CH9 程式除錯_格式化訊息_日誌_單元測試

9-1 前言

- 開發的程式正式上線後，仍然有可能因開發時的疏忽導致程式作出看似不按牌理出牌的行為

- 我們在第6章已經學到，程式錯誤可分為：語法錯誤/執行期間錯誤/邏輯錯誤

- 其中邏輯錯誤是最難察覺的，因為這在編譯階段無法察覺，也不見得會回傳錯誤值

9-1-1 臭蟲發生的原因

- 抓出臭蟲(bug)的最主要方法，就是在程式執行的過程中印出除厝訊息，來觀察程式執行過程和執行結果是否合乎預期，在本章中，我們就要來看在go程式中除錯(debugging)的一些基本方法

- 臭蟲發生的原因有以下幾種：

1. 直到開發尾聲才做測試：
   - 開發程式時會撰寫好幾個函式，但人們會偷懶，想說最後在一並做測試，但更好的做法是在每個函式完成時就做單元測試，然後依次測式個單元串連起來的結果，這稱為漸增測試
   - 這樣一來會更容易增進程式碼的穩定性，也是開發程式時須養成的習慣

2. 改良應用程式或因應需求變更：
   - 程式碼即使正式上線後，也可能會經常異動
   - 你會收到使用者的回饋或錯誤回報，客戶也可能提出新需求，但修改正式上線的程式碼有可能會無意間引入新臭蟲

3. 不切實際的開發時程：
   - 有時客戶會要求在非常急迫的時內完成程式設計，這往往導致開發者抄捷徑而非採用最佳實務做法/壓縮需求設計階段/減少測試次數，甚至來不急釐清客戶需求就逕行開發，這一切都會增加引入新臭蟲的可能性

4. 未善加處理錯誤：
   - 有些開發人員會偷懶，選擇不要處理錯誤，且忽略它們，但這樣不積極的錯誤處理態度會使程式出現臭蟲

9-1-2 除錯原則

1. 漸增式程式設計/經常性測試：
   - 在逐步開發程式時，每完成一塊就做測試，並逐漸加大測式範圍

2. 撰寫單元測試：
   - 典型的單元測試會輸入既定的測式資料，驗證處理結果是否符合預期
   - 若在變更程式碼前單元測試能通過，變更之後卻測式失敗，這代表作的變更造成了副作用

3. 所有錯誤都要處理：

- 在第6章已經談過，忽視error可能帶來的難以預料的結果，我們必須正確處理錯誤，才不會增加除錯的困難度

4. 做日誌紀錄(logging)：

- 日誌(log)是另一種除錯手段，log的種類很多，若按層級來分，常見包括trace(追蹤)/debug(除錯訊息)/info(資訊)/warn(警告)/fatal(重大錯誤)等
- 這些訊息通常用來記錄程式在臭蟲出現之前的狀況，替開發者收集諸如變數值/正在執行哪塊程式碼/傳入函式的引數值/函式的回傳值等等

- 在本章中，我們將運用go語言標準函式庫的log套件來輸出自訂的日誌訊息
- Log訊息會附帶時間戳記，這是很又用的資訊，能讓我們了解事件發生的順序
- 不過要注意的是，系統在尖峰運作期間有可能會密集輸出日誌訊息，進而影響軟體效能，設置拖慢程式反應
- 不過你不見得每次都要log來提供除錯訊息，事實上標準套件fmt就是很好用的工具，下面我們來回顧並深入探討fmt套件的格式化字串出功能

