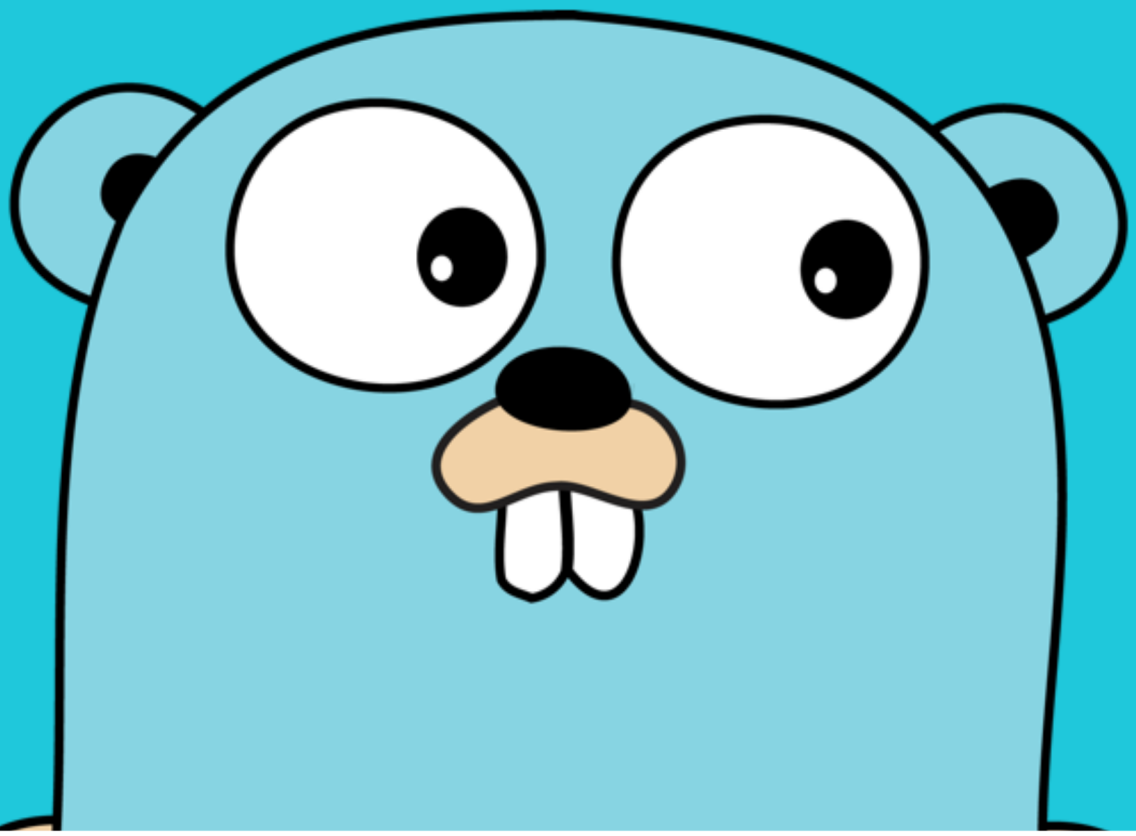
Production Go

Build modern, production-ready systems in Go



Herman Schaaf • Shawn Smith

Production Go

Build modern, production-ready web services in Go

Herman Schaaf and Shawn Smith

This book is for sale at http://leanpub.com/productiongo

This version was published on 2022-06-24

Contents

Introduction

Why Go in Production?

If you are reading this book, we assume you are interested in running Go in a production environment. Maybe you dabble in Go on your side projects, but are wondering how you can use it at work. Or perhaps you've read a company blog post about converting their codebase to Go, which now has 3 times less code and response times one tenth of what they were before. Your mileage will vary when it comes to gains in productivity and efficiency, but we have generally found a switch to Go to be more than worthwhile. Our goal in writing this book is to provide the knowledge to write a production-ready service in Go. This means not only writing the initial implementation, but also reliably deploying it, monitoring its performance, and iterating on improvements.

Go is a language that allows for fast iteration, which goes well with continuous deployment. Although it is a statically typed language, it compiles quickly and can often be used as a replacement for scripting languages like Python. Many users report that when writing Go, once a program works, it continues to "just work". This is due to the relatively simple design of the language, and the focus on readability rather than clever constructs.

In one project, we replaced existing APIs in PHP with equivalent functionality in Go. We saw performance improvements, including an order of magnitude reduction in response times, which led to both higher user retention and a reduction in server costs. We also saw developer happiness increase, because the safety guarantees in Go meant we could deploy changes regularly and safely.

This book is not meant for beginner programmers. We expect our audience to be knowledgeable of basic computer science topics and software engineering practices. Over the years, we have helped ramp up countless engineers who had no prior experience writing Go. Ideally this will be the book that people recommend to engineers writing Go for the first time, and who want to better understand the "right way" to write Go.

We hope this book will help guide you on your journey to running Go in production. It will cover many important aspects of running a production system, including topics not covered by most books on the language, like profiling the memory usage of a Go program, deploying and monitoring apps written in Go, and writing tests for web applications.

Feel free to skip around to chapters that seem more relevant to your immediate concerns or interests. We will do our best to keep the chapters fairly independent of one another in order to make that possible.

Strings

In Go, string literals are defined using double quotes, similar to other popular programming languages in the C family:

An example of a string literal

```
package main

import "fmt"

func ExampleString() {
        s := "I am a string - 你好"
        fmt.Println(s)
        // Output: I am a string - 你好
}
```

As the example shows, Go string literals and code may also contain non-English characters, like the Chinese 你好 [1].

Appending to Strings

Strings can be appended to with the addition (+) operator:

Appending to a string

```
package main

import "fmt"

func ExampleAppend() {
        greeting := "Hello, my name is "
        greeting += "Inigo Montoya"
        greeting += "."
        fmt.Println(greeting)
        // Output: Hello, my name is Inigo Montoya.
}
```

[1] 你好, pronounced *nǐ hǎo*, is Hello in Chinese.

This method of string concatenation is easy to read, and great for simple cases. But while Go does allow us to concatenate strings with the `+` (or `+=`) operator, it is not the most efficient method. It is best used only when very few strings are being added, and not in a hot code path. For a discussion on the most efficient way to do string concatenation, see the later chapter on optimization.

In most cases, the built-in `fmt.Sprintf`[2] function available in the standard library is a better choice for building a string. We can rewrite the previous example like this:

Using `fmt.Sprintf` to build a string

```
1  package main
2
3  import "fmt"
4
5  func ExampleFmtString() {
6          name := "Inigo Montoya"
7          sentence := fmt.Sprintf("Hello, my name is %s.", name)
8          fmt.Println(sentence)
9          // Output: Hello, my name is Inigo Montoya.
10 }
```

The `%s` sequence is a special placeholder that tells the `Sprintf` function to insert a string in that position. There are also other sequences for things that are not strings, like `%d` for integers, `%f` for floating point numbers, or `%v` to leave it to Go to figure out the type. These sequences allow us to add numbers and other types to a string without casting, something the `+` operator would not allow due to type conflicts. For example:

Using `fmt.Printf` to combine different types of variables in a string

```
1  package main
2
3  import "fmt"
4
5  func ExampleFmtComplexString() {
6          name := "Inigo Montoya"
7          age := 32
8          weight := 76.598
9          t := "Hello, my name is %s, age %d, weight %.2fkg"
10         fmt.Printf(t, name, age, weight)
11         // Output: Hello, my name is Inigo Montoya, age 32, weight 76.60kg
12 }
```

[2] https://golang.org/pkg/fmt/#Sprintf

Note that here we used `fmt.Printf` to print the new string directly. In previous examples, we used `fmt.Sprintf` to first create a string variable, then `fmt.Println` to print it to the screen (notice the `S` in `Sprintf`, short for string). In the above example, `%d` is a placeholder for an integer, `%.2f` a for a floating point number that should be rounded to the second decimal, and `%s` a placeholder for a string, as before. These codes are analogous to ones in the `printf` and `scanf` functions in C, and old-style string formatting in Python. If you are not familiar with this syntax, have a look at the documentation for the `fmt` package[3]. It is both expressive and efficient, and used liberally in Go code.

What would happen if we tried to append an integer to a string using the plus operator?

