Testing Go programs with `go-internal/testscript`

Giuseppe Maxia

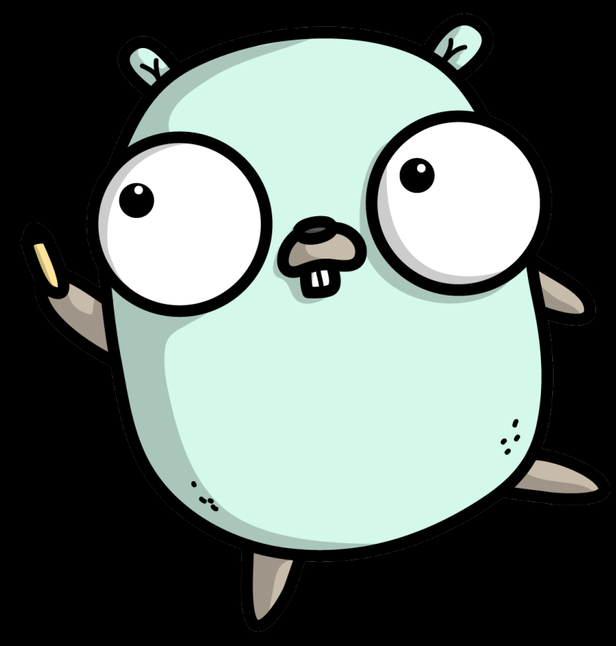

image from https://github.com/ashleymcnamara/gophers

What will you learn

- Using `testscript` to test command line programs;
- Testing the executable without intermediate steps;
- Using built-in commands and conditions;
- Creating and using custom commands and conditions.

The main goal is getting devs curious about `testscript`

why

Stating the problem, i.e. 'the old way'

If you want to test a command line program:

1. Compile the executable nd put it in a known PATH
2. Generate the testing environment
3. then:
   - 3a. run the executable with shell scripts
   - 3b. OR call the executable from Go code functions.

The problem with testing Go command line programs is that there is little coordination between the language tools and the testing of the finished artefact.
A well organised project can test single functions to make sure they do what you expect, but there is no guarantee that the tested function will be the one triggered

In the best scenario, you can run a sensible test, but you need to compile the executable first. This is not only an inconvenience but also a risk: if you forget a step after applying changes, you may be testing a stale version of the executable. Moreover, using shell scripts for testing is not the friendliest environment.

TESTSCRIPT

enter testscript

Introducing `testscript`

- It's a Go library
- But also a standalone tool
- Uses a simple file archive named `txtar`
- It was created to test the `Go` tool itself
- Now released within the `go-internal` package.

it is specifically oriented towards tools that manipulate text (such as the Go toolkit), but it adapts to any CLI.

A first example

testdata/1/hello.txtar

```
exec echo 'hello world'
stdout 'hello world\n'
! stderr .
```

hello_test.go

```
package script_test
import (
    "testing"
    "github.com/rogpeppe/go-internal/testscript"
)
func TestScript(t *testing.T) {
    testscript.Run(t, testscript.Params{
        Dir: "testdata",
    })
}
```


testscript tests have two components: the one in the Go code indicates a directory where to find the scripts, and the actual test is in a script within such directory.

In the test here, we run a command `echo` using the keyword `exec`, which, unlike the corresponding one used in shell, does not end the script.

Then the `stdout` keyword asks for a comparison with the output of the previous command. The `! stderr` expression says "make sure there is nothing in the standard error stream"

A modified first example (1)

```
exec echo 'hello world'
stdout 'h\w+ w\w+'
! stderr .
```


testdata/2/hello.txtar

The previous example may have suggested what happens here: the argument of both `stdout` and `stdin` are not blunt literal text, but regular expressions.

A modified first example (2)

```
exec echo 'hello world'
stdout 'h\w+'
stdout 'w\w+'
! stderr .
```


testdata/3/hello-split.txtar

We can use multiple `stdout` or `stderr` expressions to evaluate a single output.

Using local files

```
exec cat data.txt
stdout 'hello world\n'
! stderr .
exec cat dir1/data2.txt
stdout something

-- data.txt --
hello world

-- dir1/data2.txt --
something else
```


testdata/4/hello-data.txtar

The bread and butter of testscript environments is that we can build files on-the-fly within the testing environment, using a simple syntax: a file name between double dashes (--) followed by zero or more lines of text.

The `testscript` main *commands* (1)

- `exec` runs an executable
- `stdout` checks the standard output with a regular expression
- `stderr` checks the standard error with a regular expression
- `stdin` provides standard input for the next command
- `exists` checks that a file exists
- `stop`, `skip` interrupt the test

Note: the `!` symbol before a keyword reverses the check.

