Introduction to Programming in Go

12. Testing in Go

Module Topics

1. Testing in Go
2. Unit Testing in Go
3. Benchmarking in Go
4. Profiling in Go

Testing in Go

Go Testing

1. The go test tool performs three kinds of testing:
2. i) Unit testing using test functions.
3. ii) Benchmarking using benchmark functions.
4. iii) Examples, which we will not do in the course.
5. The go test tool works like go build does but does an alternative kind of build that is test oriented.
6. This is an attempt to reduce dependencies on third party tools.

Unit Testing in Go

Go Testing

1. For each file that contains functions we want to test we create a corresponding test file with addition of "_test" at the end of the file name.

2. For example, the test file for "counter.go" is "counter_test.go"

3. Go build/install tools ignore the test files, Go test does not.

4. The test functions must all be of the form

   ```
   func Testxxx(t *testing.T)
   ```

5. The xxx can be whatever you want.

6. The testing.T struct accesses the test framework and logging structs to record the test results.

7. The "testing" package must be imported.

A Function to Test

```
// Example 12-01 Buggy Program to Test
...
func count(s string) (vowels, cons int) {
   for _, letter := range s {
      switch letter {
      case 'a', 'e', 'i', 'o', 'u':
         vowels++
      default:
         cons++
      }
   }
   return
}
func main() {
   input := "This is a test"
   v, c := count(input)
   fmt.Printf("vowels=%d, consonants=%d\n", v, c)
}
```

```
[Module12]$ go run ex12-01.go
vowels=4, consonants=10
```

The Test Functions

```
// Example 12-01 Unit Tests
import "testing"

func TestOne(t *testing.T) {
   v, c := count("a test case")
   if c != 5 {
      t.Error("Test One: Expected 5 cons, got ", c)
   }
   if v != 4 {
      t.Error("Test One: Expected 4 vowels, got ", v)
   }
}
func TestTwo(t *testing.T) {
   v, c := count("And more stuff")
   if c != 8 {
      t.Error("Test Two: Expected 8 cons, got ", c)
   }
   if v != 4 {
      t.Error("Test Two: Expected 4 vowels, got ", v)
   }}
```

Running the Tests

```
[Module12-01]$ go test
--- FAIL: TestOne (0.00s)
     ex12-01_test.go:9: Test One: Expected 5 cons, got  7
--- FAIL: TestTwo (0.00s)
     ex12-01_test.go:18: Test Two: Expected 8 cons, got  11
     ex12-01_test.go:21: Test Two: Expected 4 vowels, got  3
FAIL
exit status 1
FAIL    examples/Module12/ex12-01 0.002s
```

A Fixed Function to Test

```
// Example 12-02 Buggy Program to Test Fixed
...
func count(s string) (vowels, cons int) {
   for _, letter := range s {
      switch letter {
      case 'a', 'e', 'i', 'o', 'u':
         vowels++
      case 'A', 'E', 'I', 'O', 'U':
         vowels++
      case ' ':
         break
      default:
         cons++
      }
   }
   return
}
```

```
[Module12-02]$ go test
PASS
ok    examples/Module12/ex12-02 0.002s
```

Coverage

1. Coverage is a measure of how many statements were executed by the tests.

2. Coverage as measured by the tool is not a precise metric.

3. Low coverage values often mean that you are not testing the code you think you are testing.

4. Coverage should always be considered an estimate.

5. The next slide shows coverage measures for the examples presented so far

Coverage Measures

```
Module12-01]$ go test -cover
--- FAIL: TestOne (0.00s)
     ex12-01_test.go:9: Test One: Expected 5 cons, got  7
--- FAIL: TestTwo (0.00s)
     ex12-01_test.go:18: Test Two: Expected 8 cons, got  11
     ex12-01_test.go:21: Test Two: Expected 4 vowels, got  3
FAIL
coverage: 62.5% of statements
exit status 1
FAIL    examples/Module12/ex12-01 0.002s
```

```
[Module12-02]$ go test -cover
PASS
coverage: 70.0% of statements
ok    examples/Module12/ex12-02 0.002s
```

Benchmarking

Benchmarking

1. Benchmarking is measuring the performance of a function under a fixed workload.

2. Benchmarking functions have the forms:

   ```
   func BenchmarkXxx(b *testing.B)
   ```

   and they are in the same file as the unit tests.

3. testing.B is a struct that is used to track benchmarking information.

4. By default no benchmarking functions are run by “go test,” they have to be specified on the command line with a regular expression that matches the names of the benchmark functions to be run.

5. Benchmarks are run in a for loop where the benchmarking program runs the function to be measured b.N times with increasingly large values of N until the benchmark values stabilize.

