

組合型別：陣列、切片、映射、結構



# 組合型別

組合(複合)型別，顧名思義是建構在基礎型別之上，把前面的型別組合在一起。

## 陣列(Array)

- 同一個陣列中只能存放**同一個型態**的值，比如整數陣列、浮點數陣列、字串陣列。
- 陣列的**長度在宣告之後無法改變**。
- 值也可以一開始就給，可部分宣告值，沒值的預設為**零值**。
- **這種[]內沒任何東西的宣告是屬於Slice，而不是省略長度的Array：**

```
e := []int{1, 2}
```

```
var a [5]int
a[0] = 10
a[1] = 100
a[2] = 1000
fmt.Println(a)
```

```
b := [5]int{1, 2, 3} //單行宣告
fmt.Println(b)

c := [5]int{
    10,
    20,
    30,
    55, //使用多行宣告的話，最後一個元素要逗號
}
fmt.Println(c[1:3])

d := [...]int{4, 6, 8} //用...省略符號，讓go判斷長度
fmt.Println(d)
/* result:
[10 100 1000 0 0]
[1 2 3 0 0]
[20 30]
[4 6 8]
*/
```

# 組合型別

## 陣列(Array)

- 宣告陣列的時候，要注意一下賦值的是int還是string：

```
package main

import "fmt"

func main() {
    var arr [10]int
    var i,j int

    var langs [4]string
    langs[0] = "Go"
    fmt.Printf("langs[0] = %s\n", langs[0] )

    for i = 0; i < 10; i++ {
        arr[i] = i
    }

    /* 輸出 */
    for j = 0; j < 10; j++ {
        fmt.Printf("arr[%d] = %d\n", j, arr[j] )
    }
}
```

# 組合型別

## 陣列(Array)

- 取得相關數值
- 要求讀寫超過陣列長度的值，程式會錯：

```
package main
import "fmt"
func main(){
    jelly := [...]string{"小櫻", "知世", "桃矢", "雪兔", "小櫻他爸"}
    fmt.Println(jelly[5])
}
```

執行結果：

```
# command-line-arguments
.\lesson07.go:5:22: invalid array index 5 (out of bounds for 5-element array)
```

- 利用 `len()` 取得陣列長度：

```
package main
import "fmt"
func main(){
    jelly := [...]string{"小櫻", "知世", "桃矢", "雪兔", "小櫻他爸"}
    fmt.Printf("陣列長度：%d", len(jelly))
}
```

執行結果：

陣列長度：5

- 因為Golang不是物件導向語言所以是用`len()`這種函式的型式來取得陣列長度(和python一樣)，這個取陣列長度在每個語言中都有各自的作法，比如：C 完全不能取得長度(除非自己實作)、java 是用`.length()`、JavaScript和dart是用`.length`、PHP 是用`count()`，各家的寫法都不一樣。

# 組合型別

## 陣列(Array)

- 走訪陣列
  - 既然都知道陣列的長度了，那能否透過 for 迴圈把陣列的內容 print 出來呢？

```
package main
import "fmt"
func main(){
    jelly := [...]string{"小櫻", "知世", "桃矢", "雪兔", "小櫻他爸"}
    for i:=0; i<len(jelly); i=i+1{
        fmt.Println(jelly[i])
    }
}
```

執行結果：

小櫻  
知世  
桃矢  
雪兔  
小櫻他爸

# 組合型別

## 陣列(Array)

- 走訪陣列
  - **Golang** 提供另一個方法來走訪陣列，這個方法可以用來走訪所有有「迭代器」的資料結構。
  - **for** 使用迭代器，為一種走訪容器的方法，藉由此方法，程式設計師不需要知道陣列內部的結構，就可以走訪容器內部的元素：

```
package main

import (
    "fmt"
)

func main() {
    var count [3]int64
    count[0] = 1
    count[1] = 2
    count[2] = 3

    for index, value := range count {
        fmt.Println(fmt.Sprintf("index%d: %d", index, value))
    }
}
```

```
index0: 1
index1: 2
index2: 3
```

藉由迭代器的方法，印出陣列中每個 **index** 所代表的元素為多少。

# 組合型別

## 陣列(Array)

- 走訪陣列
  - 因為 Go 語言一旦宣告變數後，就必須得使用，但當今天要使用迭代器印出陣列裡面的值時，又不需要 `index`，那可以用**底線**來取代變數：

```
package main

import (
    "fmt"
)

func main() {
    var count [3]int64
    count[0] = 1
    count[1] = 2
    count[2] = 3

    for _, value := range count {
        fmt.Println(value)
    }
}
```

藉由此方法，即可不必為 `index` 宣告一個變數。  
相反的，如果只想取 `index`，就不需宣告 `value`：

```
for index := range count {
    fmt.Println(index)
}
```

```
0
1
2
```

# 組合型別

## 陣列(Array)

- 走訪陣列
- 另一個要注意的地方是，使用 `range` 來走訪陣列時，直接更改 `value` 值並不會對陣列內的值造成影響，但是仍可以透過 `index` 去更改：

```
package main
import "fmt"
func main(){
    jelly := [...]string{"小櫻", "知世", "桃矢", "雪兔", "小櫻他爸"}
    for k, v := range jelly{
        jelly[k] = v + v
    }
    for _, v := range jelly{
        fmt.Println(v)
    }
}
```

執行結果：

小櫻小櫻  
知世知世  
桃矢桃矢  
雪兔雪兔  
小櫻他爸小櫻他爸



# 組合型別

## 陣列(Array)

- 宣告陣列長度時不能使用變數來宣告
  - 陣列長度在編譯完後就不能再更動，陣列長度就必需是確定的，變數值是執行時才被確定的，所以不能使用變數來設定陣列的長度：

```
package main
import "fmt"
func main(){
    length := 5
    jelly := [length]string{"小櫻", "知世", "桃矢", "雪兔", "小櫻他爸"}
    for _, v := range jelly{
        fmt.Printf("%s ", v)
    }
}
```

執行結果：

# command-line-arguments

.\\lesson07.go:5:14: non-constant array bound length

- 如果仍希望透過類似的手法來實作可以選擇宣告一個常數(constant)：

```
package main
import "fmt"
func main(){
    const length int = 5 // 利用 const 宣告一個常數
    jelly := [length]string{"小櫻", "知世", "桃矢", "雪兔", "小櫻他爸"}
    for _, v := range jelly{
        fmt.Printf("%s ", v)
    }
}
```

