Prefer table driven tests | Dave Cheney

by Dave Cheney

I'm a big fan of testing, specifically unit testing and TDD (done correctly, of course). A practice that has grown around Go projects is the idea of a table driven test. This post explores the how and why of writing a table driven test.

Let's say we have a function that splits strings:

```
// Split slices s into all substrings separated by sep and
// returns a slice of the substrings between those separators.
func Split(s, sep string) []string {
    var result []string
    i := strings.Index(s, sep)
    for i > -1 {
        result = append(result, s[:i])
        s = s[i+len(sep):]
        i = strings.Index(s, sep)
    }
    return append(result, s)
}
```

In Go, unit tests are just regular Go functions (with a few rules) so we write a unit test for this function starting with a file in the same directory, with the same package name, `strings`.

```
package split
import (
    "reflect"
    "testing"
)

func TestSplit(t *testing.T) {
    got := Split("a/b/c", "/")
    want := []string{"a", "b", "c"}
    if !reflect.DeepEqual(want, got) {
        t.Fatalf("expected: %v, got: %v", want, got)
    }
```

}

Tests are just regular Go functions with a few rules:

The name of the test function must start with `Test`.

The test function must take one argument of type `*testing.T`. A `*testing.T` is a type injected by the testing package itself, to provide ways to print, skip, and fail the test.

In our test we call `Split` with some inputs, then compare it to the result we expected.

Code coverage

The next question is, what is the coverage of this package? Luckily the go tool has a built in branch coverage. We can invoke it like this:

```
% go test -coverprofile=c.out
PASS
coverage: 100.0% of statements
ok      split   0.010s
```

Which tells us we have 100% branch coverage, which isn't really surprising, there's only one branch in this code.

If we want to dig in to the coverage report the go tool has several options to print the coverage report. We can use `go tool cover -func` to break down the coverage per function:

```
% go tool cover -func=c.out
split/split.go:8:       Split           100.0%
total:                  (statements)    100.0%
```

Which isn't that exciting as we only have one function in this package, but I'm sure you'll find more exciting packages to test.

Spray some .bashrc on that

This pair of commands is so useful for me I have a shell alias which runs the test coverage and the report in one command:

```
cover () {
    local t=$(mktemp -t cover)
    go test $COVERFLAGS -coverprofile=$t $@ \
        && go tool cover -func=$t \
        && unlink $t
```

```
}
```

Going beyond 100% coverage

So, we wrote one test case, got 100% coverage, but this isn't really the end of the story. We have good branch coverage but we probably need to test some of the boundary conditions. For example, what happens if we try to split it on comma?

```
func TestSplitWrongSep(t *testing.T) {
    got := Split("a/b/c", ",")
    want := []string{"a/b/c"}
    if !reflect.DeepEqual(want, got) {
        t.Fatalf("expected: %v, got: %v", want, got)
    }
}
```

Or, what happens if there are no separators in the source string?

```
func TestSplitNoSep(t *testing.T) {
    got := Split("abc", "/")
    want := []string{"abc"}
    if !reflect.DeepEqual(want, got) {
        t.Fatalf("expected: %v, got: %v", want, got)
    }
}
```

We're starting build a set of test cases that exercise boundary conditions. This is good.

Introducing table driven tests

However the there is a lot of duplication in our tests. For each test case only the input, the expected output, and name of the test case change. Everything else is boilerplate. What we'd like to to set up all the inputs and expected outputs and feel them to a single test harness. This is a great time to introduce table driven testing.

```
func TestSplit(t *testing.T) {
    type test struct {
        input string
        sep   string
        want  []string
    }
    tests := []test{
```

```
        {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        {input: "abc", sep: "/", want: []string{"abc"}},
    }

    for _, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("expected: %v, got: %v", tc.want, got)
        }
    }
}
```

We declare a structure to hold our test inputs and expected outputs. This is our table. The `tests` structure is usually a local declaration because we want to reuse this name for other tests in this package.

In fact, we don't even need to give the type a name, we can use an anonymous struct literal to reduce the boilerplate like this:

```
func TestSplit(t *testing.T) {
    tests := []struct {
        input string
        sep   string
        want  []string
    }{
        {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        {input: "abc", sep: "/", want: []string{"abc"}},
    }
    for _, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("expected: %v, got: %v", tc.want, got)
        }
    }
}
```

Now, adding a new test is a straight forward matter; simply add another line the `tests` structure. For example, what will happen if our input string has a trailing separator?

```
{input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
```

```
{input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
{input: "abc", sep: "/", want: []string{"abc"}},
{input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}}, // trailing sep
```

But, when we run `go test`, we get

```
% go test
--- FAIL: TestSplit (0.00s)
    split_test.go:24: expected: [a b c], got: [a b c ]
```

Putting aside the test failure, there are a few problems to talk about.