9-2 以fmt套件做格式化輸出

9-2-1 fmt套件

- fmt套件的主要用途，就是將資料輸出到主控台 (命令提示字元/powershell或終端機)或回傳格式化後的字串

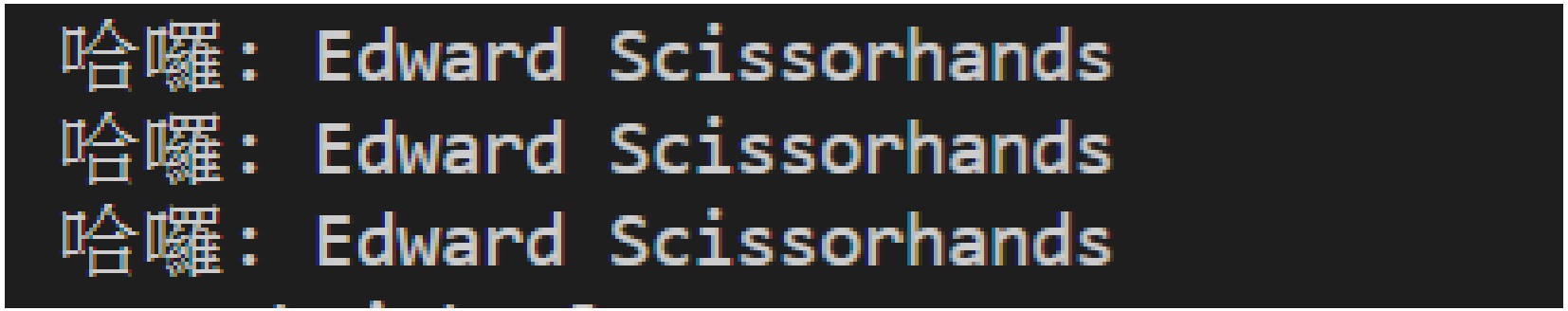
- 下面是fmt套件的一些常用函式，你能從他們的名稱看出其用途：

- Print() : 於主控台印出多筆資料，非字串的資料之間以空格隔開
- Println() : 於主控台印出多筆資料，但所有資料之間一定加入多筆資料，結尾也會加上換行字元
- Printf() : 於主控台印出多筆資料，使用格式化字串(結尾無煥行)
- Sprint() : 同Print()但回傳string
- Sprintln() : 同Println()但回傳string
- Sprintf() : 同printf()但傳回string

練習：用fmt套件印出字串

```go
package main

import "fmt"

func greeting(fname, lname string) string {
    return fmt.Sprintln("哈囉:", fname, lname)
}

func main() {
    fname := "Edward"
    lname := "Scissorhands"
    fmt.Println("哈囉:", fname, lname)
    fmt.Printf("哈囉: %v %v\n", fname, lname)
    fmt.Print(greeting(fname, lname))
}
```

執行結果：


哈囉: Edward Scissorhands
哈囉: Edward Scissorhands
哈囉: Edward Scissorhands

9-2-2 fmt的格式化輸出

- 現在我們要正式來看fmt.Printf(), 它可以使用更彈性的方式組合不同的值，輸出我們想要的字串，我們也不需要先將那些值轉成字串

- 在go語言中，格式化符號也稱為動詞(verbs)或指定符號(specifier), 而這些會在格式化字串中成為占位符(placeholder), 好讓Printf()之道該在何處插入轉換過的值

- 格式化用的動詞：

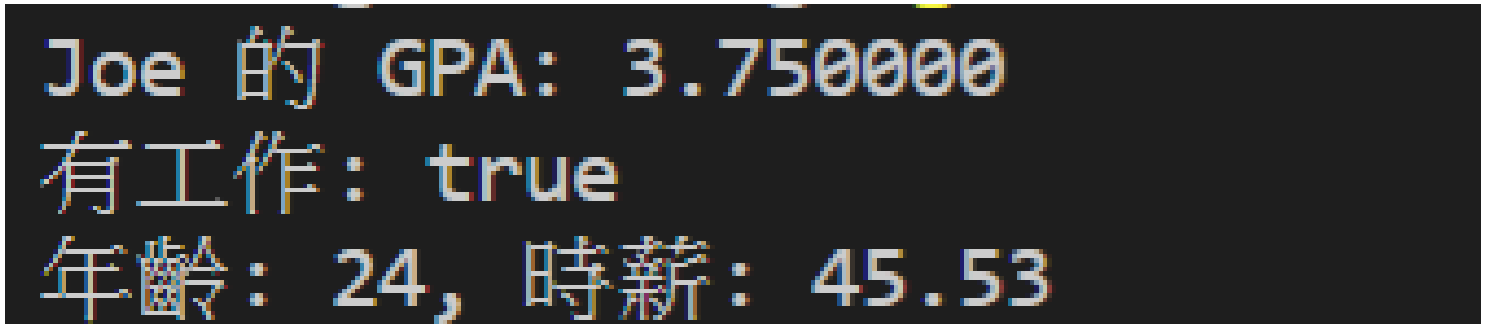
符號	格式化結果
%d	印出10進位整數
%f	印出浮點數
%e	印出帶有科學記號的浮點數
%t	印出不林值
%s	印出字串
%b	印出2進位值
%x	印出16進位值

