Module Twelve

Testing and Benchmarking

More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded - indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest.

Boris Beizer

12.1 Testing and Benchmarking in Go

In the first module, we mentioned that Go was intended to be a integrated software production environment which had as one of its design goals the reduction of third party tool dependencies used in the process of building and delivering software.

One of the current best practices in modern software engineering is the use of test driven development and other “test as you go” strategies. The effectiveness of these approaches has led to a wide proliferation of tools like Junit, TestNG, nUnit, RSpec and whole raft of others. Of course these are exactly sort of third party tool dependencies that the developers of Go wanted to eliminate, but not at the cost of reducing the use of the good test supported development practices.

The go tool supports a limited unit test and benchmarking environment implemented as a lightweight automated testing framework. Of course like with any other tool, the Go test utility does not “do testing,” is merely automates your testing activities – if you write bad tests or use testing techniques poorly, automating those does not produce any significant benefit. The actual concepts of how to write unit tests, how to plan a testing strategy and all of the other testing related content is outside the scope of this course. All we will be looking at in this course is how to use the tools.

12.1.1 Tool Capabilities

The go test tool has the following capabilities:

1. *Test functions:* these are special functions that reside in files that are named <filename>_test.go. These correspond to the usual sort of functional unit tests that we would run to see how correct our code is. Functions that implement tests have a specific naming convention and signature we will look at a bit later.
2. *Benchmarks:* These are functions that have names that start with ‘Benchmark’ and which measure performance – more or less our non-functional tests.
3. *Examples;* These are functions that start with ‘Example’ and provides machine checked documentation. Since these are part of the documentation function, we will not be looking at them in this course.

12.1.2 Test Independence

One of the principles of automated testing is that the testing environment should not pollute the production environment with any testing artifacts. Normally the test functions are ignored in the build process, but when "go test" is invoked, an alternative build is run to run the tests, report the results and then remove any traces of the test build.

```
// Example 12-01 Trivial Function To Test
package main

import "fmt"

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
```

12.1.3 Writing a Basic Test

The function we are going to test is a very badly written and trivial function that counts the number of vowels and consonants in a string. And does it wrong.

Creating the test cases and how we set them up conceptually is outside the scope of this course. However we will assume that there is a correctly constructed set of unit tests that have been developed so that the focus of this module can just be on the automation of the unit testing process.

We will assume that we have two unit tests, one which is the string “a test case” and which should produce 5 consonants and 4 vowels, and the test case consisting of the string “And more stuff” which should produce 8 consonants and 4 vowels.

12.2 Test Automation Framework

There are only a few steps needed to set up the unit tests.

12.2.1 Creating the Test File

For each source file in our directory that contains functions we want to unit test, say a file named “counter.go” for example, we create a corresponding test file named the same but with the suffix “_test” added to the name. For our counter.go file, the file containing the tests would be in counter_test.go.

The test file is in the same package and same directory as the file that it contains test for. There are two different build processes in Go, the normal production build process that ignores all files named like “*_test.go” and a test build process which adds these files and then creates its own main() function entry point to run the tests instead of running the application.

The production build process is called by “go build” and the test build process is called by “go test”.

Each one of the test files must import the “testing” package in order to be able to hook into the automated Go testing framework

12.2.2 Creating the Test Functions

Go relies a lot on the names of things encoding information about what kinds of things they are. In the test files there are test functions that will be automatically called when the test build is executed. These test functions must all be of the form

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

where the xxx can be any character string you want to make the function name meaningful.

The structure T is a data object that is passed to each test function to maintain test state and the test logs – it is data Go needs to manage the testing process.

It is up to us to write the test logic which usually calls the function under test, gives it the test data and then examines the results to ensure that they are correct. The Go test framework has no idea whether a test has passed or failed unless we tell it. If a test fails, we basically enter that into the test logs maintained in that *testing.T structure, usually by calling the Error() method. The Error() method should be provided with a description of what happened so that when we are reviewing the results, we have enough information to go back and correct the errors that we made.

```
// Example 12-01b Testing the function count

package main

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)
  }
}
```

Example 12-01b illustrates the implementation of the test functions for the two test cases. Generally we have one test function for each test case.

We can now run the tests by locating to the directory and executing ‘go test’. Doing so produces the following output for out two test cases. Notice that the main() function in our original code is ignored for the test build.

```
[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
```

Now corrections are made in the original source code and the test are rerun and the now all the tests pass.

```
// Example 12-02 Corrected Function

package main

import "fmt"

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
```

12.2.3 Test Coverage

Coverage is a term that refers to how well a set of tests cover or exercise our code. Go does provide a basic coverage tool that tell you what percentage of your executable statements actually were exercised during the test execution.