The `testscript` main *commands* (2)

- `cmp`, `cmpenv`: compare two files or streams
- `env` sets a variable
- `cat`, `cd`, `cp`, `chmod`, `mkdir`, `mv`, `rm`: as in a shell

The `testscript` main *conditions*

- `[exec:file_name]` checks that an executable is in `$PATH`
- `[unix]` checks that the test runs under a Unix OS
- `[net]` checks that network connection is available
- `[go1.x]` checks that at least the wanted Go version is used
- `[$GOARCH]`checks that we are using the wanted architecture
- `[$GOOS]` checks that the given operating system is being used

The `testscript` environment

Main environment variables:

- `WORK=<temporary-directory>`
- `HOME=/no-home`
- `TMPDIR=$WORK/tmp`

The scripts run in `$WORK` (Different for each script)

Each testscript run is isolated within its own environment.

For each script we can set all the environment variables that we need, including
overwriting system variables such as `$HOME`, `$PATH`, `$TMPDIR`

Sample environment in action

```
go test -run 'TestScriptGeneric/testdata/1/hello' -v ./no-main/
=== RUN   TestScriptGeneric
=== RUN   TestScriptGeneric/testdata/1
=== RUN   TestScriptGeneric/testdata/1/hello

    testscript.go:558: WORK=$WORK
        PATH=/usr/bin:/usr/local/bin:/usr/sbin
        GOTRACEBACK=system
        HOME=/no-home
        TMPDIR=$WORK/.tmp
        devnull=/dev/null
        /=/
        :=:
        $=$
        exe=

        > exec echo 'hello world'
        [stdout]
        hello world
        > stdout 'hello world\n'
        > ! stderr .
        PASS

--- PASS: TestScriptGeneric (0.01s)
```


Running the test with the verbose option (-v) shows all the hidden details of the environment.

Examples with commands and conditions

```
! [unix] skip This test requires a Unix operating system
[linux] exec echo 'good choice of operating system!'
[exec:seq] exec echo 'command "seq" was found'
[go.1.18] exec echo 'we can run generics!'
exists file1.txt
! exists file2.txt
cp file1.txt file2.txt
exists file2.txt

-- file1.txt --
this is file 1
```


Note that the built-in commands run without the `exec` keyword.

Conditions are enclosed in square brackets

Custom commands

```
# test custom command 'sleep_for'
sleep_for 1

# test custom command 'check_files'
check_files $WORK file1.txt file2.txt

-- file1.txt --
-- file2.txt --
```

Where do these commands come from?

we can create custom commands to run operations that would be impossible or cumbersome with the built-in ones.

custom commands definition

```
func TestWordCountAdvanced(t *testing.T) {
    testscript.Run(t, testscript.Params{
        Dir:                 "testdata/advanced",
        Cmds:                customCommands(),    // <<<<
    })
}
```


To create commands, we add a map of named functions to testscript.Run

custom commands creation (1)

- The `Cmds` parameter is a map of functions
- Each function accepts the following parameters:
  - a testscript object;
  - a `negation` Boolean flag;
  - a list of string arguments

each command is a function paired with a name

custom commands creation (2)

Each function should return nothing when the execution was successful;
It should call `testscript.Fatal` if something goes wrong.

commands implementation (1)

```
func customCommands() map[string]func(ts *testscript.TestScript, neg bool, args []string) {
    return map[string]func(ts *testscript.TestScript, neg bool, args []string){

        // check_files will check that a given list of files exists
        // invoke as "check_files workdir file1 [file2 [file3 [file4]]]"
        // The command can be negated, i.e. it will succeed if the given files do not exist
        // "! check_files workdir file1 [file2 [file3 [file4]]]"
        "check_files": checkFiles,

        // sleep_for will pause execution for the required number of seconds
        // Invoke as "sleep_for 3"
        // If no number is passed, it pauses for 1 second
        "sleep_for": sleepFor,
    }
}
```


Here's an example of a function that implements two commands: It returns a map, where the keys are the commands and the values are the related functions

commands implementation (2)

```go
// sleepFor is a testscript command that pauses the execution for the required number of seconds
func sleepFor(ts *testscript.TestScript, neg bool, args []string) {
    duration := 0
    var err error
    if len(args) == 0 {
        duration = 1
    } else {
        duration, err = strconv.Atoi(args[0])
        ts.Check(err)
    }
    time.Sleep(time.Duration(duration) * time.Second)
}
```


For example, this is the implementation of `sleep_for`.

The function checks the arguments, and interprets the first one as the requested time, using a default of '1' if none was provided.

commands implementation (3)

```go
// checkFile is a testscript command that checks the existence of a list of files
// inside a directory
func checkFiles(ts *testscript.TestScript, neg bool, args []string) {
    if len(args) < 1 {
        ts.Fatalf("syntax: check_file directory_name file_name [file_name ...]")
    }
    dir := args[0]

    for i := 1; i < len(args); i++ {
        f := path.Join(dir, args[i])
        if neg {
            if fileExists(f) {
                ts.Fatalf("file %s found", f)
            }
        }
        if !fileExists(f) {
            ts.Fatalf("file not found %s", f)
        }
    }
}
```


The command `check_files` requires at least two arguments: a directory, and then one or more file names. This commands will fail if any of the names was not found in the directory.