The first is by rewriting each test from a function to a row in a table we've lost the name of the failing test. We added a comment in the test file to call out this case, but we don't have access to that comment in the `go test` output.

There are a few ways to resolve this. You'll see a mix of styles in use in Go code bases because the table testing idiom is evolving as people continue to experiment with the form.

Enumerating test cases

As tests are stored in a slice we can print out the index of the test case in the failure message:

```
func TestSplit(t *testing.T) {
    tests := []struct {
        input string
        sep   string
        want  []string
    }{
        {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        {input: "abc", sep: "/", want: []string{"abc"}},
        {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
    }

    for i, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("test %d: expected: %v, got: %v", i+1, tc.want, got)
        }
    }
```

```
}
```

Now when we run `go test` we get this

```
% go test
--- FAIL: TestSplit (0.00s)
   split_test.go:24: test 4: expected: [a b c], got: [a b c ]
```

Which is a little better. Now we know that the fourth test is failing, although we have to do a little bit of fudging because slice indexing—and range iteration—is zero based. This requires consistency across your test cases; if some use zero base reporting and others use one based, it's going to be confusing. And, if the list of test cases is long, it could be difficult to count braces to figure out exactly which fixture constitutes test case number four.

Give your test cases names

Another common pattern is to include a name field in the test fixture.

```
func TestSplit(t *testing.T) {
   tests := []struct {
      name  string
      input string
      sep   string
      want  []string
   }{
      {name: "simple", input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
      {name: "wrong sep", input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
      {name: "no sep", input: "abc", sep: "/", want: []string{"abc"}},
      {name: "trailing sep", input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
   }
   for _, tc := range tests {
      got := Split(tc.input, tc.sep)
      if !reflect.DeepEqual(tc.want, got) {
         t.Fatalf("%s: expected: %v, got: %v", tc.name, tc.want, got)
      }
   }
}
```

Now when the test fails we have a descriptive name for what the test was doing. We no longer have to try to figure it out from the output—also, now have a string we can search on.

```
% go test
```

```
--- FAIL: TestSplit (0.00s)
    split_test.go:25: trailing sep: expected: [a b c], got: [a b c ]
```

We can dry this up even more using a map literal syntax:

```
func TestSplit(t *testing.T) {
    tests := map[string]struct {
        input string
        sep   string
        want  []string
    }{
        "simple":       {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
        "wrong sep":    {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
        "no sep":       {input: "abc", sep: "/", want: []string{"abc"}},
        "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
    }
    for name, tc := range tests {
        got := Split(tc.input, tc.sep)
        if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("%s: expected: %v, got: %v", name, tc.want, got)
        }
    }
}
```

Using a map literal syntax we define our test cases not as a slice of structs, but as map of test names to test fixtures. There's also a side benefit of using a map that is going to potentially improve the utility of our tests.

Map iteration order is *undefined* [1] This means each time we run `go test`, our tests are going to be potentially run in a different order.

This is super useful for spotting conditions where test pass when run in statement order, but not otherwise. If you find that happens you probably have some global state that is being mutated by one test with subsequent tests depending on that modification.

Introducing sub tests

Before we fix the failing test there are a few other issues to address in our table driven test harness.

The first is we're calling `t.Fatalf` when one of the test cases fails. This means after the first failing test case we stop testing the other cases. Because test cases are run in an undefined order, if there is a test failure, it would be nice to know if it was the only failure or just the first.

The testing package would do this for us if we go to the effort to write out each test case as its own function, but that's quite verbose. The good news is since Go 1.7 a new feature was added that lets us do this easily for table driven tests. They're called sub tests.

```
func TestSplit(t *testing.T) {
   tests := map[string]struct {
      input string
      sep   string
      want  []string
   }{
      "simple":       {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
      "wrong sep":    {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
      "no sep":       {input: "abc", sep: "/", want: []string{"abc"}},
      "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
   }
   for name, tc := range tests {
      t.Run(name, func(t *testing.T) {
         got := Split(tc.input, tc.sep)
         if !reflect.DeepEqual(tc.want, got) {
            t.Fatalf("expected: %v, got: %v", tc.want, got)
         }
      })
   }
}
```

As each sub test now has a name we get that name automatically printed out in any test runs.

```
% go test
--- FAIL: TestSplit (0.00s)
   --- FAIL: TestSplit/trailing_sep (0.00s)
      split_test.go:25: expected: [a b c], got: [a b c ]
```

Each subtest is its own anonymous function, therefore we can use `t.Fatalf`, `t.Skipf`, and all the other `testing.T`helpers, while retaining the compactness of a table driven test.

