CHAPTER 15

Writing Tests

Since the 2000s, the widespread adoption of automated testing has probably done more to improve code quality than any other software engineering technique. As a language and ecosystem focused on improving software quality, it's not surprising that Go includes testing support as part of its standard library. Go makes it so easy to test your code, there's no excuse to not do it.

In this chapter, you'll see how to test Go code, group tests into unit and integration tests, examine code coverage, write benchmarks, and learn how to check code for concurrency issues by using the Go data race detector. Along the way, I'll discuss how to write code that is testable and why this improves our code quality.

Understanding the Basics of Testing

Go's testing support has two parts: libraries and tooling. The `testing` package in the standard library provides the types and functions to write tests, while the `go test` tool that's bundled with Go runs your tests and generates reports. Unlike many other languages, Go places its tests in the same directory and the same package as the production code. Since tests are located in the same package, they are able to access and test unexported functions and variables. You'll see in a bit how to write tests that ensure that you are testing only a public API.

Complete code samples for this chapter are found in the Chapter 15 repository.

Let's write a simple function and then a test to make sure the function works. In the *sample_code/adder* directory, in the file *adder.go*, you have the following:

```
func addNumbers(x, y int) int {
    return x + x
}
```

The corresponding test is in *adder_test.go*:

```
func Test_addNumbers(t *testing.T) {
    result := addNumbers(2,3)
    if result != 5 {
        t.Error("incorrect result: expected 5, got", result)
    }
}
```

Every test is written in a file whose name ends with *_test.go*. If you are writing tests against *foo.go*, place your tests in a file named *foo_test.go*.

Test functions start with the word `Test` and take in a single parameter of type `*testing.T`. By convention, this parameter is named `t`. Test functions do not return any values. The name of the test (apart from starting with the word "Test") is meant to document what you are testing, so pick something that explains what you are testing. When writing unit tests for individual functions, the convention is to name the unit test `Test` followed by the name of the function. When testing unexported functions, some people use an underscore between the word `Test` and the name of the function.

Also note that you use standard Go code to call the code being tested and to validate that the responses are as expected. When there's an incorrect result, you report the error with the `t.Error` method, which works like the `fmt.Print` function. You'll see other error-reporting methods in a bit.

You've just seen the library portion of Go's test support. Now let's take a look at the tooling. Just as `go build` builds a binary and `go run` runs a program, the command `go test` runs the test functions in the current directory:

```
$ go test
--- FAIL: Test_addNumbers (0.00s)
    adder_test.go:8: incorrect result: expected 5, got 4
FAIL
exit status 1
FAIL    test_examples/adder     0.006s
```

It looks like you found a bug in your code. Taking a second look at `addNumbers`, you see that you are adding `x` to `x`, not `x` to `y`. Let's change the code and rerun the test to verify that the bug is fixed:

```
$ go test
PASS
ok      test_examples/adder     0.006s
```

The `go test` command allows you to specify which packages to test. Using `./...` for the package name specifies that you want to run tests in the current directory and all subdirectories of the current directory. Include a `-v` flag to get verbose testing output.

Reporting Test Failures

There are several methods on `*testing.T` for reporting test failures. You've already seen `Error`, which builds a failure description string out of a comma-separated list of values.

If you'd rather use a `Printf`-style formatting string to generate your message, use the `Errorf` method instead:

```
t.Errorf("incorrect result: expected %d, got %d", 5, result)
```

While `Error` and `Errorf` mark a test as failed, the test function continues running. If you think a test function should stop processing as soon as a failure is found, use the `Fatal` and `Fatalf` methods. The `Fatal` method works like `Error`, and the `Fatalf` method works like `Errorf`. The difference is that the test function exits immediately after the test failure message is generated. Note that this doesn't exit *all* tests; any remaining test functions will execute after the current test function exits.

When should you use `Fatal`/`Fatalf` and when should you use `Error`/`Errorf`? If the failure of a check in a test means that further checks in the same test function will always fail or cause the test to panic, use `Fatal` or `Fatalf`. If you are testing several independent items (such as validating fields in a struct), then use `Error` or `Errorf` so you can report many problems at once. This makes it easier to fix multiple problems without rerunning your tests over and over.

Setting Up and Tearing Down

Sometimes you have some common state that you want to set up before any tests run and remove when testing is complete. Use a `TestMain` function to manage this state and run your tests:

```
var testTime time.Time

func TestMain(m *testing.M) {
    fmt.Println("Set up stuff for tests here")
    testTime = time.Now()
    exitVal := m.Run()
    fmt.Println("Clean up stuff after tests here")
    os.Exit(exitVal)
}

func TestFirst(t *testing.T) {
    fmt.Println("TestFirst uses stuff set up in TestMain", testTime)
}
```

```
func TestSecond(t *testing.T) {
    fmt.Println("TestSecond also uses stuff set up in TestMain", testTime)
}
```

Both `TestFirst` and `TestSecond` refer to the package-level variable `testTime`. Assume that it needs to be initialized in order for the tests to run properly. You declare a function called `TestMain` with a parameter of type `*testing.M`. If there's a function named `TestMain` in a package, `go test` calls it instead of the test functions. It is the responsibility of the `TestMain` function to set up any state that's necessary to make the tests in the package run correctly.

Once the state is configured, the `TestMain` function calls the `Run` method on `*testing.M`. This runs the test functions in the package. The `Run` method returns the exit code; `0` indicates that all tests passed. Finally, the `TestMain` function must call `os.Exit` with the exit code returned from `Run`.

Running `go test` on this produces the output:

```
$ go test
Set up stuff for tests here
TestFirst uses stuff set up in TestMain 2020-09-01 21:42:36.231508 -0400 EDT
    m=+0.000244286
TestSecond also uses stuff set up in TestMain 2020-09-01 21:42:36.231508 -0400
    EDT m=+0.000244286
PASS
Clean up stuff after tests here
ok      test_examples/testmain  0.006s
```

Be aware that `TestMain` is invoked once, not before and after each individual test. Also be aware that you can have only one `TestMain` per package.

`TestMain` is useful in two common situations:

- When you need to set up data in an external repository, such as a database
- When the code being tested depends on package-level variables that need to be initialized

As mentioned before (and will be again!), you should avoid package-level variables in your programs. They make it hard to understand how data flows through your program. If you are using `TestMain` for this reason, consider refactoring your code.

The `Cleanup` method on `*testing.T` is used to clean up temporary resources created for a single test. This method has a single parameter, a function with no input parameters or return values. The function runs when the test completes. For simple tests,

you can achieve the same result by using a `defer` statement, but `Cleanup` is useful when tests rely on helper functions to set up sample data, as you see in Example 15-1. It's fine to call `Cleanup` multiple times. Just like `defer`, the functions are invoked in last-added, first-called order.

Example 15-1. Using `t.Cleanup`

```
// createFile is a helper function called from multiple tests
func createFile(t *testing.T) (_ string, err error) {
    f, err := os.Create("tempFile")
    if err != nil {
        return "", err
    }
    defer func() {
        err = errors.Join(err, f.Close())
    }()
    // write some data to f
    t.Cleanup(func() {
        os.Remove(f.Name())
    })
    return f.Name(), nil
}

func TestFileProcessing(t *testing.T) {
    fName, err := createFile(t)
    if err != nil {
        t.Fatal(err)
    }
    // do testing, don't worry about cleanup
}
```

If your test uses temporary files, you can avoid writing cleanup code by taking advantage of the `TempDir` method on `*testing.T`. This method creates a new temporary directory every time it is invoked and returns the full path of the directory. It also registers a handler with `Cleanup` to delete the directory and its contents when the test has completed. You can use it to rewrite the previous example:

```
// createFile is a helper function called from multiple tests
func createFile(tempDir string) (_ string, err error) {
    f, err := os.CreateTemp(tempDir, "tempFile")
    if err != nil {
        return "", err
    }
    defer func() {
        err = errors.Join(err, f.Close())
    }()
    // write some data to f
    return f.Name(), nil
}
```

```
func TestFileProcessing(t *testing.T) {
    tempDir := t.TempDir()
    fName, err := createFile(tempDir)
    if err != nil {
        t.Fatal(err)
    }
    // do testing, don't worry about cleanup
}
```

Testing with Environment Variables

It's a common (and very good) practice to configure applications with environment variables. To help you test your environment-variable–parsing code, Go provides a helper method on `testing.T`. Call `t.Setenv()` to register a value for an environment variable for your test. Behind the scenes, it calls `Cleanup` to revert the environment variable to its previous state when the test exits:

```
// assume ProcessEnvVars is a function that processes environment variables
// and returns a struct with an OutputFormat field
func TestEnvVarProcess(t *testing.T) {
    t.Setenv("OUTPUT_FORMAT", "JSON")
    cfg := ProcessEnvVars()
    if cfg.OutputFormat != "JSON" {
        t.Error("OutputFormat not set correctly")
    }
    // value of OUTPUT_FORMAT is reset when the test function exits
}
```

While it is good to use environment variables to configure your application, it is also good to make sure that most of your code is entirely unaware of them. Be sure to copy the values of environment variables into configuration structs before your program starts its work, in your `main` function or soon afterward. Doing so makes it easier to reuse and test code, since *how* the code is configured has been abstracted away from *what* the code is doing.