Breaking code that tries to append an integer to a string

```
1  package main
2
3  func main() {
4          s := "I am" + 32 + "years old"
5  }
```

Running this with `go run`, Go returns an error message during the build phase:

```
1  $ go run bad_append.go
2  # command-line-arguments
3  ./bad_append.go:4: cannot convert "I am" to type int
4  ./bad_append.go:4: invalid operation: "I am" + 32 (mismatched types string an\
5  d int)
```

As expected, Go's type system catches our transgression, and complains that it cannot append an integer to a string. We should rather use `fmt.Sprintf` for building strings that mix different types.

Next we will have a look at a very useful standard library package that allows us to perform many common string manipulation tasks: the built-in `strings` package.

Splitting strings

The `strings` package is imported by simply adding `import "strings"`, and provides us with many string manipulation functions. One of these is a function that split a string by separators, and obtain a slice of strings:

[3] https://golang.org/pkg/fmt/

Splitting a string

```
package main

import "fmt"
import "strings"

func ExampleSplit() {
        l := strings.Split("a,b,c", ",")
        fmt.Printf("%q", l)
        // Output: ["a" "b" "c"]
}
```

The `strings.Split` function takes a string and a separator as arguments. In this case, we passed in `"a,b,c"` and the separator ",", and received a string slice containing the separate letters `a`, `b`, and `c` as strings.

Counting and finding substrings

Using the `strings` package, we can also count the number of non-overlapping instances of a substring in a string with the aptly-named `strings.Count`. The following example uses `strings.Count` to count occurrences of both the single letter `a`, and the substring `ana`. In both cases we pass in a string[4]. Notice that we get only one occurrence of `ana`, even though one may have expected it to count `ana` both at positions 1 and 3. This is because `strings.Count` returns the count of *non-overlapping* occurrences.

Count occurrences in a string

```
package main

import (
        "fmt"
        "strings"
)

func ExampleCount() {
        s := "banana"
        c1 := strings.Count(s, "a")
        c2 := strings.Count(s, "ana")
        fmt.Println(c1, c2)
```

[4]Remember, Go does not support function overloading, so a single character should be passed as a string if the function expects a string, like most functions in the `strings` standard library.

```
13          // Output: 3 1
14  }
```

If we want to know whether a string contains, starts with, or ends with some substring, we can use the `strings.Contains`, `strings.HasPrefix`, and `strings.HasSuffix` functions, respectively. All of these functions return a boolean:

Count occurrences in a string

```
1   package main
2
3   import (
4           "fmt"
5           "strings"
6   )
7
8   func ExampleContains() {
9           str := "two gophers on honeymoon"
10          if strings.Contains(str, "moon") {
11                  fmt.Println("Contains moon")
12          }
13          if strings.HasPrefix(str, "moon") {
14                  fmt.Println("Starts with moon")
15          }
16          if strings.HasSuffix(str, "moon") {
17                  fmt.Println("Ends with moon")
18          }
19          // Output: Contains moon
20          // Ends with moon
21  }
```

For finding the index of a substring in a string, we can use `strings.Index`. Index returns the index of the first instance of substr in `s`, or -1 if substr is not present in s:

Using strings.Index to find substrings in a string

```
package main

import "fmt"
import "strings"

func ExampleIndex() {
        an := strings.Index("banana", "an")
        am := strings.Index("banana", "am")
        fmt.Println(an, am)
        // Output: 1 -1
}
```

The `strings` package also contains a corresponding `LastIndex` function, which returns the index of the *last* (ie. right-most) instance of a matching substring, or -1 if it is not found.

The `strings` package contains many more useful functions. To name a few: `ToLower`, `ToUpper`, `Trim`, `Equals` and `Join`, all performing actions that match their names. For more information on these and other functions, refer to the `strings package docs`[5]. As a final example, let's see how we might combine some of the functions in the `strings` package in a real program, and discover some of its more surprising functions.

Advanced string functions

The program below repeatedly takes input from the user, and declares whether the typed sentence is palindromic. For a sentence to be palindromic, we mean that the words should be the same when read forwards and backwards. We wish to ignore punctuation, and assume the sentence is in English, so there are spaces between words. Take a look and notice how we use two new functions from the `strings` package, `FieldsFunc` and `EqualFold`, to keep the code clear and concise.

[5] https://golang.org/pkg/strings/

A program that declares whether a sentence reads the same backward and forward, word for word

```
package main

import (
        "bufio"
        "fmt"
        "os"
        "strings"
        "unicode"
)

// getInput prompts the user for some text, and then
// reads a line of input from standard input. This line
// of text is then returned.
func getInput() string {
        fmt.Print("Enter a sentence: ")
        scanner := bufio.NewScanner(os.Stdin)
        scanner.Scan()
        return scanner.Text()
}

func isNotLetter(c rune) bool {
        return !unicode.IsLetter(c)
}

// isPalindromicSentence returns whether or not the given sentence
// is palindromic. To calculate this, it splits the string into words,
// then creates a reversed copy of the word slice. It then checks
// whether the reverse is equal (ignoring case) to the original.
// It also ignores any non-alphabetic characters.
func isPalindromicSentence(s string) bool {
        // split into words and remove non-alphabetic characters
        // in one operation by using FieldsFunc and passing in
        // isNotLetter as the function to split on.
        w := strings.FieldsFunc(s, isNotLetter)

        // iterate over the words from front and back
        // simultaneously. If we find a word that is not the same
        // as the word at its matching from the back, the sentence
```

```
39          // is not palindromic.
40          l := len(w)
41          for i := 0; i < l/2; i++ {
42                  fw := w[i]     // front word
43                  bw := w[l-i-1] // back word
44                  if !strings.EqualFold(fw, bw) {
45                          return false
46                  }
47          }
48
49          // all the words matched, so the sentence must be
50          // palindromic.
51          return true
52  }
53
54  func main() {
55          // Go doesn't have while loops, but we can use for loop
56          // syntax to read into a new variable, check that it's not
57          // empty, and read new lines on subsequent iterations.
58          for l := getInput(); l != ""; l = getInput() {
59                  if isPalindromicSentence(l) {
60                          fmt.Println("... is palindromic!")
61                  } else {
62                          fmt.Println("... is not palindromic.")
63                  }
64          }
65  }
```

Save this code to `palindromes.go`, and we can then run it with `go run palindromes.go`.

An example run of the palindrome program

```
1  $ go run palindromes.go
2  Enter a sentence: This is magnificent!
3  ... is not palindromic.
4  Enter a sentence: This is magnificent, is this!
5  ... is palindromic!
6  Enter a sentence:
```

As expected, when we enter a sentence that reads the same backwards and forwards, ignoring

punctuation and case, we get the output `... is palindromic!`. Now, let's break down what this code is doing.

The `getInput` function uses a `bufio.Scanner` from the `bufio` package[6] to read one line from standard input. `scanner.Scan()` scans until the end of the line, and `scanner.Text()` returns a string containing the input line.

The meat of this program is in the `isPalindromicSentence` function. This function takes a string as input, and returns a boolean indicating whether the sentence is palindromic, word-for-word. We also want to ignore punctuation and case in the comparison. First, on line 34, we use `strings.FieldsFunc` to split the string at each Unicode code point for which the `isNotLetter` function returns true. In Go, you can pass around functions like any other value. A function's type signature describes the types of its arguments and return values. Our `isNotLetter` function satisfies the function signature specified by `FieldsFunc`, which is to take a rune as input, and return a boolean. Runes are a special character type in the Go language - for now, just think of them as more or less equivalent to a single character, like `char` in Java.

In `isNotLetter`, we return `false` if the passed in rune is a letter as defined by the Unicode standard, and `true` otherwise. We can achieve this in a single line by using `unicode.IsLetter`, another built-in function provided by the standard `unicode` library.

Putting it all together, `strings.FieldsFunc(s, isNotLetter)` will return a slice of strings, split by sequences of non-letters. In other words, it will return a slice of words.

Next, on line 40, we iterate over the slice of words. We keep an index `i`, which we use to create both `fw`, the word at index `i`, and `bw`, the matching word at index `l - i - 1`. If we can walk all the way through the slice without finding two words that are not equal, we have a palindromic sentence. And we can stop halfway through, because then we have already done all the necessary comparisons. The next table shows how this process works for an example sentence as `i` increases. As we walk through the slice, words match, and so we continue walking until we reach the middle. If we were to find a non-matching pair, we can immediately return `false`, because the sentence is not palindromic.

