CH19 Go語言的特殊套件

提要

- 在最後的章節，我們來看go語言一些較為罕見的進階功能，雖然使用時機較少，但仍能在開發時提供一些協助

- 這兩個套件分別是實作反射(reflection)功能的reflect套件，能在執行階段檢視物件的型別與值，以及能同樣在執行階段直接存取記憶體，繞過型別系統的unsafe套件

19-1 反射(reflection)

- “反射” 指程式在有能力執行時期檢視和修改自己的內容，特別是透過型別操縱自身的資料，而go語言提供了reflect套件來實現這一點

- 注意：reflect套件能讓你用更底層的方式轉換和操縱資料，繞過正常語法的限制，但它並沒有非得使用的必要；此外，有鑑於reflect套件的功能相當複雜，下面只會示範一些基礎來介紹

19-1-1 TypeOf()和ValueOf()

取得空介面的動態型別與值

- 為了使用go語言的反射，你得先了解reflect套件的兩個函式：

```
func  TypeOf(i  interface{})  Type
func  ValueOf(i  interface{})  Value
```

這兩個函數都接收一個空介面，但會分別回傳reflet的Type和Value型別，這讓我們能檢視空介面底下含有的動態型別和動態值

以下範例會建立幾個不同型別的變數，並會印出將這些值傳給TypeOf()和ValueOf()後得到的結果：

```go
package main

import (
    "fmt"
    "reflect"
)

func Print(i interface{}) {
    fmt.Println("Type :", reflect.TypeOf(i))  //取得動態型別
    fmt.Println("Value:", reflect.ValueOf(i)) //取得動態值
}

func main() {
    a := 5
    Print(a)
    b := &a
    Print(b)
    c := []string{"test"}
    Print(c)
    d := map[string]string{"a": "b"}
    Print(d)
}

```

執行結果：

```
PS D:\git\GO> go run "d:\git\GO\ch19\19-1-1\main.go"
Type : int
Value: 5
Type : *int
Value: 0xc000012088
Type : []string
Value: [test]
Type : map[string]string
Value: map[a:b]
```

- 可以發現reflect套件能夠讀出空介面的形別及動態值，而不需要透過介面形別斷言 (當然你不會直接取得原始型別，而是透過TypeOf()傳回reflect.Type型別再做進一步的轉換)

- reflect.Type型別實際上是一個介面，代表透過反射取出的型別資訊和能做的各種行為；reflect.Value介面則代表用反射取出的值以及其他相關行為

19-1-2 取得指標值和修改之

- 現在我們來更仔細的看看前面範例中的變數b：它是整數a的指標，所以前面呼叫ValueOf()時只能印出其記憶體位址

- 若要在reflect取得指標指向的值，得呼叫reflect.Value的Elem()方法：

```
func  (v  Value)  Elem()  Value
```

- Elem()也會傳回reflect.Value, 但這回這就是指標指向的值
- 而既然我們是透過指標存取這個值，你也可以修改它
- 以下範例：

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
    a := 5
    b := &a

    v := reflect.ValueOf(b).Elem()           //取得b指向的指標
    fmt.Printf("%v %T\n", v.Int(), v.Int()) //轉成整數, 用fmt查看形別和值

    v.SetInt(10) //修改b指向的值
    fmt.Printf("%v %T\n", v.Int(), v.Int())
    fmt.Printf("%v %T\n", v.Interface(), v.Interface())
    fmt.Printf("%v %T\n", *b, *b)
}

```

- 如果你知道reflect.Value的值實際上是什麼型別，你可以直接轉換它 (呼叫 Int(), Float(), Bool(), String() 等方法，若無法轉換會引發 panic), 或用interface{}方法轉成空介面

- 我們在此甚至透過reflect.Value來修改b的值，並用不同方式印出

- 以下是這個範例的執行結果：

```
PS D:\git\GO> go run "d:\git\GO\ch19\19-1-2\main.go"
5 int64
10 int64
10 int
10 int
```

19-1-3 取得結構的欄位名稱/型別/值

- 現在來看一個更複雜的例子，這次要用reflect套件來直接檢視一個結構變數所用有的欄位/欄位型別/欄位值：

```go
package main

import (
    "fmt"
    "reflect"
)

type User struct {
    Name    string  `des:"userName"` //欄位帶有標籤
    Age     int     `des:"userAge"`
    Balance float64 `des:"bankBalance"`
    Member  bool    `des:"isMember"`
}