Rather than writing this code yourself, you should strongly consider using a third-party configuration library, like Viper or envconfig. Also, look at GoDotEnv as a way to store environment variables in *.env* files for development or continuous integration machines.

Storing Sample Test Data

As `go test` walks your source code tree, it uses the current package directory as the current working directory. If you want to use sample data to test functions in a package, create a subdirectory named *testdata* to hold your files. Go reserves this directory name as a place to hold test files. When reading from *testdata*, always use

a relative file reference. Since `go test` changes the current working directory to the current package, each package accesses its own *testdata* via a relative file path.

The `text` package in the *sample_code* directory of the Chapter 15 repository demonstrates how to use *testdata*.

Caching Test Results

Just as you learned in Chapter 10 that Go caches compiled packages if they haven't changed, Go also caches test results when running tests across multiple packages if they have passed and their code hasn't changed. The tests are recompiled and rerun if you change any file in the package or in the *testdata* directory. You can also force tests to always run if you pass the flag `-count=1` to `go test`.

Testing Your Public API

The tests that you've written are in the same package as the production code. This allows you to test both exported and unexported functions.

If you want to test just the public API of your package, Go has a convention for specifying this. You still keep your test source code in the same directory as the production source code, but you use `packagename_test` for the package name. Let's redo the initial test case, using an exported function instead. You can find the code in the *sample_code/pubadder* directory in the Chapter 15 repository. If you have the following function in the `pubadder` package

```
func AddNumbers(x, y int) int {
    return x + y
}
```

then you can test it as public API by using the following code in a file in the `pubadder` package named *adder_public_test.go*:

```
package pubadder_test

import (
    "github.com/learning-go-book-2e/ch15/sample_code/pubadder"
    "testing"
)

func TestAddNumbers(t *testing.T) {
    result := pubadder.AddNumbers(2, 3)
    if result != 5 {
        t.Error("incorrect result: expected 5, got", result)
    }
}
```

Notice that the package name for the test file is `pubadder_test`. You have to import `github.com/learning-go-book-2e/ch15/sample_code/pubadder` even though the files are in the same directory. To follow the convention for naming tests, the test function name matches the name of the `AddNumbers` function. Also note that you use `pubadder.AddNumbers`, since you are calling an exported function in a different package.

If you are typing this code in by hand, you'll need to create a module with a *go.mod* file that has the module declaration:

```
module github.com/learning-go-book-2e/ch15
```

and that puts the source code in the *sample_code/pubadder* directory within the module.

Just as you can call exported functions from within a package, you can test your public API from a test that is in the same package as your source code. The advantage of using the `_test` suffix in the package name is that it lets you treat your tested package as a "black box." You are forced to interact with it only via its exported functions, methods, types, constants, and variables. Also be aware that you can have test source files with both package names intermixed in the same source directory.

Using go-cmp to Compare Test Results

Writing a thorough comparison of a compound type's two instances can be verbose. While you can use `reflect.DeepEqual` to compare structs, maps, and slices, there's a better way. Google released a third-party module called `go-cmp` that does the comparison for you and returns a detailed description of what does not match. Let's see how it works by defining a simple `struct` and a factory function that populates it. You can find this code in the *sample_code/cmp* directory in the Chapter 15 repository:

```
type Person struct {
    Name      string
    Age       int
    DateAdded time.Time
}

func CreatePerson(name string, age int) Person {
    return Person{
        Name:      name,
        Age:       age,
        DateAdded: time.Now(),
    }
}
```

In your test file, you need to import `github.com/google/go-cmp/cmp`, and your test function looks like this:

```
func TestCreatePerson(t *testing.T) {
    expected := Person{
        Name: "Dennis",
        Age:  37,
    }
    result := CreatePerson("Dennis", 37)
    if diff := cmp.Diff(expected, result); diff != "" {
        t.Error(diff)
    }
}
```

The `cmp.Diff` function takes in the expected output and the output that was returned by the function that you're testing. It returns a string that describes any mismatches between the two inputs. If the inputs match, it returns an empty string. You assign the output of the `cmp.Diff` function to a variable called `diff` and then check whether `diff` is an empty string. If it is not, an error occurred.

When you build and run the test, you'll see the output that `go-cmp` generates when two struct instances don't match:

```
$ go test
--- FAIL: TestCreatePerson (0.00s)
    ch13_cmp_test.go:16:   ch13_cmp.Person{
              Name:      "Dennis",
              Age:       37,
        -     DateAdded: s"0001-01-01 00:00:00 +0000 UTC",
        +     DateAdded: s"2020-03-01 22:53:58.087229 -0500 EST m=+0.001242842",
          }

FAIL
FAIL    ch13_cmp    0.006s
```

The lines with a `-` and `+` indicate the fields whose values differ. The test failed because the dates didn't match. This is a problem because you can't control what date is assigned by the `CreatePerson` function. You have to ignore the `DateAdded` field. You do that by specifying a comparator function. Declare the function as a local variable in your test:

```
comparer := cmp.Comparer(func(x, y Person) bool {
    return x.Name == y.Name && x.Age == y.Age
})
```

Pass a function to the `cmp.Comparer` function to create a customer comparator. The function that's passed in must have two parameters of the same type and return a `bool`. It also must be symmetric (the order of the parameters doesn't matter), deterministic (it always returns the same value for the same inputs), and pure (it must

not modify its parameters). In your implementation, you are comparing the `Name` and `Age` fields and ignoring the `DateAdded` field.

Then change your call to `cmp.Diff` to include `comparer`:

```
if diff := cmp.Diff(expected, result, comparer); diff != "" {
    t.Error(diff)
}
```

This is only a quick preview of the most useful features in `go-cmp`. Check its documentation to learn more about how to control what is compared and the output format.

Running Table Tests

Most of the time, it takes more than a single test case to validate that a function is working correctly. You could write multiple test functions to validate your function or multiple tests within the same function, but you'll find that a great deal of the testing logic is repetitive. You set up supporting data and functions, specify inputs, check the outputs, and compare to see if they match your expectations.

Rather than writing this over and over, you can take advantage of a pattern called *table tests*. Let's take a look at a sample. You can find this code in the *sample_code/table* directory in the Chapter 15 repository. Assume you have the following function in the `table` package:

```
func DoMath(num1, num2 int, op string) (int, error) {
    switch op {
    case "+":
        return num1 + num2, nil
    case "-":
        return num1 - num2, nil
    case "*":
        return num1 + num2, nil
    case "/":
        if num2 == 0 {
            return 0, errors.New("division by zero")
        }
        return num1 / num2, nil
    default:
        return 0, fmt.Errorf("unknown operator %s", op)
    }
}
```

To test this function, you need to check the different branches, trying out inputs that return valid results, as well as inputs that trigger errors. You could write code like this, but it's very repetitive:

```
func TestDoMath(t *testing.T) {
    result, err := DoMath(2, 2, "+")
```

```
    if result != 4 {
        t.Error("Should have been 4, got", result)
    }
    if err != nil {
        t.Error("Should have been nil error, got", err)
    }
    result2, err2 := DoMath(2, 2, "-")
    if result2 != 0 {
        t.Error("Should have been 0, got", result2)
    }
    if err2 != nil {
        t.Error("Should have been nil error, got", err2)
    }
    // and so on...
}
```