The palindromic sentence algorithm by example

	"Fall"	"leaves"	"as"	"soon"	"as"	"leaves"	"fall"	EqualFold
i=0	fw						bw	true
i=1		fw				bw		true
i=2			fw		bw			true

The equality check of strings is performed on line 44 using `strings.EqualFold` - this function compares two strings for equality, ignoring case.

Finally, on line 58, we make use of the semantics of the Go for loop definition. The basic for loop has three components separated by semicolons:

- the init statement: executed before the first iteration

[6] https://golang.org/pkg/bufio/

- the condition expression: evaluated before every iteration
- the post statement: executed at the end of every iteration

We use these definition to instantiate a variable `l` and read into it from standard input, conditionally break from the loop if it is empty, and set up reading for each subsequent iteration in the post statement.

Ranging over a string

When the functions in the `strings` package don't suffice, it is also possible to range over each character in a string:

Iterating over the characters in a string

```
package main

import "fmt"

func ExampleIteration() {
        s := "ABC你好"
        for i, r := range s {
                fmt.Printf("%q(%d) ", r, i)
        }
        // Output: 'A'(0) 'B'(1) 'C'(2) '你'(3) '好'(6)
}
```

You might be wondering about something peculiar about the output above. The printed indexes start from 0, 1, 2, 3 and then jump to 6. Why is that? This is the topic in the next chapter, Supporting Unicode.

Testing

A critical part of any production-ready system is a complete test suite. If you have not written tests before, and wonder why they are important, this introduction is for you. If you already understand the importance of proper testing, you can skip to the particulars of writing tests in Go, in writing tests.

Why do we need tests?

A line of reasoning we sometimes hear, is that "my code clearly works, why do I need to write tests for it?" This is a natural enough question, but make no mistake, a modern production-ready system absolutely must have automated tests. Let's use an analogy from the business world to understand why this is so: double entry bookkeeping.

Double entry is the idea that every financial transaction has equal and opposite effects in at least two different accounts. For example, if you spend $10 on groceries, your bank account goes down by $10, and your groceries account goes up by $10. This trick allows you to see, at a glance, simultaneously how much money is in your bank account, and how much you spent on groceries. It also allows you to spot mistakes. Suppose a smudge in your books made the $10 entry look like $18. The total balance of your assets would no longer match your liabilities plus equity - there would be an $8 difference. We can compare entries in the bank account with entries in the groceries account to discover which amount is incorrect. Before double-entry bookkeeping, it was much harder to prove mistakes, and impossible to see different account balances at a glance. The idea revolutionized bookkeeping, and underpins accounting to this day.

Back to tests. For every piece of functionality we write, we also write a test. The test should prove that the code works in all reasonable scenarios. Like double-entry bookkeeping, tests are our way to ensure our system is correct, and remains correct. Your system might work now - you might even prove it to yourself by trying out some cases manually. But systems, especially production systems, require changes over time. Requirements change, environments change, bugs emerge, new features become needed, inefficiencies are discovered. All these things will require changes to be made to the code. After making these changes, will you still be sure that the system is correct? Will you run through manual test cases after every change? What if someone else is maintaining the code? Will they know how to test changes? How much time will it take you to manually perform these test cases?

Automated tests cost up-front investment, but they uncover bugs early, improve maintainability, and save time in the long run. Tests are the checks and balances to your production system.

Many books and blog posts have been written about good testing practice. There are even movements that promote writing tests first[7], before writing the code. We don't think it's necessary to be quite that extreme, but if it helps you write good tests, then more power to you. No production system is complete without a test suite that makes sensible assertions on the code to prove it correct.

Now that we have discussed the importance of testing in general, let's see how tests are written in Go. As we'll see, testing was designed with simplicity in mind.

Writing Tests

Test files in Go are located in the same package as the code being tested, and end with the suffix `_test.go`. Usually, this means having one `_test.go` to match each code file in the package. Below is the layout of a simple package for testing prime numbers.

- prime
  - prime.go
  - prime_test.go
  - sieve.go
  - sieve_test.go

This is a very typical Go package layout. Go packages contain all files in the same directory, including the tests. Now, let's look at what the test code might look like in `prime_test.go`.

A simple test in `prime_test.go` that tests a function called IsPrime

```
1  package main
2
3  import "testing"
4
5  // TestIsPrime tests that the IsPrime function
6  // returns true when the input is prime, and false
7  // otherwise.
8  func TestIsPrime(t *testing.T) {
9          // check a prime number
10         got := IsPrime(19)
11         if got != true {
12                 t.Errorf("IsPrime(%d) = %t, want %t", 19, got, true)
13         }
14
15         // check a non-prime number
```

[7] https://en.wikipedia.org/wiki/Test-driven_development

```
16          got = IsPrime(21)
17          if got != false {
18                  t.Errorf("IsPrime(%d) = %t, want %t", 21, got, false)
19          }
20  }
```

We start by importing the `testing` package. Then, on line 8, we define a test as a normal Go function taking a single argument: `t *testing.T`. All tests must start with the word `Test` and have a single argument, a pointer to `testing.T`. In the function body, we call the function under test, `IsPrime`. First we pass in the integer 19, which we expect should return `true`, because 19 is prime. We check this assertion with a simple `if` statement on line 11, `if got != true`. If the statement evaluates to false, `t.Errorf` is called. `Errorf` formats its arguments in a way analogous to `Printf`, and records the text in the error log. We repeat a similar check for the number 21, this time asserting that the `IsPrime` function returns `true`, because 21 is not prime.

We can run the tests in this package using `go test`. Let's see what happens:

```
$ go test
PASS
ok      _/Users/productiongo/code/prime    0.018s
```

It passed! But did it actually run our `TestIsPrime` function? Let's check by adding the `-v` (verbose) flag to the command:

```
$ go test -v
=== RUN   TestIsPrime
--- PASS: TestIsPrime (0.00s)
PASS
ok      _/Users/productiongo/code/prime    0.019s
```

Our test is indeed being executed. The `-v` flag is a useful trick to remember, and we recommend running tests with it turned on most of the time.

All tests in Go follow essentially the same format as the `TestIsPrime`. The Go authors made a conscious decision not to add specific assertion functions, advising instead to use the existing control flow tools that come with the language. The result is that tests look very similar to normal Go code, and the learning curve is minimal.

Table-driven tests

Our initial `TestIsPrime` test is a good start, but it only tests two numbers. The code is also repetitive. We can do better by using what is called a *table-driven* test. The idea is to define all the inputs and expected outputs first, and then loop through each case with a `for` loop.

A table-driven test in `prime_test.go` that tests a function called IsPrime

```
package main

import "testing"

// TestIsPrime tests that the IsPrime function
// returns true when the input is prime, and false
// otherwise.
func TestIsPrimeTD(t *testing.T) {
        cases := []struct {
                give int
                want bool
        }{
                {19, true},
                {21, false},
                {10007, true},
                {1, false},
                {0, false},
                {-1, false},
        }

        for _, c := range cases {
                got := IsPrime(c.give)
                if got != c.want {
                        t.Errorf("IsPrime(%d) = %t, want %t", c.give, got, c.want)
                }
        }
}
```

In the refactored test, we use a slice of an anomymous struct to define all the inputs we want to test. We then loop over each test case, and check that the output matches what we want. This is much cleaner than before, and it only took a few keystrokes to add more test cases into the mix. We now also check some edge cases: inputs of 0, 1, 10007, and negative inputs. Let's run the test again and check that it still passes:

```
$ go test -v
=== RUN   TestIsPrimeTD
--- PASS: TestIsPrimeTD (0.00s)
PASS
ok      _/Users/productiongo/code/prime   0.019s
```

It looks like the `IsPrime` function works as advertised! To be sure, let's add a test case that we expect to fail:

```
    ...
        {-1, false},

        // 17 is prime, so this test should fail:
        {17, false},
    }
    ...
```

We run `go test -v` again to see the results:

```
$ go test -v
=== RUN   TestIsPrimeTD
--- FAIL: TestIsPrimeTD (0.00s)
    prime_test.go:25: IsPrime(17) = true, want false
FAIL
exit status 1
FAIL    _/Users/productiongo/code/prime   0.628s
```

This time `go test` reports that the test failed, and we see the error message we provided to `t.Errorf`.