//用reflect檢視結構內容
func PrintStruct(s interface{}) {
    sT := reflect.TypeOf(s) //取得reflect.Type
    sV := reflect.ValueOf(s) //取得reflect.Value

    //印出結構型別名稱和其基礎型別名稱
    fmt.Printf("type %s %v {\n", sT.Name(), sT.Kind().String())

    //走訪結構欄位
    for i := 0; i < sT.NumField(); i++ {
        field := sT.Field(i) //取得第i個欄位的型別 (reflect.Type)
        value := sV.Field(i) //取得第i個欄位的值 (reflect.Value)

```

```go
        //印出欄位名稱/型別/值 以及標籤des的字串
        fmt.Printf("\t%s\t%s\t= %v\t(description: %s)\n",
            field.Name, field.Type.String(),
            value.Interface(), field.Tag.Get("des"))
    }

    fmt.Println("}")
}

func main() {
    u1 := User{
        Name:    "Tracy",
        Age:     51,
        Balance: 98.43,
        Member:  true,
    }

    PrintStruct(u1) //用reflect印出u1內容

    //透過u1的指標用reflect指名欄位名稱，以便更改欄位值
    v := reflect.ValueOf(&u1)
    v.Elem().FieldByName("Name").SetString("Grace")
    v.Elem().FieldByName("Age").SetInt(45)
    v.Elem().FieldByName("Balance").SetFloat(56.97)
    v.Elem().FieldByName("Member").SetBool(false)

    PrintStruct(u1) //再次印出u1內容
}
```

執行結果：

```
PS D:\git\GO> go run "d:\git\GO\ch19\19-1-3\main.go"
type User struct {
        Name    string  = Tracy (description: userName)
        Age     int     = 51    (description: userAge)
        Balance float64 = 98.43 (description: bankBalance)
        Member  bool    = true  (description: isMember)
}
type User struct {
        Name    string  = Grace (description: userName)
        Age     int     = 45    (description: userAge)
        Balance float64 = 56.97 (description: bankBalance)
        Member  bool    = false (description: isMember)
}
PS D:\git\GO>
```

- 這個範例展示了reflect不僅能處理基本型別，更可以深入結構找出欄位名稱/型別/值, 甚至和前面一樣可以修改值

- reflect.Type和reflect.Value都有以下兩個方法能存取結構欄位：
  - Field(I int) 用索引傳回欄位
  - FieldByName(name string) 用名稱傳回欄位
  - 若你是透過reflect.Type呼叫以上方法，會傳回reflect.StructField型別；StructFiel的Type屬性是另一個reflect.Type形別物件，代表結構欄位本身的型別
  - 若用reflect.Value呼叫以上方法，就會得到代表該欄位的reflect.Value物件

Type vs. Kind

- 注意到範例中印出結構u1型別時，呼叫的是reflect.Type的Kind()方法，這會傳回reflect.Kind結構，這和Type有何不同？
  - 每個複合型別因為定義的不同，會被視為不同型別，但其實同樣類型的複合型別(陣列,切片，map等)仍屬於同一種 “類型”(kind)

```
func  main()  {
        a := []int{1, 2, 3}
        b := string{“apple”, “banana”, “mango”}

        fmt.Println(reflect.TypeOf(a))  //印出 []int
        fmt.Println(reflect.TypeOf(b))  //印出[]string
        fmt.Println(reflect.TypeOf(a).Kind())  //印出slice
        fmt.Println(reflect.TypeOf(b).Kind())  //印出slice
}
```

- 以前面的結構u1來說，其形別會是main.User, 但類型就會是struct

19-1-4 練習：用reflect取代介面斷言

```go
package main

import (
    "errors"
    "fmt"
    "reflect"
)

func doubler(i interface{}) (string, error) {
    t := reflect.TypeOf(i)
    v := reflect.ValueOf(i)

    //用形別名稱來判斷，以便呼叫reflect.Value的正確轉值方法
    switch t.String() {
    case "string":
        return v.String() + v.String(), nil
    case "bool":
        if v.Bool() {
            return "truetrue", nil
        }
        return "falsefalse", nil
    case "float32", "float64":
        if t.String() == "float64" {
            return fmt.Sprint(v.Float() * 2), nil
        }
        return fmt.Sprint(float32(v.Float()) * 2), nil
```