當然還有其他動詞存在，後面會再介紹

- 來看以下範例：

```go
package main

import "fmt"

func main() {
    fname := "Joe"
    gpa := 3.75
    hasJob := true
    age := 24
    hourlyWage := 45.53
    //印出字串和浮點數
    fmt.Printf("%s 的 GPA: %f\n", fname, gpa)
    //印出布林值
    fmt.Printf("有工作: %t\n", hasJob)
    //印出整數與一般值
    fmt.Printf("年齡: %d, 時薪: %v\n", age, hourlyWage)
}
```

執行結果



練習：印出數值的10進位/2進位/16進位

- 在這個練習中，我們要用迴圈從整數1數到255, 然後觀察10進位/2進位/16進位印出的效果

```go
package main

import (
    "fmt"
)

func main() {
    for i := 1; i <= 255; i++ {
        fmt.Printf("10 進位: %3.d | ", i)
        fmt.Printf("2 進位: %8.b | ", i)
        fmt.Printf("16 進位: %2.x\n", i)
    }
}
```

執行結果(部分省略)

```
10 進位:   1 | 2 進位:        1 | 16 進位:  1
10 進位:   2 | 2 進位:       10 | 16 進位:  2
10 進位:   3 | 2 進位:       11 | 16 進位:  3
10 進位:   4 | 2 進位:      100 | 16 進位:  4
10 進位:   5 | 2 進位:      101 | 16 進位:  5
10 進位:   6 | 2 進位:      110 | 16 進位:  6
10 進位:   7 | 2 進位:      111 | 16 進位:  7
10 進位:   8 | 2 進位:     1000 | 16 進位:  8
10 進位:   9 | 2 進位:     1001 | 16 進位:  9
10 進位:  10 | 2 進位:     1010 | 16 進位:  a
10 進位:  11 | 2 進位:     1011 | 16 進位:  b
```

9-2-3 印出浮點數的進階格式化

- 在前面的程式中，浮點數gpa印出時後面有4個零，但也許我們也許會希望只要印出到小數第二位就好

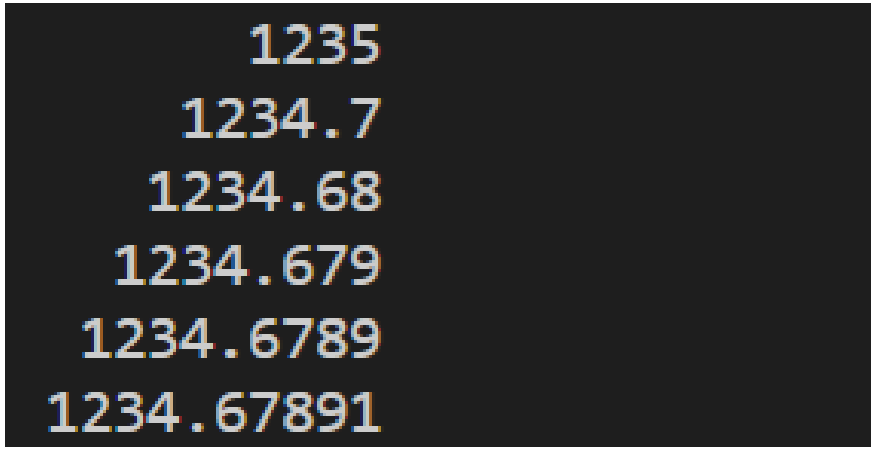
- 這時可以改寫%f動詞，在中間加上一個.2來代表把結果近位到小數第二位

- 你可以更進一步控制浮點數的整體長度，例如%10.2f就代表分配10個字元的長度給浮點數，包括一個小數點和2個小數位，因此整數部分顯示的長度便是7字元，若實際長度不足，就會把數字向右對齊，不足部分以空白填補

- 下面來看一個例子，能反映不同浮點數格式對於輸出結果的影響：

```go
package main

import "fmt"

func main() {
    v := 1234.678915
    fmt.Printf("%10.0f\n", v)
    fmt.Printf("%10.1f\n", v)
    fmt.Printf("%10.2f\n", v)
    fmt.Printf("%10.3f\n", v)
    fmt.Printf("%10.4f\n", v)
    fmt.Printf("%10.5f\n", v)
}
```