Let's replace this repetition with a table test. First, you declare a slice of anonymous structs. The struct contains fields for the name of the test, the input parameters, and the return values. Each entry in the slice represents another test:

```
data := []struct {
    name     string
    num1     int
    num2     int
    op       string
    expected int
    errMsg   string
}{
    {"addition", 2, 2, "+", 4, ""},
    {"subtraction", 2, 2, "-", 0, ""},
    {"multiplication", 2, 2, "*", 4, ""},
    {"division", 2, 2, "/", 1, ""},
    {"bad_division", 2, 0, "/", 0, `division by zero`},
}
```

Next, loop over each test case in `data`, invoking the `Run` method each time. This is the line that does the magic. You pass two parameters to `Run`, a name for the subtest and a function with a single parameter of type `*testing.T`. Inside the function, you call `DoMath` by using the fields of the current entry in `data`, using the same logic over and over. When you run these tests, you'll see that not only do they pass, but when you use the `-v` flag, each subtest also now has a name:

```
for _, d := range data {
    t.Run(d.name, func(t *testing.T) {
        result, err := DoMath(d.num1, d.num2, d.op)
        if result != d.expected {
            t.Errorf("Expected %d, got %d", d.expected, result)
        }
        var errMsg string
        if err != nil {
            errMsg = err.Error()
        }
```

```
        if errMsg != d.errMsg {
            t.Errorf("Expected error message `%s`, got `%s`",
                d.errMsg, errMsg)
        }
    })
}
```

Comparing error messages can be fragile, because there may not be any compatibility guarantees on the message text. The function that you are testing uses `errors.New` and `fmt.Errorf` to make errors, so the only option is to compare the messages. If an error has a custom type or a named sentinel error, use `errors.Is` or `errors.As` to check that the correct error is returned.

Running Tests Concurrently

By default, unit tests are run sequentially. Since each unit test is supposed to be independent from every other unit test, they make ideal candidates for concurrency. To make a unit test run concurrently with other tests, call the `Parallel` method on `*testing.T` as the first line in your test:

```
func TestMyCode(t *testing.T) {
    t.Parallel()
    // rest of test goes here
}
```

Parallel tests run concurrently with other tests marked as parallel.

The advantage of parallel tests is that it can speed up long-running test suites. There are some disadvantages, though. If you have multiple tests that rely on the same shared mutable state, do not mark them as parallel, because you will get inconsistent results. (But after all of my warnings, you don't have any shared mutable state in your application, right?) Also be aware that your test will panic if you mark it as parallel and use the `Setenv` method in your test function.

Be careful when running table tests in parallel. When table tests run in parallel, it's just as you saw in "Goroutines, for Loops, and Varying Variables" on page 298, where you launched multiple goroutines within a `for` loop. If you run this example using Go 1.21 or earlier (or on Go 1.22 or later with the Go version set to 1.21 or earlier in the `go` directive in the *go.mod* file), a reference to the variable `d` is shared by all the parallel tests, so they all see the same value:

```
func TestParallelTable(t *testing.T) {
    data := []struct {
        name   string
        input  int
        output int
    }{
```

```
        {"a", 10, 20},
        {"b", 30, 40},
        {"c", 50, 60},
    }
    for _, d := range data {
        t.Run(d.name, func(t *testing.T) {
            t.Parallel()
            fmt.Println(d.input, d.output)
            out := toTest(d.input)
            if out != d.output {
                t.Error("didn't match", out, d.output)
            }
        })
    }
}
```

You can run this code on The Go Playground or in the *sample_code/parallel* directory in the Chapter 15 repository. Take a look at the output and you'll see that you test the last value in the table test three times:

```
=== CONT  TestParallelTable/a
50 60
=== CONT  TestParallelTable/c
50 60
=== CONT  TestParallelTable/b
50 60
```

This problem is common enough that Go 1.20 added a `go vet` check for this problem. When you run `go vet` on this code, you get the message `loop variable d captured by func literal` on each line where `d` is captured.

The `for` loop changes in Go 1.22 or later resolve this issue. If you cannot use Go 1.22, you can avoid this bug by shadowing `d` within the `for` loop before invoking `t.Run`:

```
    for _, d := range data {
        d := d // THIS IS THE LINE THAT SHADOWS d!
        t.Run(d.name, func(t *testing.T) {
            t.Parallel()
            fmt.Println(d.input, d.output)
            out := toTest(d.input)
            if out != d.output {
                t.Error("didn't match", out, d.output)
            }
        })
    }
```

Now that you have a way to run lots of tests, you'll learn about code coverage to find out what your tests are testing.

Checking Your Code Coverage

Code coverage is a very useful tool for knowing whether you've missed any obvious cases. However, reaching 100% code coverage doesn't guarantee that there aren't bugs in your code for some inputs. First you'll see how `go test` displays code coverage and then you'll look at the limitations of relying on code coverage alone.

Adding the `-cover` flag to the `go test` command calculates coverage information and includes a summary in the test output. If you include a second flag `-coverprofile`, you can save the coverage information to a file. Let's go back to the *sample_code/table* directory in the Chapter 15 repository and gather code coverage information:

```
$ go test -v -cover -coverprofile=c.out
```

If you run your table test with code coverage, the test output now includes a line that indicates the amount of test code coverage, 87.5%. That's good to know, but it'd be more useful if you could see what you missed. The `cover` tool included with Go generates an HTML representation of your source code with that information:

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

When you run it, your web browser should open and show you a page that looks like Figure 15-1.

Every file that's tested appears in the combo box in the upper left. The source code is in one of three colors. Gray is used for lines of code that aren't testable, green is used for code that's been covered by a test, and red is used for code that hasn't been tested. (The reliance on color is unfortunate for readers of the print edition and those who have red-green color blindness. If you are unable to see the colors, the lighter gray is the covered lines.) From looking at this, you can see that you didn't write a test to cover the default case, when a bad operator is passed to your function. Add that case to your slice of test cases:

```
{"bad_op", 2, 2, "?", 0, `unknown operator ?`},
```

When you rerun `go test -v -cover -coverprofile=c.out` and `go tool cover -html=c.out`, you see in Figure 15-2 that the final line is covered and you have 100% test code coverage.