```
27          case "int", "int8", "int16", "int32", "int64":
28              return fmt.Sprint(v.Int() * 2), nil
29          case "uint", "uint8", "uint16", "uint32", "uint64":
30              return fmt.Sprint(v.Uint() * 2), nil
31          default:
32              return "", errors.New("傳入了未支援的值")
33          }
34      }
35
36      func main() {
37          res, _ := doubler(-5)
38          fmt.Println("-5  :", res)
39          res, _ = doubler(5)
40          fmt.Println("5   :", res)
41          res, _ = doubler("yum")
42          fmt.Println("yum :", res)
43          res, _ = doubler(true)
44          fmt.Println("true:", res)
45          res, _ = doubler(float32(3.14))
46          fmt.Println("3.14:", res)
47      }
48
```

執行結果：

```
PS D:\git\GO> go run "d:\git\GO\ch19\19-1-4\main.go"
-5   : -10
5    : 10
yum : yumyum
true: truetrue
3.14: 6.28
```

19-1-5 DeepEqual

- 前面已經知道，go語言中可用 == 判斷兩值是否相等，這適用於陣列和結構，但切片跟map卻無法如此必較，這時就可以用DeepEqual()：

```go
package main

import (
    "fmt"
    "reflect"
)

func runDeepEqual(a, b interface{}) {
    fmt.Printf("%v DeepEqual %v : %v\n", a, b, reflect.DeepEqual(a, b))
}

func main() {
    runDeepEqual([3]int{1, 2, 3}, [3]int{1, 2, 3})
    runDeepEqual([]int{1, 2, 3}, []int{1, 2, 3})

    a := map[int]string{1: "one", 2: "two"}
    b := map[int]string{1: "one", 2: "two"}
    runDeepEqual(a, b)

    var c, d interface{}
    c = map[int]string{1: "one", 2: "two"}
    d = map[int]string{1: "one", 2: "two"}
    runDeepEqual(c, d)
}

```

執行結果：

```
[1 2 3] DeepEqual [1 2 3] : true
[1 2 3] DeepEqual [1 2 3] : true
map[1:one 2:two] DeepEqual map[1:one 2:two] : true
map[1:one 2:two] DeepEqual map[1:one 2:two] : true
```

- 可見DeepEqual()能比較切片，map甚至空介面，只要兩者內容相同就會回傳true

- 你可以將DeepEqual()看成 == 的延伸版

19-2 unsafe套件

- go語言雖然是靜態語言，但其執行環境會自動配和回收記憶體
- 但總是有時候開發者會需要直接存取記憶體，用更低階的方式提高程式效率，或故意繞過形別檢查，這時就要用到unsafe套件
- unsafe套件正如其名，使用時可能會造成問題

19-2-1 unsafe.Pointer指標

- 為了安全起見，指標型別是不能轉換成其他型別的：

```
a := int64(100)
b := &a
fmt.Println((*int32)(b) //嘗試執行時會被告知*int64不能轉*int32
```

go語言不允許更動指標型別，就是要避免不安全的資料操作；出於同樣理由，指標變數本身也不能用來數學運算

使用unsafe.Pointer指標

- 若真的想轉換指標型別，可透過unsafe.Pointer指標

- 這種指標可以和任何指標型別互轉：

```
a := int64(100)
b := (*int32)(unsafe.Pointer(&a))
fmt.Println(*b, reflect.TypeOf(b))   //印出100  *int32
```

因為unsafe套件能直接讀取記憶體，繞過了go語言本身的型別系統

- 既然unsafe.Pointer可和其他指標互轉 ,, 那原本一些無法直接轉換的型別也可以藉此互轉 :

```
a := int8(1)
fmt.Println(*(*bool()(unsafe.Pointer(&a)))  //將int8轉為bool, 印出true
```

須注意轉換後值不一定會正確 , 因為unsafe會直接讀取指標指向的記憶體位置 , 並試圖用另一種形別來解讀 ; 若目標型別使用不同的儲存格式或使用的空間不同 , 就有可能得到錯誤的值

結構型別的轉換

- 下面定義了兩個結構型別；由於他們擁有同樣的欄位組成，因此用unsafe轉換型別時資料就能保持原狀

```go
package main

import (
    "fmt"
    "unsafe"
)

type User struct {
    name string
    age  int
}

type Employee struct {
    name string
    age  int
}

func main() {
    a := User{"John", 42}
    fmt.Printf("a = %#v\n", a)