執行結果：



- 若你希望輸出結果左側補滿0, 可以這樣寫 : %010.2f
- 若你希望輸出結果靠左而不是靠右對齊 , 可以這樣寫 : %-10.2f

9-3 使用log提供追蹤訊息/日誌

- 程式有臭蟲是開發人員必經之事，而你可以用下面的基本除錯技巧來找臭蟲，或搜尋有關資訊：

1. 印出變數值
2. 印出變數型別
3. 印出追蹤訊息
4. 輸出 訊息到檔案：
   - 因為有些錯誤只會發生在正式上線的環境，或因某些原因無法透過主控台監控log訊息，這時就能用檔案先蒐集錯誤訊息

9-3-1 印出錯誤訊息

- 除錯的初步動作之一，就是先用追蹤訊息找出臭蟲在程式中的大略位置

- 通常追蹤訊息就只是用fmt.Println()等功能對主控台印出一段訊息

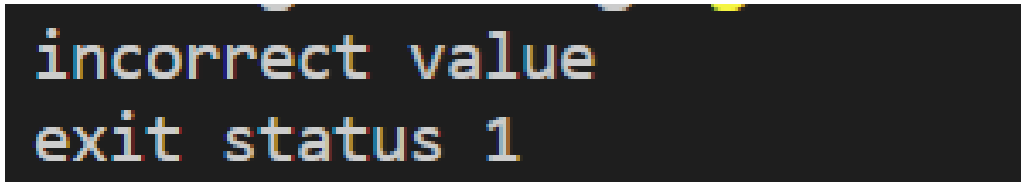
用追蹤訊息找出錯誤發生位置

- 下面來看一個例子，這個程式會隨機產生一個介於1~20之間的整數，而它呼叫的a()和b()函式一定會有一個回傳一個錯誤：a會在數字小於10時回傳error, b則會在數字大於10時回傳error

```go
package main

import (
    "errors"
    "fmt"
    "math/rand"
    "os"
    "time"
)

func main() {
    r := random(1, 20)
    err := a(r)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    err = b(r)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}

```

```go
func random(min, max int) int {
    rand.Seed(time.Now().UTC().UnixNano())
    return rand.Intn((max-min)+1) + min
}

func a(i int) error {
    if i < 10 {
        return errors.New("incorrect value")
    }
    return nil
}

func b(i int) error {
    if i >= 10 {
        return errors.New("incorrect value")
    }
    return nil
}
```

執行結果



- 但這隻程式無法確切知道錯誤發生的位置，這時你可以這樣寫:

```go
func a(i int) error {
    if i < 10 {
        fmt.Println("錯誤發生在 a()")
        return errors.New("incorrect value")
    }
    return nil
}

func b(i int) error {
    if i >= 10 {
        fmt.Println("錯誤發生在 b()")
        return errors.New("incorrect value")
    }
    return nil
}
```

重新執行的結果：

```
錯誤發生在 b()
incorrect value
exit status 1
```

9-3-2 用log套件輸出日誌

使用log.Println()

- 與其使用fmt.Println()或類似的功能輸出追蹤訊息，go語言的log套件能讓我們用更豐富的細節紀錄程式執行資訊：

- 請參考以下範例：

```go
package main

import (
    "errors"
    "log"
)

func main() {
    log.Println("Start of our app")
    err := errors.New("application aborted!")
    if err != nil {
        log.Println(err)
    }
    log.Println("End of out app")
}
```

執行結果：

```
2022/08/05 14:23:20 Start of our app
2022/08/05 14:23:20 application aborted!
2022/08/05 14:23:20 End of out app
```

- 如上所見，log.Println()用起來和fmt.Println()一樣，但還會加上訊息的時間戳記，這對於日後檢視日誌，釐清臭蟲發生時間跟順序就非常有用

- 出於同理，log套件的Print()與Printf()的使用方式就跟fmt套件的同名函式是一樣的

- 你甚至可以自訂log套件的日誌訊息格式，辦法是使用SetFlags()：

```go
func main() {
    log.SetFlags(log.Ldate | log.Lmicroseconds | log.Llongfile)
    log.Println("Start of our app")
    err := errors.New("application aborted!")
    if err != nil {
        log.Println(err)
    }
    log.Println("End of out app")
}
```