執行結果：

小櫻 知世 桃矢 雪兔 小櫻他爸

宣告成常數後就不可以再更動其值，因為該常數能在編譯前確定，所以可以當作陣列的長度

# 組合型別

## 切片(Slice)

- 陣列(array)使用上非常方便，但每次都得**預先宣告(長度需要為固定)**一個長度、撥一段記憶體空間給陣列使用。
- Google說這邊有一個物件叫 **Slice**，只能裝載相同性質的元素，但是**長度是可以做彈性改變的**，使得在使用上，更加方便，講明了就是List。
- **宣告時中括號[]裡為空**，把**Slice**拆開來看，這玩意包含了三樣東西：

1. 指標 ptr：透過指標，**可以與別人共用同一個地方**
2. 長度 len：**現在**的長度
3. 容量 cap：**最大能容納**的長度



# 組合型別

## 切片(Slice)

- 宣告Slice的方法之一：

```
Variable := make([]Type, Len, Cap)
```

```
b := make([]int, 5, 10)
```

宣告變數b為len:5、cap:10的整數切片。

- 以下是幾種宣告Slice的方法：

```
a := make([]int, 10) //設定 len:10，現在長度10了，容量雖然沒給，但最大容納長度當然不可能小於10吧，所以就是10了
fmt.Println(a, len(a), cap(a), len(a) == 0, a == nil)

b := make([]int, 5, 10) //設定 len:5、cap:10
fmt.Println(b, len(b), cap(b), len(b) == 0, b == nil)

var c = []int{} //初始化slice
fmt.Println(c, len(c), cap(c), len(c) == 0, c == nil)

var d []int //尚未實體化，此時等於nil
fmt.Println(d, len(d), cap(d), len(d) == 0, d == nil)

e := []string{"youtube.com", "facebook.com"} //直接賦值
fmt.Println(e, len(e), cap(e), len(e) == 0, e == nil)

/* result:
[0 0 0 0 0 0 0 0 0 0] 10 10 false false
[0 0 0 0 0] 5 10 false false
[] 0 0 true false
[] 0 0 true true
[youtube.com facebook.com]
*/
```

初始化沒給值的話都是零值

# 組合型別

## 切片(Slice)

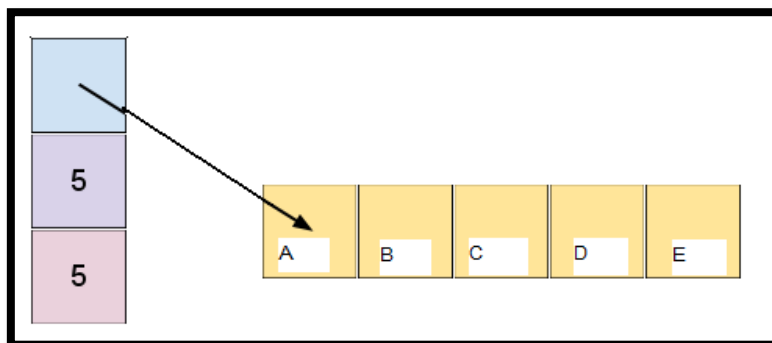
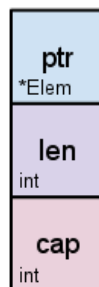
- 如果以 code 範例示意，初始化圖會像下面這樣：

```
package main

import (
    "fmt"
)

func main() {
    var a []string
    a = make([]string, 5, 5)
    a[0] = "A"
    a[1] = "B"
    a[2] = "C"
    a[3] = "D"
    a[4] = "E"
    fmt.Printf("array %#v\n", a)
}

// slice 需要使用 make 做初始化才能開始使用
```



# 組合型別

## 切片(Slice)

- 切片截取：切片能改變長度的特性，也多了許多方便性，可以透過設定的上限以及下限來截取切片：

```
package main
import "fmt"
func main() {
    count := []int64{1,2,3,4,5}
    subCount := count[0:2]

    fmt.Println(subCount) // [1 2]
}
```

先宣告名為 `count` 的切片，而今天有一個需求是要印出 `index 0` 到 `2` 的值，所以宣告另一個切片，設置下限為 `0`，上限為 `2`，然後對 `count` 做截取，如此一來，即可達到要的需求。

# 組合型別

## 切片(Slice)

- **make**：切片也可以在**執行時期動態產生**，這時候會使用 **make** 做為關鍵字。

在下方舉一個例子，動態產生一個長度為 3 的切片：

```
package main

import "fmt"

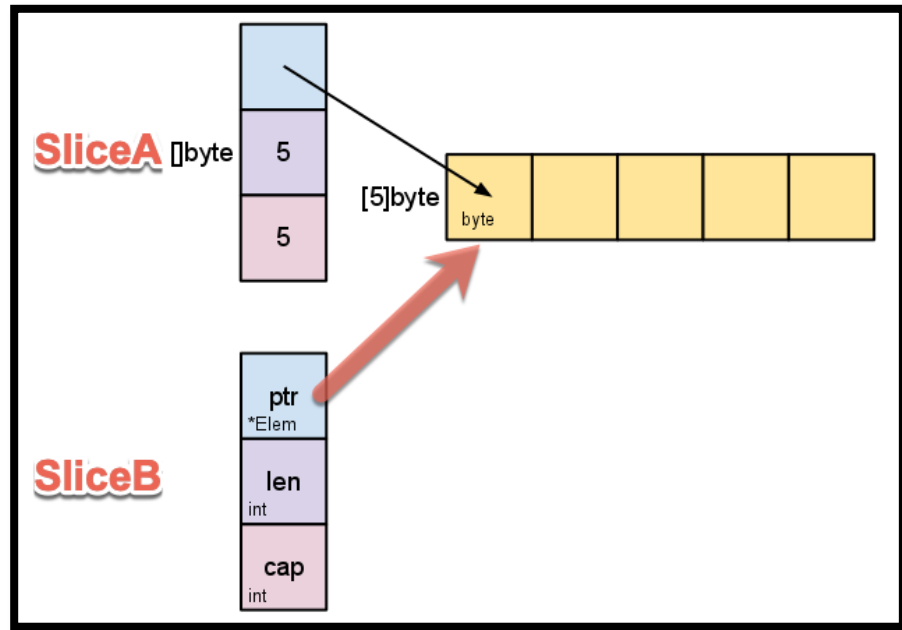
func main() {
    slice := make([]int, 3)

    for i := 0; i < len(slice); i++ {
        y := i + 1
        slice[i] = y * 2
    }
    fmt.Println(slice) // [2 4 6]
}
```

# 組合型別

## 切片(Slice)

- **Slice**並不是真正存值，而是透過**指標**指到更下面的地方，某個陣列，這也是為什麼說可以跟別人共用的原因。如果**SliceA**跟**SliceB**都是一樣的值，底層陣列只要一份資料就好，夠省空間吧！



