CH17 運用Go語言工具

17-1 前言

- 到目前為止的章節中，各位已經學到了go語言的諸多實用功能，但go語言在安裝時也提供了許多實用工具

- 例如，我們會用go run執行程式，第8章用go mod產生模組檔，用go build 編譯執行檔，用go get下載第三方套件，第9章則用了go test做單元測試

- 但go語言提供的工具遠不只如此，這一章來看看go語言工具中最常用的功能

17-2 go build工具 : 編譯可執行檔

17-2-1 使用go build

- go build 會將到專案的原始碼(包括它用的套件)編譯成單一可執行檔
- 你也可以指定要編譯的特定原始檔，且能指定執行檔的名稱及產生位置：

  go　build　-o　執行檔路徑與名稱　模組/套件或檔案名稱

- 若使用套件名稱，專案資料夾或其父目錄必須有go.mod指定模組路徑
- 以下練習假設各位都有啟用go modules功能

練習：用go build產生執行檔

- 下面是個非常簡單的程式，但我們在專案的bin子目錄夾下產生一個叫hello_world.exe的執行檔：

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello World!!")
}

```

```
問題 13  輸出  偵錯主控台  GITLENS  JUPYTER  終端機
PS D:\git\Golang\ch17\17-2-1> go build -o .\bin\hello_world.exe main.go
PS D:\git\Golang\ch17\17-2-1> .\bin\hello_world.exe
Hello World!!
```

17-2-2 編譯條件:選擇要編譯的檔案

- Go語言程式能夠在多種作業系統和CPU架構下運作，但你可能會在編譯時希望針對當下的平台只編譯特定檔案，這時你就要使用編譯條件(build constraints)或編譯標籤(build tag)

- go語言編譯條件可以使用下面兩種方式呈現：

  1. 在.go程式檔開頭使用 //+build<平台> 特殊註解

  2. 將.go檔名命名為 XXX_<平台>.go

- 例如 , 你能在一個.go程式檔的開頭(與package<套件名稱>隔一航)加入類似的編譯條件 :

  // +build windows

- 意思是此檔案只會在Windows系統上被編譯 , 或著 :

  // +build amd64,darwin !386,windows

- 逗號代表AND(相當於&&), 空格代表OR(相當於||); 因此這樣的寫法代表 (amd64 AND darwin) OR ((NOT 386) AND windows) , 也就是針對AMD64電腦(即x86-64)的Darwin系統 , 或非386電腦的Windows系統來編譯

- 第二個方式是透過檔名來表明編譯平台，例如：

  main.go　← 無限制
  main_linux.go　← Linux系統,不限CPU架構
  main_windows_amd64.go　← Windows系統, AMD64架構

- 若以檔名來提供編譯條件，格式必須符合以下其中一種：

  XXX_<作業系統>.go
  XXX_<CPU 架構>.go
  XXX_<作業系統>_<CPU架構>.go

- 在你的系統上，作業系統和CPU架構會由GOOS即GOARCH環境變數來記錄，你可以在主控台下指令檢視：

```
PS D:\git\Golang\ch17\17-2-1> go tool dist list
aix/ppc64
android/386
android/amd64
android/arm
android/arm64
darwin/amd64
darwin/arm64
dragonfly/amd64
freebsd/386
freebsd/amd64
freebsd/arm
freebsd/arm64
illumos/amd64
ios/amd64
ios/arm64
js/wasm
linux/386
linux/amd64
linux/arm
linux/arm64
linux/mips
```

- 下面我們就來看如何在中使用編譯條件，讓你能在不同系統選擇編譯特定的檔案；你甚至能藉此撰寫同一個函式的不同實作，配合不同平台的需求

練習：對go程式加上編譯條件

- 在這個練習中會包含四個.go檔案，其中兩個針對Windows，另外兩個針對Linux

- 首先是兩支不同的主程式，特別留意它們的檔名：

ch17 > 17-2-2 > main_windows.go > ...

```go
//檔名_windows.go,只會在Windows系統編譯
package main

import "fmt"

func main() {
    fmt.Println("Hello Windows!")
    fmt.Println(greetings()) //呼叫來自main套件,位於其他檔案的函式
}
```

ch17 > 17-2-2 > main_linux.go

```go
//檔名是_linux.go, 因此只會Linux系統編譯
package main

import "fmt"

func main() {
    fmt.Println("Hello Linux!")
    fmt.Println(greetings())
}
```

- 接著是兩個同樣屬於main套件的 .go程式，各自有一個greetings()函式，但擁有不同的編譯條件：

ch17 > 17-2-2 > greet1.go > ...