    //把a從User轉成Employee結構型別
    b := *(*Employee)(unsafe.Pointer(&a))
    fmt.Printf("b = %#v\n", b)
}
```

執行結果：

```
PS D:\git\GO> go run "d:\git\GO\ch19\19-2-1\main.go"
a = main.User{name:"John", age:42}
b = main.Employee{name:"John", age:42}
PS D:\git\GO>
```

19-2-2 以uintptr搭配unsafe存取記憶體位址

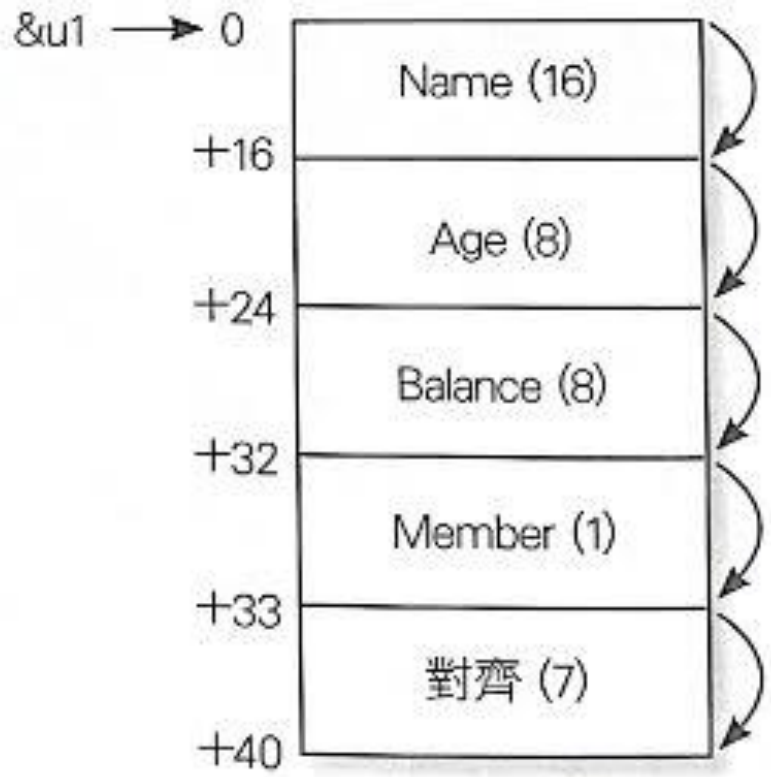
- unsafe能做的不僅於此，你更能直接存取複合型別的記憶體空間，該套件提供了以下三個函式：

```
func  Alignof(x  ArbitraryType)  uintptr  // x 在記憶體的最大對齊大小
func  Offsetof(x ArbitraryType)  uintptr // x(結構欄位)的記憶體偏移大小
func  SizeOf(x  ArbitraryType)  uintptr  // x的記憶體大小
```

ArbitraryType不是真正的型別，只是在說x能填入任一型別的值，那uintptr型別式甚麼值呢？

- 以前面19-1-3的User結構變數u1為例，每個欄位的大小和offset(偏移)如下：

欄位	型別	大小 (byte)	偏移
Name	string	16	0
Age	int	8	16
Balance	float64	8	24
Member	bool	1	32



- 在記憶體中，u1 的內容會由一系列連續的記憶體區塊組成，而&u1會指向這些記憶體的開頭(位置0)

- 若想在記憶體中直接存取某欄位的資料，就要加上offset, 例如u1.Balance位於&u1 + 24

- 注意若你用Sizeof()來測量u1的大小，會得到40而非33, 因為og語言得顧及跨平台運作的一致性，並提高記憶體效率，每個記憶體區塊都得有一定的大小 (及 “對齊”) ; Alignof()函式傳回的值，就是傳入的型別至多得填補的位元數

- 然而go語言的指標無法直接用於運算，就連unsafe.Pointer也一樣，因此得透過uintptr來參照記憶體位址；uintptr是go語言內建的特殊正整數型別，用來儲存記憶體位址和其偏移值，且能和unsafe.Pointer互轉

- 例如，下面用兩種方式印出u1 的記憶體位址：

  fmt.Println(unsafe.Pointer(&u1)) → 0xc0001003c0
  fmt.Println(uintptr(unsafe.Pointer(&u1))) → 824634770368(0xc0001003c0的十進位)\

  只要拿824634770368加上欄位的偏移值，就能存取欄位的記憶體空間，有此可見unsafe.Pointer扮演了中介角色，讓我們能輕易地計算記憶體的相對位址

練習：用unsafe和uintptr修改結構變數欄位

- 在這個練習裡, 要透過記憶體位址來存取u1結構變數的某些欄位，並修改他們的值，好讓你了解unsafe是如何透過記憶體操縱資料的

