

14 Testing

Go 有自帶一個 Unit Test 的工具包。程式寫作時，可以自動做 unit test。使用上的慣例：在當下的目錄下，為每一個程式檔案，再新增一個 xxx_test.go 的檔案，裏面撰寫 unit test 程式。

VSCode Go Plugins 設定：

```
{
  "terminal.integrated.shell.osx": "/bin/zsh",
  "go.coverOnSave": true,
  "go.coverageDecorator": "gutter",
  "go.testFlags": ["-v"]
}
```

目錄與檔案

```
.
├── util.go
└── util_test.go
```

測試 function 命名是以 **Test** 開頭，通常會針對要測試的 function 來命名，比如：有一個 Sum 的 function, 測試 Sum 的 function 則命名為 TestSum。

util.go

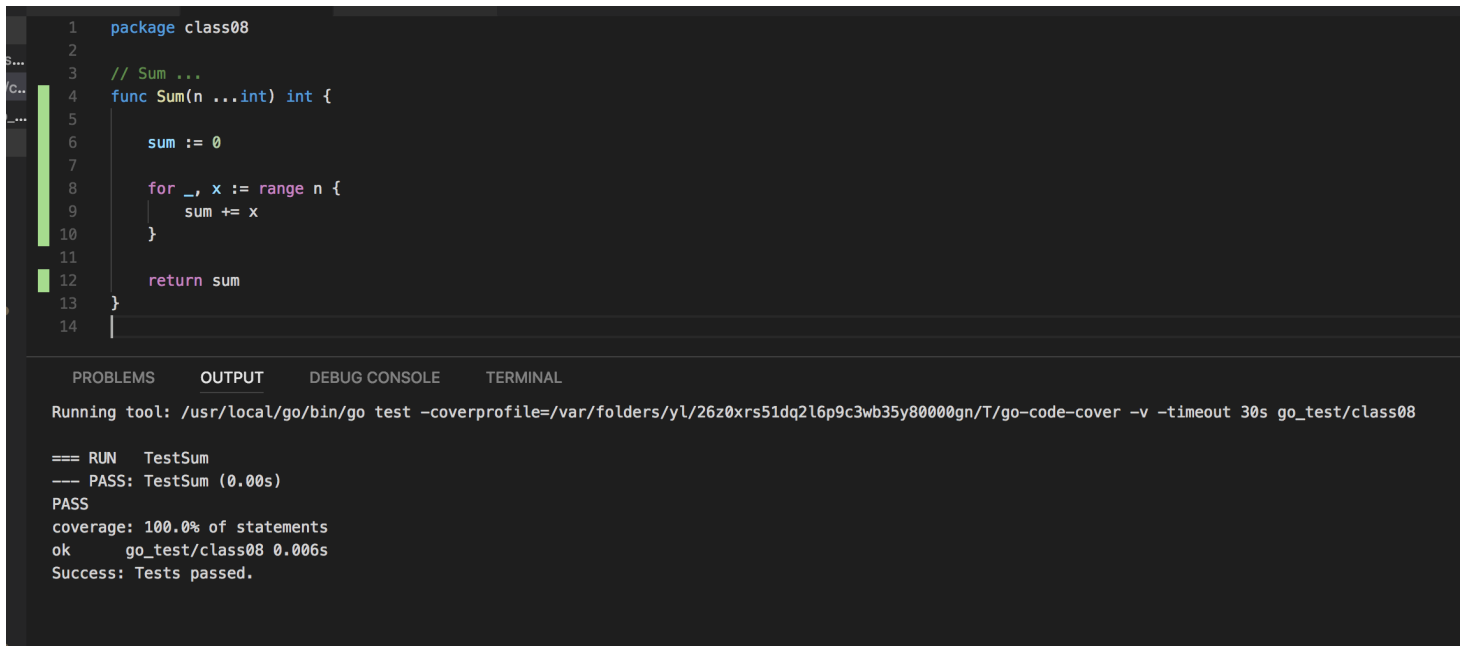
```
1 package util
2
3 import "fmt"
4
5 func init() {
6     fmt.Println("util init")
7 }
8
9 const (
10     defaultSum = 0 // package util_test 無法存取
11 )
12
13 // Sum ...
14 func Sum(x ...int) int {
15     s := 0
16
17     for _, v := range x {
18         s += v
19     }
20
21     return s
22 }
```

util_test.go

```
1 package util_test
2
3 import (
4     "fmt"
5     "os"
6     "testing"
7
8     . "util"
9 )
10
11 func TestSum(t *testing.T) {
12
13     x := Sum(1, 2, 3, 4, 5)
14
15     if x != 15 {
16         t.Fatal("sum error")
17     }
18 }
19
20 func TestMain(m *testing.M) {
21     // initialize test resource
22
23     exitCode := m.Run()
24
25     // destroy test resource
26
27     os.Exit(exitCode)
28 }
29
30 func BenchmarkSum(b *testing.B) {
31     for i := 0; i < b.N; i++ {
32         Sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
33     }
34 }
35
36 func ExampleSum() {
37     fmt.Println("hello world")
38
39     fmt.Println(Sum(1, 2, 3))
40     // Output:
41     // hello world
42     // 6
43 }
44
45 func ExampleHello() {
46     fmt.Println("hello world")
47
48     fmt.Println(Sum(1, 2, 3))
49     // Unordered output:
50     // 6
```

```
51 | // hello world
52 | }
```