Writing error messages

In the tests above, we had the following code:

```
if got != c.want {
    t.Errorf("IsPrime(%d) = %t, want %t", c.give, got, c.want)
}
```

The ordering of the if statement is not accidental: by convention, it should be `actual != expected`, and the error message uses that order too. This is the recommended way to format test failure messages

in Go [8]. In the error message, first state the function called and the parameters it was called with, then the actual result, and finally, the result that was expected. We saw before that this results in a message like

```
prime_test.go:25: IsPrime(17) = true, want false
```

This makes it clear to the reader of the error message what function was called, what happened, and what should have happened. The onus is on you, the test author, to leave a helpful message for the person debugging the code in the future. It is a good idea to assume that the person debugging your failing test is not you, and is not your team. Make both the name of the test and the error message relevant.

Testing HTTP Handlers

Let's look at an example of testing that comes up often when developing web applications: testing an HTTP handler. First, let's define a comically simple HTTP handler that writes a friendly response:

A very simple HTTP handler

```
1  package main
2
3  import (
4          "fmt"
5          "net/http"
6  )
7
8  // helloHandler writes a friendly "Hello, friend :)" response.
9  func helloHandler(w http.ResponseWriter, r *http.Request) {
10         fmt.Fprintln(w, "Hello, friend :)")
11 }
12
13 func main() {
14         http.HandleFunc("/hello", helloHandler)
15         http.ListenAndServe(":8080", nil)
16 }
```

The `httptest` package provides us with the tools we need to test this handler as if it were running in a real web server. The `TestHTTPHandler` function in the following example illustrates how to use `httptest.NewRecorder()` to send a real request to our friendly `helloHandler`, and read the resulting response.

[8] https://github.com/golang/go/wiki/CodeReviewComments#useful-test-failures

Testing an HTTP handler

```
package main

import (
        "net/http"
        "net/http/httptest"
        "testing"
)

func TestHTTPHandler(t *testing.T) {
        // Create a request to pass to our handler.
        req, err := http.NewRequest("GET", "/hello", nil)
        if err != nil {
                t.Fatal(err)
        }

        // We create a ResponseRecorder, which satisfies
        // the http.ResponseWriter interface, to record
        // the response.
        r := httptest.NewRecorder()
        handler := http.HandlerFunc(helloHandler)

        // Our handler satisfies http.Handler, so we can call
        // the ServeHTTP method directly and pass in our
        // Request and ResponseRecorder.
        handler.ServeHTTP(r, req)

        // Check that the status code is what we expect.
        if r.Code != http.StatusOK {
                t.Errorf("helloHandler returned status code %v, want %v",
                        r.Code, http.StatusOK)
        }

        // Check that the response body is what we expect.
        want := "Hello, friend :)\n"
        got := r.Body.String()
        if got != want {
                t.Errorf("helloHandler returned body %q want %q",
                        got, want)
```

```
39          }
40  }
```

In this example we see the `t.Fatal` method used for the first time. This method is similar to `t.Error`, but unlike `t.Error`, if `t.Fatal` is called, the test will not execute any further. This is useful when a condition happens that will cause the rest of the test to be unnecessary. In our case, if our call to create a request on line 11 were to fail for some reason, the call to `t.Fatal` ensures that we log the error and abandon execution immediately. Anagolous to `t.Errorf`, there is also a `t.Fatalf` method, which takes arguments the same way as `fmt.Printf`.

On line 19 we create a new `httptest.Recorder` with which to record the response. We also create `handler`, which is `helloHandler`, but now of type `http.HandlerFunc`. We can do this, because `helloHandler` uses the appropriate signature defined by `http.HandlerFunc`[9]:

```
type HandlerFunc func(ResponseWriter, *Request)
```

`http.HandlerFunc` is an adapter to allow the use of ordinary functions as HTTP handlers. As the final step of the setup, we pass the recorder and the request we created earlier in to `handler.ServeHTTP(r, req)`. Now we can use the fields provided by `httptest.Recorder`, like `Code` and `Body`, to make assertions against our HTTP handler, as shown in the final lines of the test function.

Mocking

Imagine you need to test code that uses a third party library. Perhaps this library is a client library to an external API, or perhaps it performs database operations. In your unit tests, it is best to assume that the library does its job, and only test your functions and their interactions. This allows your test case failures to accurately reflect where the problem is, rather than leave the question of whether it's your function, or the library, that's at fault. There is a place for tests that include third party libraries, and that place is in integration tests, not unit tests.

Interfaces

How do we go about testing our functions, but not the libraries they use? The answer: interfaces. Interfaces are an incredibly powerful tool in Go.

In Java, interfaces need to be explicitly implemented. You rely on your third party vendor to provide an interface that you can use to stub methods for tests. In Go, we don't need to rely on the third party author; we can define our own interface. As long as our interface defines a subset of the methods implemented by the library, the library will automatically implement our interface.

The next example illustrates one particular case where mocking is very useful: testing code that relies on random number generation.

[9] https://golang.org/pkg/net/http/#HandlerFunc

Using an interface to abstract away API calls

```
package eightball

import (
        "math/rand"
        "time"
)

// randIntGenerator is an interface that includes Intn, a
// method in the built-in math/rand package. This allows us
// to mock out the math/rand package in the tests.
type randIntGenerator interface {
        Intn(int) int
}

// EightBall simulates a very simple magic 8-ball,
// a magical object that predicts the future by
// answering yes/no questions.
type EightBall struct {
        rand randIntGenerator
}

// NewEightBall returns a new EightBall.
func New() *EightBall {
        return &EightBall{
                rand: rand.New(rand.NewSource(time.Now().UnixNano())),
        }
}

// Answer returns a magic eightball answer
// based on a random result provided by
// randomGenerator. It supports only four
// possible answers.
func (e EightBall) Answer(s string) string {
        n := e.rand.Intn(3)
        switch n {
        case 0:
                return "Definitely not"
        case 1:
```

```
39                 return "Maybe"
40         case 2:
41                 return "Yes"
42         default:
43                 return "Absolutely"
44         }
45 }
```

We define a simple `eightball` package that implements a simple Magic 8-Ball[10]. We ask it a yes/no question, and it will return its prediction of the future. As you might expect, it completely ignores the question, and just makes use of a random number generator. But random numbers are hard to test, because they change all the time. One option would be to set the random seed in our code, or in our tests. This is indeed an option, but it doesn't allow us to specifically test the different outcomes without some trial and error. Instead, we create an `randIntGenerator` interface, which has only one method, `Intn(int) int`. This method signature is the same as the `Intn`[11] method implemented by Go's built-in `math/rand` package. Instead of using the `math/rand` package directly in `Answer`, we decouple our code by referencing the `Intn` method on the EightBall's rand interface. Since `EightBall.rand` is not exported, users of this package will not be aware of this interface at all. To create the struct, they will need to call the `New` method, which assigns the built-in struct from `math/rand` struct to satisfy our interface. So to package users the code looks the same, but under the hood, we can now mock out the call to `Intn` in our tests:

Testing using our interface

```
1  package eightball
2
3  import (
4          "testing"
5  )
6
7  type fixedRandIntGenerator struct {
8          // the number that should be "randomly" generated
9          randomNum int
10
11         // record the paramater that Intn gets called with
12         calledWithN int
13 }
14
15 func (g *fixedRandIntGenerator) Intn(n int) int {
16         g.calledWithN = n
```

[10] https://en.wikipedia.org/wiki/Magic_8-Ball
[11] https://golang.org/pkg/math/rand/#Rand.Intn

```
17          return g.randomNum
18  }
19
20  func TestEightBall(t *testing.T) {
21          cases := []struct {
22                  randomNum int
23                  want      string
24          }{
25                  {0, "Definitely not"},
26                  {1, "Maybe"},
27                  {2, "Yes"},
28                  {3, "Absolutely"},
29                  {-1, "Absolutely"}, // default case
30          }
31
32          for _, tt := range cases {
33                  g := &fixedRandIntGenerator{randomNum: tt.randomNum}
34                  eb := EightBall{
35                          rand: g,
36                  }
37
38                  got := eb.Answer("Does this really work?")
39                  if got != tt.want {
40                          t.Errorf("EightBall.Answer() is %q for num %d, want %q",
41                                  got, tt.randomNum, tt.want)
42                  }
43
44                  if g.calledWithN != 3 {
45                          t.Errorf("EightBall.Answer() did not call Intn(3) as expected")
46                  }
47          }
48  }
```

Sometimes, when the existing code uses a specific library implementation, it takes refactoring to use interfaces to mock out impementation details. However, the resulting code is more decoupled. The tests run faster (e.g. when mocking out external network calls) and are more reliable. Don't be afraid to make liberal use of interfaces. This makes for more decoupled code and more focused tests.