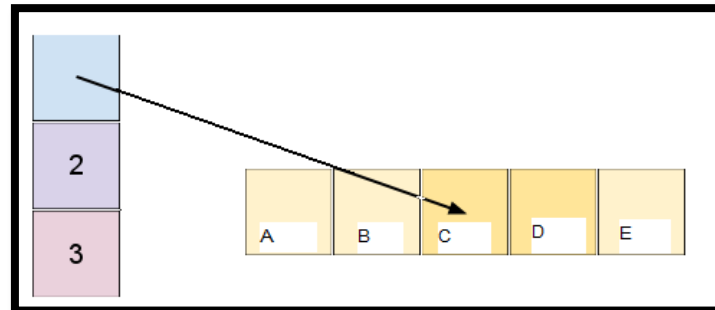
```
package main

import (
    "fmt"
)

func main() {
    var a []string
    a = make([]string, 5, 5)
    a[0] = "A"
    a[1] = "B"
    a[2] = "C"
    a[3] = "D"
    a[4] = "E"
    fmt.Printf("a array %#v\n", a)

    s := a[2:4]
    //s[0] == "C"
    //s[1] == "D"
    //len(s) == 2
    //cap(s) == 3
    s[0] = "Z"
    // a[2] == "Z" 這段是拿來證明 s是參照到 a
    fmt.Printf("s array %#v\n", s)
    fmt.Printf("a array %#v\n", a)
}
```

Slice 切片出來的東西，都是建立一個新的 Slice 指向原本的 Slice，所以原本的 abcde slice，被切成 2-4，就會變成只有 C 跟 D，但容量(cap)一開始規劃出來是 5， $5-2=3$ ，長度(len)則是  $4-2=2$ 。



# 組合型別

## 切片(Slice)

- 常常需要取用某個小段的 **Slice**，但 **Slice** 的原理都是給出參照，以上面為例，可能只需要 C, D 兩值，但是用 **Slice** 取出來卻會造成，其他 A, B, E 都一直存活著，比較好的作法是用 **copy**，要如何 **copy** 呢，來看個範例：(copy 是 built-in function)

```
package main

import (
    "fmt"
)

func main() {
    var a []string
    a = make([]string, 5, 5)
    a[0] = "A"
    a[1] = "B"
    a[2] = "C"
    a[3] = "D"
    a[4] = "E"
    fmt.Printf("a array %#v\n", a)

    s := make([]string, len(a))
    copy(s, a)
    s[2] = "Z"
    fmt.Printf("s array %#v\n", s)
    fmt.Printf("a array %#v\n", a)
}
```



# 組合型別

## 切片(Slice)

- 通常會用**append**這個方法來操作Slice：

```
x := []int{1, 2, 3}
x = append(x, 4, 5, 6)
fmt.Println(x)

/* result:
[1 2 3 4 5 6]
*/
```

- Slice** 有一個重點就是在於 **長度(len)**不可大於**(cap)**否則會噴錯。
- 當想對切片最後面增加一個元素時，則可以使用**append**函式：
- 容量**cap**不夠時、長度**len**超過容量**cap**的時候，再來個**append**就好，容量**cap**會自動擴充，但在足夠容量中沒發生災情，但是在**超小容量**中卻因為後來的 `_` 做了 `append`，導致值也被改掉了？這就是可以看到指標(**pointer**)的地方，原因在於底層的陣列被後來的`_`蓋掉了。

```
a := make([]int, 0, 10) //給足夠容量
b := append(a, 1, 2, 3)
_ = append(a, 99, 88, 77)
fmt.Println(b)

//-----

aa := make([]int, 0, 2) //給超小容量
bb := append(aa, 1, 2, 3)
_ = append(aa, 99, 88, 77)
fmt.Println(bb)

/* result:
[99 88 77]
[1 2 3]
*/
```

# 組合型別

## 切片(Slice)

- 也可以利用多維切片製作**矩陣(matrix)**，見下例：

```
package main

import "fmt"

func main() {
    matrix := [][]float64{
        []float64{1, 2, 3},
        []float64{4, 5, 6},
    }

    fmt.Println(matrix)
}
```

# 組合型別

## 映射(map)

- 如果有用過Python，那Map就是類似Dictionary的概念。
- map有很多種翻譯，名詞叫地圖，動詞有**映射**、**對應**、**對照**的意思，概念就是一個key對應一個value。假設要記錄每位同學的身高，那每位同學的姓名就是key，身高就是value，那資料就會記錄成這樣：

Key	Value
"Tony"	168
"Mary"	159
"Alen"	185

- 在資料結構上的呈現就會是這樣：`{"Tony": 168, "Mary": 159, "Alen": 185}`，要注意的是key會是獨一無二的，"Tony"這個key只會有一個。

# 組合型別

## 映射(map)

- 宣告：

```
var Variable = map[Type]Type{}  
  
var a = map[int]string{}
```

- 宣告一個空白的map，map後[]內的是key的型態，[]後的是value的型態

- 可以像這樣bool對應到任何string：

```
var Male = map[bool]string{  
    true: "公",  
    false: "母",  
}
```

- 或是設定string對應到int：

```
var Number = map[string]int{  
    "零": 0,  
    "壹": 1,  
    "貳": 2,  
}  
Number["參"] = 3
```

- string對應到string也可以：

```
var Size = map[string]string{  
    "big": "大",  
    "medium": "中",  
    "small": "小",  
}
```

# 組合型別

## 映射(map)

- 宣告：

- 或是可以用make，make算是宣告資料結構的好幫手：

```
var h = make(map[string]int)
```

- make的另一種寫法：

```
h := make(map[string]int)
```

要注意:=和=是不同的

- 巢狀式宣告：

```
package main

import (
    "fmt"
)

var (
    a map[string]map[string]string
)

func main() {
    a = make(map[string]map[string]string)

    a["s1"] = map[string]string{
        "a1": "apple",
    }

    fmt.Println(a)
}
```

# 組合型別

## 映射(map)

- 設定初始值：

```
h := map[string]int{"Tony": 168, "Mary": 159, "Alen", 185}
```

  - 其實就只是把值填入{}中而已，格式是{key:value,...}
- 存取操作：

```
h["Tony"] // 168
```

  - 直接將key放上[]內就會取得value了：
- 放新的map進去：

```
h["George"] = 163
```

  - 其實就跟指定操作一樣，不過會是新的key
  - 要注意的是，key獨一無二的特性，如果今天又操作了h["Tony"]，那只會對"Tony"進行操作，並不會產生新的map

# 組合型別

## 映射(map)

- 長度：`len(h)`
  - `len()`這個function其實在各種有長度的地方都可以使用，如果Map裡的資料組數太多可以用這個看到底多少組了
- 刪除key value：`delete(h, "Tony")`
  - 直接用`delete()`就行，也算是滿萬用的function