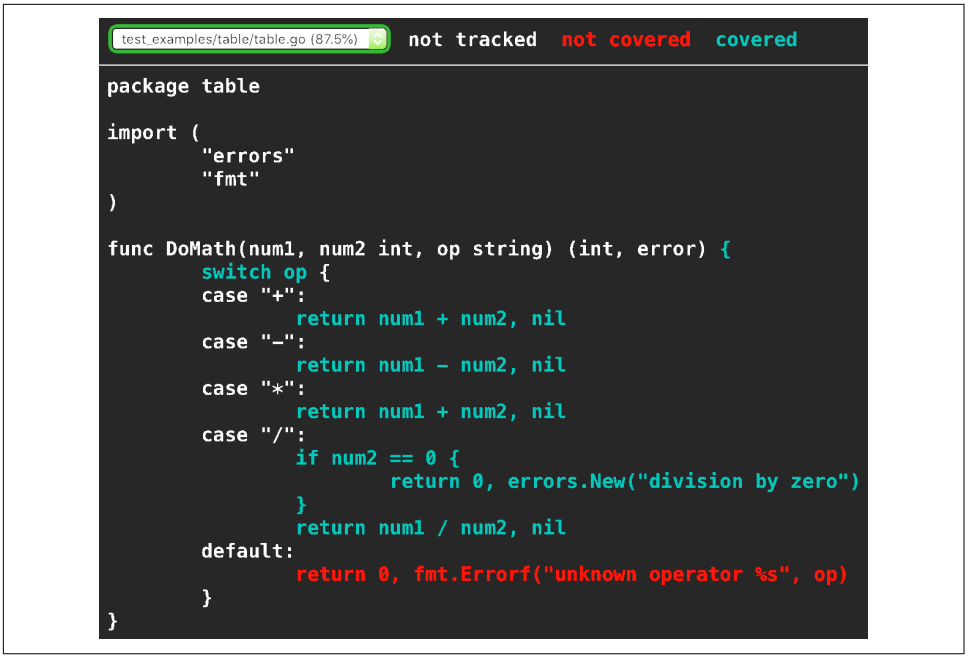

Figure 15-1. Initial code coverage

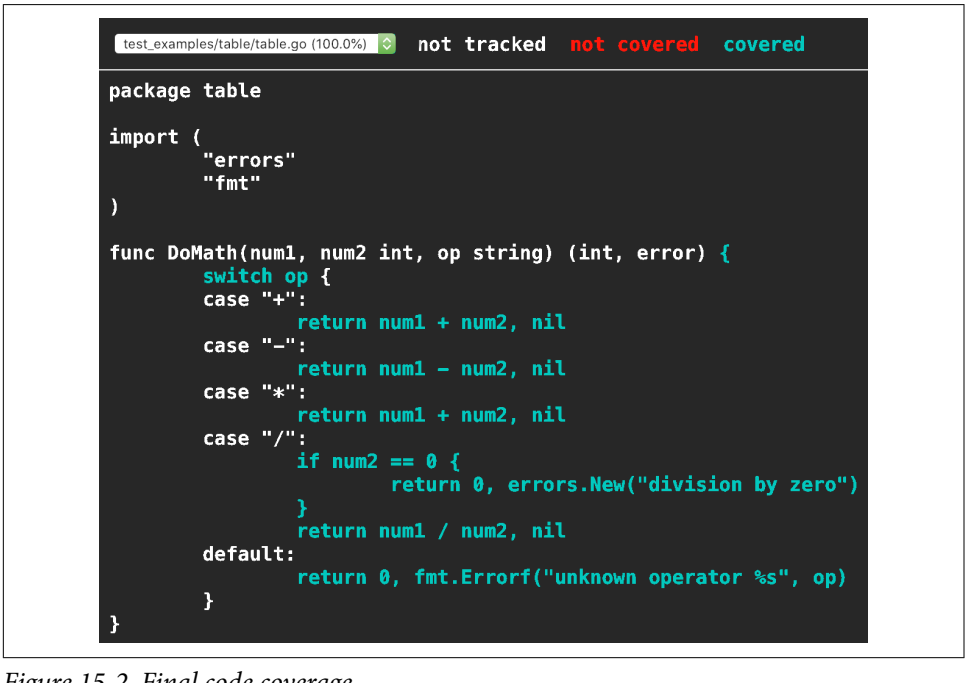

Figure 15-2. Final code coverage

Code coverage is a great thing, but it's not enough. There's actually a bug in your code, even though you have 100% coverage. Have you noticed it? If not, add another test case and rerun your tests:

```
{"another_mult", 2, 3, "*", 6, ""},
```

You should see the error:

```
table_test.go:57: Expected 6, got 5
```

Your case for multiplication has a typo. It adds the numbers together instead of multiplying them. (Beware the dangers of copy-and-paste coding!) Fix the code, rerun `go test -v -cover -coverprofile=c.out` and `go tool cover -html=c.out`, and you'll see that the tests pass again.

Code coverage is necessary but not sufficient. You can have 100% code coverage and still have bugs in your code!

Fuzzing

One of the most important lessons that every developer eventually learns is that all data is suspect. No matter how well specified a data format is, you will eventually have to process input that doesn't match what you expect. This doesn't happen for only malicious reasons. Data can be corrupted in transit, in storage, and even in memory. Programs that process data might have bugs, and specifications for data formats always have corner cases that can be interpreted differently by different developers.

Even when developers do the work of writing good unit tests, it's impossible to think of everything. As you've seen, even 100% unit test code coverage is no guarantee that your code is bug free. You need to supplement unit tests with generated data that could break your program in ways you didn't anticipate. That's where fuzzing helps.

Fuzzing is a technique for generating random data and submitting it to code to see whether it properly handles unexpected input. The developer can provide a *seed corpus* or set of known good data, and the fuzzer uses that as a basis for generating bad input. Let's see how to use the fuzzing support in Go's testing tools to discover additional test cases.

Assume that you are writing a program to processes data files. You can find the code for this example on GitHub. You are sending a list of strings but want to efficiently allocate memory, so the number of strings in a file is sent as the first line, while the remaining lines are the lines of text. Here's the sample function for processing this data:

```
func ParseData(r io.Reader) ([]string, error) {
    s := bufio.NewScanner(r)
    if !s.Scan() {
        return nil, errors.New("empty")
    }
    countStr := s.Text()
    count, err := strconv.Atoi(countStr)
    if err != nil {
        return nil, err
    }
    out := make([]string, 0, count)
    for i := 0; i < count; i++ {
        hasLine := s.Scan()
        if !hasLine {
            return nil, errors.New("too few lines")
        }
        line := s.Text()
        out = append(out, line)
    }
    return out, nil
}
```

You use a `bufio.Scanner` to read line-by-line from an `io.Reader`. If there's no data to be read, an error is returned. You then read the first line and attempt to convert it to an int named `count`. If you can't, an error is returned. Next, you allocate the memory for your slice of strings, and read `count` number of lines from the scanner. If there aren't enough lines, an error is returned. If all goes well, you return the lines you read.

A unit test has already been written to validate the code:

```
func TestParseData(t *testing.T) {
    data := []struct {
        name   string
        in     []byte
        out    []string
        errMsg string
    }{
        {
            name:   "simple",
            in:     []byte("3\nhello\ngoodbye\ngreetings\n"),
            out:    []string{"hello", "goodbye", "greetings"},
            errMsg: "",
        },
        {
            name:   "empty_error",
            in:     []byte(""),
            out:    nil,
            errMsg: "empty",
        },
        {
            name:   "zero",
            in:     []byte("0\n"),
```

```
            out:    []string{},
            errMsg: "",
        },
        {
            name:   "number_error",
            in:     []byte("asdf\nhello\ngoodbye\ngreetings\n"),
            out:    nil,
            errMsg: `strconv.Atoi: parsing "asdf": invalid syntax`,
        },
        {
            name:   "line_count_error",
            in:     []byte("4\nhello\ngoodbye\ngreetings\n"),
            out:    nil,
            errMsg: "too few lines",
        },
    }
    for _, d := range data {
        t.Run(d.name, func(t *testing.T) {
            r := bytes.NewReader(d.in)
            out, err := ParseData(r)
            var errMsg string
            if err != nil {
                errMsg = err.Error()
            }
            if diff := cmp.Diff(d.out, out); diff != "" {
                t.Error(diff)
            }
            if diff := cmp.Diff(d.errMsg, errMsg); diff != "" {
                t.Error(diff)
            }
        })
    }
}
```

The unit test has 100% line coverage for `ParseData`, handling all its error cases. You might think the code is ready for production, but let's see if fuzzing can help you discover errors that you hadn't considered.

Be aware that fuzzing uses a lot of resources. A fuzz test can allocate (or attempt to allocate) many gigabytes of memory and might write several gigabytes of data to your local disk. If you are running something else on the same machine at the same time as a fuzz test, don't be surprised if it slows down.

You start by writing a fuzz test:

```
func FuzzParseData(f *testing.F) {
    testcases := [][]byte{
        []byte("3\nhello\ngoodbye\ngreetings\n"),
        []byte("0\n"),
    }
```

```
    for _, tc := range testcases {
        f.Add(tc)
    }
    f.Fuzz(func(t *testing.T, in []byte) {
        r := bytes.NewReader(in)
        out, err := ParseData(r)
        if err != nil {
            t.Skip("handled error")
        }
        roundTrip := ToData(out)
        rtr := bytes.NewReader(roundTrip)
        out2, err := ParseData(rtr)
        if diff := cmp.Diff(out, out2); diff != "" {
            t.Error(diff)
        }
    })
}
```

A fuzz test looks similar to a standard unit test. The function name starts with `Fuzz`, the only parameter is of type `*testing.F`, and it has no return values.

Next, you set up a seed corpus, which is composed of one or more sets of sample data. The data can run successfully, error out, or even panic. The important things are that you know how your program behaves when this data is provided and that this behavior is accounted for by your fuzz test. These samples are mutated by the fuzzer to generate bad inputs. This example uses only a single field of data for each entry (a slice of bytes), but you can have as many fields as needed. The fields in a corpus entry are currently limited to only certain types:

- Any integer type (including unsigned types, `rune`, and `byte`)
- Any floating-point type
- `bool`
- `string`
- `[]byte`

Each entry in the corpus is passed to the `Add` method on the `*testing.F` instance. In the example, you have a slice of bytes for each entry:

```
f.Add(tc)
```

If the function being fuzzed needed an `int` and a `string`, the call to `Add` would look like this:

```
f.Add(1, "some text")
```

It's a runtime error to pass a value of an invalid type to `Add`.