```go
// +build windows

package main

func greetings() string {
    return "Greetings from Windows!"
}

```

ch17 > 17-2-2 > greet2.go

```go
// +build linux

package main

func greetings() string {
    return "Greetings from Linux!"
}

```

- 最後也請在專案內建立一個go.mod, 以便go build運作

- 現在來嘗試編譯它：

  windows系統編譯和執行結果：

```
PS D:\git\Golang\ch17\17-2-2> go build -o main.exe
PS D:\git\Golang\ch17\17-2-2> ./main
Hello Windows!
Greetings from Windows!
```

如何跨平台編譯

- 在Linux系統上，可以使用以下指令：

```
GOOS=windows GOARCH=amd64 go build -o main.exe
```

- 在Windows上，得先修改go環境變數，例如：

```
go env -w GOOS=linux
go env -w GOARCH=arm
```

然後再使用go build編譯即可

17-3 go run 工具 : 執行程式

- go run 和 go build 很像 , 差別在後者會將指定的模組,套件或檔案編譯成一個二進未執行檔 , 而go run會直接執行他 (臨時產生一個執行檔, 執行後刪除) , 非常適合拿來做測試

- 下面來看幾個應用方式 :

練習：用go run 執行程式

- 這裡要沿用17-2-1的練習，但先替他加入模組路徑(go mod init 17-2-1)，以便觀察不同的執行指令會有何效果

- 接著執行它：

```
PS D:\git\Golang\ch17\17-2-1> go run main.go
Hello World!!
PS D:\git\Golang\ch17\17-2-1> go run 17-2-1
Hello World!!
PS D:\git\Golang\ch17\17-2-1> go run .
Hello World!!
```

直接執行單一檔案

需要有go.mod

需要有go.mod

- 以上展示了幾種可以執行go程式的方法
- 和go build一樣，若指定的是模組套件名稱，必須先建立go.mod來提供模組路徑

練習：用go run 偵測記憶體資源爭奪

- 在16章講到併行性運算時，看到了記憶體資源爭奪的問題，即使是經驗老到的開發者也可能意外造成這種現象，解該問題發生的時間可能很短，導致難以察覺

- 不過，如前一章所示，可以用go run 或 go test 加上 –race 旗標來檢查是否發生記憶體資源競爭

- 下面的程式中，程式試圖透過goroutine對一個切片 name加入新元素，同時主程式的goroutine又要讀取name的內容，導致記憶體資源爭奪發生

- 由於程式可以正常執行，只有加上-race才能發現問題

```go
package main

import "fmt"

func main() {
    finished := make(chan bool)
    names := []string{"Packt"}

    //goruoutine嘗試在names加入值
    go func() {
        names = append(names, "Electric")
        names = append(names, "Boogaloo")
        finished <- true
    }()
    //但同時又被main()嘗試讀取names
    for _, name := range names {
        fmt.Println(name)
    }
    <-finished
}

```

執行結果：

```
PS D:\git\Golang\ch17\17-3> go run -race .
Packt
==================
WARNING: DATA RACE
Write at 0x00c000004078 by goroutine 7:
  main.main.func1()
      D:/git/Golang/ch17/17-3/main.go:11 +0xee

Previous read at 0x00c000004078 by main goroutine:
  main.main()
      D:/git/Golang/ch17/17-3/main.go:16 +0x184

Goroutine 7 (running) created at:
  main.main()
      D:/git/Golang/ch17/17-3/main.go:10 +0x17a
==================
Found 1 data race(s)
exit status 66
```

17-4 gofmt工具 : 程式碼格式化

- 一致且整齊的程式碼意味著有量好的閱讀性 , 使程式碼容易維護,不過開發實人們卻很常忽略了程式碼風格的重要性
- 為了克服這種問題 , go語言提供了使風格一致化的工具 : gofmt
- 只要團隊都使用gofmt, 那每個人提交的程式碼風格就是一樣的 , 有相同的縮排等等
- 語法 : gofmt -w 檔案名稱

練習：使用gofmt格式化程式碼

- 在以下練習中，各位將看到如何使用gofmt格式化程式碼
- 我們故意儲存一個格式亂七八糟的.go程式檔：