# 組合型別

## 映射(map)

- 確認key是否存在：

```
if val, exists := h["Mary"]; exists {  
    do something  
}
```

- 第一個會回傳「對應的值」，第二個會回傳「布林值」用來表示該鍵是否存在。
- 可以利用兩個回傳值去存取，這樣會找h中是否有"Mary"這個key，如果有exists=true，沒有則exists = false

```
package main  
import "fmt"  
func main(){  
    // 更快地建立一個由 string 映射到 int 的 map  
    tall := map[string]int{  
        "小櫻" : 153,  
        "知世" : 155,  
        "小狼" : 156,  
    }  
    val, ok := tall["小櫻"]  
    fmt.Println(val, ok)  
    val, ok = tall["小可"]  
    fmt.Println(val, ok)  
}
```

執行結果：

153 true

0 false

如果該鍵並不存在，則在讀取該值時其實並不會出錯，而是會以預設值充當對應值回傳，這與部份語言的特性不太一樣



# 組合型別

## 映射(map)

- 確認key是否存在：
  - 如果只是想確認該值是否存在而不想知道該值，則可以在回傳對應值的部分設為\_，以免程式因為沒使用宣告出來的變數而爆錯：

```
package main
import "fmt"
func main(){
    // 更快地建立一個由 string 映射到 int 的 map
    tall := map[string]int{
        "小櫻" : 153,
        "知世" : 155,
        "小狼" : 156,
    }
    _, ok := tall["小狼"]
    fmt.Printf("tall[\"小狼\"] 存在嗎?%t", ok) // 第 11 行
}
```

執行結果：

tall["小狼"] 存在嗎? true

在第 11 行 print 的地方當要 print 出雙引號時，為了避免被誤認成字串用的雙引號，可以在雙引號前加上反斜線 \ 去告訴程式這是一般的字不是識別字串起頭結尾的字

# 組合型別

## 映射(map)

- 透過for range關鍵字，遍歷造訪映射內的每個元素：

```
var Size = map[string]string{
    "big":    "大",
    "medium": "中",
    "small":  "小",
}
```

```
for key, value := range Size {
    fmt.Println(key, value)
}
/* result:
big 大
medium 中
small 小
*/
```

# 組合型別

## 映射(map)

- 利用 `for` 迴圈走訪時，通常會用 `key, value` 去接 `range` 的回傳值，其實也可以只用一個參數，而這個參數預設是 `key`，所以如果只想走訪 `key` 時就可以不用使用 `k, _ := range xxx` 這種寫法，可以直接寫成 `k := range xxx`：

```
package main
import "fmt"
func main(){
    tall := map[string]int{
        "小櫻" : 153,
        "知世" : 155,
        "小狼" : 156,
    }

    for k := range tall{
        fmt.Println(k)
    }
}
```

執行結果：

小櫻  
知世  
小狼

# 組合型別

## 結構(struct)

- 先前介紹的變數都是儲存單一的值或是多個**相同型態**的值，那如果要用變數表示較複雜的概念，像是紀錄一個人的名字、年齡或是身高時，由於這些是**不同的資料型態**，所以要記錄下來時，就必須使用不同的容器，這裡會介紹 Go 語言中的結構 **Struct**。
- 其實**Struct**就是有點類似**OOP**(物件導向)的概念。比如說今天想要建立一個型態"人"，那人有姓名和身高，這時候就可以使用**Struct**：

```
person {  
    name  
    height  
}
```

# 組合型別

## 結構(struct)

- 建立結構(定義型態)
  - Go 使用 `struct` 做為結構的關鍵字，這是承襲 C 的慣例。下面建立一個結構：
  - 通常會在宣告結構時一併定義新的型別，因為種結構在程式裡，可能會多次使用，便於後續程式呼叫，**使用 `type` 可以宣告新型別**，在上述的例子建立一個叫 `Person` 的 `struct`，而裡面的結構組成有 `name(string)`、`year (int64)`和 `heigh(float64)`。接著以此結構，宣告一個變數，並填入其裡頭的屬性：

```
type Person struct {  
    name string  
    year  int64  
    heigh float64  
}
```

```
package main  
  
import "fmt"  
  
type Person struct {  
    name string  
    year  int64  
    heigh float64  
}  
  
func main() {  
    jack := Person{  
        name: "Jack",  
        year: 18,  
        heigh: 178,  
    }  
  
    fmt.Println(jack) // {Jack 18 178}  
}
```

# 組合型別

## 結構(struct)

- 建立結構(定義型態)
  - 透過使用結構，可以更有效率地處理資料數據，以下示範如何使用結構裡的數據，來進來一些運算及判斷：

```
package main
import (
    "fmt"
)
type Rectangle struct {
    length float64
    width  float64
}
func main() {
    x := Rectangle{
        length: 3,
        width: 3,
    }

    if x.length == x.width{
        fmt.Println("這是個正方形，且面積為", x.length * x.width)
    } else{
        fmt.Println("這只是個長方形，面積為", x.length * x.width)
    }
}
```

指定操作：如果想要存取struct裡面的內容，只要在後面加"."就好了  
先宣告一個名為 **Rectangle** 的結構，裡面屬性有 **length** 和 **width**，再以此結構宣告一個變數 **x** 並填入其屬性，然後依照屬性來判斷其是否為正方形，並計算其面積。

# 組合型別

## 結構(struct)

- 建立結構(定義型態)
- 不使用 `type` 直接使用 `struct` 宣告：

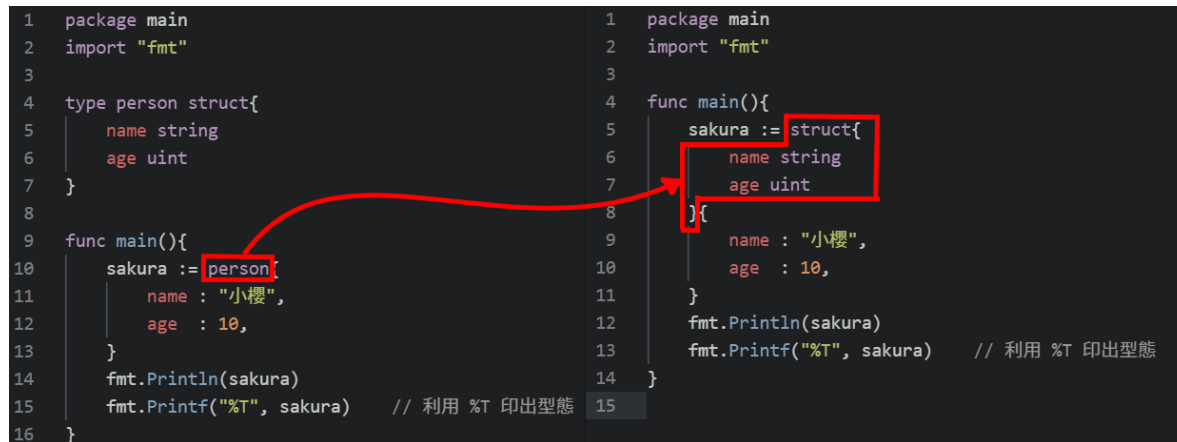
```
package main
import "fmt"

func main(){
    sakura := struct{
        name string
        age uint
    }{
        name : "小櫻",
        age  : 10,
    }
    fmt.Println(sakura)
    fmt.Printf("%T", sakura)    // 利用 %T 印出型態
}
```