Custom conditions

custom conditions

```
# the actual version is passed to this process in the Setup clause of testscript.Params
exec wordcount -version
cmpenv stdout version.txt

# test the custom condition about version
[version_is_at_least:0.2] stop 'this test is satisfied'

# if we use a lower version, we enter this impossible comparison and the test fails
exec echo 'aaa'
stdout 'bbb'

-- version.txt --
$WORDCOUNT_VERSION
```

Where do these conditions come from?

There are two interesting things in this script: The first one is that we are checking the executable version against an environment variable that is inside a file. The second thing is that we are using a custom condition to check that the version is at least 0.2. If it is, the test stops, and all is well. If it isn't the test will fail because it will meet an impossible comparison.

custom conditions definition

```
func TestWordCountAdvanced(t *testing.T) {
    testscript.Run(t, testscript.Params{
        Dir:                 "testdata/advanced",
        Condition:           customConditions,  // <<<<
        Cmds:                customCommands(),
        RequireExplicitExec: true,
        Setup: func(env *testscript.Env) error {
            env.Setenv("WORDCOUNT_VERSION", cmd.Version)  // <<<
            return nil
        },
    })
}
```


Defining the condition is similar to the commands. The main difference is that the condition handler is not a map of function, but a single function, which will be in charge of parsing the input. In this example we also see how the version environment variable is set: using the testscript `Setup` predefined function, we pass the environment value we need, using the value that is currently in the wordcount code.

custom conditions creation (1)

The `Condition` parameter points to a single function:
 * receiving a string as input
 * returning a boolean and error

The function that defines the condition will read a string, parse it to split between condition name and arguments, and return true if the condition passes

custom conditions creation (2)

The function must parse the input and eventually extract the parameters, if any were designed.

It returns `true` if the condition was met.

Condition implementation (1)

```
// customConditions is a testscript function that handles all the conditions defined for this test
func customConditions(condition string) (bool, error) {
    // assumes arguments are separated by a colon (":")
    elements := strings.Split(condition, ":")
    if len(elements) == 0 {
        return false, fmt.Errorf("no condition found")
    }
    name := elements[0]
    switch name {
    case "version_is_at_least":
        return versionIsAtLeast(elements)
    case "exists_within_seconds":
        return existsWithinSeconds(elements)
    default:
        return false, fmt.Errorf("unrecognized condition '%s'", name)
    }
}
```


For example: in this function, we assume the condition name and its parameters are separated by a colon (:). The we pass the resulting data to the corresponding function for each condition.

Condition implementation (2)

```
func versionIsAtLeast(elements []string) (bool, error) {
    if len(elements) < 2 {
        return false, fmt.Errorf("condition '%s' requires version", elements[0])
    }
    version := elements[1]
    return versionGreaterOrEqual(cmd.Version, version)
}
```


The condition that checks that the version is at least a given value checks that there is at least one argument, and uses such argument as the wanted version.

Condition implementation (3)

```go
func existsWithinSeconds(elements []string) (bool, error) {
    if len(elements) < 3 {
        return false, fmt.Errorf("condition 'exists_within_seconds' requires a file name and the number of seconds")
    }
    fileName := elements[1]
    delay, err := strconv.Atoi(elements[2])
    if err != nil {
        return false, err
    }
    if delay == 0 {
        return fileExists(fileName), nil
    }
    elapsed := 0
    for elapsed < delay {
        time.Sleep(time.Second)
        if fileExists(fileName) {
            return true, nil
        }
        elapsed++
    }
    return false, nil
}
```


a similar function runs the condition `exists_within_seconds`, which requires two arguments: a file name, and how many seconds we wait for it to appear. This is useful, for example, when we test a command that on successful task completion will create a file, and we want to make sure that the file exists. If we used the built-in command `exists`, the test would fail, because the file creation may take some seconds. Speaking of which, this check could be implemented either as a condition or as a command. The implementation depends on personal taste and different usage of such check.

Summary

- `testscript` can greatly simplify the testing of command line programs;
- Programs that manipulate texts can especially suit the environment, thanks to `txtar` files;
- No need for separate compilation of the executable;
- Built-in commands and conditions allow for quick and accurate testing;
- The testing environment is reasonably isolated, allowing parallel

Sample code and slides

https://github.com/datacharmer/wordcount

More resources

Splendid articles about `testscript`: https://bitfieldconsulting.com/golang/tag/testscript

The original documentation: https://pkg.go.dev/github.com/rogpeppe/go-internal/testscript

Presentations about testscript:
* https://github.com/qba73/belfast-go-meetup
* https://github.com/qba73/dublin-go-meetup