個 map

- 不過建議不要用 map 來儲存有序資料。即使你用 int 整數型別當map 的索引鍵，這也不保證走訪整個 map 時資料會照索引鍵大小排序。

- 事實上 Go語言在你用 for range 時會故意打亂元素順序（不是隨機而是非決定性 (non-deterministic) 的），好確保沒人只靠元素的次序取值。

- 若你真的需要以特定順序走訪資料，最好還是改用陣列或切片

map的定義與新增元素方式

- map的定義語法如下：

  map[鍵型別]值型別

- 由於go語言部會替你初始化map的鍵與值，必須在定義map時一併賦予鍵與值：

  map[鍵型別]值型別{鍵1:值1, 鍵2:值2, ..., 鍵N,值N}

- 若你嘗試對一個未經初始化的map賦值，會引發執行期間panic(第6章介紹)，所以請盡量避免定義一個零值map

- 更常用的方式是使用make()傳回一個經過初始化的map, 但make()在此傳入的參數會和初始化切片時有所不同：

  make(map[鍵型別]值型別, 容量)

- 和切片不同的是，go語言不會替map建立索引鍵，所以無法在make()指定map的長度

- 至於map的容量是可選擇的，但不能用cap()查詢map的容量

- 還有注意map和切片一樣，也有複製上需要注意的特質
- 若要複製map，就要用迴圈走訪其中一個map，然後寫入另一個map
- map初始化後，就可以用中括號對map加入元素(鍵與值)，且不用擔心其長度問題：

  map名稱[索引鍵] = 值

練習：建立/讀取和寫入一個map

- 在這個練習中會定義一個map，放一些資料來初始化，再增添一個新元素，最後印出map的內容

```go
package main

import "fmt"

func getUsers() map[string]string {
    users := map[string]string{
        "305": "Sue",
        "204": "Bob",
        "631": "Jake",
    }
    users["073"] = "Tracy"
    return users
}

func main() {
    fmt.Println("Users:", getUsers())
}
```

- 顯示結果：

```
Users: map[073:Tracy 204:Bob 305:Sue 631:Jake]
```

- 可以發現，map的處理方式和切片或陣列很相似

- 至於map的使用時機，取決於你要儲存何種資料，以及取用資料的方式

4-5-2 從map讀取元素

- 只有用索引鍵在map取值時，才會知道該鍵存在與否

- 萬一索引鍵不存在，go語言會回傳map資料型別的nil值，所以藉由nil值判斷索引鍵是否存在也是可行的，但你無法保證nil就代表查無此鍵 (nil也有可能是有用的資料)

- 在這種情況下，你在取值時可從map多接收一個參數，寫法如下：

  值, 存在狀態 := map名稱[索引鍵]
  - 存在狀態是布林值，若索引存在為ture，反之為false

練習：讀取map元素並檢查它存在與否

- 這個練習將以直接存取和迴圈走訪兩種方式，從一個map讀取資料，並檢視某個索引鍵是否存在

```go
package main

import (
    "fmt"
    "os"
)

func getUsers() map[string]string {
    return map[string]string{
        "305": "Sue",
        "204": "Bob",
        "631": "Jake",
        "073": "Tracy",
    }
}

func getUser(id string) (string, bool) {
    users := getUsers()
    user, exists := users[id]
    return user, exists
}

```

```go
func main() {
    if len(os.Args) < 2 { //檢查是否至少傳入一個參數
        fmt.Println("未傳入使用者 ID")
        os.Exit(1)
    }
    userID := os.Args[1]
    name, exists := getUser(userID) //取值並檢查鍵是否存在
    if !exists {
        fmt.Printf("查無傳入的使用者 ID (%v).\n使用者列表:\n", userID)
        for key, value := range getUsers() {
            fmt.Println("使用者 ID:", key, "  名字:", value)
        }
        os.Exit(1)
    }
    fmt.Println("查得名字:", name)
}

```

- 測試結果：

```
PS D:\git\go lan> go run "d:\git\go lan\ch4\4-5-2.go" 123
查無傳入的使用者 ID (123).
使用者列表:
使用者 ID: 631    名字: Jake
使用者 ID: 073    名字: Tracy
使用者 ID: 305    名字: Sue
使用者 ID: 204    名字: Bob
exit status 1
PS D:\git\go lan> go run "d:\git\go lan\ch4\4-5-2.go" 305
查得名字: Sue
```