執行結果：

{小櫻 10}

struct{ name string; age uint }



```
1 package main
2 import "fmt"
3
4 type person struct{
5     name string
6     age uint
7 }
8
9 func main(){
10     sakura := person{
11         name : "小櫻",
12         age  : 10,
13     }
14     fmt.Println(sakura)
15     fmt.Printf("%T", sakura)    // 利用 %T 印出型態
16 }
```

```
1 package main
2 import "fmt"
3
4 func main(){
5     sakura := struct{
6         name string
7         age uint
8     }{
9         name : "小櫻",
10         age  : 10,
11     }
12     fmt.Println(sakura)
13     fmt.Printf("%T", sakura)    // 利用 %T 印出型態
14 }
15
```

# 組合型別

## 結構(struct)

- 建立結構(多重定義)(巢狀結構)

- 比如說人比較多了，想要建立群組，裡面有群組名稱及人：

```
group {  
    name  
    person  
}
```

這時候就算是struct裡面又包了struct。

- 定義

- 首先是person的部分：

```
type person struct {  
    name string  
    height int  
}
```

- 再來是group，裡面包含struct person：

```
type group struct {  
    name string  
    person  
}
```

- 宣告：

```
g := group{"LINE", person{name: "Emily", height: 158}}
```

- 存取操作：那要如何存取struct裡面的struct呢？基本上就是一直.就對了：

```
g.person.name
```

- 如果想存取height在group裡面是沒有定義的變數，可以偷懶一點像是：

```
g.height
```

效果會等同於

```
g.person.height
```

- 那像是group和person都有name呢？那這時候就不能用這種偷懶的寫法了



# 組合型別

## 結構(struct)

- function操作(基本方法)

- 假設要寫一個method給person用，為Greeting()：

```
func (p person) Greeting() {  
    fmt.Println("Hi~")  
}
```

- 如何使用：`p.Greeting()`

- override(覆寫)

- 那如果也想要幫group也做一個Greeting()怎麼辦呢？這時候就是override：

- 這時候使用group的Greeting()就會不一樣了：

```
g.Greeting()
```

```
func (p person) Greeting() {  
    fmt.Println("Hi~")  
}  
  
func (g group) Greeting() {  
    fmt.Println("We are a group")  
}
```

# 組合型別

## 結構(struct)

- 在函式中使用 struct
- 以下為錯誤範例：(試著將 person 中的 age 加一)

```
package main
import "fmt"

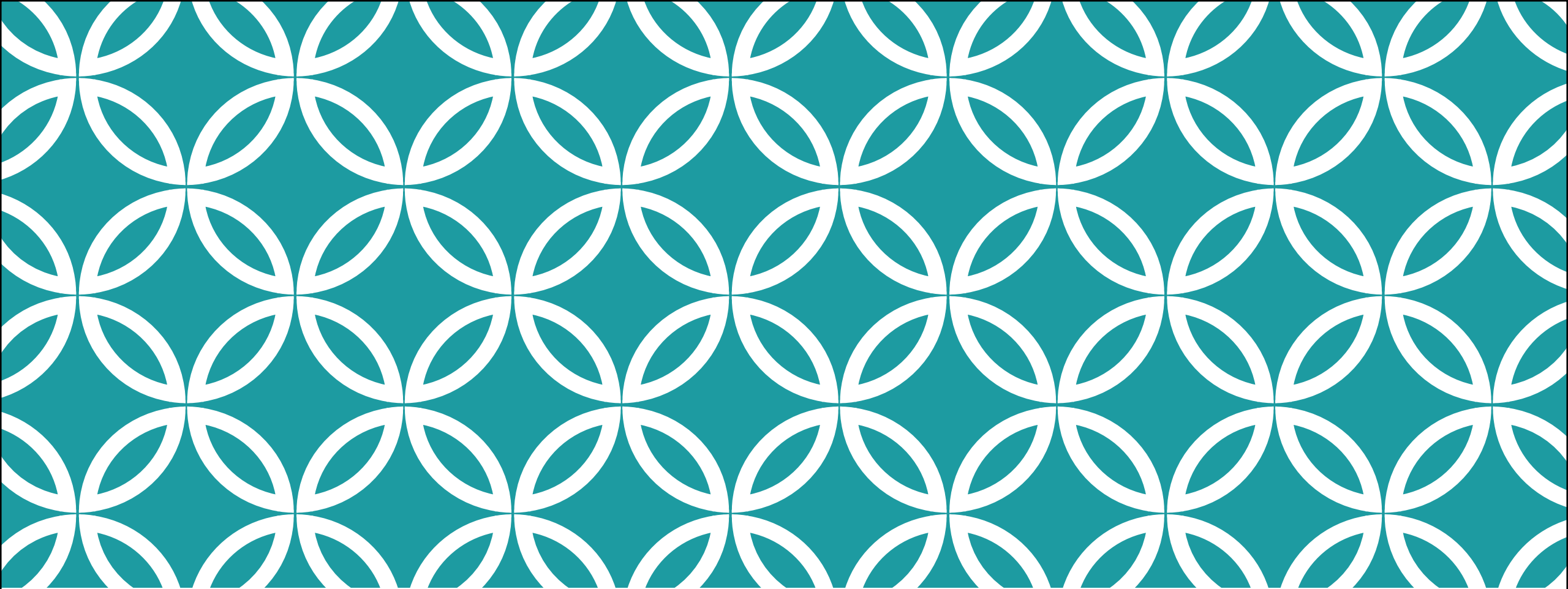
type person struct{
    name string
    age uint
}

func plusOne(p person){
    p.age = p.age + 1
}

func main(){
    sakura := person{
        name : "小櫻",
        age  : 10,
    }
    fmt.Println(sakura)
    plusOne(sakura)
    fmt.Println(sakura)
}
```