GoMock

Another way to generate mocks for tests is GoMock[12]. WIP

Generating Coverage Reports

To generate test coverage percentages for your code, run the `go test -cover` command. Let's make a quick example and a test to go with it.

We're going to write a simple username validation function. We want our usernames to only contain letters, numbers, and the special characters "-", "_", and ".". Usernames also cannot be empty, and they must be less than 30 characters long. Here's our username validation function:

Username validation function

```
package validate

import (
        "fmt"
        "regexp"
)

// Username validates a username. We only allow
// usernames to contain letters, numbers,
// and special chars "_", "-", and "."
func Username(u string) (bool, error) {
        if len(u) == 0 {
                return false, fmt.Errorf("username must be > 0 chars")
        }
        if len(u) > 30 {
                return false, fmt.Errorf("username too long (must be < 30 chars)")
        }
        validChars := regexp.MustCompile(`^[a-zA-Z1-9-_.]+$`)
        if !validChars.MatchString(u) {
                return false, fmt.Errorf("username contains invalid character")
        }

        return true, nil
}
```

Now let's write a test for it:

[12] https://github.com/golang/mock

Test for username validation function

```
package validate

import "testing"

var usernameTests = []struct {
        in        string
        wantValid bool
}{
        {"gopher", true},
}

func TestUsername(t *testing.T) {
        for _, tt := range usernameTests {
                valid, err := Username(tt.in)
                if err != nil && tt.wantValid {
                        t.Fatal(err)
                }

                if valid != tt.wantValid {
                        t.Errorf("Username(%q) = %t, want %t", tt.in, valid, tt.wantValid)
                }
        }
}
```

As you can see, we're not covering very many cases. Let's see what exactly our test coverage is for this function:

```
$ go test -cover
PASS
coverage: 62.5% of statements
ok      github.com/gopher/validate 0.008s
```

62.5% is a bit too low. This function is simple enough that we can get close to 100% coverage. We'd like to know exactly what parts of the function are not being covered. This is where the coverage profile and HTML report come in.

To generate a test coverage profile, we run `go test -coverprofile=coverage.out`:

```
$ go test -coverprofile=coverage.out
PASS
coverage: 62.5% of statements
ok      github.com/gopher/validate 0.008s
```

We can now get a breakdown of coverage percentages per function, although we only have one function so it's not very interesting:

```
go tool cover -func=coverage.out
github.com/gopher/validate/validate.go:11: Username    62.5%
total:                  (statements)    62.5%
```

What we really want to see is a line-by-line breakdown. We can get this with the HTML report, which we'll cover in the next section.

HTML Coverage Reports

We can generate an HTML coverage report using the same `coverage.out` file from before, by running the following command:

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

This should open up a browser and show us an HTML page like the following:

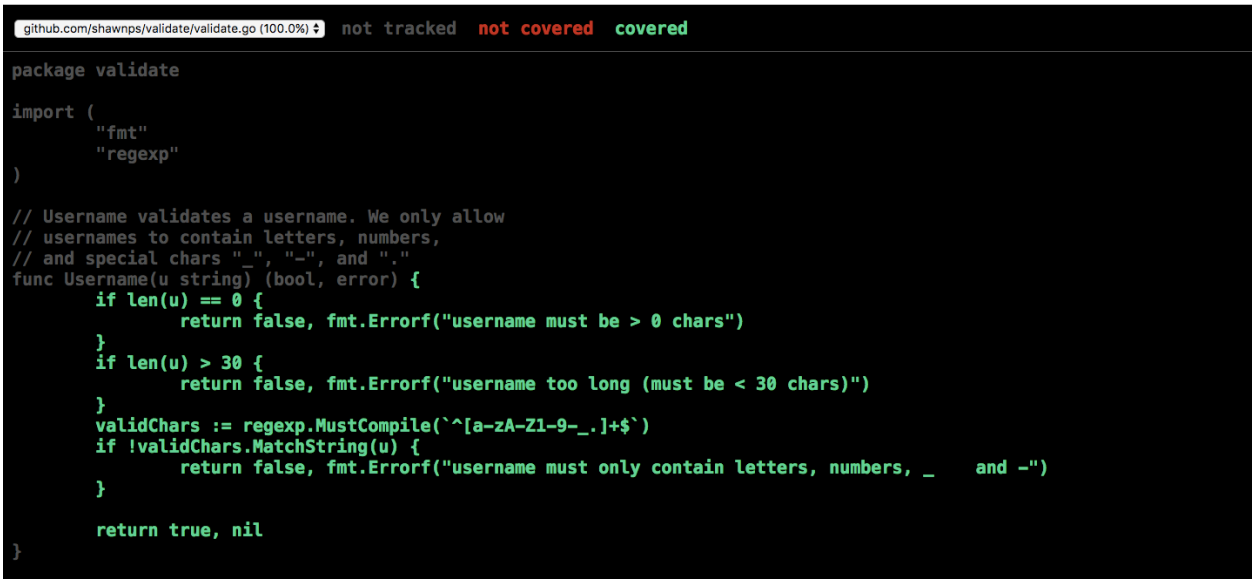
Username test coverage

Now we can see exactly where we need to improve our coverage. We need to cover the cases where the username length is either 0 or > 30, as well as the case where the username contains an invalid character. Let's update our test for those cases:

Test for username validation function, 100% coverage

```
package validate

import "testing"

var usernameTests = []struct {
        in        string
        wantValid bool
}{
        {"", false},
        {"gopher", true},
        {"gopher$", false},
        {"abcdefghijklmnopqrstuvwxyzabcde", false},
}

func TestUsername(t *testing.T) {
        for _, tt := range usernameTests {
                valid, err := Username(tt.in)
                if err != nil && tt.wantValid {
                        t.Fatal(err)
                }

                if valid != tt.wantValid {
                        t.Errorf("Username(%q) = %t, want %t", tt.in, valid, tt.wantValid)
                }
        }
}
```

Now if we re-run `go test -coverprofile=coverage.out` to get a new coverage profile, and then `go tool cover -html=coverage.out` to view the HTML report again, we should see all green:



Username test coverage 100%

Writing Examples

We can also write example code and the `go test` tool will run our examples and verify the output. `godoc` renders examples underneath the function's documentation.

Let's write an example for our username validation function:

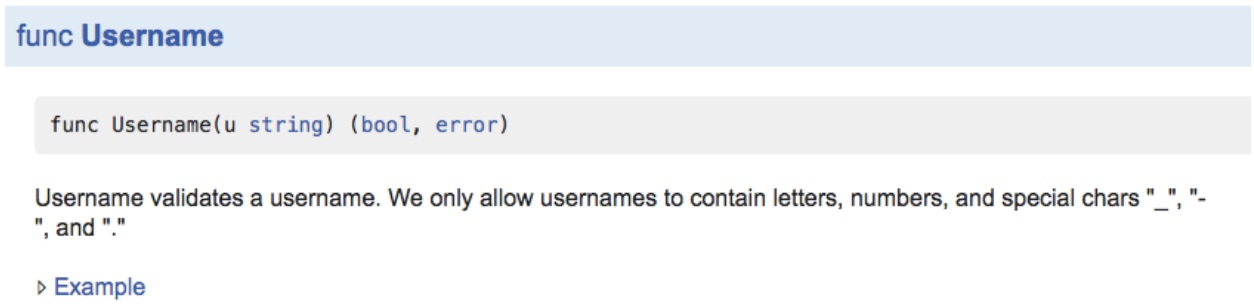
Username validation function example test

```
1  package validate
2
3  import (
4          "fmt"
5          "log"
6  )
7
8  func ExampleUsername() {
9          usernames := []struct {
10                 in    string
11                 valid bool
12         }{
13                 {"", false},
14                 {"gopher", true},
15                 {"gopher$", false},
16                 {"abcdefghijklmnopqrstuvwxyzabcde", false},
```

```
17          }
18          for _, tt := range usernames {
19                  valid, err := Username(tt.in)
20                  if err != nil && tt.valid {
21                          log.Fatal(err)
22                  }
23
24                  fmt.Printf("%q: %t\n", tt.in, valid)
25          }
26          // Output:
27          // "": false
28          // "gopher": true
29          // "gopher$": false
30          // "abcdefghijklmnopqrstuvwxyzabcde": false
31  }
```