```go
package main

import (
    "fmt"
    "unsafe"
)

type User struct {
    Name    string
    Age     int
    Balance float64
    Member  bool
}

func main() {
    u1 := User{
        Name:    "Tracy",
        Age:     51,
        Balance: 98.43,
        Member:  true,
    }
    fmt.Println(u1)

```

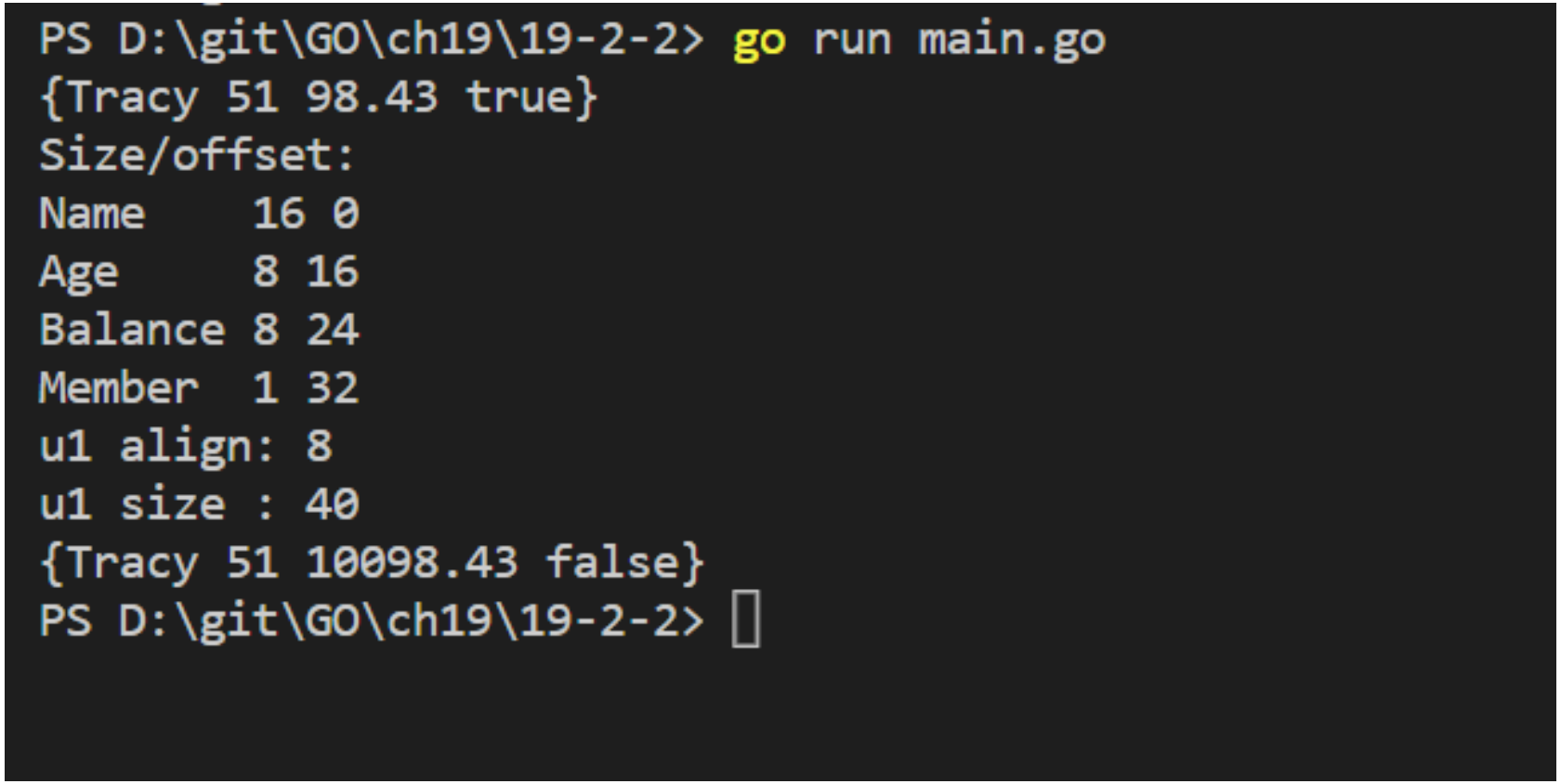
```
24      //顯示u1個欄位的記憶體大小及offset
25      fmt.Println("Size/offset:")
26      fmt.Println("Name   ", unsafe.Sizeof(u1.Name), unsafe.Offsetof(u1.Name))
27      fmt.Println("Age    ", unsafe.Sizeof(u1.Age), unsafe.Offsetof(u1.Age))
28      fmt.Println("Balance", unsafe.Sizeof(u1.Balance), unsafe.Offsetof(u1.Balance))
29      fmt.Println("Member ", unsafe.Sizeof(u1.Member), unsafe.Offsetof(u1.Member))
30      //u1的對其大小和總大小
31      fmt.Println("u1 align:", unsafe.Alignof(u1))
32      fmt.Println("u1 size :", unsafe.Sizeof(u1))
33
34      //建立指標指向u1.Balance
35      //u1(u1.Name)位址 + 16 + 8 = u1.Balance位址
36      balance := (*float64)(unsafe.Pointer(uintptr(unsafe.Pointer(&u1)) + unsafe.Sizeof(u1.Name) + unsafe.Sizeof(u1.Age)))
37      *balance += 10000
38      //建立指標指向u1.Member
39      //u1(u1.Name位址) + 32 = u1.Member位址
40      member := (*bool)(unsafe.Pointer(uintptr(unsafe.Pointer(&u1)) + unsafe.Offsetof(u1.Member)))
41      *member = false
42
43      fmt.Println(u1) //印出修改後的u1
44  }
45
```

- 上面想修改的對象為u1.Balance和u1.Member, 並分別用Sizeof()和Offset()來跳到正確的記憶體位置