- 你看到的map輸出不見的和以上相同，正是因為go語言會在使用for range時故意打亂元素

4-5-3 從map刪除元素

- 若你要從map中移除元素，做法會跟陣列或切片都不一樣
- 陣列的元素是無法移除的，因為長度已經固定，頂多將該元素設為零值
- 至於切片，可以用零值清空該元素，或是用append()組合新切片範圍，並在過程中去掉某元素

- 但在map中，你可以將元素變成零值，但元素仍然存在，所以你的程式檢查時會得到“鍵仍存在”的錯誤結論

- Map也無法像切片那樣用擷取範圍來移除元素，所以若要移除map元素必須用到內建函式delete()：

  delete(map名稱, 索引鍵)

- delete()沒有回傳值，所以就算移除不存在的鍵，也不會有任何問題

練習：從map刪除一個元素

- 這個練習要定義一個map，然後依據使用者輸入的鍵來刪除一個元素，最後把刪減過的map顯示出來

```go
package main

import (
    "fmt"
    "os"
)

var users = map[string]string{
    "305": "Sue",
    "204": "Bob",
    "631": "Jake",
    "073": "Tracy",
}

func deleteUser(id string) {
    delete(users, id)
}

```

```
18
19  func main() {
20      if len(os.Args) < 2 {
21          fmt.Println("未傳入使用者 ID")
22          os.Exit(1)
23      }
24      userID := os.Args[1]
25      deleteUser(userID)
26      fmt.Println("使用者列表:", users)
27  }
```

- 測試結果 :

```
PS D:\git\go lan> go run "d:\git\go lan\ch4\4-5-3.go" 305
使用者列表: map[073:Tracy 204:Bob 631:Jake]
```

- 注意 : 只有map可以用delete()函數; 不能對陣列或切片 使用

4-6 簡易自訂型別(custom types)

- 你可以用go語言的核心型別當成起點，建立你的自訂型別：

  type 自訂型別名稱 核心型別

- 例如想以字串為基礎建立一個叫做id的型別，可以寫成如下:

  type id string

- 自訂型別的行為就跟其他核心型別一樣，包括擁有零值 / 能跟同型別的資料比對等等

- 但自訂型別只不過是拿另一個型別換個名稱，還有甚麼用呢？

  - 重點在於，你可以對自訂型別加上自訂的行為(函式或”方法”)，但你不能對核心型別這麼做

  - 下一節談到結構型別時，再來看是甚麼 “方法”

練習：定義一個簡易的自訂型別

- 在這個練習中，我們要定義一個簡單、以字串為基礎的自訂型別，拿它來建立一些資料，並彼此做對比，甚至和字串型別做對比

```go
package main

import "fmt"

type id string //自訂型別

func getIDs() (id, id, id) {
    //用自訂型別建立變數
    var id1 id
    var id2 id = "1234-5678"
    var id3 id
    id3 = "1234-5678"
    return id1, id2, id3
}

func main() {
    id1, id2, id3 := getIDs()
    //自訂型別互相比對
    fmt.Println("id1 == id2         :", id1 == id2)
    fmt.Println("id2 == id3         :", id2 == id3)
    //轉成字串後跟字串比對
    fmt.Println("id2 == \"1234-5678\":", string(id2) == "1234-5678")
}
```

- 顯示結果：

```
id1 == id2          : false
id2 == id3          : true
id2 == "1234-5678": true
```

- 在現實中，會常看到用自訂型別來包裝資料

- 當你讓型別的名稱反映你需要處理的資料時，程式碼會變得更好理解和維護

4-7 結構(struct)

- 陣列、切片 、map 集合都十分適合用來把型別及用途相同的資料放在一起，但 GO 語言還有另一種收集資料的方式，用途也不相同。

- 當簡單的字串、數值或布林值都無法正確地捕捉你手中資料的本質時，你就需要用上以下這種更複雜的資料型別。

- 舉個例，以我們先前建立的使用者 map 來說，一筆使用者資料會由不重複的識別碼 ID 和使用者的名字構成。

- 然而，你需要儲存的個人資料可能五花八門，像是姓名、稱謂、生日、身高、體重、甚至工作地點都有可能，而map 這種形式根本不夠應付所有類型的資料。

- 當然，你可以用多個map來維護這些資料，每個都使用相同的索引鍵，以便對到不同型別的值，但這樣做起來既麻煩又難以維護

- 因此理想的做法是把這些型別各異的資料收在一個單獨的資料結構之中，你可以任意調整和控制它。這就是 Go 語言的結構(struct) 型別：

    - 它是一種自訂型別，你可以命名它、並指定其中的欄位 (field）名稱以及其型別

4-7-1 結構的定義

- 語法：

```
type 結構型別名稱 struct {
        欄位1 型別
        欄位2 型別
        …
        欄位N 型別
}
```

