Testing Essentials In Go

Israel Miles Follow
Dec 16 · 8 min read ★

From powerful test flags to code coverage and benchmarking!

From Danielle

Every software engineer needs to know how to test in their language of choice. Even a simple layer of unit tests can greatly improve a program's resilience and ensure it runs as the engineer expects. Go has a fantastic built-in testing support system with comprehensive flag options to help an engineer create tested programs efficiently.

In this article we will detail the following:

- Running basic tests in Go
- Using test flags
- Benchmarking & performance
- Generating code coverage

Let's learn some Go!

A Basic Test Suite

Let's start from scratch. In order to run our tests, we will need to move to our Go source folder. From there we can create a new folder that we will use as a sandbox for our tests. Let's also create two directories `parent` with a sub-directory `child`. Finally, we will add two test files `parent_test.go` and `child_test.go`. Adding "_test" at the end of a Go file signals to the compiler that we want to run unit tests from this source.

```
$ go env GOPATH
/Users/israelmiles/go
$ cd /Users/israelmiles/go/src
$ mkdir testing-go
$ cd testing-go
$ mkdir parent
$ mkdir parent/child
$ touch parent/parent_test.go
$ touch parent/child/child_test.go
```

If you need help setting your `GOPATH`, look here:

How do I SET the GOPATH environment variable on Ubuntu? What file must I edit?

You can use the "export" solution just like what other guys have suggested. I'd like to provide you with another...

stackoverflow.com

We are going to have `parent` and `child` directories to showcase different testing methods. First let's add the following code to our `parent_test.go` file.

parent_test.go

In our `parent` package our first step is to import the `testing` package which gives us access to different testing methods. As you can see, we take a reference to the object `testing.T` for our test function `TestMath()`. This function then runs two different tests against addition and subtraction in go. The structure of `t.Run()` is to include a name (Addition in go) and an anonymous function that accepts that same reference to `testing.T`. If we don't pass some conditions we set, we call `t.Fail()` as an option to showcase the results of our tests.

The method `t.Run()` may be called simultaneously from multiple goroutines, but all such calls must return before the outer test function for the testing object `t` returns.

```
package parent

import "testing"

func TestMath(t *testing.T) {
	t.Run("Addition in Go", func(t *testing.T) {
		if 1+1 != 2 {
			t.Fail()
		}
	})
	t.Run("Subtraction in Go", func(t *testing.T) {
		if 1-1 != 0 {
			t.Fail()
		}
	})
}

```

```go
func TestStrings(t *testing.T) {
	t.Run("Concatenation in Go", func(t *testing.T) {
		if "Hello, "+"World!" != "Hello, World!" {
			t.Fail()
		}
	})
}
```

parent_test.go hosted with ❤ by GitHub view raw

It's important to capitalize the first letter and use camel case for the Go compiler to recognize our tests. Using test functions with individual tests layered within is a basic but effective way to organize our test suites.

go test

If we enter the `parent` directory and run `go test`, the compiler will automatically run any files ending in _test.go.

```
$ pwd
/Users/israelmiles/go/src/testing-go/parent
$ go test
PASS
ok   testing-go/parent 0.220s
```

Cool, our tests passed! But what if we want more information? We can include the `-v` flag for verbose to gain additional insight into how our tests ran.

```
$ go test -v
=== RUN   TestMath
=== RUN   TestMath/Addition_in_Go
=== RUN   TestMath/Subtraction_in_Go
--- PASS: TestMath (0.00s)
    --- PASS: TestMath/Addition_in_Go (0.00s)
    --- PASS: TestMath/Subtraction_in_Go (0.00s)
=== RUN   TestStrings
=== RUN   TestStrings/Concatenation_in_Go
--- PASS: TestStrings (0.00s)
    --- PASS: TestStrings/Concatenation_in_Go (0.00s)
PASS
ok   testing-go/parent 0.085s
```

Now we can see which functions were ran with each individual test layered within. We can also run a recursive test suite if we have a layered directory structure.

If you want to **run a single test**, you can use the `-run` flag.

```
$ pwd
/Users/israelmiles/go/src/testing-go/parent
$ go test -v -run TestMath
=== RUN   TestMath
=== RUN   TestMath/Addition_in_Go
=== RUN   TestMath/Subtraction_in_Go
--- PASS: TestMath (0.00s)
    --- PASS: TestMath/Addition_in_Go (0.00s)
    --- PASS: TestMath/Subtraction_in_Go (0.00s)
PASS
ok   testing-go/parent 0.243s
```

You can also specify the package to search for your tests to run. The structure is then `go test -run <Test expression> <packages to search>`.