- 如前所述, u1會指向記憶體區塊的開頭(這位置也代表u1.Name); 於是我們先將u1的位址轉成uintptr, 以便進行運算 , 加上正確的offset後再轉回unsafe.Pointer

- 注意：由於uintptr儲存的記憶體位址數字沒有參照到任何go套件 , 若你試圖用他來建立變數就會立刻被go回收, 導致無法使用 , 因此必須在含有unsafe.Pointer的運算式中直接使用uinptr值

執行結果：


PS D:\git\GO\ch19\19-2-2> go run main.go
{Tracy 51 98.43 true}
Size/offset:
Name   16 0
Age   8 16
Balance 8 24
Member 1 32
u1 align: 8
u1 size : 40
{Tracy 51 10098.43 false}
PS D:\git\GO\ch19\19-2-2>

19-2-3 go語言標準套件中的unsafe

- 既然有資料安全問題，go語言自然不希望你用unsafe, 但仍有不少人使用他來繞過形別系統，讓執行效率變快

- 其時unsafe套件並非以go語言撰寫，而是內建在編譯器內的功能，這也表示unsafe套件的運作和go語言所在的平台有更直接的關係：若你在程式中使用unsafe, 跨平台可能就會遇到表現不一致的問題

- 人們常使用unsafe的另一個原因是cgo內建函式庫，它允許在go程式中呼叫C語言程式碼；為了讓go語言與cgo的C語言型別互轉，就會用到unsafe

- 注意：cgo套件會用到gcc工具才能執行

```go
package main

/*
//以header形式提供給cgo的c程式碼
#include <stdio.h>
#include <stdlib.h>

static void myprint(char* s) {
    printf("%s\n", s);
}
*/
regenerate cgo definitions
import "C" // 匯入cgo
import (
    "fmt"
    "unsafe"
)

func main() {
    s := "Hello C!" //原始字串

```

```
21      //把字串轉為CString形別
22      cString := C.CString(s)
23      //結束時釋放CString指向的記憶體空間'
24      defer C.free(unsafe.Pointer(cString))
25      //呼叫C程式的函式
26      C.myprint(cString)
27
28      //將CString轉回成go的[]byte切片
29      b := C.GoBytes(unsafe.Pointer(cString), C.int(len(s)))
30      fmt.Println(string(b))
31  }
32
```

- 注意cgo匯入時必須獨立寫import, 且C語言程式要以註解形式直接置於它前面

- 執行結果：

本章結束