- 欄位就是隸屬於結構的變數，每個欄位的名稱必須獨一無二，但其型別毫無限制，可以是集合/指標，甚至另一個結構

- 讀取欄位和對它們賦值的語法如下：

  結構變數名稱.欄位名稱

- 一旦定義好結構型別，就可以用它來建立變數

- 用結構建立變數的方式很多，以下練習就來看看有哪些做法

練習：定義結構型別和建立變數

- 這個練習要定義一個使用者結構
- 會在該結構中定義若干不同型別不同的欄位，然後用幾種不同的方式對結構賦值

```go
package main

import "fmt"

type user struct { //自訂結構型別
    name    string //定義欄位名稱與型別
    age     int
    balance float64
    member  bool
}
```

```go

func getUsers() []user {
    u1 := user{
        name:    "Tracy", //搭配欄位名稱賦值法 (不必照順序)
        age:     51,
        balance: 98.43,
        member:  true,
    }
    u2 := user{
        age:  19,
        name: "Nick",
        //其餘沒有賦值的欄位會是零值
    }
    u3 := user{
        "Bob", //不使用欄位名稱的賦值法(必須照順序 , 資料不能有缺)
        25,
        0,
        false,
    }
    var u4 user
    u4.name = "Sue" //透過 "結構.欄位 = 值" 賦值
    u4.age = 31
    u4.member = true
    u4.balance = 17.09

    return []user{u1, u2, u3, u4}
}
```

```
38
39  func main() {
40      users := getUsers()
41      for i := 0; i < len(users); i++ {
42          fmt.Printf("%v: %#v\n", i, users[i])
43      }
44  }
45
```

- 執行結果:

```
0: main.user{name:"Tracy", age:51, balance:98.43, member:true}
1: main.user{name:"Nick", age:19, balance:0, member:false}
2: main.user{name:"Bob", age:25, balance:0, member:false}
3: main.user{name:"Sue", age:31, balance:17.09, member:true}
```

- 注意：對欄位賦值時(u1~u3)，欄位間不一定要換行，但一定要用逗號隔開,如果有換行，每行結尾一定要是逗號

匿名結構

- 結構型別一般會在套件層級宣告，因為可能會有多重函式需要用它建立變數，但你也可以把結構型別定義在函式層級內，只是這麼一來就只能在該函式內使用

- 這種於函式內定義的結構型別（注意不是結構變數）稱為匿名結構(anonymous strut)。

- 這種結構沒有名稱，並可在宣告時一併對欄位賦值。這樣的結構型別就只能有一個變數，沒辦法再用來建立其他結構。

- 匿名結構的寫法如下：

```
結構變數名稱 := struct {
    欄位1 型別
    欄位2 型別
    …
    欄位N 型別
}{
    值1,
    值2,
    …
    值N,
}
```

4-7-2 結構的相互比較

- 若結構中的每個欄位都相同，且使用可以比較的型別，那麼該結構型別的變數就可以相比較

- 比如，若你的結構型別是以字串和整數構成，它就能和同型別的結構比較，並在欄位值完全相同時回傳true

- 但如使用切片最為欄位，就不能相比了

- Go語言屬於強行別語言，只有相同型別才能做比較，但在結構這方面較為彈性

- 如果在函式中定義的匿名結構型別和其他結構型別擁有相同欄位，那也可以做比較

練習：比較結構

- 這次的練習，要定義幾個可比較的結構型別(包含匿名結構型別)和拿來建立變數，而這些結構型別都有相同的名稱與型別的欄位

- 最後會比較這些結構，並顯示結果

```go
package main

import "fmt"

type point struct { //具名結構
    x int
    y int
}

func compare() (bool, bool) {
    point1 := struct { //匿名結構(有給初始值)
        x int
        y int
    }{
        10,
        10,
    }
    point2 := struct { //匿名結構(無初始值)
        x int
        y int
    }{}
    point2.x = 10 //設定欄位值
    point2.y = 5
    point3 := point{10, 10}
    return point1 == point2, point1 == point3
}