If you want to **run a test multiple times**, you can use the `-count` flag. This could be useful if you were performing endurance tests for example.

```
$ go test -count 1 testing-go/parent
ok   testing-go/parent 0.079s
$ go test -count 10 testing-go/parent
ok   testing-go/parent 0.310s
Israels-MacBook-Pro:parent israelmiles$
```

child_test.go

Within our `parent` directory is the `child` directory including `child_test.go`. Here's an example test we could add to the file:

```go
package child

import "testing"

func TestMultiplication(t *testing.T) {
	if 10*2 != 20 {
		t.Error("Failed to multiply correctly")
```

```
                t.Error("failed to multiply correctly")
        }
}
```

child_test.go hosted with ❤ by GitHub view raw

Here we have an even more simple setup where we just check a condition and if it's false we call the `Error()` method from our testing object.

Recursive testing

We can run a **recursive test** with `go test` by adding `/...` at the end of the package to test. For example:

```
$ pwd
/Users/israelmiles/go/src/testing-go
$ go test -v testing-go/parent/...
=== RUN   TestMath
=== RUN   TestMath/Addition_in_Go
=== RUN   TestMath/Subtraction_in_Go
--- PASS: TestMath (0.00s)
    --- PASS: TestMath/Addition_in_Go (0.00s)
    --- PASS: TestMath/Subtraction_in_Go (0.00s)
=== RUN   TestStrings
=== RUN   TestStrings/Concatenation_in_Go
--- PASS: TestStrings (0.00s)
    --- PASS: TestStrings/Concatenation_in_Go (0.00s)
PASS
ok   testing-go/parent (cached)
=== RUN   TestMultiplication
--- PASS: TestMultiplication (0.00s)
PASS
ok   testing-go/parent/child 0.295s
```

Now you know how to:

- Run tests with `go test`
- Run tests multiple times with the `-count` flag
- Create tests with `testing.T.Run()`
- Print test information with the `-v` flag
- Execute a single test with the `-run` flag

- Execute a recursive test suite with by appending `/...` to your package

Benchmarking your tests

Let's beef up our project structure. We will add three files to our project root folder to calculate and test the fibonacci sequence using a loop structure and a recursive format.

fibonacci.go

Two ways to find the fibonacci sequence using slices and recursion.

```go
package main

//FibonacciLoop finds f(n) with slices
func FibonacciLoop(n int) int {
	f := make([]int, n+1, n+2)
    if n < 2 {
        f = f[0:2]
    }
    f[0] = 0
    f[1] = 1
    for i := 2; i <= n; i++ {
        f[i] = f[i-1] + f[i-2]
    }
    return f[n]
}

//FibonacciRecursion finds f(n) with recursion
func FibonacciRecursion(n int) int {
    if n <= 1 {
        return n
    }
    return FibonacciRecursion(n-1) + FibonacciRecursion(n-2)
}
```

fibonacci.go hosted with ❤ by GitHub view raw

fibonacci_test.go

Here we have the first 12 values of the fibonacci sequence to test against. Then we call two tests against the loop and recursive methods of calculating the sequence.

```go
package main

import (
	"testing"
)

func TestFib(t *testing.T) {
	first12FibNums := []int{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89}
	fibNum30 := 832040

	t.Run("Fibonnaci Loop", func(t *testing.T) {
		for i, fibNum := range first12FibNums {
			if FibonacciLoop(i) != fibNum {
				t.Fail()
			}
		}
		if FibonacciLoop(30) != fibNum30 {
			t.Fail()
		}
	})
	t.Run("Fibonnaci Recursion", func(t *testing.T) {
		for i, fibNum := range first12FibNums {
			if FibonacciRecursion(i) != fibNum {
				t.Fail()
			}
		}
		if FibonacciRecursion(30) != fibNum30 {
			t.Fail()
		}
	})
}
```

fibonacci_test.go hosted with ❤ by GitHub view raw

main.go

The driver file `main.go` simple calls the fibonacci methods 10 times each, using `strconv.Itoa()` to convert from an int to a string. You can run this to see the output of each Fibonacci function.

```go
package main

import (
	"fmt"
```

```go
        "strconv"
)

func main() {
    for i := 0; i <= 9; i++ {
        fmt.Print(strconv.Itoa(FibonacciLoop(i)) + " ")
    }
    fmt.Println("")
    for i := 0; i <= 9; i++ {
        fmt.Print(strconv.Itoa(FibonacciRecursion(i)) + " ")
    }
    fmt.Println("")
}
```