Individual sub test cases can be executed directly

Because sub tests have a name, you can run a selection of sub tests by name using the `go test -run` flag.

```
% go test -run=.*/trailing -v
=== RUN   TestSplit
```

```
=== RUN   TestSplit/trailing_sep
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:25: expected: [a b c], got: [a b c ]
```

Comparing what we got with what we wanted

Now we're ready to fix the test case. Let's look at the error.

```
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:25: expected: [a b c], got: [a b c ]
```

Can you spot the problem? Clearly the slices are different, that's what `reflect.DeepEqual` is upset about. But spotting the actual difference isn't easy, you have to spot that extra space after `c`. This might look simple in this simple example, but it is any thing but when you're comparing two complicated deeply nested gRPC structures.

We can improve the output if we switch to the `%#v` syntax to view the value as a Go(ish) declaration:

```
got := Split(tc.input, tc.sep)
if !reflect.DeepEqual(tc.want, got) {
    t.Fatalf("expected: %#v, got: %#v", tc.want, got)
}
```

Now when we run our test it's clear that the problem is there is an extra blank element in the slice.

```
% go test
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:25: expected: []string{"a", "b", "c"}, got: []string{"a", "b", "c", ""}
```

But before we go to fix our test failure I want to talk a little bit more about choosing the right way to present test failures. Our `Split` function is simple, it takes a primitive string and returns a slice of strings, but what if it worked with structs, or worse, pointers to structs?

Here is an example where `%#v` does not work as well:

```
func main() {
    type T struct {
        I int
    }
    x := []*T{{1}, {2}, {3}}
```

```
    y := []*T{{1}, {2}, {4}}
    fmt.Printf("%v %v\n", x, y)
    fmt.Printf("%#v %#v\n", x, y)
}
```

The first `fmt.Printf`prints the unhelpful, but expected slice of addresses; `[0xc000096000 0xc000096008 0xc000096010] [0xc000096018 0xc000096020 0xc000096028]`. However our `%#v` version doesn't fare any better, printing a slice of addresses cast to `*main.T`; `[]*main.T{(*main.T)(0xc000096000), (*main.T)(0xc000096008), (*main.T)(0xc000096010)} []*main.T{(*main.T)(0xc000096018), (*main.T)(0xc000096020), (*main.T)(0xc000096028)}`

Because of the limitations in using any `fmt.Printf` verb, I want to introduce the go-cmp library from Google.

The goal of the cmp library is it is specifically to compare two values. This is similar to `reflect.DeepEqual`, but it has more capabilities. Using the cmp pacakge you can, of course, write:

```
func main() {
    type T struct {
        I int
    }
    x := []*T{{1}, {2}, {3}}
    y := []*T{{1}, {2}, {4}}
    fmt.Println(cmp.Equal(x, y)) // false
}
```

But far more useful for us with our test function is the `cmp.Diff` function which will produce a textual description of what is different between the two values, recursively.

```
func main() {
    type T struct {
        I int
    }
    x := []*T{{1}, {2}, {3}}
    y := []*T{{1}, {2}, {4}}
    diff := cmp.Diff(x, y)
    fmt.Printf(diff)
}
```

Which instead produces:

```
% go run
{[]*main.T}[2].I:
        -: 3
        +: 4
```

Telling us that at element 2 of the slice of `Ts` the `I`field was expected to be 3, but was actually 4.

Putting this all together we have our table driven go-cmp test

```
func TestSplit(t *testing.T) {
   tests := map[string]struct {
      input string
      sep   string
      want  []string
   }{
      "simple":       {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
      "wrong sep":    {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
      "no sep":       {input: "abc", sep: "/", want: []string{"abc"}},
      "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
   }
   for name, tc := range tests {
      t.Run(name, func(t *testing.T) {
         got := Split(tc.input, tc.sep)
         diff := cmp.Diff(tc.want, got)
         if diff != "" {
            t.Fatalf(diff)
         }
      })
   }
}
```

Running this we get

```
% go test
--- FAIL: TestSplit (0.00s)
   --- FAIL: TestSplit/trailing_sep (0.00s)
      split_test.go:27: {[]string}[?->3]:
            -: <non-existent>
            +: ""
FAIL
```

exit status 1
FAIL	split	0.006s

Using `cmp.Diff` our test harness isn't just telling us that what we got and what we wanted were different. Our test is telling us that the strings are different lengths, the third index in the fixture shouldn't exist, but the actual output we got an empty string, "". From here fixing the test failure is straight forward.

Please don't email me to argue that map iteration order is *random*. It's not.