```

```
28    func main() {
29        a, b := compare()
30        fmt.Println("point1 == point2:", a)
31        fmt.Println("point1 == point3:", b)
32    }
```

- 顯示結果:

```
point1 == point2: false
point1 == point3: true
```

4-7-3 內嵌結構

- 雖然說Go 語言的結構型不支援繼承，但其設計者倒是提供了個有趣的替代方式：在結構型別中內嵌 (embedding）其他結構。

- 內嵌和拿其他結構型別當成欄位是不一樣的；當你把結構 B 內嵌到結構A時，B 的欄位會被提升 (promoted) 成 A 結構自身的欄位，就好像是直接在A結構型別內定義的一樣。

- 要內嵌一個結構，寫法就像平時定義欄位一樣，只是不必指定欄位名稱只需在加上來源的結構型別名稱即可：

```
type 型別名稱 struct {
        欄位 型別 //正常的欄位定義
        結構型別 //內嵌結構型別 (不寫欄位名稱)
}
```

- 可以內嵌任何型別到結構內，只是內嵌核心型別不是常見的做法，也沒什麼用處

- 可以用下面的語法存取內嵌型別：

  結構變數.內嵌(結構)型別

- 於是, 你就可以用同樣的方式存取內嵌結構的欄位，這麼做的效果跟直接存取提升的欄位是一樣的：

  結構變數.內嵌結構型別.欄位名稱
  結構變數.提升的欄位名稱

- 但若要直接存取提升的欄位名稱，這個欄位的名字必須是獨一無二的，不能跟目標結構的欄位撞名

- 若發生了同名欄位的現象，go語言還是會允許內嵌，但是子結構的欄位不會被提升，這時唯一存取該欄位的方式就是透過內嵌結構型別名稱了

- 在初始化欄位時，不能直接對內嵌的欄位賦值，必須透過其型別名稱 (下面會看到作法)

- 練習 ：嵌入結構賦予初始值

  這次的練習要定義若干結構和自訂型別，並把這些型別嵌入到另一個結構中

```go
package main

import (
    "fmt"
)

type name string //自訂型別

type location struct { //結構型別
    x int
    y int
}

type size struct { //結構型別
    width  int
    height int
}

type dot struct {
    name     //內嵌自訂型別
    location //內嵌結構
    size     //內嵌結構
}

```

```go
func getDots() []dot {
    var dot1 dot

    dot2 := dot{}
    dot2.name = "A" //存取內嵌的自訂型別
    dot2.x = 5      //內嵌結構型別的欄位被提升了
    dot2.y = 6
    dot2.width = 10
    dot2.height = 20

    dot3 := dot{
        name: "B",
        location: location{ //透過嵌入型別賦予初始值
            x: 13,
            y: 27,
        },
        size: size{
            width:  5,
            height: 7,
        },
    }

```

```
47      dot4 := dot{}
48      dot4.name = "C"
49      dot4.location.x = 101
50      dot4.location.y = 209
51      dot4.size.width = 87
52      dot4.size.height = 43
53
54      return []dot{dot1, dot2, dot3, dot4}
55  }
56
57  func main() {
58      dots := getDots()
59      for i := 0; i < len(dots); i++ {
60          fmt.Printf("dot%v: %#v\n", i+1, dots[i])
61      }
62
63  }
```

- 顯示結果：

```
dot1: main.dot{name:"", location:main.location{x:0, y:0}, size:main.size{width:0, height:0}}
dot2: main.dot{name:"A", location:main.location{x:5, y:6}, size:main.size{width:10, height:20}}
dot3: main.dot{name:"B", location:main.location{x:13, y:27}, size:main.size{width:5, height:7}}
dot4: main.dot{name:"C", location:main.location{x:101, y:209}, size:main.size{width:87, height:43}}
```

- 在現實中，嵌入的用法並不常見，因為會額外帶來複雜度跟問題，因此通常會直接將其他結構當成目標結構的具名欄位

4-7-4 替自訂型別加上方法 (method)

值接收器 VS 指標接收器

- 結構可以包含不同型別的欄位，用起來很方便，但你能不能讓它也具備一些功能，像是擁有自己的函式可供呼叫？

- Go 語言雖不是物件導向語言，沒有所謂的類別 (class)，但我們確實可以在定義函式時，將它們指向特定型別的變數 (通常是個結構變數)

