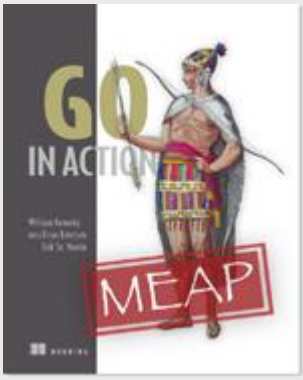

Go in Action: Benchmarking

By William Kennedy with Brian Ketelsen and Erik St. Martin

In this article, excerpted from the book Go in Action, we will look at a set of benchmark functions that reveal the fastest way to convert an integer value to a string.

Benchmarking is a way to test the performance of code. It is useful when you want to test the performance between different solutions to the same problem and see which solution performs better. It can also be useful to identify cpu or memory issues for a particular piece of code that might be critical to the performance of your application. Many developers use benchmarking, for instance, to test different concurrency patterns or to help configure work pools to make sure they are configured properly for the best throughput.

Let's look at a set of benchmark functions that reveal the fastest way to convert an integer value to a string. In the standard library there are three different ways to convert an integer value to a string:

Listing 1: listing28_test.go : lines 01 - 10

```
01    // Sample benchmarks to test which function is better for converting
02    // an integer into a string. First using the fmt.Sprintf function,03 //
         then the strconv.FormatInt function and then strconv.Itoa.

04 package listing28_test

05
06    import (
07    "fmt"
08    "strconv"
09    "testing"
10    )
```

Listing 1 shows the initial code for the `listing28_test.go` benchmarks. As with the unit test files, the file name must end in `_test.go`. The `testing` package must also be imported. Next, let's look at one of the benchmark functions:

Listing 2: listing28_test.go : lines 12 - 22

```
12 // BenchmarkSprintf provides performance numbers for the
13 // fmt.Sprintf function.
14      func BenchmarkSprintf(b *testing.B) {
15      number := 10
16      17      b.ResetTimer()
18
19      for i := 0; i < b.N; i++ {
20      fmt.Sprintf("%d", number)
21      }
22      }
```

On line 14 in listing 2, we see the first benchmark called `BenchmarkSprintf`. Benchmark functions begin with the word `Benchmark` and take as their only parameter a pointer of type `testing.B`. In order for the benchmarking framework to calculate performance, it must run the code over and over again for a period of time. This is where the for loop comes in:

Listing 3: listing28_test.go : lines 19 - 22

```
19      for i := 0; i < b.N; i++ {
20      fmt.Sprintf("%d", number)
21      }
22      }
```

The `for` loop on line 19 in listing 3, shows how to use the `b.N` value. On line 20, we have the call to the `Sprintf` function from the `fmt` package. This is the function we are benchmarking to convert an integer value into a string.

By default, the benchmarking framework will call the benchmark function over and over again for at least one second. Each time the framework calls the benchmark function, it will increase the value of `b.N`. On the very first call, the value of `b.N` will be 1. It is important to place all the code to benchmark inside the loop and to use the `b.N` value. If this is not done, the results can not be trusted.

If we just want to run benchmark functions, we need to use the `-bench` option:

Listing 4: Running the benchmark test.

```
        go test -v -run="none" -bench="BenchmarkSprintf"
```

In our call to `go test`, we specified the `-run` option passing the string `"none"` to make sure no unit tests are run prior to running the specified benchmark function. Both these options take a regular expression to filter the tests to run. Since there is no unit test function that has `none` in its name, it eliminates any unit tests from running. When we issue this command, we get the following output:

```
$ go test -v -run="none" -bench="BenchmarkSprintf"
testing: warning: no tests to run
PASS
BenchmarkSprintf         5000000               257 ns/op
ok      github.com/goinaction/code/chapter9/listing05   1.551s
```

Figure 1: Running a single benchmark.

It starts out specifying that there are no tests to run and then proceeds to run the `BenchmarkSprintf` benchmark. After the word `PASS`, we see the result of running the benchmark function. The first number `5000000` represents the number of times the code inside the loop was executed. In this case that is five million times. The next numbers represents the performance of the code based on the number of nanoseconds per operation. So, the performance of using the `Sprintf` function in this context is it takes 257 nanoseconds on average per call.