執行結果：  
{小櫻 10}  
{小櫻 10}

sakura 是一個 person 型態 的變數，並不是「指標變數」，  
若想要能在函式中更改 sakura，那麼需要取得 sakura 的**指標**才行



指標

⇒ GO



# 指標

## 指標(point)

- 值的儲存

- Go語言採用了一個簡單的記憶體管理系統叫堆疊(stack)，每個參數都會在堆疊中獲得自己的記憶體，缺點是有越多值在函式之間傳遞，這樣重複的動作就會消耗越多的記憶體。
- 但其實還有一種在函式傳值的替代方式，使用的記憶體較少，這種方式不會複製值，而是建立稱指標(pointer)的東西傳遞給函式。

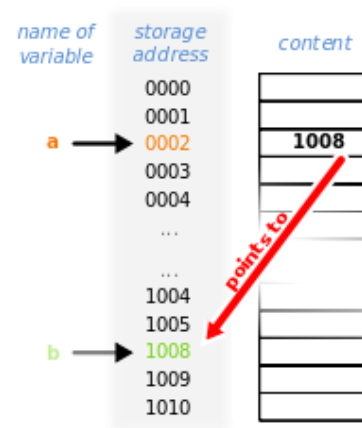
- 指標(址)的儲存

- 建立一個指標後，Go語言會將值放在所謂的堆積(heap)記憶體空間，堆積允許一個值存在，直到沒有任何指標參照為止。Go語言會用所謂的垃圾回收機制(garbage collection)程序來回收這些記憶體。這種機制會在背景定期運作，無需特別操心。
- 有時就算值沒有設置任何指標，也會因為其他原因被放進堆積裡面，值究竟該放到堆疊還是堆積上，無法介入，因為記憶體管理並非Go規格的一部分，而是被視為內部實作細節，所以上述所說的特性只是一般性指引，非鐵則。
- 指標會有未設定(is not set)的狀態，沒有儲存目標值的地址時會回傳nil，而nil這個特殊值在Go語言中代表無值。

# 指標

## 指標(point)

- 何謂指標？ 指標就是指到記憶體的位置，並不是直接存值
- 如上圖所述，`pointer` 只是指出該變數的「住址」，並不是該變數的值，在 `golang` 裡面，要取「址」需要用 `&` 這個符號。`*` 符號則是傳回該「位址」的值



```
package main

import (
    "fmt"
)

func main() {
    var a int
    a = 15

    fmt.Println(&a)
}
```

可以看到它印出 `a` 這個變數的記憶體位址

```
package main

import "fmt"

func main() {
    var p *int // 宣告p是一個int的指標，但此時他要指向哪還不知道
    a := 10 // a佔用了一個記憶體空間

    p = &a // 將p指到a的記憶體位置

    fmt.Println(p) // p所指到的記憶體位置
    fmt.Println(*p) // *代表顯示該記憶體位置的值
}
```

# 指標

## 指標(point)

- 指標變數
  - 指標通常指的是，一個指向記憶體位址(memory address)的變數，而這也意味著指標是一種型別，就和函式變數一樣，可以當作變數使用的都是一種型別，因此可以宣告一個指標變數，如下：

```
var ptr *string // 宣告一個指向字串的指標變數ptr

fmt.Printf("%p \n", ptr) // 印出: 0x0 (尚未指派記憶體位置)

if ptr == nil {
    fmt.Println("ptr is nil")
}
```

- 指標變數的初始值一律是nil，表示尚未指派記憶體位址，指標可以透過格式化輸出(%p)印出內容。

# 指標

## 指標(point)

- 指標變數

```
str := "Iron Man" // 宣告一個字串變數str

var ptr *string // 宣告一個指向字串的指標變數ptr

ptr = &str //取出str的記憶體位址給ptr

fmt.Printf("%p \n", ptr) // 印出: 0xc000044740 (str變數記憶體位置, 以16進位表示)

fmt.Printf("%s \n", *ptr) // 印出: Iron Man
```

- 範例中，在完成宣告變數後，透過取址符號 `&` 取出字串`str`的記憶體位址，然後賦值給指標`ptr`。再來透過格式化輸出`(%p)`印出指標內容，也就是16進位的記憶體位址。如果想要取出記憶體位址中的實際內容，就必須使用取值符號 `*` 來取出指標指向的變數內容。

# 指標

## 指標(point)

- 指標變數
  - 而指標變數的型別是以指向的型別來做區分：

```
str := "Iron Man" // 宣告一個字串變數str  
  
var ptr *int // 宣告一個指向整數的指標變數ptr  
  
ptr = &str //取出str的記憶體位址給ptr
```

- 這樣寫就會出錯：

```
$ go run ./main.go  
# command-line-arguments  
./main.go:14:6: cannot use &str (type *string) as type *int in assignment
```

- 最後，因為考量安全問題，go語言不支援指標運算



# 指標

## 指標(point)

- 有了Pointer的概念，就比較好理解function是怎麼傳值的：

```
package main

import "fmt"

func foo(x int) {
    fmt.Println(&x) // function內x的記憶體位置
}

func main() {
    a := 10
    fmt.Println(&a) // main裡面a的記憶體位置
    foo(a)
}
```

結果會看到兩個記憶體位置是不一樣的。代表在function傳值過去後，function複製該值到另一個空間來操作，對於原本main裡面的值是不會影響的

# 指標

## 指標(point)

- 傳指標(pass-by-pointer)：Go像C語言一樣有指標的概念，同樣可以傳遞

```
package main

import "fmt"

func foo(x *int) {
    fmt.Println(x) // function內x的記憶體位置
}

func main() {
    a := 10
    fmt.Println(&a) // main裡面a的記憶體位置
    foo(&a)
}
```

- 當然，想要在傳指標到function內操作也是可以的
- 可以看到傳指標過去後，兩邊操作的是同一個東西

```
package main

import "fmt"

func change(x *string) {
    *x = "Tom"
}

func main() {
    name := "Troy"
    fmt.Println(&name) // name的記憶體位置

    change(&name) // 傳記憶體位置

    fmt.Println(&name) // 記憶體位置不變
    fmt.Println(name) // 但內容有變
}
```

# 指標

## 指標(point)

- 用兩個範例說明運用指標的差異：

```
package main

import (
    "fmt"
)

func Compute(a int) {
    a = 0
}

func main() {
    x := 5
    Compute(x)
    fmt.Println(x)
}
```

左邊經過 `Compute` 這個 `func`，`x` 值並無改變，因為在 `golang`，運作行為是『複製』變數傳進去，所以兩個變數是完全獨立且分開的。

右邊為什麼 `x`，會有不一樣？是因為這時候 `golang` 行為是傳送位址進去，所以在 `Compute` 裡面所改到的值是原始 `x`，所改到的值。

`golang` 裡面 `slice` & `map` 都是 `pointer` 型別，所以如果把它當作參數傳入使用，要注意。

```
package main

import (
    "fmt"
)

func Compute(a *int) {
    *a = 0
}

func main() {
    x := 5
    Compute(&x)
    fmt.Println(x)
}
```