```
1   package main
2
3       import "fmt"
4
5   func
6   main(){
7     firstVar := 1
8           secondVar :=    2
9
10    fmt.Println(firstVar)
11                      fmt.Println(secondVar)
12        fmt.    Println("Hello Packt")
13                        }
14
```

- 若直接用gofmt, 它只會輸出格式化程式碼該有的樣子，並不會改變原本的檔案：

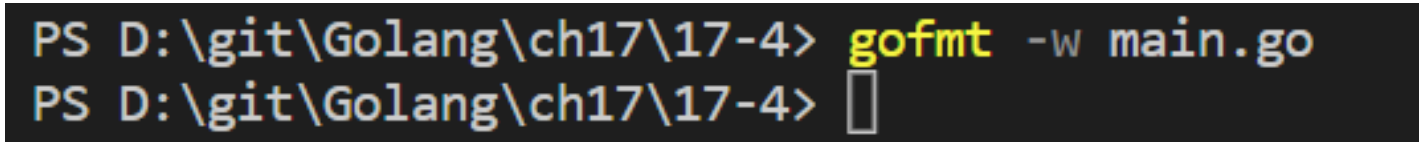
```
PS D:\git\Golang\ch17\17-4> gofmt main.go
package main

import "fmt"

func main() {
        firstVar := 1
        secondVar := 2

        fmt.Println(firstVar)
        fmt.Println(secondVar)
        fmt.Println("Hello Packt")
}
```

- 若希望gofmt能自動拿格式化後的結果複寫這個檔案，就得加上 –w 參數：



17-5 go vet : 程式靜態分析工具

- 儘管編譯時go語言會指出你可能犯的錯誤 , 但偶爾還是會漏掉一些小問題 , 而這些臭蟲會跟著程式一起正式上線 , 直到很久之後才被發現

- 例如 , Printf()傳入太多的引數 , 卻沒有足夠的格式化動詞來接收 ; 或是將非指標介面傳入Unmarshal(), 編譯器會認為合法 , 但Unmarshal()會沒辦法寫入資料到介面中

- 因此go語言設計了go vet工具 , 這是個能對你的程式碼做靜態分析的工具 , 使你能在臭蟲變成大問題時及時補救 :

  go vet 檔案名稱

練習：使用go vet找出程式問題

- 在這個練習中，會使用go vet找出上述的Sprintf()引數問題，讓你知道傳入的引數數量正不正確：

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       helloString := "Hello"
7       packtString := "Packt"
8
9       //傳2個引數給Sprintf, 卻只有一個%s
10      jointString := fmt.Sprintf("%s", helloString, packtString)
11      fmt.Println(jointString)
12  }
13
```

- 若執行這個程式, 他會毫無錯誤的通過編譯並執行 , 儘管輸出結果有問題

- 現在用go vet檢查main.go, 他會正確地指出錯誤 :

```
PS D:\git\Golang\ch17\17-5> go vet main.go
# command-line-arguments
.\main.go:10:17: fmt.Sprintf call needs 1 arg but has 2 args
```

- 修正方式:

```
10      jointString := fmt.Sprintf("%s %s", helloString, packtString)
```

- 修正後再次執行go vet :

```
PS D:\git\Golang\ch17\17-5> go vet main.go
PS D:\git\Golang\ch17\17-5> go run .
Hello Packt
```

17-6 go duc 工具 : 產生文件

- 程式規格文件(documentation)是許多軟體專案中經常被忽略的部分 , 因為撰寫和更新文件是很麻煩的過程 , 但這些文件對使用者至關重要

- 因此 , go語言設計了一個能從程式碼自動產生文件的工具 , 叫做 go doc

- go doc 用法十分簡單：只要在套件和有匯出的型別/函式前加上註解，go doc 就會把他們連同定義一起轉成規格文件

- 當你在大型專案需要與其他人合作時，這些文件就有助於讓別人理解你的套件要如何使用

- 與其花時間維護文件和溝通，專案各團隊的程式設計師就可以用 go doc 快速產生文件並分享給其他團隊

- 替go程式碼加入註解的規則如下：
  - 只有套件中有匯出(自首為大寫)的型別/函式才會出現在go doc產生的文件中
  - 撰寫文件的格式為：

    ```
    // <型別或函式名稱> 說明
    // 說明
    // ...
    <有匯出的型別或函式宣告>
    ```

  - 一段文件說明可以有多行，但第一行的第一個詞必須是型別或函式的名稱，習慣上第二個詞會是動詞，用來說明這個型別或函式在做甚麼
  - 套件本身的文件格式是：//package <套件名稱> 說明 ....