The final output from running the benchmark shows *ok* to represent the benchmark finished properly. Then the name of the code file that was executed is displayed and finally the total time the benchmark ran. The default minimum run time for a benchmark is 1 second. You can see how the framework still ran the test for approximately a second and a half. There is another option called `-benchtime` if you want to have the test run longer. Let's run the test again using a bench time of three seconds:

```
$ go test -v -run="none" -bench="BenchmarkSprintf" -benchtime="3s"
testing: warning: no tests to run
PASS
BenchmarkSprintf        20000000               250 ns/op
ok      github.com/goinaction/code/chapter9/listing05   5.275s
```

Figure 2: Running a single benchmark with the -benchtime option.

This time the `Sprintf` function was run twenty million times for a period of 5.275 seconds. The performance of the function didn't change much. This time the performance was 250

nanoseconds per operation. Sometimes by increasing the bench time, you can get a more accurate reading of performance. Increasing the bench time over three seconds for most tests tends to not provide any difference for an accurate reading. But each benchmark is different.

Let's look at the other two benchmark functions and then run all three benchmarks together to see what is the fastest way to convert an integer value to a string:

Listing 5: listing28_test.go : lines 24 - 46

```
24 // BenchmarkFormat provides performance numbers for the 25 //
strconv.FormatInt function.
26      func BenchmarkFormat(b *testing.B) {
27      number := int64(10)

28

29     b.ResetTimer()

30

31      for i := 0; i < b.N; i++ {
32      strconv.FormatInt(number, 10)
33      }
34      }

35

36 // BenchmarkItoa provides performance numbers for the 37 //
strconv.Itoa function.
38      func BenchmarkItoa(b *testing.B) {
39      number := 10

40

41     b.ResetTimer()

42

43      for i := 0; i < b.N; i++ {
44      strconv.Itoa(number)
45      }
46      }
```

Listing 5 shows the other two benchmark functions. The `BenchmarkFormat` function benchmarks the use of the `FormatInt` function from the `strconv` package. The `BenchmarkItoa` function benchmarks the use of the `Itoa` function from the same `strconv` package. You can see the same pattern in these two other benchmark function as in the `BenchmarkSprintf` function. The call is inside the `for` loop using `b.N` to control the number of iterations for each call.

One thing we skipped over was the call to `b.ResetTimer`, which is used in all three benchmark functions. This method is useful to reset the timer when initialization is required before the code can start executing the loop. To have the most accurate benchmark times you can, use this method.

When we run all the benchmark functions for a minimum of three seconds, we get the following result:

```
$ go test -v -run="none" -bench=. -benchtime="3s"
testing: warning: no tests to run
PASS
BenchmarkSprintf        20000000               258 ns/op
BenchmarkFormat 100000000               46.8 ns/op
BenchmarkItoa   100000000               50.1 ns/op
ok      github.com/goinaction/code/chapter9/listing05   15.228s
```

Figure 3: Running all three benchmarks.

The results show that the `BenchmarkFormat` test function runs the fastest at 46.8 nanoseconds per operation. The `BenchmarkItoa` comes in a close second at 50.1 nanoseconds per operation. Both of those benchmarks were much faster than the use of the `Sprintf` function.

Another great option you can use when running benchmarks is the `-benchmem` option. It will provide information about the number of allocations and bytes per allocation for a given test. Let's use that option with the benchmark:

```
$ go test -v -run="none" -bench=. -benchtime="3s" -benchmem
testing: warning: no tests to run
PASS
BenchmarkSprintf        20000000               262 ns/op              16 B/op          2 allocs/op
BenchmarkFormat 100000000               47.1 ns/op             2 B/op          1 allocs/op
BenchmarkItoa   100000000               50.2 ns/op             2 B/op          1 allocs/op
ok      github.com/goinaction/code/chapter9/listing05   15.339s
```

Figure 4: Running benchmark with the -benchmem option.

This time with the output we see two new values. A value for `B/op` and one for `allocs/op`. The `allocs/op` value represents the number of heap allocations per operation. We can see the `Sprintf` functions allocate two values on the heap per operation and the other two function allocation one value per operation. The `B/op` value represents the number of bytes per

operation. We can see that those two allocations from the `Sprintf` function results in sixteen bytes of memory being allocated per operation. The other two functions only allocated two bytes per operation.

There are many different options you can use when running tests and benchmarks. I suggest you explore all those options and leverage this testing framework to the fullest extent when writing your packages and projects. The community expects package authors to provide comprehensive tests when publishing packages for open use by the community.