Function to Benchmark

```
// Example 12-03 Program to be Benchmarked
...

func Fibonacci(n int) int {
   if n < 2 {
      return n
   }
   return Fibonacci(n-1) + Fibonacci(n-2)
}

func main() {
   fmt.Println(Fibonacci(30))
}
```

```
[Module12-03]$ go run ex12-03.go
832040
```

The Benchmark Function

```
// Example 12-03 Benchmarking

import "testing"

func BenchmarkFib20(b *testing.B) {
   for n := 0; n < b.N; n++ {
      Fibonacci(20)
   }
}
```

```
[Module12-03]$ go test -bench=.
testing: warning: no tests to run
PASS
BenchmarkFib20-12          30000              40771 ns/op
ok      _examples/Module12/ex12-03 1.728s
```

Benchmarking

1. The middle number is the final value of b.N, the test ran for 30,000 loops with an average time of of 40771 nanoseconds per loop.

2. A complete breakdown of benchmarking in general is beyond this introduction.

Profiling

Profiling

1. Measures the performance of critical code.

2. Samples a number of events during the program execution and then extrapolates to produce a statistical summary called a profile.

3. Three different kinds of profiling are supported by Go testing.

   CPU profiling - functions whose execution requires the most CPU time.

   Heap profiling - statements responsible for allocating the most memory.

   Blocking profiling - operations responsible for blocking goroutines the longest.

4. A profile is collected in a log file which can then be examined using the pprof Go tool.

5. No more than one profile at a time should be done or the sampling will be skewed.

Profiles

```
[Module12-04]$ go test -bench=. -cpuprofile=cout.log
testing: warning: no tests to run
PASS
BenchmarkFib20-12          30000             40828 ns/op
ok      _/examples/Module12/ex12-04 1.671s
```

```
[Module12-04]$ go tool pprof -text ex12-04.test cout.log
1.67s of 1.67s total (  100%)
      flat  flat%   sum%        cum   cum%
     1.65s 98.80% 98.80%      1.65s 98.80%  Module12/ex12-04.Fibonacci
     0.01s   0.6% 99.40%      0.01s   0.6%  runtime.(*mspan).sweep
     0.01s   0.6%   100%      0.01s   0.6%  runtime.usleep
         0     0%   100%      1.65s 98.80%  Module12/ex12-04.BenchmarkFib20
         0     0%   100%      0.01s   0.6%  runtime.(*gcWork).get
         0     0%   100%      0.01s   0.6%  runtime.GC
         0     0%   100%      0.01s   0.6%  runtime.findrunnable
         0     0%   100%      0.01s   0.6%  runtime.gcDrain
         0     0%   100%      0.01s   0.6%  runtime.gcMarkTermination
         0     0%   100%      0.01s   0.6%  runtime.gcMarkTermination.func2
         0     0%   100%      0.01s   0.6%  runtime.gcStart
         0     0%   100%      0.01s   0.6%  runtime.gcSweep
         0     0%   100%      0.01s   0.6%  runtime.gchelper
         0     0%   100%      0.01s   0.6%  runtime.getfull
         0     0%   100%      1.66s 99.40%  runtime.goexit
         0     0%   100%      0.01s   0.6%  runtime.mstart
         0     0%   100%      0.01s   0.6%   runtime.mstart1 ...
```

1. The way of profiling demonstrated so far is cumbersome.

2. Dave Cheney has written a nice interface to wrap around profiling code.

3. The utility is available from a github repository.

4. Its use is demonstrated in the example.

Function to Profile

```
// Profiling non-test functions
...
import (
   "fmt"
   "github.com/pkg/profile"
)
func Fibonacci(n int) int {
   if n < 2 {
      return n
   }
   return Fibonacci(n-1) + Fibonacci(n-2)
}

func main() {
   defer profile.Start().Stop()
   fmt.Println(Fibonacci(30))
}
```

Output

```
[Module12-04]$ go run ex12-04.go
2016/09/22 07:34:59 profile: cpu profiling enabled, /tmp/profile457278947/
cpu.pprof
Fibonacci num for 30 is 1346269
```

```
[Module12-04]$  go tool pprof -text ex12-04  /tmp/profile457278947/cpu.pprof
10ms of 10ms total (  100%)
      flat  flat%   sum%        cum   cum%
      10ms   100%   100%       10ms   100%  main.fib
         0     0%   100%       10ms   100%  main.main
         0     0%   100%       10ms   100%  runtime.goexit
         0     0%   100%       10ms   100%  runtime.main
```

Lab 12: Testing