# 指標

## 指標(point)

- 設計一個變數交換的函式：

```
func swap(a, b int) (int, int) {  
    temp := a  
    a = b  
    b = temp  
    return a, b  
}
```

```
func swap(a, b *int) {  
    temp := *a  
    *a = *b  
    *b = temp  
}  
  
func main() {  
    a, b := 1, 2  
    swap(&a, &b)  
    println(a, b)  
}  
// 執行結果: 2 1
```

- 傳入兩個整數後，做完交換就回傳給呼叫者，看似很合理。但如果函式不想有要回傳值的話，有辦法做的到嗎？又該如何把結果交給呼叫者？這時候就可以使用指標的概念來設計這個函式：
- 右上範例用指標改寫了swap函式，取值符號 \* 可以讀取指標資料，也可以修改指標資料。這樣一來不必回傳值，就可以把函式中的計算結果交給呼叫者。

# 指標

## 指標(point)

- 切片其實是一種指標
- 切片的本質上類似一個指標，所以傳遞切片時可以**直接更改切片中的值**

```
package main
import "fmt"

func double(nums []int){
    for i := 0; i < len(nums); i = i+1{
        nums[i] = nums[i] * 2
    }
}

func main(){
    nums := []int{1, 3, 4, 7, 9}
    double(nums)
    for _, v := range nums{
        fmt.Printf("%d ", v)
    }
}
```

執行結果：  
2 6 8 14 18

# 指標

指標(point)

- **Map 也是一種指標**

```
package main
import "fmt"

func plusOne(m map[string]int){
    for k, v := range m{
        m[k] = v + 1
    }
}

func main(){
    tall := map[string]int{
        "小櫻" : 153,
        "知世" : 155,
        "小狼" : 156,
    }
    plusOne(tall)
    for _, v := range tall{
        fmt.Printf("%d ", v)
    }
}
```

執行結果

154 156 157

# 指標

## 指標(point)

- 陣列不是指標

- 這個部分跟 C 不太一樣，基本上 C 中的陣列就是單純的指標，但在 Go 中陣列其實加了一些料，所以不能當指標來用。也因此，比起 C 語言的指標，Go 中的指標算是好學許多

```
package main
import "fmt"

func double(nums [5]int){    // 第 4 行
    for i := 0; i < len(nums); i = i+1{
        nums[i] = nums[i] * 2
    }
}

func main(){
    nums := [5]int{1, 3, 4, 7, 9}
    double(nums)
    for _, v := range nums{
        fmt.Printf("%d ", v)
    }
}
```

執行結果

13479

雖然在函式 `double()` 中更動了 `nums`，但因為陣列 `nums` 傳進函式 `double()` 是傳送「值」而不是「址」所以並不會影響原先的陣列

有一點要注意的是，第4行中以陣列當參數時型態中要指定長度，要是沒有指定長度，**Golang**會把它當成切片而不是陣列

# 指標

## 指標(point)

- 陣列透過**取址**也是可以當指標傳遞
- 然而，如果很執著想要透過函式更改陣列的值也不是不行，透過取址的方式仍然可以實現

```
package main
import "fmt"

func double(nums *[5]int){
    for i := 0; i < len(nums); i = i+1{
        (*nums)[i] = (*nums)[i] * 2
    }
}

func main(){
    nums := [5]int{1, 3, 4, 7, 9}
    double(&nums)
    for _, v := range nums{
        fmt.Printf("%d ", v)
    }
}
```

執行結果：

2 6 8 14 18



# 指標

## 指標(point)

- 傳結構(struct)：自訂型別struct也是一樣可以傳的，其實也是傳pointer的概念

```
package main

import "fmt"

type stuff struct {
    name string
    price int
}

func main() {
    p := stuff{"pencil", 10}
    fmt.Println(p.price)
    inprice(&p)
    fmt.Println(p.price)
}

func inprice(s *stuff) {
    s.price += 10
}
```

```
type Hero struct {
    name      string
    age, power int
}

func main() {
    var tony = &Hero{"IronMan", 30, 666}

    fmt.Println("Before change:", *tony)

    changeHero(tony)

    fmt.Println("After change:", *tony)
}

func changeHero(h *Hero) {
    h.name = "Dr.Strange"
    h.age = 55
    h.power = 9999
}
```

```
Before change: {IronMan 30 666}
After change: {Dr.Strange 55 9999}
```

上面的結果中，發現結構內容被修改了，因為這次是傳遞指標參數，函式會透過記憶體位址修改原本的結構

# 指標

## 指標(point)

- 指標、結構、位址

```
package main

import "fmt"

type Wallet struct {
    Blue1000 int // 藍色小朋友
    Red100    int // 紅色國父
    Card      string
}

type PencilBox struct {
    Pencil string
    Pen     string
}

type Bag struct {
    Wallet // 直接放入結構就好
    PencilBox // 直接放入結構就好
    Books    string
}
```

```
type Person struct {
    Bag // 放Bag這個物件
    Name string
}

func main() {
    var bag = &Bag{
        Wallet{Card: "世華泰國信用無底洞卡", Red100: 5},
        PencilBox{Pen: "Cross", Pencil: "Pentel"},
        "Go繁不及備載",
    }

    var Tommy = Person{}
    Tommy.Name = "Tommy"
    Tommy.Bag = *bag // 透過`取值`來取出bag位址裡面的東西

    fmt.Printf("%+v", Tommy)
}

/* result:
{Bag:{Wallet:{Blue1000:0 Red100:5 Card:世華泰國信用無底洞卡} PencilBox:{Pencil:Pentel Pen:Cross} Books:Go繁不及備載} Name:Tommy}
*/
```

印出bag 就要透過\*來取值

# 指標

## 指標(point)

- 指標、結構、位址
- 如果將Person裡的Bag改成 \*Bag：這樣子就會印出bag的位址

```
package main

import "fmt"

type Wallet struct {
    Blue1000 int // 藍色小朋友
    Red100    int // 紅色國父
    Card      string
}

type PencilBox struct {
    Pencil string
    Pen     string
}

type Bag struct {
    Wallet    // 直接放入結構就好
    PencilBox // 直接放入結構就好
    Books     string
}
```

```
type Person struct {
    *Bag // 放指標
    Name string
}

func main() {
    var bag = &Bag{ // 指到位址
        Wallet{Card: "世華泰國信用無底洞卡", Red100: 5},
        PencilBox{Pen: "Cross", Pencil: "Pentel"},
        "Go繁不及備載",
    }

    var Tommy = Person{}
    Tommy.Name = "Tommy"
    Tommy.Bag = bag // 這裡就印出bag位址

    fmt.Printf("%+v", Tommy)
}

/* result:
{Bag:0xc000048050 Name:Tommy}
*/
```