執行結果：

```
2022/08/05 14:30:29.821628 d:/git/Golang/ch9/9-3-2.go:10: Start of our app
2022/08/05 14:30:29.840941 d:/git/Golang/ch9/9-3-2.go:13: application aborted!
2022/08/05 14:30:29.841568 d:/git/Golang/ch9/9-3-2.go:15: End of out app
```

- 以上程式碼中，log.SetFlags()以聯集算符|串聯了三個旗標，這些旗標都是log套件提供的常數

- 若想了解還有哪些旗標，可以參閱官方文件：https://pkg.go.dev/log#pkg-constants

使用log.Fatal()和log.Panic紀錄嚴重錯誤

- Fatal() / Fatalln() / Fatalf() 方法的作用與log或fmt的Print() / Println() / Printf() 相同 , 唯一差別在於Fatal()與其他姊妹函式在輸出訊息後 , 接著會呼叫os.Exit(1)來中止程式

- 除此之外 , log還有Panic() / Panicln() / Panicf(), 用法和Fatal系列相同 , 差別在他們會引發panic , 如同第六章所說 , panic可以用recover函式救回來 , 但os.Exit()就不行了

- 換言之，當有重大錯誤發生時，你可以在輸出日誌追蹤資訊的同時決定是否要終止程式，而且該不該給使用者機會挽救

- 例如，若錯誤可能會令應用程式的資料受損，或發生難以預期的行為，那最好在事態嚴重前就讓程式當掉

- 若程式結束時需要做一些安全性操作，例如透果defer延遲執行的函式來關閉檔案或資料庫，那麼使用log.Panic()會是比較好的選擇

- 下面來修改前面的範例，用log.Fatal()讓程式在遭遇錯誤時當掉：

```go
package main

import (
    "errors"
    "log"
)

func main() {
    log.SetFlags(log.Ldate | log.Lmicroseconds | log.Llongfile)
    log.Println("Start of our app")
    err := errors.New("application aborted!")
    if err != nil {
        log.Fatal(err)        You, 1 秒前 • Uncommitted changes
    }
    log.Println("End of out app")
}
```

執行結果：

```
2022/08/05 15:31:44.225974 d:/git/Golang/myproject/test.go:10: Start of our app
2022/08/05 15:31:44.273934 d:/git/Golang/myproject/test.go:13: application aborted!
exit status 1
```

9-3-3 建立自訂logger物件

- 到目前為止，本章使用的log印出日誌時，事實上都是透過該套件提供的標準logger(standard logger)

- 你也可以依需要建立自己的logger, 以便針對不同的環境輸出訊息：

```
logger  :=  log.New(io.Writer介面，  前綴詞，  旗標)
```

- 標準logger的第一個參數會使用os.Stdout, 這個符合io.Writer介面的物件其實就是將訊息印出到主控台

- 這當然也可以換成其他物件; 例如 , 地12章要討論的os.File結構就是另一個符合io.Writer介面的東西 , 這是你能將日誌訊息寫到檔案

- 前綴詞是個字串 , 會加在log訊息的最前面 , 除非你用log.Lmsgprefix旗標讓他挪到使用者的訊息之前 (旗標參數則和前面的SetFlags()設定一樣 , 能用來決定logger訊息的格式)

- 來看下面範例 , 我們將log套件的標準logger換成自訂的logger, 不果使用方式依然相同 :

```go
package main

import (
    "errors"
    "log"
    "os"
)

func main() {
    logger := log.New(os.Stdout, "log ", log.Ldate|log.Lmicroseconds|log.Llongfile)
    logger.Println("Srart of out app")
    err := errors.New("application aborted")
    if err != nil {
        logger.Fatal(err)
    }
    logger.Println("End of out app")
}
```

執行結果：

```
log 2022/08/05 15:45:03.035530 d:/git/Golang/ch9/9-3-3.go:11: Srart of out app
log 2022/08/05 15:45:03.064607 d:/git/Golang/ch9/9-3-3.go:14: application aborted
exit status 1
```

9-4 撰寫單元測試

- 最後我們來看如何替go語言撰寫簡單的單元測試(unit test), 並使用go test工具來替我們測試函式與套件

- 下面我們沿用第8章所建立的shape套件，來測試它對於不同形狀傳回的名稱及面積是否正確

- 在第8章中，shape套件唯一匯出的函式只會直接印出訊息到主控台，為了能夠示範如何套用單元測試，下面稍微修改了程式碼

```go
package shape

type Shape interface {
    area() float64
    name() string
}

type Triangle struct {
    Base   float64
    Height float64
}

type Rectangle struct {
    Length float64
    Width  float64
}

type Square struct {
    Side float64
}

func GetName(shape Shape) string { //修改過的新函式，傳回shape介面的名稱
    return shape.name()
}

func GetArea(shape Shape) float64 { //修改過的新函式，傳回shape介面的面積
    return shape.area()
}