Next you call the `Fuzz` method on your `*testing.F` instance. This looks a bit like a call to `Run` when writing a table test in a standard unit test. `Fuzz` takes in a single parameter, a function whose first parameter is of type `*testing.T` and whose remaining parameters exactly match the types, order, and count of the values that were passed in to `Add`. This also specifies the type of data generated by the fuzzing engine during fuzzing. There's no way for the Go compiler to enforce this constraint, so it's a runtime error if the convention is not followed.

Finally, let's look at the body of the fuzz target. Remember, fuzzing is used to find cases where bad input is not handled correctly. Since the input is randomly generated, you can't write tests that have knowledge of what the output should be. Instead, you have to use test conditions that will be true for all inputs. In the case of `ParseData`, you can check for two things:

- Does the code return an error for bad input, or does it panic?
- If you convert the slice of strings back to a slice of bytes and re-parse it, do you get the same result?

Let's see what happens when you run the fuzz test:

```
$ go test -fuzz=FuzzParseData
fuzz: elapsed: 0s, gathering baseline coverage: 0/243 completed
fuzz: elapsed: 0s, gathering baseline coverage: 243/243 completed,
    now fuzzing with 8 workers
fuzz: minimizing 289-byte failing input file
fuzz: elapsed: 3s, minimizing
fuzz: elapsed: 6s, minimizing
fuzz: elapsed: 9s, minimizing
fuzz: elapsed: 10s, minimizing
--- FAIL: FuzzParseData (10.48s)
    fuzzing process hung or terminated unexpectedly while minimizing: EOF
    Failing input written to testdata/fuzz/FuzzParseData/
        fedbaf01dc50bf41b40d7449657cdc9af9868b1be0421c98b2910071de9be3df
    To re-run:
    go test -run=FuzzParseData/
        fedbaf01dc50bf41b40d7449657cdc9af9868b1be0421c98b2910071de9be3df
FAIL
exit status 1
FAIL    file_parser     10.594s
```

If you don't specify the `-fuzz` flag, your fuzz tests will be treated like unit tests and will be run against the seed corpus. Only a single fuzz test can be fuzzed at a time.

If you want to get the full experience, delete the contents of the *testdata/fuzz/FuzzParseData* directory. This will cause the fuzzer to generate new seed corpus entries. Since the fuzzer generates random input, your samples might be different from the ones shown. The different entries will likely produce similar errors, though maybe not in the same order.

The fuzz test runs for several seconds and then fails. In this case, the `go` command reports that it has crashed. You don't want programs that crash, so let's look at the input that was generated. Every time a test case fails, the fuzzer writes it to the *testdata/fuzz/TESTNAME* subdirectory in the same package as the failed test, adding a new entry to the seed corpus. The new seed corpus entry in the file now becomes a new unit test, one that was automatically generated by the fuzzer. It is run anytime `go test` runs the `FuzzParseData` function, and acts as a regression test once you fix your bug.

Here are the contents of the file:

```
go test fuzz v1
[]byte("300000000000")
```

The first line is a header to indicate that this is test data for a fuzz test. The subsequent lines have the data that caused the failure.

The failure message tells you how to isolate this failing case when you rerun the test:

```
$ go test -run=FuzzParseData/
    fedbaf01dc50bf41b40d7449657cdc9af9868b1be0421c98b2910071de9be3df
signal: killed
FAIL    file_parser     15.046s
```

The problem is that you are trying to allocate a slice with the capacity to hold 300,000,000,000 strings. That requires quite a bit more RAM than my computer (and probably yours) has. You need to limit the number of expected text elements to a reasonable number. Set the maximum number of rows to 1,000 by adding the following code to `ParseData` after you parse the number of expected rows:

```
    if count > 1000 {
        return nil, errors.New("too many")
    }
```

Run the fuzzer again to see whether it finds any more errors:

```
$ go test -fuzz=FuzzParseData
fuzz: elapsed: 0s, gathering baseline coverage: 0/245 completed
fuzz: elapsed: 0s, gathering baseline coverage: 245/245 completed,
    now fuzzing with 8 workers
fuzz: minimizing 29-byte failing input file
fuzz: elapsed: 2s, minimizing
--- FAIL: FuzzParseData (2.20s)
```

```
    --- FAIL: FuzzParseData (0.00s)
        testing.go:1356: panic: runtime error: makeslice: cap out of range
            goroutine 23027 [running]:
            runtime/debug.Stack()
                /usr/local/go/src/runtime/debug/stack.go:24 +0x104
            testing.tRunner.func1()
                /usr/local/go/src/testing/testing.go:1356 +0x258
            panic({0x1003f9920, 0x10042a260})
                /usr/local/go/src/runtime/panic.go:884 +0x204
            file_parser.ParseData({0x10042a7c8, 0x14006c39bc0})
                file_parser/file_parser.go:24 +0x254
[...]
    Failing input written to testdata/fuzz/FuzzParseData/
        03f81b404ad91d092a482ad1ccb4a457800599ab826ec8dae47b49c01c38f7b1
    To re-run:
    go test -run=FuzzParseData/
        03f81b404ad91d092a482ad1ccb4a457800599ab826ec8dae47b49c01c38f7b1
FAIL
exit status 1
FAIL    file_parser     2.434s
```

This time you get a fuzz result that produces a panic. Looking at the file generated by `go fuzz`, you see this:

```
go test fuzz v1
[]byte("-1")
```

The line that's generating the panic is shown here:

```
    out := make([]string, 0, count)
```

You are trying to create a slice with negative capacity, which panics. Add another condition to your code to look for negative numbers:

```
    if count < 0 {
        return nil, errors.New("no negative numbers")
    }
```

Run your fuzzer again, and it generates another error:

```
$ go test -fuzz=FuzzParseData
fuzz: elapsed: 0s, gathering baseline coverage: 0/246 completed
fuzz: elapsed: 0s, gathering baseline coverage: 246/246 completed,
    now fuzzing with 8 workers
fuzz: elapsed: 3s, execs: 288734 (96241/sec), new interesting: 0 (total: 246)
fuzz: elapsed: 6s, execs: 418803 (43354/sec), new interesting: 0 (total: 246)
fuzz: minimizing 34-byte failing input file
fuzz: elapsed: 7s, minimizing
--- FAIL: FuzzParseData (7.43s)
    --- FAIL: FuzzParseData (0.00s)
        file_parser_test.go:89:   []string{
            -   "\r",
            +   "",
              }
```

```
    Failing input written to testdata/fuzz/FuzzParseData/
        b605c41104bf41a21309a13e90cfc6f30ecf133a2382759f2abc34d41b45ae79
    To re-run:
    go test -run=FuzzParseData/
        b605c41104bf41a21309a13e90cfc6f30ecf133a2382759f2abc34d41b45ae79
FAIL
exit status 1
FAIL    file_parser     7.558s
```

Looking at the file it created, you are generating blank lines with only `\r` (return) characters. Blank lines are not what you expect in your input, so add some code to the loop that reads lines of text from the `Scanner`. You'll check whether a line consists only of whitespace characters. If it does, return an error:

```
        line = strings.TrimSpace(line)
        if len(line) == 0 {
            return nil, errors.New("blank line")
        }
```

Run your fuzzer yet again:

```
$ go test -fuzz=FuzzParseData
fuzz: elapsed: 0s, gathering baseline coverage: 0/247 completed
fuzz: elapsed: 0s, gathering baseline coverage: 247/247 completed,
    now fuzzing with 8 workers
fuzz: elapsed: 3s, execs: 391018 (130318/sec), new interesting: 2 (total: 249)
fuzz: elapsed: 6s, execs: 556939 (55303/sec), new interesting: 2 (total: 249)
fuzz: elapsed: 9s, execs: 622126 (21734/sec), new interesting: 2 (total: 249)
[...]
fuzz: elapsed: 2m0s, execs: 2829569 (0/sec), new interesting: 16 (total: 263)
fuzz: elapsed: 2m3s, execs: 2829569 (0/sec), new interesting: 16 (total: 263)
^Cfuzz: elapsed: 2m4s, execs: 2829569 (0/sec), new interesting: 16 (total: 263)
PASS
ok      file_parser     123.662s
```

After a few minutes, no more errors are found, so hit Ctrl-C to end fuzzing.