main.go hosted with ❤ by GitHub view raw

Running our tests with `go test -v` will thus give:

```
=== RUN   TestFib
=== RUN   TestFib/Fibonnaci_Loop
=== RUN   TestFib/Fibonnaci_Recursion
--- PASS: TestFib (0.00s)
    --- PASS: TestFib/Fibonnaci_Loop (0.00s)
    --- PASS: TestFib/Fibonnaci_Recursion (0.00s)
PASS
ok   testing-go 0.092s
```

Benchmarking

If you want to get a good idea of how efficient your tests are, you can use the `-bench` flag in addition to creating a benchmark method. First, add the below code to `fibonacci_test.go`. To create a benchmark method, simply start the function name with `Benchmark` and take a reference to `testing.B` as a parameter. The value `b.N` is assigned by the compiler and is the number of times we will run `FibonacciLoop`.

```go
func BenchmarkFibonacciLoop1(b *testing.B) {
        for i := 0; i < b.N; i++ {
                FibonacciLoop(1)
        }
}

func BenchmarkFibonacciLoop100(b *testing.B) {
```

```go
8         for i := 0; i < b.N; i++ {
9                 FibonacciLoop(100)
10        }
11  }
```

benchmark.go hosted with ❤ by GitHub view raw

Above, our first benchmark is ran with the `FibonacciLoop()` method for the first sequence value. The second benchmark method on line 7 is for the 100th fibonacci sequence value. If we run a benchmark against the tests, we can see how it becomes progressively more difficult to calculate the fibonacci sequence.

```
$ go test -v -bench Fib
=== RUN   TestFib
=== RUN   TestFib/Fibonnaci_Loop
=== RUN   TestFib/Fibonnaci_Recursion
--- PASS: TestFib (0.01s)
    --- PASS: TestFib/Fibonnaci_Loop (0.00s)
    --- PASS: TestFib/Fibonnaci_Recursion (0.01s)
goos: darwin
goarch: amd64
pkg: testing-go
BenchmarkFibonacciLoop1
BenchmarkFibonacciLoop1-8      43454248         25.2 ns/op
BenchmarkFibonacciLoop100
BenchmarkFibonacciLoop100-8     4285092        277 ns/op
PASS
ok   testing-go 2.694s
```

See how our regular expression `Fib` executes both the `FibonacciLoop` methods? Not just that, but we can see that calculating the 1st fibonacci sequence value takes about 25.2 nanoseconds per call, while the 100th fibonacci sequence value takes about 277 nanoseconds per call!

Furthermore we can see how much memory our functions use by including the `-benchmem` flag.

```
$ go test -v -bench . -benchmem
=== RUN   TestFib
=== RUN   TestFib/Fibonnaci_Loop
=== RUN   TestFib/Fibonnaci_Recursion
```

```
--- PASS: TestFib (0.01s)
    --- PASS: TestFib/Fibonnaci_Loop (0.00s)
    --- PASS: TestFib/Fibonnaci_Recursion (0.01s)
goos: darwin
goarch: amd64
pkg: testing-go
BenchmarkFibonacciLoop1
BenchmarkFibonacciLoop1-8   42072594 24.9 ns/op 32 B/op  1 allocs/op
BenchmarkFibonacciLoop100
BenchmarkFibonacciLoop100-8 4268587  280 ns/op  896 B/op 1 allocs/op
PASS
ok   testing-go 2.780s
```

Now we can see that the 1st sequence value takes 32 Bytes per call, and the 100th value requires 896 Bytes. How much do you think the recursive method takes?

Generating code coverage

Alright, you've learned a lot so far! Let's wind down with one last useful tool for testing in Go. If we want to see how much of our code is covered by test cases, we can use the `-cover` flag.

```
$ go test -v -cover testing-go
=== RUN   TestFib
=== RUN   TestFib/Fibonnaci_Loop
=== RUN   TestFib/Fibonnaci_Recursion
--- PASS: TestFib (0.01s)
    --- PASS: TestFib/Fibonnaci_Loop (0.00s)
    --- PASS: TestFib/Fibonnaci_Recursion (0.01s)
PASS
coverage: 64.7% of statements
ok   testing-go 0.112s
```

So we cover 64.7% of all the code in `fibonacci.go` and `main.go` . If you were to get rid of the print statements in `main.go` , you would increase the coverage up to 73.3%. Further targeted test cases against the fibonacci methods would bump your coverage even more.