練習：用go doc產生規格文件

```go
// package main: Exercise 17-6
package main

import "fmt"

// Calc defines a calculator construct
type Calc struct{}

// Add returns the total of two integers added together
func (c Calc) Add(a, b int) int {
    return a + b
}

```

```go
// Multiply returns the total of one integers multipled by the other
func (c Calc) Multiply(a, b int) int {
    return a * b
}

// PrintResult prints out the received integer argument
func PrintResult(i int) {
    fmt.Println(i)
}

func main() {
    calc := Calc{}
    PrintResult(calc.Add(1, 1))
    PrintResult(calc.Multiply(2, 2))
}

```

- 注意到程式中的Calc結構型別和其他方法的定義前面都有註解，而且第一個詞會以型別或函式名稱開始

- 現在你就可以在同一個目錄使用go doc -all 或 doc doc –all main.go 來產生說明文件：

```
PS D:\git\Golang\ch17\17-6> go doc -all
package main: Exercise 17-6

FUNCTIONS

func PrintResult(i int)
    PrintResult prints out the received integer argument


TYPES

type Calc struct{}
    Calc defines a calculator construct

func (c Calc) Add(a, b int) int
    Add returns the total of two integers added together

func (c Calc) Multiply(a, b int) int
    Multiply returns the total of one integers multipled by the other
```

17-6 go get 工具：下載模組或套件

- go get能讓你下載網路上的套件，我們曾在第8章使用過
- 儘管go語言內建不少套件，但是使用網路上由第三方開發的套件能大大擴充你的程式功能
- 下面我們來看一個例子：

練習：用 go get 下載套件

- 下面是一個簡單的伺服器程式，使用gorilla/mux這個套件 (https://github.com/gorilla/mux)，而非go語言的http套件來處理請求路徑

```go
package main

import (
    "fmt"
    "log"
    "net/http"

    "github.com/gorilla/mux"
)

func exampleHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, "<h1>Hello Golang with gorilla/mux</h1>")
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/", exampleHandler)
    log.Fatal(http.ListenAndServe(":8080", r))
}

```

- 首先先用go mod init替專案建立模組路徑 ：

```
PS D:\git\Golang\ch17\17-7> go mod init 17-7
go: creating new go.mod: module 17-7
go: to add module requirements and sums:
        go mod tidy
```

- 然後使用go get 下載套件：

```
PS D:\git\Golang\ch17\17-7> go get -u github.com/gorilla/mux
go: added github.com/gorilla/mux v1.8.0
```

- 查看go.mod會發現他已列入了此套件的路徑與版本：

ch17 > 17-7 > go.mod

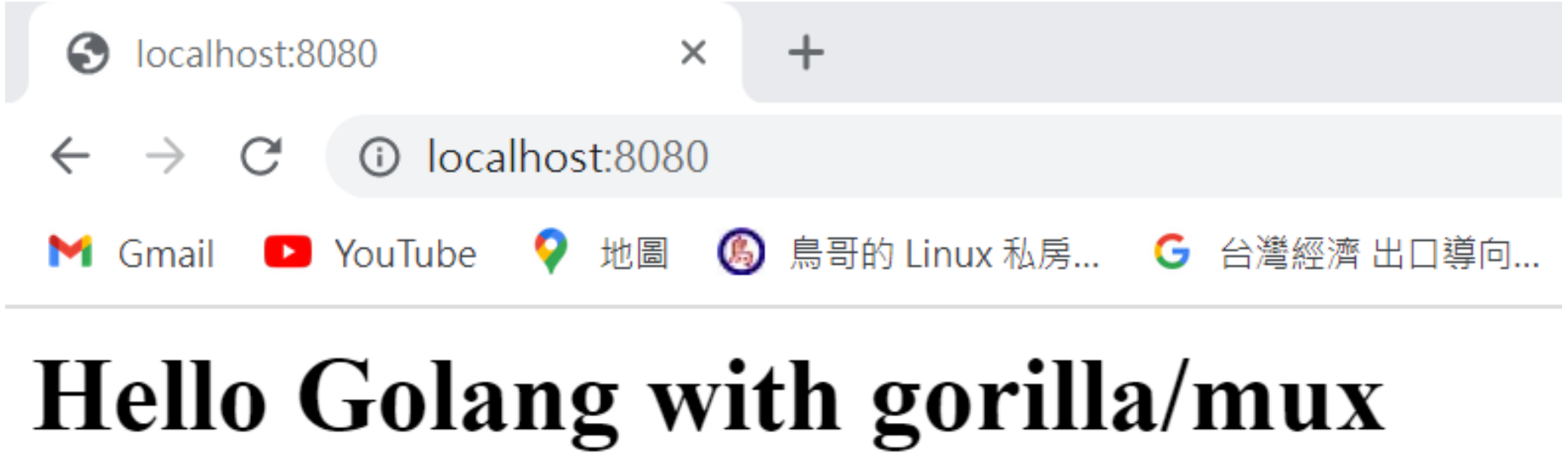
Run go mod tidy | Create vendor directory

```
1  module 17-7
2
3  go 1.18
4
```

Check for upgrades | Upgrade transitive dependencies |

```
5  require github.com/gorilla/mux v1.8.0
6
```

- 最後就可以執行程式，並在瀏覽器打開: localhost:8080 :



本章結束