Just because the fuzzer didn't find additional issues doesn't mean that the code is now bug free. However, fuzzing allowed you to automatically find some significant oversights in the original code. Writing fuzz tests takes practice, as they require a slightly different mindset than writing unit tests. Once you get the hang of them, they become an essential tool for validating how your code handles unexpected user input.

Using Benchmarks

Determining how fast (or slow) code runs is surprisingly difficult. Rather than trying to figure it out yourself, you should use the benchmarking support that's built into Go's testing framework. Let's explore it with a function in the *sample_code/bench* directory in the Chapter 15 repository:

```
func FileLen(f string, bufsize int) (int, error) {
    file, err := os.Open(f)
    if err != nil {
        return 0, err
    }
    defer file.Close()
    count := 0
    for {
        buf := make([]byte, bufsize)
        num, err := file.Read(buf)
        count += num
        if err != nil {
            break
        }
    }
    return count, nil
}
```

This function counts the number of characters in a file. It takes in two parameters, the name of the file and the size of the buffer that you are using to read the file (you'll see the reason for the second parameter in a moment).

Before seeing how fast it is, you should test your library to make sure that it works (it does). Here's a simple test:

```
func TestFileLen(t *testing.T) {
    result, err := FileLen("testdata/data.txt", 1)
    if err != nil {
        t.Fatal(err)
    }
    if result != 65204 {
        t.Error("Expected 65204, got", result)
    }
}
```

Now you can see how long it takes your file-length function to run. Your goal is to find out what size buffer you should use to read from the file.

Before you spend time going down an optimization rabbit hole, be sure that you need to optimize. If your program is already fast enough to meet your responsiveness requirements and is using an acceptable amount of memory, your time is better spent on adding features and fixing bugs. Your business requirements determine what "fast enough" and "acceptable amount of memory" mean.

In Go, *benchmarks* are functions in your test files that start with the word `Benchmark` and take in a single parameter of type `*testing.B`. This type includes all the functionality of a `*testing.T` as well as additional support for benchmarking. Let's start by looking at a benchmark that uses a buffer size of 1 byte:

```
var blackhole int

func BenchmarkFileLen1(b *testing.B) {
    for i := 0; i < b.N; i++ {
        result, err := FileLen("testdata/data.txt", 1)
        if err != nil {
            b.Fatal(err)
        }
        blackhole = result
    }
}
```

The `blackhole` package-level variable is interesting. You write the results from `FileLen` to this package-level variable to make sure that the compiler doesn't get too clever and decide to optimize away the call to `FileLen`, ruining your benchmark.

Every Go benchmark must have a loop that iterates from 0 to `b.N`. The testing framework calls your benchmark functions over and over with larger and larger values for `N` until it is sure that the timing results are accurate. You'll see this in the output in a moment.

You run a benchmark by passing the `-bench` flag to `go test`. This flag expects a regular expression to describe the name of the benchmarks to run. Use `-bench=.` to run all benchmarks. A second flag, `-benchmem`, includes memory allocation information in the benchmark output. All tests are run before the benchmarks, so you can benchmark code only when tests pass.

Here's the output for the benchmark on my computer:

```
BenchmarkFileLen1-12  25  47201025 ns/op  65342 B/op  65208 allocs/op
```

Running a benchmark with memory allocation information produces output with five columns. Here's what each one means:

BenchmarkFileLen1-12
: The name of the benchmark, a hyphen, and the value of `GOMAXPROCS` for the benchmark.

25
: The number of times that the test ran to produce a stable result.

47201025 ns/op
: The amount of time it took to run a single pass of this benchmark, in nanoseconds (there are 1,000,000,000 nanoseconds in a second).

65342 B/op
: The number of bytes allocated during a single pass of the benchmark.

65208 allocs/op
: The number of times bytes had to be allocated from the heap during a single pass of the benchmark. This will always be less than or equal to the number of bytes allocated.

Now that you have results for a buffer of 1 byte, let's see what the results look like when you use buffers of different sizes:

```
func BenchmarkFileLen(b *testing.B) {
    for _, v := range []int{1, 10, 100, 1000, 10000, 100000} {
        b.Run(fmt.Sprintf("FileLen-%d", v), func(b *testing.B) {
            for i := 0; i < b.N; i++ {
                result, err := FileLen("testdata/data.txt", v)
                if err != nil {
                    b.Fatal(err)
                }
                blackhole = result
            }
        })
    }
}
```

Just as you launched table tests using `t.Run`, you're using `b.Run` to launch benchmarks that vary based only on input. Here are the results of this benchmark on my computer:

```
BenchmarkFileLen/FileLen-1-12          25  47828842 ns/op   65342 B/op  65208 allocs/op
BenchmarkFileLen/FileLen-10-12        230   5136839 ns/op  104488 B/op   6525 allocs/op
BenchmarkFileLen/FileLen-100-12      2246    509619 ns/op   73384 B/op    657 allocs/op
BenchmarkFileLen/FileLen-1000-12    16491     71281 ns/op   68744 B/op     70 allocs/op
BenchmarkFileLen/FileLen-10000-12   42468     26600 ns/op   82056 B/op     11 allocs/op
BenchmarkFileLen/FileLen-100000-12  36700     30473 ns/op  213128 B/op      5 allocs/op
```

These results aren't surprising; as you increase the size of the buffer, you make fewer allocations and your code runs faster, until the buffer is bigger than the file. When the buffer is bigger than the size of the file, extra allocations slow the output. If you expect files of roughly this size, a buffer of 10,000 bytes would work best.

But you can make a change that improves the numbers more. You are reallocating the buffer every time you get the next set of bytes from the file. That's unnecessary. If you move the byte slice allocation before the loop and rerun your benchmark, you see an improvement:

```
BenchmarkFileLen/FileLen-1-12          25  46167597 ns/op     137 B/op  4 allocs/op
BenchmarkFileLen/FileLen-10-12        261   4592019 ns/op     152 B/op  4 allocs/op
BenchmarkFileLen/FileLen-100-12      2518    478838 ns/op     248 B/op  4 allocs/op
BenchmarkFileLen/FileLen-1000-12    20059     60150 ns/op    1160 B/op  4 allocs/op
BenchmarkFileLen/FileLen-10000-12   62992     19000 ns/op   10376 B/op  4 allocs/op
BenchmarkFileLen/FileLen-100000-12  51928     21275 ns/op  106632 B/op  4 allocs/op
```

The number of allocations are now consistent and small, just four allocations for every buffer size. What is interesting is that you now can make trade-offs. If you are tight on memory, you can use a smaller buffer size and save memory at the expense of performance.

Profiling Your Go Code

If benchmarking reveals that you have a performance or memory problem, the next step is figuring out exactly what the problem is. Go includes profiling tools that gather CPU and memory usage data from a running program as well as tools that help you visualize and interpret the generated data. You can even expose a web service endpoint to gather profiling information remotely from a running Go service.

Discussing the profiler is a topic that's beyond the scope of this book. Many great resources are available online. A good starting point is the blog post "Profiling Go Programs with pprof" by Julia Evans.

Using Stubs in Go

So far, you've written tests for functions that didn't depend on other code. This is not typical, as most code is filled with dependencies. As you saw in Chapter 7, Go allows you to abstract function calls in two ways: defining a function type and defining an interface. These abstractions help you write not only modular production code but also unit tests.

When your code depends on abstractions, it's easier to write unit tests!

Let's take a look at an example in the *sample_code/solver* directory in the Chapter 15 repository. You define a type called `Processor`:

```
type Processor struct {
    Solver MathSolver
}
```

It has a field of type `MathSolver`:

```
type MathSolver interface {
    Resolve(ctx context.Context, expression string) (float64, error)
}
```

You'll implement and test `MathSolver` in a bit.