# 指標

## 指標(point)

- 指標、結構、位址
- invalid recursive type

```
type PencilBox struct {
    Pencil string
    Pen     string
    Bag      // 你中有我 我中有你
}

type Bag struct {
    Wallet
    PencilBox
    Books    string
}
```

```
type PencilBox struct {
    Pencil string
    Pen     string
    *Bag      // 你中有針
}

type Bag struct {
    Wallet
    PencilBox
    Books    string
}

func main() {
    var bag = Bag{
        Wallet{Card: "世華泰國信用無底洞卡", Red100: 5},
        PencilBox{Pen: "Cross", Pencil: "Pentel"},
        "Go繁不及備載",
    }
    bag.PencilBox.Bag = &bag // 包包裡放針

    fmt.Printf("%+v", *bag.PencilBox.Bag)
}

/* result:
{Wallet:{Blue100:0 Red100:5 Card:世華泰國信用無底洞卡} PencilBox:{Pencil:Pentel
Pen:Cross Bag:0xc00046060} Books:Go繁不及備載}
*/
```

雖然放物件會出現錯誤，  
但是放指標不會

# 指標

## 指標(point)

- 匿名結構
  - Go的結構還有一個特殊的宣告方式-匿名結構，感覺有點像匿名函式。不用事先定義好結構型別，在變數宣告時產生一次性的結構型別，如下：

```
// 一般變數宣告的匿名結構
var tony = &struct {
    name      string
    age, power int
}{"IronMan", 30, 666}

fmt.Println(*tony)

// 短變數宣告的匿名結構
stephen := &struct {
    name      string
    age, power int
}{name: "Dr.Strange"}

fmt.Println(*stephen)
```

```
{IronMan 30 666}
{Dr.Strange 0 0}
```

# 指標

## 指標(point)-New Func

- Go語言提供了一種方法，可以動態產生一個變數空間，如下：

```
ptr := new(string) // 動態配置一個字串型別的變數

*ptr = "Iron Man"

fmt.Printf("%s \n", *ptr) // 印出: Iron Man
```

- 這結果和原本的範例一樣，只差在有沒有先宣告str變數。**new()**這種方式稱作動態配置記憶體，而原本常用的 **var** 和 **:=** 宣告變數就稱作靜態配置記憶體。動態配置適合用於操作大量且不固定的記憶體空間。

# 指標

## 指標(point)-New Func

- 透過Struct物件Pointer，可以自製New Obj Func，這個func回傳被實體化物件的指標，相當於用New產生一個物件。

```
type cat struct {
    name string
}

func main() {
    var c = &cat{name: "始祖貓"}
    fmt.Println(c, &c)

    n1 := newCat("")
    n2 := newCat("複製貓三號")

    fmt.Println(n1, &n1)
    fmt.Println(n2, &n2)

    var c2 = new(cat) // 內建的新方法
    fmt.Println(c2, &c2)
}

func newCat(n string) *cat {
    return &cat{name: n}
}

/* result:
&{始祖貓} 0xc0000ca018
&{} 0xc0000ca028
&{複製貓三號} 0xc0000ca030
&{} 0xc0000ca038
*/
```

# 指標

## 指標(point)-nil

- Nil 在一般的程式語言其實就是 null, NULL, 或 None
- 在宣告一個變數時Golang會給一個預設值，比如 int, uint 會給 0、string 會給 ""，那麼如果是一個指標變數，會給什麼呢？  
Golang會給一個全部都是 0 的位址，稱為 nil。
- 要注意的是，nil 常常會導致程式出錯。就舉先前的程式為例：

為什麼會這樣呢？因為在預設情況下，宣告新的 person 時，crush 會預設為 nil，nil 就是一個沒有指向任何地方的指標，因此如果想要取得 nil.name 當然會報錯

```
執行結果：
奈緒子的本名是？ 柳澤奈緒子
panic: runtime error: invalid memory address or nil pointer dereference
[signal 0xc0000005 code=0x0 addr=0x8 pc=0x49136d]

goroutine 1 [running]:
main.(*person).getLoverName(...)
省略...
```

```
package main
import "fmt"

type person struct{
    name    string
    groups  []string
    crush   *person
}

func (p *person) getName() string{
    return p.name
}

func (p *person) setCrush(crush *person){
    p.crush = crush
}

func (p *person) getCrushName() string{
    return p.crush.name
}

func main(){
    sakura := &person{
        name    : "木之本櫻",
        groups  : []string{"啦啦隊"},
    }

    tomoyo := &person{
        name    : "大盜寺知世",
        groups  : []string{"合唱團"},
    }

    naoko := &person{
        name    : "柳澤奈緒子",
        groups  : []string{"啦啦隊"},
    }

    touya := &person{
        name    : "木之本桃矢",
    }

    yukito := &person{
        name    : "月城雪兔",
    }

    // naoko 並沒有設定喜歡的人
    sakura.setCrush(yukito)
    tomoyo.setCrush(sakura)
    touya.setCrush(yukito)
    yukito.setCrush(touya)

    // 對 naoko 存取喜歡的人的名字會...?
    fmt.Println("奈緒子的本名是？", naoko.getName())
    fmt.Println("奈緒子喜歡的人是？", naoko.getCrushName())
}
```



# 指標

指標(point)-nil

- 改寫 getCrushName() :

```
func (p *person) getCrushName() string{
    if p.crush == nil{
        return "沒有"
    }
    return p.crush.name
}
```

執行結果：

奈緒子的本名是？柳澤奈緒子  
奈緒子喜歡的人是？沒有

```
package main
import "fmt"

type person struct{
    name    string
    groups  []string
    crush   *person
}

func (p *person) getName() string{
    return p.name
}

func (p *person) setCrush(crush *person){
    p.crush = crush
}

func (p *person) getCrushName() string{
    return p.crush.name
}

func main(){
    sakura := &person{
        name    : "木之本櫻",
        groups  : []string{"啦啦隊"},
    }

    tomoyo := &person{
        name    : "大盜寺知世",
        groups  : []string{"合唱團"},
    }

    naoko := &person{
        name    : "柳澤奈緒子",
        groups  : []string{"啦啦隊"},
    }

    touya := &person{
        name    : "木之本桃矢",
    }

    yukito := &person{
        name    : "月城雪兔",
    }

    // naoko 並沒有設定喜歡的人
    sakura.setCrush(yukito)
    tomoyo.setCrush(sakura)
    touya.setCrush(yukito)
    yukito.setCrush(touya)

    // 對 naoko 存取喜歡的人的名字會...?
    fmt.Println("奈緒子的本名是？", naoko.getName())
    fmt.Println("奈緒子喜歡的人是？", naoko.getCrushName())
}
```