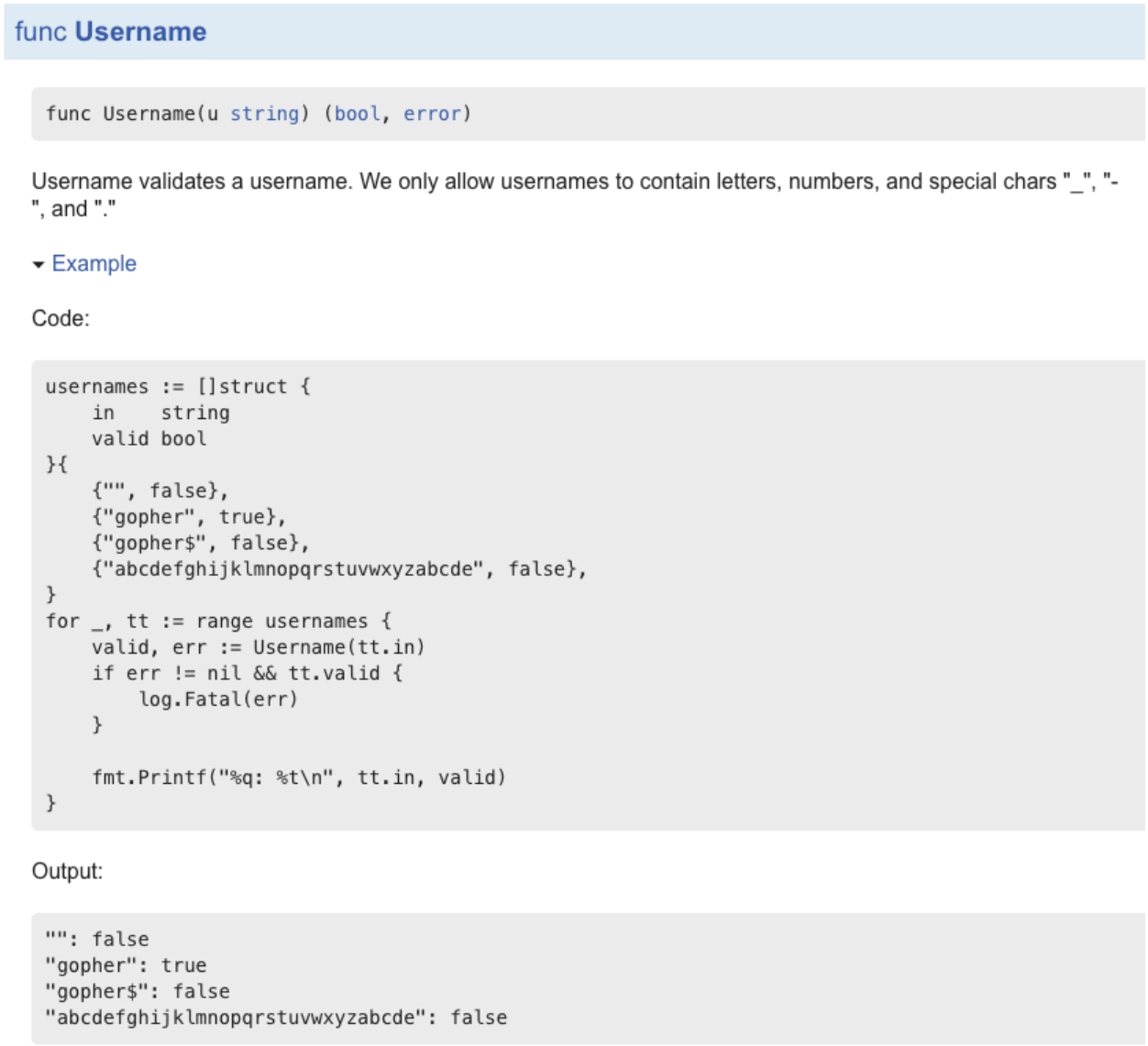
Note the `Output:` at the bottom. That's a special construct that tells `go test` what the standard output of our example test should be. `go test` is actually going to validate that output when it runs the tests.

If we run a local `godoc` server with `godoc -http:6060`, and navigate to our `validate` package, we can also see that `godoc` renders the example, as expected:



Godoc example

If we click "Example" we'll see our example code:



func **Username**

```
func Username(u string) (bool, error)
```

Username validates a username. We only allow usernames to contain letters, numbers, and special chars "_", "-", and "."

▾ Example

Code:

```
usernames := []struct {
    in    string
    valid bool
}{
    {"", false},
    {"gopher", true},
    {"gopher$", false},
    {"abcdefghijklmnopqrstuvwxyzabcde", false},
}
for _, tt := range usernames {
    valid, err := Username(tt.in)
    if err != nil && tt.valid {
        log.Fatal(err)
    }

    fmt.Printf("%q: %t\n", tt.in, valid)
}
```

Output:

```
"": false
"gopher": true
"gopher$": false
"abcdefghijklmnopqrstuvwxyzabcde": false
```

Godoc example full

Another note about examples is that they have a specific naming convention. We named our example above `ExampleUsername` because we wrote an example for the `Username` function. But what if we want to write an example for a method on a type? Let's say we had a type `User` with a method `ValidateName`:

```
type User struct {
    Name string
}

func (u *User) ValidateName() (bool, error) {
        ...
}
```

Then our example code would look like this:

```
func ExampleUser_ValidateName() {
    ...
}
```

where the convention for writing examples for methods on types is `ExampleT_M()`.

If we need multiple examples for a single function, we append an underscore and a lowercase letter. For example with our `Validate` function, we could have `ExampleValidate`, `ExampleValidate_second`, `ExampleValidate_third`, and so on.

In the next chapter, we will discuss one last important use of the Go testing package: benchmarking.

Benchmarks

The Go testing package contains a benchmarking tool for examining the performance of our Go code. In this chapter, we will use the benchmark utility to progressively improve the performance of a piece of code. We will then discuss advanced benchmarking techniques to ensure that we are measuring the right thing.

A simple benchmark

Let's suppose we have a simple function that computes the nth Fibonacci number. The sequence F_n of Fibonacci numbers is defined by the recurrence relation, $F_n = F_{n-1} + F_{n-2}$, with $F_0 = 0, F_1 = 1$. That is, every number after the first two is the sum of the two preceding ones:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
```

Because the sequence is recursively defined, a function that calculates the nth Fibonacci number is often used to illustrate programming language recursion in computer science text books. Below is such a function that uses the definition to recursively calculate the nth Fibonacci number.

A function that recursively obtains the nth Fibonacci number

```
1  package fibonacci
2
3  // F returns the nth Fibonnaci number.
4  func F(n int) int {
5          if n <= 0 {
6                  return 0
7          } else if n == 1 {
8                  return 1
9          }
10         return F(n-1) + F(n-2)
11 }
```

Let's make sure it works by writing a quick test, as we saw in the chapter on Testing.

A test for the function that recursively obtains the n^{th} Fibonacci number

```
// fibonacci_test.go
package fibonacci

import "testing"

func TestF(t *testing.T) {
        cases := []struct {
                n    int
                want int
        }{
                {-1, 0},
                {0, 0},
                {1, 1},
                {2, 1},
                {3, 2},
                {8, 21},
        }

        for _, tt := range cases {
                got := FastF(tt.n)
                if got != tt.want {
                        t.Errorf("F(%d) = %d, want %d", tt.n, got, tt.want)
                }
        }
}
```

Running the test, we see that indeed, our function works as promised:

```
$ go test -v
=== RUN   TestF
--- PASS: TestF (0.00s)
PASS
ok      _/home/productiongo/benchmarks/fibonacci    0.001s
```

Now, this recursive Fibonacci function works, but we can do better. How much better? Before we rewrite this function, let's establish a baseline to which we can compare our future efficiency improvements. Go provides a benchmark tool as part of the testing package. Anagalous to `TestX(t *testing.T)`, we create benchmarks with `BenchmarkX(b *testing.B)`:

A benchmark for the Fibonacci function

```
1  // fibonacci_bench_test.go
2  package fibonacci
3
4  import "testing"
5
6  var numbers = []int{
7          0, 10, 20, 30,
8  }
9
10 func BenchmarkF(b *testing.B) {
11         // run F(n) b.N times
12         m := len(numbers)
13         for n := 0; n < b.N; n++ {
14                 F(numbers[n%m])
15         }
16 }
```

The BenchmarkF function can be saved in any file ending with `_test.go` to be included by the testing package. The only real surprise in the code is the for loop defined on line 13,

```
13 for n := 0; n < b.N; n++ {
```

The benchmark function must run the target code `b.N` times. During benchmark execution, `b.N` is adjusted until the benchmark function lasts long enough to be timed reliably.

To run the benchmark, we need to instruct `go test` to run benchmarks using the `-bench` flag. Similar to the `-run` command-line argument, `-bench` also accepts a regular expression to match the benchmark functions we want to run. To run all the benchmark functions, we provide `-bench=.`. `go test` will first run all the tests (or those matched by `-run`, if provided), and then run the benchmarks. The output for our benchmark above looks is as follows:

```
$ go test -bench=.
goos: linux
goarch: amd64
BenchmarkF-4        1000        1255534 ns/op
PASS
ok      _/home/productiongo/benchmarks/fibonacci 1.387s
```

The output tells us that the benchmarks ran on a Linux x86-64 environment. Furthermore, the testing package executed our one benchmark, `BenchmarkF`. It ran the `b.N` loop 1000 times, and each iteration (i.e. each call to `F`) lasted 1,255,534ns (~1.2ms) on average.

1.2ms per call seems a bit slow! Especially considering that the numbers we provided to the Fibonacci function were quite small. Let's improve our original function by not using recursion.

An improved Fibonacci function

```
1  package fibonacci
2
3  // FastF returns the nth Fibonnaci number,
4  // but does not use recursion.
5  func FastF(n int) int {
6          var a, b int = 0, 1
7          for i := 0; i < n; i++ {
8                  a, b = b, a+b
9          }
10         return a
11 }
```

This new function `FastF`, is equivalent to the original, but uses only two variables and no recursion to calculate the final answer. Neat! Let's check whether it's actually any faster. We can do this by adding a new benchmark function for `FastF`:

```
func BenchmarkFastF(b *testing.B) {
    // run FastF(n) b.N times
    m := len(numbers)
    for n := 0; n < b.N; n++ {
        FastF(numbers[n%m])
    }
}
```

Again we run `go test -bench=.`. This time we will see the output of both benchmarks:

```
$ go test -bench=.
goos: linux
goarch: amd64
BenchmarkF-4            1000       1245008 ns/op
BenchmarkFastF-4    50000000            20.3 ns/op
PASS
ok      _/home/productiongo/benchmarks/fibonacci 2.444s
```

The output is telling us that `F` still took around 1245008ns per execution, but `FastF` took only 20.3ns! The benchmark proves that our non-recursive `FastF` is indeed orders of magnitude faster than the textbook recursive version, at least for the provided inputs.

Comparing benchmarks

The `benchcmp` tool parses the output of two `go test -bench` runs and compares the results.

To install, run:

```
go get golang.org/x/tools/cmd/benchcmp
```

Let's output the benchmark for the original `F` function from earlier to a file, using `BenchmarkF`:

```
$ go test -bench . > old.txt
```

The file will look as follows:

```
goos: darwin
goarch: amd64
BenchmarkF-4        1000       1965113 ns/op
PASS
ok      _/Users/productiongo/benchmarks/benchcmp   2.173s
```

Now instead of implementing `FastF`, we copy the `FastF` logic into our original `F` function:

Fast F implementation

```
package fibbonaci

// F returns the nth Fibonnaci number.
func F(n int) int {
        var a, b int = 0, 1
        for i := 0; i < n; i++ {
                a, b = b, a+b
        }
        return a
}
```

and re-run the benchmark, outputting to a file called `new.txt`:

```
$ go test -bench . > new.txt
```

`new.txt` should look like this:

```
goos: darwin
goarch: amd64
BenchmarkF-4    50000000            25.0 ns/op
PASS
ok      _/Users/productiongo/benchmarks/benchcmp    1.289s
```

Now let's run `benchcmp` on the results:

```
benchcmp old.txt new.txt
benchmark         old ns/op     new ns/op     delta
BenchmarkF-4      1965113       25.0          -100.00%
```

We can see the old performance, new performance, and a delta. In this case, the new version of `F` performs so well that it reduced the runtime of the original by 99.9987%. Thus rounded to two decimals, we get a delta of -100.00%.

Resetting benchmark timers

We can reset the benchmark timer if we don't want the overall benchmark timing to include the execution time of our setup code.

A benchmark from the `crypto/aes` package in the Go source code provides an example of this:

crypto/aes BenchmarkEncrypt

```
package aes

import "testing"

func BenchmarkEncrypt(b *testing.B) {
        tt := encryptTests[0]
        c, err := NewCipher(tt.key)
        if err != nil {
                b.Fatal("NewCipher:", err)
        }
        out := make([]byte, len(tt.in))
        b.SetBytes(int64(len(out)))
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                c.Encrypt(out, tt.in)
        }
}
```

As we can see, there is some setup done in the benchmark, then a call to `b.ResetTimer()` to reset the benchmark time and memory allocation counters.

Benchmarking memory allocations

The Go benchmarking tools also allow us to output the number memory allocations by the benchmark, alongside the time taken by each iteration. We do this by adding the `-benchmem` flag. Let's see what happens if we do this on our Fibonnaci benchmarks from before.

```
$ go test -bench=. -benchmem
goos: linux
goarch: amd64
BenchmarkF-4            1000       1241017 ns/op           0 B/op          0 \
allocs/op
BenchmarkFastF-4    100000000            20.6 ns/op         0 B/op          0 \
allocs/op
PASS
ok      _/Users/productiongo/benchmarks/fibonacci 3.453s
```

We now have two new columns on the right: the number of bytes per operation, and the number of heap allocations per operation. For our Fibonnaci functions, both of these are zero. Why is this? Let's add the `-gcflags=-m` option to see the details. The output below is truncated to the first 10 lines:

```
$ go test -bench=. -benchmem -gcflags=-m
# _/home/herman/Dropbox/mastergo/manuscript/code/benchmarks/fibonacci
./fibonacci_bench_test.go:10:20: BenchmarkF b does not escape
./fibonacci_fast_bench_test.go:6:24: BenchmarkFastF b does not escape
./fibonacci_test.go:22:5: t.common escapes to heap
./fibonacci_test.go:6:15: leaking param: t
./fibonacci_test.go:22:38: tt.n escapes to heap
./fibonacci_test.go:22:38: got escapes to heap
./fibonacci_test.go:22:49: tt.want escapes to heap
./fibonacci_test.go:10:3: TestF []struct { n int; want int } literal does not\
 escape
./fibonacci_test.go:22:12: TestF ... argument does not escape
...
```

The Go compiler performs escape analysis[13]. If an allocation does not escape the function, it can be stored on the stack. Variables placed on the stack avoid the costs involved with a heap allocation and the garbage collector. The omission of the `fibonacci.go` file from the output above implies that no variables from our F and FastF functions escaped to the heap. Let's take another look at the FastF function to see why this is:

```
func FastF(n int) int {
    var a, b int = 0, 1
    for i := 0; i < n; i++ {
        a, b = b, a+b
    }
    return a
}
```

In this function, the `a`, `b`, and `i` variables are declared locally and do not need to be put onto the heap, because they are not used again when the function exits. Consider what would happen if, instead of storing only the last two values, we naively stored all values calculated up to `n`:

[13] https://en.wikipedia.org/wiki/Escape_analysis

A high-memory implementation of F