Statement coverage is not a very sophisticated kind of quality measure so it should not be taken as a metric for how good your tests are. However the tool does give you a rough idea of whether or not you are actually running the test you think you are running. For example, if you think you have a full set of test cases and you get 40% coverage, then it suggests that some of the tests are not running or there are some serious problems with the code so resulting in 60% of the code not executing when you may be thinking it is.

One of the more interesting uses of the cover option is to explore code that may be non-essential. If I have a set of unit tests that provide 100% functional coverage (ie. they account for all possible types of input) and I have a cover value of 50%, then I have to wonder what that other 50% of my code is actually doing.

These sorts of metrics are good for getting a general idea but they are not intended to be at the level of a dedicated testing tool. For example, if we compare the coverage of the two test we just ran, we seem to be adding more to the coverage, but we also added more statements so there are a number of factors that will affect the coverage value.

```
[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
```

12.2 Benchmarking

Generally benchmarking is defined as measuring the performance of a function under a fixed workload.

In the Go testing framework, benchmarking functions look a lot like the test functions except that they have the form

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

where B is a struct that is used to track benchmarking information.

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. Using the pattern /./ will call all the benchmarking functions to be run.

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 it seems that the benchmark values have stabilized. How exactly this works is outside the scope for this course.

In example 12-03 we have a standard Fibonacci recursive function which has the interesting property that the larger the value we provide, the length of time to compute the result increases exponentially.

```go
// Example 12-03 Benchmarking a function

package main

import "fmt"

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 Benchmarking function below is now run. Notice the use of the b.N as the loop limit so that the metering program can look for a stable extrapolation of the function performance.

```
// Example 12-03 Benchmarking

package main

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
```

In the result, the statement BenchmarkFib20-12, the -12 part refers to GOMAXPROCS which is value that controls the number of operating system threads allocated to goroutines in this program. You may see a different number on your machine when you run this.

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

12.3 Profiling

Another aspect of testing related to benchmarking is profiling which is used to measure the performance of critical code by sampling a number of events during the program execution and then extrapolating that sample to produce a statistical summary called a profile. There are several different kinds of profiling that are supported by the test package in Go.

1. A CPU profile identifies the functions whose execution requires the most CPU time.
2. A heap profile identifies the statements responsible for allocating the most memory.
3. A blocking profile identifies the operations responsible for blocking goroutines the longest.

These profiles can be generated by the use of the profiling flags. Example 12-04 demonstrates the use of the CPU profile using the same code that from example 12-03

```
[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
```

The profile is collected in the file cout.log which can then be examined using the pprof Go tool. This is illustrated on the next page.

Using the pdf output provides a more graphical format if the output of the pprof command in piped into a file to be opened in a pdf viewer.

One of the cautions of using the profiling tools is that one should not try to profile more than one of the values at a time since if more than one profile is selected, the sampling for each profile may interfere with the sampling of the others.

```
[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
         0     0%   100%      0.01s   0.6%  runtime.schedule
         0     0%   100%      0.01s   0.6%  runtime.stopm
         0     0%   100%      0.01s   0.6%  runtime.sweepone
         0     0%   100%      0.01s   0.6%  runtime.systemstack
         0     0%   100%      1.66s 99.40%  testing.(*B).launch
         0     0%   100%      1.66s 99.40%  testing.(*B).runN
```

12.3.1 Profiling Non-Test Functions

Dave Cheney has written a nice interface to allow the use of profiling in non test environments in a simple way. The profiling tool he has created is at github.com/pkg/profile. Although we have not used the go get command yet, the example has set up a go workspace with the tool installed. To use the tool, the code is “wrapped” in a profiling function that produces the same sort of output as the seen in the last section.

```
// Profiling non-test functions

package main

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()
  n := 30
  fmt.Println("Fibonacci num for", n, "is", Fibonacci(n))
}
```

```
[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
2016/09/22 07:34:59 profile: cpu profiling disabled, /tmp/
profile457278947/cpu.pprof
```

The file produced in this way is analyzed using go tool pprof as before

```
[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
```

The various parameters to the profiling tool can be set using a configuration structure as demonstrated in this example from Cheney.

```
// Profiling non-test functions

package main
import (
  "fmt"
  "github.com/pkg/profile"
)

func main() {
        cfg := profile.Config {
                MemProfile: true,
                NoShutdownHook: true, // do not hook SIGINT
        }
        // p.Stop() must be called before the program exits
        // to ensure profiling information is written to disk.
        p := profile.Start(&cfg)
        ...
}
```