`Processor` also has a method that reads an expression from an `io.Reader` and returns the calculated value:

```
func (p Processor) ProcessExpression(ctx context.Context, r io.Reader)
                                     (float64, error) {
    curExpression, err := readToNewLine(r)
    if err != nil {
        return 0, err
    }
    if len(curExpression) == 0 {
        return 0, errors.New("no expression to read")
    }
    answer, err := p.Solver.Resolve(ctx, curExpression)
    return answer, err
}
```

Let's write the code to test `ProcessExpression`. First, you need a simple implementation of the `Resolve` method to write your test:

```
type MathSolverStub struct{}

func (ms MathSolverStub) Resolve(ctx context.Context, expr string)
                                 (float64, error) {
    switch expr {
    case "2 + 2 * 10":
        return 22, nil
    case "( 2 + 2 ) * 10":
        return 40, nil
    case "( 2 + 2 * 10":
        return 0, errors.New("invalid expression: ( 2 + 2 * 10")
    }
    return 0, nil
}
```

Next, you write a unit test that uses this stub (production code should test the error messages too, but for the sake of brevity, you'll leave those out):

```
func TestProcessorProcessExpression(t *testing.T) {
    p := Processor{MathSolverStub{}}
    in := strings.NewReader(`2 + 2 * 10
( 2 + 2 ) * 10
( 2 + 2 * 10`)
    data := []float64{22, 40, 0}
    hasErr := []bool{false, false, true}
    for i, d := range data {
        result, err := p.ProcessExpression(context.Background(), in)
        if err != nil && !hasErr[i] {
            t.Error(err)
        }
        if result != d {
            t.Errorf("Expected result %f, got %f", d, result)
        }
```

```
    }
}
```

You can then run your test and see that everything works.

While most Go interfaces specify only one or two methods, this isn't always the case. You sometimes find yourself with an interface that has many methods. Let's take a look at the code in the *sample_code/stub* directory in the Chapter 15 repository. Assume you have an interface that looks like this:

```
type Entities interface {
    GetUser(id string) (User, error)
    GetPets(userID string) ([]Pet, error)
    GetChildren(userID string) ([]Person, error)
    GetFriends(userID string) ([]Person, error)
    SaveUser(user User) error
}
```

There are two patterns for testing code that depends on large interfaces. The first is to embed the interface in a struct. Embedding an interface in a struct automatically defines all the interface's methods on the struct. It doesn't provide any implementations of those methods, so you need to implement the methods that you care about for the current test. Let's assume that `Logic` is a struct that has a field of type `Entities`:

```
type Logic struct {
    Entities Entities
}
```

Assume you want to test this method:

```
func (l Logic) GetPetNames(userId string) ([]string, error) {
    pets, err := l.Entities.GetPets(userId)
    if err != nil {
        return nil, err
    }
    out := make([]string, len(pets))
    for _, p := range pets {
        out = append(out, p.Name)
    }
    return out, nil
}
```

This method uses only one of the methods declared on `Entities`: `GetPets`. Rather than creating a stub that implements every single method on `Entities` just to test `GetPets`, you can write a stub struct that implements only the method you need to test this method:

```
type GetPetNamesStub struct {
    Entities
}
```

```
func (ps GetPetNamesStub) GetPets(userID string) ([]Pet, error) {
    switch userID {
    case "1":
        return []Pet{{Name: "Bubbles"}}, nil
    case "2":
        return []Pet{{Name: "Stampy"}, {Name: "Snowball II"}}, nil
    default:
        return nil, fmt.Errorf("invalid id: %s", userID)
    }
}
```

You then write your unit test, with your stub injected into `Logic`:

```
func TestLogicGetPetNames(t *testing.T) {
    data := []struct {
        name     string
        userID   string
        petNames []string
    }{
        {"case1", "1", []string{"Bubbles"}},
        {"case2", "2", []string{"Stampy", "Snowball II"}},
        {"case3", "3", nil},
    }
    l := Logic{GetPetNamesStub{}}
    for _, d := range data {
        t.Run(d.name, func(t *testing.T) {
            petNames, err := l.GetPetNames(d.userID)
            if err != nil {
                t.Error(err)
            }
            if diff := cmp.Diff(d.petNames, petNames); diff != "" {
                t.Error(diff)
            }
        })
    }
}
```

(By the way, the `GetPetNames` method has a bug. Did you see it? Even simple methods can sometimes have bugs.)

If you embed an interface in a stub struct, make sure you provide an implementation for every method that's called during your test! If you call an unimplemented method, your tests will panic.

If you need to implement only one or two methods in an interface for a single test, this technique works well. The drawback comes when you need to call the same method in different tests with different inputs and outputs. When that happens, you need to either include every possible result for every test within the same implementation or reimplement the struct for each test. This quickly becomes difficult to

understand and maintain. A better solution is to create a stub struct that proxies method calls to function fields. For each method defined on `Entities`, you define a function field with a matching signature on your stub struct:

```
type EntitiesStub struct {
    getUser     func(id string) (User, error)
    getPets     func(userID string) ([]Pet, error)
    getChildren func(userID string) ([]Person, error)
    getFriends  func(userID string) ([]Person, error)
    saveUser    func(user User) error
}
```

You then make `EntitiesStub` meet the `Entities` interface by defining the methods. In each method, you invoke the associated function field. For example:

```
func (es EntitiesStub) GetUser(id string) (User, error) {
    return es.getUser(id)
}

func (es EntitiesStub) GetPets(userID string) ([]Pet, error) {
    return es.getPets(userID)
}
```

Once you create this stub, you can supply different implementations of different methods in different test cases via the fields in the data struct for a table test:

```
func TestLogicGetPetNames(t *testing.T) {
    data := []struct {
        name     string
        getPets  func(userID string) ([]Pet, error)
        userID   string
        petNames []string
        errMsg   string
    }{
        {"case1", func(userID string) ([]Pet, error) {
            return []Pet{{Name: "Bubbles"}}, nil
        }, "1", []string{"Bubbles"}, ""},
        {"case2", func(userID string) ([]Pet, error) {
            return nil, errors.New("invalid id: 3")
        }, "3", nil, "invalid id: 3"},
    }
    l := Logic{}
    for _, d := range data {
        t.Run(d.name, func(t *testing.T) {
            l.Entities = EntitiesStub{getPets: d.getPets}
            petNames, err := l.GetPetNames(d.userID)
            if diff := cmp.Diff(d.petNames, petNames); diff != "" {
                t.Error(diff)
            }
            var errMsg string
            if err != nil {
                errMsg = err.Error()
            }
```

```
            if errMsg != d.errMsg {
                t.Errorf("Expected error `%s`, got `%s`", d.errMsg, errMsg)
            }
        })
    }
}
```

You add a field of function type to `data`'s anonymous struct. In each test case, you specify a function that returns the data that `GetPets` would return. When you write your test stubs this way, it's clear what the stubs should return for each test case. As each test runs, you instantiate a new `EntitiesStub` and assign the `getPets` function field in your test data to the `getPets` function field in `EntitiesStub`.

Mocks and Stubs

The terms *mock* and *stub* are often used interchangeably, but they are actually two different concepts. Martin Fowler, a respected voice on all things related to software development, wrote a blog post on mocks that, among other things, covers the difference between mocks and stubs. In short, a *stub* returns a canned value for a given input, whereas a *mock* validates that a set of calls happen in the expected order with the expected inputs.

You used stubs in the examples to return canned values to a given response. You can write your own mocks by hand, or you can use a third-party library to generate them. The two most popular options are the gomock library from Google and the testify library from Stretchr, Inc.

Using httptest

It can be difficult to write tests for a function that calls an HTTP service. Traditionally, this became an integration test, requiring you to stand up a test instance of the service that the function calls. The Go standard library includes the `net/http/httptest` package to make it easier to stub HTTP services. Let's go back to the *sample_code/solver* directory in the Chapter 15 repository and provide an implementation of `MathSolver` that calls an HTTP service to evaluate expressions:

```
type RemoteSolver struct {
    MathServerURL string
    Client        *http.Client
}

func (rs RemoteSolver) Resolve(ctx context.Context, expression string)
                              (float64, error) {
    req, err := http.NewRequestWithContext(ctx, http.MethodGet,
        rs.MathServerURL+"?expression="+url.QueryEscape(expression),
        nil)
```

```
    if err != nil {
        return 0, err
    }
    resp, err := rs.Client.Do(req)
    if err != nil {
        return 0, err
    }
    defer resp.Body.Close()
    contents, err := io.ReadAll(resp.Body)
    if err != nil {
        return 0, err
    }
    if resp.StatusCode != http.StatusOK {
        return 0, errors.New(string(contents))
    }
    result, err := strconv.ParseFloat(string(contents), 64)
    if err != nil {
        return 0, err
    }
    return result, nil
}
```