- 正如前面所提，這些綁(bind) 到結構『物件』的函式，通常會被稱為結構的方法(method)。（下一章我們會更深入介紹函式的定義方式，不過各位已經在前面幾章看過不少函式的例子了。）

- 為了定義型別方法，有值接收器 (value receiver) 和『指標接收器』指標接收器 (point receiver) 兩種寫法。

- 首先來看『值接收器』：

```
func (接收器變數  型別)  函式名稱()  {
    程式碼
}
```

- 而指標接收器的語法和值接收器的語法很像 . 只差型別前面要加上* :

```
func(接收器變數  *型別)  函式名稱() {
    程式碼
}
```

- 等一下我們會來看這兩者有何差別。但函式只要有接收器，你就就能用『變數.函式()』的語法呼叫它們。

- 接收器會讓結構變數以參數的形式傳給函式，使我們能從函式內部存取特定結構的欄位和其他方法。

- 附帶一提，不同型別（自訂型別或結構型別）可以定義名稱一模一樣、但參數與傳回值完全不同的方法

- 要注意的是接收器變數有以下限制：

  1. 不能是核心型別 / 介面(interface) / 指標
  2. 方法必須與該型別定義在同一套建中

- 簡單來說，只能對同一套建定義的自訂型別/結構綁上方法，還有若某結構是定義在外部套建，就不允許替它新增方法

建立並呼叫型別方法

- 這裡修改前面的練習，替自訂型別和結構加上他們的方法

```go
package main

import (
    "fmt"
)

type name string //name型別

type point struct { //point結構型別
    x int
    y int
}

func (n name) printName() { //name型別方法(值接收器)
    fmt.Println("name", n)
}

func (p *point) setPoint(x, y int) { //point結構方法(指標接收器)
    p.x = x //存取結構欄位
    p.y = y
}

func (p point) getPoint() string { //point結構方法
    return fmt.Sprintf("(%v, %v)", p.x, p.y)
}

```

```go
func main() {
    var n name = "Golang"
    n.printName() //呼叫 n 的方法 printName()

    a, b := point{}, point{}
    a.setPoint(10, 10)                    //呼叫 a 的setPoint方法
    b.setPoint(10, 5)                     //呼叫 b 的setPoint方法
    fmt.Println("point1:", a.getPoint()) //呼叫 a 的getPoint() 方法
    fmt.Println("point2:", b.getPoint()) //呼叫 b 的getPoint() 方法
}
```

- 執行結果:

```
name Golang
point1: (10, 10)
point2: (10, 5)
```

- 我們在這裡替name自訂型別和point結構型別定義了各自的方法，且同一個方法可以套用在不同變數上

- 那麼，以point結構為例，(p point)和(p *point)究竟有何不同?

1. (p point)是值接收器，它會複製變數到p並傳給函式，函式可以讀到原本的變數欄位，但若修改p，其變動不會反映到原變數
2. (p *point)是指標接收器，它會把p轉為指標&p傳給函式，函式對p的任何修改都會反映到原變數

- 可以試試看將setPoint()方法從指標改成值接收器，然後觀察執行結果：

```
name Golang
point1: (0, 0)
point2: (0, 0)
```

- 改成值接收器後，setPoint()只修改了它接收到的複本，因此main()內的a和b結構欄位仍維持在零值

值接收器和指標的自動轉換

- 上面提到go語言在將值傳入指標接收器時，會自行將它轉成指標傳入，這其實是個雙向機制，這裡我們來觀察是如何運作的

- 我們可以修改以上程式，把其中一個point{}結構變數換成指標:

```go
package main

import (
    "fmt"
)

type point struct {
    x int
    y int
}

func (p *point) setPoint(x, y int) {
    p.x = x
    p.y = y
}

func (p point) getPoint() string {
    return fmt.Sprintf("(%v, %v)", p.x, p.y)
}

func main() {
    a := point{}  //a是結構
    b := &point{} //b是指向結構變數的指標
    a.setPoint(10, 10)
    b.setPoint(10, 5)
    fmt.Println("point1:", a.getPoint())
    fmt.Println("point2:", b.getPoint())
}
```

- 執行結果:

```
point1: (10, 10)
point2: (10, 5)
```

- 可以看到即便b變成指標 , 它在使用setPoint() (指標接收器)和getPoint() (值接收器) 時仍表現正常

- 事實上 , 當呼叫a.setPoint()時 , a也會被自動轉成指標 , 這是因為go語言會在背後做以下處理:
  1. 呼叫a.setPoint()時 , a 是一個值, setPoint()卻用了指標接收器 , 因此go語言會自動將它轉成 (&a).setPoint()