```
package fibonacci

// FastHighMemF returns the nth Fibonacci number, but
// stores the full slice of intermediate
// results, consuming more memory than necessary.
func FastHighMemF(n int) int {
        if n <= 0 {
                return 0
        }

        r := make([]int, n+1)
        r[0] = 0
        r[1] = 1
        for i := 2; i <= n; i++ {
                r[i] = r[i-1] + r[i-2]
        }
        return r[n]
}
```

Running the same test and benchmark from before on this high-memory version of `F`, we get:

```
$ go test -bench=. -benchmem
goos: linux
goarch: amd64
BenchmarkFastHighMemF-4     20000000            72.0 ns/op       132 B/op     \
      0 allocs/op
PASS
ok      _/Users/productiongo/benchmarks/fibonacci_mem 1.518s
```

This time our function used 132 bytes per operation, due to our use of a slice in the function. If you are wondering why the number is 132 specifically: the exact number of bytes is sensitive to the numbers we use in the benchmark. The higher the input `n`, the more memory the function will allocate. The average of the values used in the benchmark (`0, 10, 20, 30`) is 15. Because this was compiled for a 64-bit machine, each `int` will use 8 bytes (8x8=64 bits). The slice headers also use some bytes. We still have zero heap allocations per operation, due to all variables being contained within the function. We will discuss advanced memory profiling and optimization techniques in Optimization.

Modulo vs Bitwise-and

In our Fibonacci benchmarks so far, we have made use of a list of four integer test cases:

```
6  var nums = []int{
7      0, 10, 20, 30,
8  }
```

which we then loop over in the BenchmarkF function:

```
13     for n := 0; n < b.N; n++ {
14         FastF(nums[n%m])
15     }
```

But when it comes down to the nanoseconds, modulo is a relatively slow computation to do on every iteration. It can actually have an impact on the accuracy of our results! Let's peek at the Go assembler code. Go allows us to do this with the `go tool compile -S` command, which outputs a pseudo-assembly language called ASM. In the command below, we filter the instructions for the line we are interested in with `grep`:

```
$ go tool compile -S fibonacci.go fibonacci_bench_test.go | grep "fibonacci_b\
ench_test.go:14"
    0x0036 00054 (fibonacci_bench_test.go:14)   MOVQ    (BX)(DX*8), AX
    0x003a 00058 (fibonacci_bench_test.go:14)   MOVQ    AX, (SP)
    0x003e 00062 (fibonacci_bench_test.go:14)   PCDATA  $0, $0
    0x003e 00062 (fibonacci_bench_test.go:14)   CALL    "".F(SB)
    0x0043 00067 (fibonacci_bench_test.go:14)   MOVQ    ""..autotmp_5+16(SP),\
 AX
    0x0061 00097 (fibonacci_bench_test.go:14)   MOVQ    "".numbers(SB), BX
    0x0068 00104 (fibonacci_bench_test.go:14)   MOVQ    "".numbers+8(SB), SI
    0x006f 00111 (fibonacci_bench_test.go:14)   TESTQ   CX, CX
    0x0072 00114 (fibonacci_bench_test.go:14)   JEQ 161
    0x0074 00116 (fibonacci_bench_test.go:14)   MOVQ    AX, DI
    0x0077 00119 (fibonacci_bench_test.go:14)   CMPQ    CX, $-1
    0x007b 00123 (fibonacci_bench_test.go:14)   JEQ 132
    0x007d 00125 (fibonacci_bench_test.go:14)   CQO
    0x007f 00127 (fibonacci_bench_test.go:14)   IDIVQ   CX
    0x0082 00130 (fibonacci_bench_test.go:14)   JMP 137
    0x0084 00132 (fibonacci_bench_test.go:14)   NEGQ    AX
```

```
    0x0087 00135 (fibonacci_bench_test.go:14)   XORL    DX, DX
    0x0089 00137 (fibonacci_bench_test.go:14)   CMPQ    DX, SI
    0x008c 00140 (fibonacci_bench_test.go:14)   JCS 49
    0x008e 00142 (fibonacci_bench_test.go:14)   JMP 154
    0x009a 00154 (fibonacci_bench_test.go:14)   PCDATA  $0, $1
    0x009a 00154 (fibonacci_bench_test.go:14)   CALL    runtime.panicindex(SB)
    0x009f 00159 (fibonacci_bench_test.go:14)   UNDEF
    0x00a1 00161 (fibonacci_bench_test.go:14)   PCDATA  $0, $1
    0x00a1 00161 (fibonacci_bench_test.go:14)   CALL    runtime.panicdivide(S\
B)
    0x00a6 00166 (fibonacci_bench_test.go:14)   UNDEF
    0x00a8 00168 (fibonacci_bench_test.go:14)   NOP
```

The details of this output are not as important as it is to notice how many instructions there are. Now, let's rewrite the code to use bitwise-and (`&`) instead of modulo `%`:

```
12      m := len(nums)-1
13      for n := 0; n < b.N; n++ {
14          FastF(nums[n&m])
15      }
```

Now, the ASM code becomes:

```
$ go tool compile -S fibonacci.go fibonacci_bench_test.go | grep "fibonacci_b\
ench_test.go:14"
    0x0032 00050 (fibonacci_bench_test.go:14)  MOVQ    (BX)(DI*8), AX
    0x0036 00054 (fibonacci_bench_test.go:14)   MOVQ    AX, (SP)
    0x003a 00058 (fibonacci_bench_test.go:14)   PCDATA  $0, $0
    0x003a 00058 (fibonacci_bench_test.go:14)   CALL    "".F(SB)
    0x003f 00063 (fibonacci_bench_test.go:14)   MOVQ    "".n+16(SP), AX
    0x005e 00094 (fibonacci_bench_test.go:14)   MOVQ    "".numbers(SB), BX
    0x0065 00101 (fibonacci_bench_test.go:14)   MOVQ    "".numbers+8(SB), SI
    0x006c 00108 (fibonacci_bench_test.go:14)   LEAQ    -1(AX), DI
    0x0070 00112 (fibonacci_bench_test.go:14)   ANDQ    CX, DI
    0x0073 00115 (fibonacci_bench_test.go:14)   CMPQ    DI, SI
    0x0076 00118 (fibonacci_bench_test.go:14)   JCS 45
    0x0078 00120 (fibonacci_bench_test.go:14)   JMP 132
    0x0084 00132 (fibonacci_bench_test.go:14)   PCDATA  $0, $1
    0x0084 00132 (fibonacci_bench_test.go:14)   CALL    runtime.panicindex(SB)
```

```
    0x0089 00137 (fibonacci_bench_test.go:14)   UNDEF
    0x008b 00139 (fibonacci_bench_test.go:14)   NOP
```

This is considerably shorter than before. In other words, the Go runtime will need to perform fewer operations, but the results will be the same. We can use modulo instead of ampersand because we have exactly four items in our `nums` slice. In general, $n\%m == n\&(m-1)$ if m is a power of two. For example,

```
0 % 4 = 0 & 3 = 0
1 % 4 = 1 & 3 = 1
2 % 4 = 2 & 3 = 2
3 % 4 = 3 & 3 = 3
4 % 4 = 4 & 3 = 0
5 % 4 = 5 & 3 = 1
6 % 4 = 6 & 3 = 2
...
```

If you are not yet convinced, expand the binary version of the bitwise-and operations to show that this is true:

```
0 & 3 = 00b & 11b = 0
1 & 3 = 01b & 11b = 1
2 & 3 = 10b & 11b = 2
3 & 3 = 11b & 11b = 3
...
```

To evaluate the impact of changing from modulo to ampersand on the benchmark results, let us create two benchmarks for `FastF`, one with modulo and the other with bitwise-and:

```
$ go test -bench=BenchmarkFastF
goos: linux
goarch: amd64
BenchmarkFastFModulo-4          100000000           16.7 ns/op
BenchmarkFastFBitwiseAnd-4      200000000            7.40 ns/op
PASS
ok      _/Users/productiongo/benchmarks/bench_bitwise_and 4.058s
```

The version using bitwise-and runs twice as fast. Our original benchmark was spending half the time recalculating modulo operations! This is unlikely to have a big impact on benchmarks of bigger functions, but when benchmarking small pieces of code, using bitwise-and instead of modulo will make the benchmark results more accurate.