```

```go
func (t Triangle) area() float64 {
    return (t.Base * t.Height) / 2
}

func (t Triangle) name() string {
    return "三角形"
}

func (r Rectangle) area() float64 {
    return r.Length * r.Width
}

func (r Rectangle) name() string {
    return "長方形"
}

func (s Square) area() float64 {
    return s.Side * s.Side
}

func (s Square) name() string {
    return "正方形"
}
```

- 這個範例也附有修改過的main.go, 並建立go.mod來提供模組路徑 , 這裡就不多贅述

撰寫測試檔

- Go語言測試檔的名稱不重要，但結尾必加上”_test”, 例如 shape_test.go

- 而在這個檔案中，你必須宣告一個測試用函式：

```
func  Test名稱(t  *testing.T)
```

- 同樣, 函式名稱並不重要，但必須以Test開頭，例如TestGetName
- 此函數會接收一個型別為testing.T的指標變數t(來自testin套件)

- 現在, 於shape子目路底下建立測試檔shape_test.go, 並撰寫兩個測試函式：

```go
package shape

import "testing"

run test | debug test
func TestGetName(t *testing.T) {
    triangle := Triangle{Base: 15.5, Height: 20.1}
    rectangle := Rectangle{Length: 20, Width: 10}
    square := Square{Side: 10}

    if name := GetName(triangle); name != "三角形" {
        t.Errorf("%T 形狀錯誤: %v", triangle, name) //傳回值錯誤時回報測試錯誤
    }
    if name := GetName(rectangle); name != "長方形" {
        t.Errorf("%T 形狀錯誤: %v", rectangle, name)
    }
    if name := GetName(square); name != "正方形" {
        t.Errorf("%T 形狀錯誤: %v", square, name)
    }
}

```

```
    run test | debug test
21  func TestGetArea(t *testing.T) {
22      triangle := Triangle{Base: 15.5, Height: 20.1}
23      rectangle := Rectangle{Length: 20, Width: 10}
24      square := Square{Side: 10}
25
26      if value := GetArea(triangle); value != 155.775 {
27          t.Errorf("%T 面積錯誤: %v", triangle, value)
28      }
29      if value := GetArea(rectangle); value != 200 {
30          t.Errorf("%T 面積錯誤: %v", rectangle, value)
31      }
32      if value := GetArea(square); value != 100 {
33          t.Errorf("%T 面積錯誤: %v", square, value)
34      }
35  }
```

- 注意到上面將測試用的結構寫在個別測試函式中，而不是宣告為shape的套件層級變數，以免影響到shape套件本身

- 測試函式會使用這些結構來測試shape公開函式的回傳值，看看結果是否與已知的正確結果相符

- 測試檔寫好後，就能用go test來跑測試，這裡我們也加上旗標 -v來印出更詳細的測試過程

- go test會自動尋找目錄中的go語言測試檔執行之，若只想執行特定測試檔，可用go test 檔名 的寫法：

```
PS D:\git\Golang\ch9\9-4\shape> go test -v
=== RUN   TestGetName
--- PASS: TestGetName (0.00s)
=== RUN   TestGetArea
--- PASS: TestGetArea (0.00s)
PASS
ok      Example09.04/shape      0.827s
```

- 現在我們來模擬一個程式在修改後意外產生bug的情形：開發人員不小心將shape套件Rectangle結構的main()傳回名稱改為矩形，而Triangle結構的area()方法則忘記將底乘高後除以2，在這種前提底下再次執行go test會看到以下結果：

```
PS D:\git\Golang\ch9\9-4\shape> go test -v
=== RUN   TestGetName
    shape_test.go:14: shape.Rectangle 形狀錯誤: 矩形
--- FAIL: TestGetName (0.00s)
=== RUN   TestGetArea
    shape_test.go:27: shape.Triangle 面積錯誤: 311.55
--- FAIL: TestGetArea (0.00s)
FAIL
exit status 1
FAIL    Example09.04/shape      0.807s
```

本章結束