  2. 呼叫 b.getPoint()時 , b是一個指標 , getPoint()卻使用了值接收器 , go語言會自行轉成(*b).getPoint()

4-8 介面與型別檢查

4-8-1 型別轉換

- 在開發go語言時，兩個不同型別的值是不能彼此互動的
- 這時可以進行型別轉換 (type conversion)： 型別(值)
- 字串的轉換不會有損失，但數值型別的轉換可能發生溢位錯誤
- Go 語言對於未定義型別會嘗試推斷其型別:
  - b := ture //true會被推斷為bool型別
  - v := 1 //1會被推斷位int型別
  - v := 1 * unit64(5) //1因為和unit64()型別運算，因此會被推斷成unit64
  - s := “test” //”test”會被推斷為 string 型別

數值型別轉換

- 這次練習要做幾個數值型別轉換，並刻意造成溢位

```go
package main

import (
    "fmt"
    "math"
)

func convert() string {
    var i8 int8 = math.MaxInt8
    i := 128
    f64 := 3.14
    m := fmt.Sprintf("int8    = %v  > in64    = %v\n", i8, int64(i8))
    m += fmt.Sprintf("int     = %v  > in8     = %v\n", i, int8(i))
    m += fmt.Sprintf("int8    = %v  > float32 = %v\n", i8, float64(i8))
    m += fmt.Sprintf("float64 = %v > int     = %v\n", f64, int(f64))
    return m
}

func main() {
    fmt.Print(convert())
}
```

- 顯示結果：

```
int8    = 127  > in64    = 127
int     = 128  > in8     = -128
int8    = 127  > float32 = 127
float64 = 3.14 > int     = 3
```

4-8-2 型別斷言與interface{}空介面

- 在第7章會再詳談甚麼是介面，現在你知要知道：介面型別是個規範，會列出若干函式的定義，而任何型別只要具備相同的函式，就會被視為屬於該介面型別

- Interface{}空介面是一個沒有指定任何函式的空型別，那就代表所有型別都符合其規範，可以接收任何型別的參數

- 不過一旦參數以空介面傳入函式後，go 語言會強迫你遵循該介面獨有的協議，你只能存取介面裡列出的方法(interface{}沒有任何方法)，這也是位甚麼空介面型別依然能維持型別安全

- 若想挖出被intrface()掩蓋的資料型別功能，就得使用下面介紹的型別斷言(type assertion)

使用型別斷言

- 型別斷言會嘗試把值轉換成指定的型別並傳回，其語法如下:
  - 值 := 變數名稱.(型別)
  - 你還可以接收第二個選擇性回傳值, 代表轉換成功與否”
  - 值, ok := 變數名稱.(型別)

- 若不接收第二個回傳值，而型別斷言轉換失敗的話，go語言會引發 panic

- 若你把某個值傳入interface{}形別參數，go語言不會從中移除任何內容，但你也因此無法存取原始的值

- 唯一將該值解鎖的方法就是用型別斷言告訴go語言你想要取得這個值

練習：用型別斷言檢查型別

- 以下練習要進行若干型別斷言，並確認做型別斷言時都有附上型別安全檢查

- 我們要寫一個函式，它能把特定型別的值加倍，並對不符的型別丟出錯誤

```go
package main

import (
    "errors"
    "fmt"
)

//接收一個interface{}型別的函數
func doubler(v interface{}) (string, error) {
    if i, ok := v.(int); ok { //嘗試轉成int
        return fmt.Sprint(i * 2), nil
    }
    if s, ok := v.(string); ok { //嘗試轉成string
        return s + s, nil
    }
    //型別不符前面的檢查 , 傳回錯誤
    return "", errors.New("傳入了未支援的值")
}

func main() {
    res, _ := doubler(5)
    fmt.Println("5   :", res)
    res, _ = doubler("yum")
    fmt.Println("yum :", res)
    _, err := doubler(true)
    fmt.Println("true:", err)
}
```

- 顯示結果：

```
5    : 10
yum  : yumyum
true: 傳入了未支援的值
```

- 只要結合interface{}和型別斷言，就可以幫你繞過go語言嚴格的型別控制，讓你建立可接受任何型別作為參數的函式，缺點是你會失去go語言在編譯階段提供的型別安全保護，確保型別安全的責任轉移到你身上，只要沒有處理好就可能碰到麻煩的執行階段錯誤