Note: If you have a test failure, the `-cover` flag will actually rewrite your source code before it compiles it (don't ask me how). So, *do not* use the `-cover` flag in conjunction with any benchmarking tests.

But what parts of my code are covered?

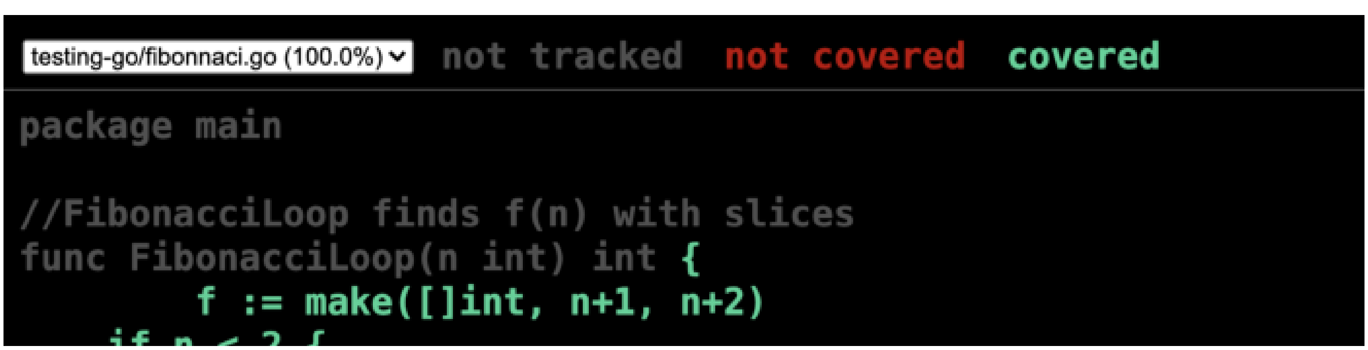
If you want to see which of your code statements are covered by tests, you can include the `-coverprofile` flag. This requires a file to append information to, which we will call `cover.out`. If you were to try to immediately read from `cover.out`, you wouldn't get much useful information.

```
$ go test -coverprofile cover.out
$ cat cover.out
mode: set
testing-go/fibonnaci.go:4.31,6.14 2 1
testing-go/fibonnaci.go:9.5,11.29 3 1
testing-go/fibonnaci.go:14.5,14.16 1 1
testing-go/fibonnaci.go:6.14,8.6 1 1
testing-go/fibonnaci.go:11.29,13.6 1 1
testing-go/fibonnaci.go:18.36,19.15 1 1
testing-go/fibonnaci.go:22.5,22.61 1 1
testing-go/fibonnaci.go:19.15,21.6 1 1
testing-go/main.go:3.13,4.29 1 0
testing-go/main.go:7.5,7.29 1 0
testing-go/main.go:4.29,6.6 1 0
testing-go/main.go:7.29,9.6 1 0
```

In order to see our coverage in a more useful format, use the `go tool cover` command. If you call the `-html` flag, you can assign it to our new output file `cover.out` in order to generate a comprehensive report.

```
$ go tool cover -html=cover.out
```

This will then open a summary of the coverage in your default web browser:



```
    if n < 2 {
        f = f[0:2]
    }
    f[0] = 0
    f[1] = 1
    for i := 2; i <= n; i++ {
        f[i] = f[i-1] + f[i-2]
    }
    return f[n]
}

//FibonacciRecursion finds f(n) with recursion
func FibonacciRecursion(n int) int {
    if n <= 1 {
        return n
    }
    return FibonacciRecursion(n-1) + FibonacciRecursion(n-2)
}
```

As we can see, the fibonacci methods are actually fully covered! But if we look at the main Go file…

```
testing-go/main.go (0.0%)   not tracked   not covered   covered

package main

func main() {
    for i := 0; i <= 9; i++ {
        FibonacciLoop(i)
    }
    for i := 0; i <= 9; i++ {
        FibonacciRecursion(i)
    }
}
```

We see that this is actually the source for our lack of coverage.

I hope you enjoyed this article and learned something new. There is plenty more information to learn when it comes to testing in Go, but this article covers the main basics you need to get up and running. If you saw anything noteworthy, or would like to hear more about a section of the article, I encourage you to leave a comment below! Thanks so much for reading.

Golang Tutorial Programming Testing Technology