Now let's see how to use the `httptest` library to test this code without standing up a server. The code is in the `TestRemoteSolver_Resolve` function in *sample_code/solver/remote_solver_test.go* in the Chapter 15 repository, but here are the highlights. First, you want to make sure that the data that's passed into the function arrives on the server. So in your test function, you define a type called `info` to hold your input and output and a variable called `io` that is assigned the current input and output:

```
type info struct {
    expression string
    code       int
    body       string
}
var io info
```

Next, you set up your fake remote server and use it to configure an instance of `RemoteSolver`:

```
server := httptest.NewServer(
    http.HandlerFunc(func(rw http.ResponseWriter, req *http.Request) {
        expression := req.URL.Query().Get("expression")
        if expression != io.expression {
            rw.WriteHeader(http.StatusBadRequest)
            fmt.Fprintf(rw, "expected expression '%s', got '%s'",
                io.expression, expression)
            return
        }
        rw.WriteHeader(io.code)
        rw.Write([]byte(io.body))
    }))
defer server.Close()
```

```
rs := RemoteSolver{
    MathServerURL: server.URL,
    Client:        server.Client(),
}
```

The `httptest.NewServer` function creates and starts an HTTP server on a random unused port. You need to provide an `http.Handler` implementation to process the request. Since this is a server, you must close it when the test completes. The `server` instance has its URL specified in the `URL` field of the `server` instance and a preconfigured `http.Client` for communicating with the test server. You pass these into `RemoteSolver`.

The rest of the function works like every other table test that you've seen:

```
data := []struct {
    name   string
    io     info
    result float64
}{
    {"case1", info{"2 + 2 * 10", http.StatusOK, "22"}, 22},
    // remaining cases
}
for _, d := range data {
    t.Run(d.name, func(t *testing.T) {
        io = d.io
        result, err := rs.Resolve(context.Background(), d.io.expression)
        if result != d.result {
            t.Errorf("io `%f`, got `%f`", d.result, result)
        }
        var errMsg string
        if err != nil {
            errMsg = err.Error()
        }
        if errMsg != d.errMsg {
            t.Errorf("io error `%s`, got `%s`", d.errMsg, errMsg)
        }
    })
}
```

The interesting thing to note is that the variable `io` has been captured by two closures: the one for the stub server and the one for running each test. You write to it in one closure and read it in the other. This is a bad idea in production code, but it works well in test code within a single function.

Using Integration Tests and Build Tags

Even though `httptest` provides a way to avoid testing against external services, you should still write *integration tests*, automated tests that connect to other services. These validate that your understanding of the service's API is correct. The challenge is figuring out how to group your automated tests; you want to run integration tests only when the support environment is present. Also, integration tests tend to be slower than unit tests, so they are usually run less frequently.

In "Using Build Tags" on page 284, I covered build tags, which are used by the Go compiler to control when a file is compiled. While they are primarily intended to allow developers to write code that's intended only for a specific operating system, CPU, or Go version, you can take advantage of the ability to specify custom build tags to control when integration tests are compiled and run.

Let's try this out with the math-solving project. Use Docker to download a server implementation with `docker pull jonbodner/math-server` and then run the server locally on port 8080 with `docker run -p 8080:8080 jonbodner/math-server`.

If you don't have Docker installed or if you want to build the code for yourself, you can find it on GitHub.

You need to write an integration test to make sure that your `Resolve` method properly communicates with the math server. The *sample_code/solver/remote_solver_integration_test.go* file in the Chapter 15 repository has a complete test in the `TestRemoteSolver_ResolveIntegration` function. The test looks like every other table test that you've written. The interesting thing is the first line of the file, separated from the package declaration by a newline:

```
//go:build integration
```

To run your integration test alongside the other tests you've written, use this:

```
$ go test -tags integration -v ./...
```

You should be aware that some people in the Go community prefer to use environment variables rather than build tags to create groups of integration tests. Peter Bourgon describes the concept in a blog post with the unsubtle title "Don't Use Build Tags for Integration Tests". His argument is that it's hard to find out what build tags to set in order to run integration tests. An explicit check for an environment variable in each integration test, combined with a detailed message in a `t.Skip` method call, makes it clear that tests aren't being run and how to run them. In the end, it's a trade-off between verbosity and discoverability. Feel free to use either technique.

Using the -short Flag

Yet another way to group tests is to use `go test` with the `-short` flag. If you want to skip over tests that take a long time, label your slow tests by placing the following code at the start of the test function:

```
if testing.Short() {
    t.Skip("skipping test in short mode.")
}
```

When you want to run only short tests, pass the `-short` flag to `go test`.

There are a few problems with the `-short` flag. If you use it, there are only two levels of testing: short tests and all tests. By using build tags, you can group your integration tests, specifying which service they need in order to run. Another argument against using the `-short` flag to indicate integration tests is philosophical. Build tags indicate a dependency, while the `-short` flag is meant to indicate only that you don't want to run tests that take a long time. Those are different concepts. Finally, I find the `-short` flag unintuitive. You should run short tests all the time. It makes more sense to require a flag to *include* long-running tests, not to *exclude* them.

Finding Concurrency Problems with the Data Race Detector

Even with Go's built-in support for concurrency, bugs still happen. It's easy to accidentally reference a variable from two different goroutines without acquiring a lock. The computer science term for this is a *data race*. To help find these sorts of bugs, Go includes a *race checker*. It isn't guaranteed to find every single data race in your code, but if it finds one, you should put proper locks around what it finds.

Look at a simple example in the file *sample_code/race/race.go* in the Chapter 15 repository:

```
func getCounter() int {
    var counter int
    var wg sync.WaitGroup
    wg.Add(5)
    for i := 0; i < 5; i++ {
        go func() {
            for i := 0; i < 1000; i++ {
                counter++
            }
            wg.Done()
        }()
    }
    wg.Wait()
```

```
    return counter
}
```

This code launches five goroutines, has each of them update a shared `counter` variable 1,000 times, and then returns the result. You'd expect it to be 5,000, so let's verify this with a unit test in *sample_code/race/race_test.go*:

```
func TestGetCounter(t *testing.T) {
    counter := getCounter()
    if counter != 5000 {
        t.Error("unexpected counter:", counter)
    }
}
```

If you run `go test` a few times, you'll see that sometimes it passes, but most of the time it fails with an error message like this:

```
unexpected counter: 3673
```

The problem is that there's a data race in the code. In a program this simple, the cause is obvious: multiple goroutines are trying to update `counter` simultaneously, and some of their updates are lost. In more complicated programs, these sorts of races are harder to see. Let's see what the data race detector does. Use the flag `-race` with `go test` to enable it:

```
$ go test -race
==================
WARNING: DATA RACE
Read at 0x00c000128070 by goroutine 10:
  test_examples/race.getCounter.func1()
      test_examples/race/race.go:12 +0x45

Previous write at 0x00c000128070 by goroutine 8:
  test_examples/race.getCounter.func1()
      test_examples/race/race.go:12 +0x5b
```

The traces make it clear that the line `counter++` is the source of your problems.

Some people try to fix race conditions by inserting "sleeps" into their code, trying to space out access to the variable that's being accessed by multiple goroutines. *This is a bad idea.* Doing so might eliminate the problem in some cases, but the code is still wrong and will fail in some situations.

You can also use the `-race` flag when you build your programs. This creates a binary that includes the data race detector and that reports any races it finds to the console. This allows you to find data races in code that doesn't have tests.

If the data race detector is so useful, why isn't it enabled all the time for testing and production? A binary with `-race` enabled runs approximately 10 times slower than a normal binary. That isn't a problem for test suites that take a second to run, but for large test suites that take several minutes, a 10× slowdown reduces productivity.

For more information on the data race detector, check out its official documentation.

Exercises

Now that you've learned about writing tests and using the code-quality tools included with Go, complete these exercises to apply that knowledge to a sample application.

1. Download the Simple Web App program. Write unit tests for the program and get as close to 100% code coverage as you can. If you find any bugs, fix them.
2. Use the race detector to find a concurrency problem in the program and fix it.
3. Write a fuzz test against the `parser` function and fix any problems that you find.

Wrapping Up

In this chapter, you've learned how to write tests and improve code quality by using Go's built-in support for testing, code coverage, benchmarking, fuzzing, and data race checking. In the next chapter, you're going to explore some Go features that allow you to break the rules: the `unsafe` package, reflection, and `cgo`.