4-8-3 型別switch

用switch搭配型別斷言

- 若要進一步改寫前面練習中的doubler()函式，使它能夠涵蓋所有整數型別，你勢必要寫出一堆的if判斷式

- 但go語言有個方法可以簡化型別斷言，也就是型別switch：

```
switch 值 := 變數.型別 {
case 型別:
        敘述
case 型別, 型別:
        敘述
default:
        敘述
```

- 型別switch只會執行符合型別的case所對應的程式敘述，並把值設定為那個型別

- 你可以在case內比對一種以上的型別，但這樣go語言就無法自動調整值型別, 還是必須在case底下加入額外的型別斷言

- 型別switch跟第二章的switch不一樣，型別switch後面必須寫成 值 := 變數.型別，無第二種寫法，也不能使用fallthrough 敘述

練習：使用型別switch

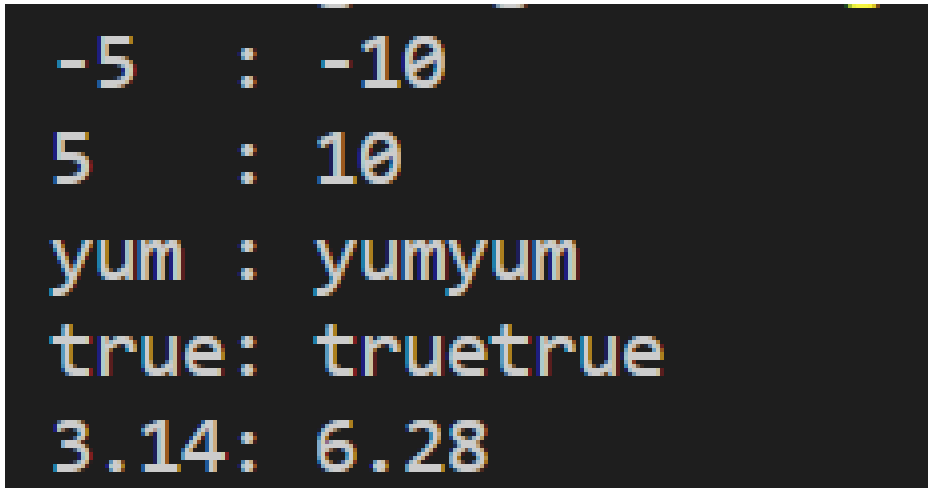
- 在這個練習中，我們要利用型別switch改寫前面的doubler()函式，將它的判斷範圍擴大

```go
package main

import (
    "errors"
    "fmt"
)

func doubler(v interface{}) (string, error) {
    switch t := v.(type) {
    case string: //型別為字串時
        return t + t, nil
    case bool: //型別為布林值時
        if t {
            return "truetrue", nil
        }
        return "falsefalse", nil
    case float32, float64: //型別為浮點數時
        if f, ok := t.(float64); ok {
            return fmt.Sprint(f * 2), nil
        }
        return fmt.Sprint(t.(float32) * 2), nil
```

```
22      case int: //其餘整數型別檢查
23          return fmt.Sprint(t * 2), nil
24      case int8:
25          return fmt.Sprint(t * 2), nil
26      case int16:
27          return fmt.Sprint(t * 2), nil
28      case int32:
29          return fmt.Sprint(t * 2), nil
30      case int64:
31          return fmt.Sprint(t * 2), nil
32      case uint:
33          return fmt.Sprint(t * 2), nil
34      case uint8:
35          return fmt.Sprint(t * 2), nil
36      case uint16:
37          return fmt.Sprint(t * 2), nil
38      case uint32:
39          return fmt.Sprint(t * 2), nil
40      case uint64:
41          return fmt.Sprint(t * 2), nil
42      default:
43          return "", errors.New("傳入了未支援的值")
44      }
45  }
46
```

```
47  func main() {
48      res, _ := doubler(-5)
49      fmt.Println("-5  :", res)
50      res, _ = doubler(5)
51      fmt.Println("5   :", res)
52      res, _ = doubler("yum")
53      fmt.Println("yum :", res)
54      res, _ = doubler(true)
55      fmt.Println("true:", res)
56      res, _ = doubler(float32(3.14))
57      fmt.Println("3.14:", res)
58  }
```

- 執行結果：



本章結束