如果 VSCode 有設定正確的話，在每次修改 util.go 存檔後，會自動執行 unit test，並回報覆蓋度。如下圖：



```
1 package class08
2
3 // Sum ...
4 func Sum(n ...int) int {
5
6     sum := 0
7
8     for _, x := range n {
9         sum += x
10    }
11
12    return sum
13 }
14
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Running tool: /usr/local/go/bin/go test -coverprofile=/var/folders/yl/26z0xrs51dq2l6p9c3wb35y80000gn/T/go-code-cover -v -timeout 30s go_test/class08

```
=== RUN    TestSum
--- PASS: TestSum (0.00s)
PASS
coverage: 100.0% of statements
ok      go_test/class08 0.006s
Success: Tests passed.
```

或者到專案的目錄下，執行 `go test -v`，輸出的結果如下：

```
util init
=== RUN    TestSum
--- PASS: TestSum (0.00s)
=== RUN    ExampleSum
--- PASS: ExampleSum (0.00s)
=== RUN    ExampleHello
--- PASS: ExampleHello (0.00s)
PASS
ok      util      0.005s
```

testing.T

`testing.T` 是做 unit test 會帶入的參數，它的功能很多 (可參考[官方說明](#))，以下列出常用的 function。

1. Log, Logf: 輸出訊息
2. Fail: 標註目前測試，發生錯誤，但繼續執行
3. FailNow: 標註目前測試，發生錯誤，中斷執行
4. Error, Errorf: Log + Fail
5. Fatal, Fatalf: Log + FailNow

通常會用到的是 `Log` , `Logf` , `Error` , `Errorf` , `Fatal` , `Fatalf`

testing.M

很多情況下，unit test 會需要先產生測試資料，在完成後，刪除測試資料。此時，撰寫 unit test 就好像在寫一個完整的執行程式，此時就會用到 `testing.M` .

```
1 func TestSum(t *testing.T) {
2
3     x := Sum(1, 2, 3, 4, 5)
4
5     if x != 15 {
6         t.Fatal("sum error")
7     }
8 }
9
10 func TestMain(m *testing.M) {
11     // initialize test resource
12
13     exitCode := m.Run()
14
15     // destroy test resource
16
17     os.Exit(exitCode)
18 }
```

Benchmark

Go Unit Test 套件，也可以做 benchmark 測試，程式碼撰寫在 `xxx_test.go` 中，function 命名與 `Test` 類似，以 **Benchmark** 開頭。

```
1 func BenchmarkSum(b *testing.B) {
2     for i := 0; i < b.N; i++ {
3         Sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
4     }
5 }
```

VS Code 預設不會執行 benchmark，因此可以在 console 下，切換到專案目錄，執行 `go test -bench=."`。可以得到以下的結果：

```
goos: darwin
goarch: amd64
pkg: go_test/class10
BenchmarkSum-4
2000000000          8.01 ns/op
PASS
ok      go_test/class10 2.421s
```

以中 2000000000 8.01 ns/op 是指本次 benchmark 執行 **2000000000** 次數，**8.01 ns/op** 每次花費 **8.01 ns**。

1 ns/op** 每次花費 **8.01 ns**。

Example

Example 開頭的 function 也可用來測試程式，主要是比對輸出是否正確。在程式碼中，需加入一段**註解**來說明該程式正確的輸出結果為何？

- // Output：比對輸出結果，且順序都要一致。
- // Unordered Output：比對輸出結果，但順序可以不同。

```
1 func ExampleSum() {
2     fmt.Println("hello world")
3
4     fmt.Println(Sum(1, 2, 3))
5     // Output:
6     // hello world
7     // 6
8 }
9
10 func ExampleHello() {
11     fmt.Println("hello world")
12
13     fmt.Println(Sum(1, 2, 3))
14     // Unordered output:
15     // 6
16     // hello world
17 }
```

Package 命名

在上例中，util 目錄下，有兩個 packages: util 及 util_test。Golang 在同一個目錄下，只能有一個 package (util)及對應的測試 package(util_test)。

Test Code Package Comparison

- Black-box Testing: Use package `myfunc_test`, which will ensure you're only using the [exported identifiers](#).
- White-box Testing: Use package `myfunc` so that you have access to the non-exported identifiers. Good for unit tests that require access to non-exported variables, functions, and methods.

Comparison of Strategies Listed in Question

- Strategy 1: The file `myfunc_test.go` uses package `myfunc` — In this case the test code in `myfunc_test.go` will be in the same package as the code being tested in `myfunc.go`, which is `myfunc` in this example.
- Strategy 2: The file `myfunc_test.go` uses package `myfunc_test` — In this case the test code in `myfunc_test.go` "will be compiled as a separate package, and then linked and run with the main test binary." [Source: Lines 58–59 in the `test.go` source code]
- Strategy 3: The file `myfunc_test.go` uses package `myfunc_test` but imports `myfunc` using the dot notation — This is a variant of Strategy 2, but uses the dot notation to import `myfunc`.

from [Proper package naming for testing with the Go language](#)

- Black-box Testing: 只管 Input/Output 測試，建議 package 命名用 `xxx_test`。
- White-Box Testing: 測試程式內部邏輯，建議放在相同的 package。