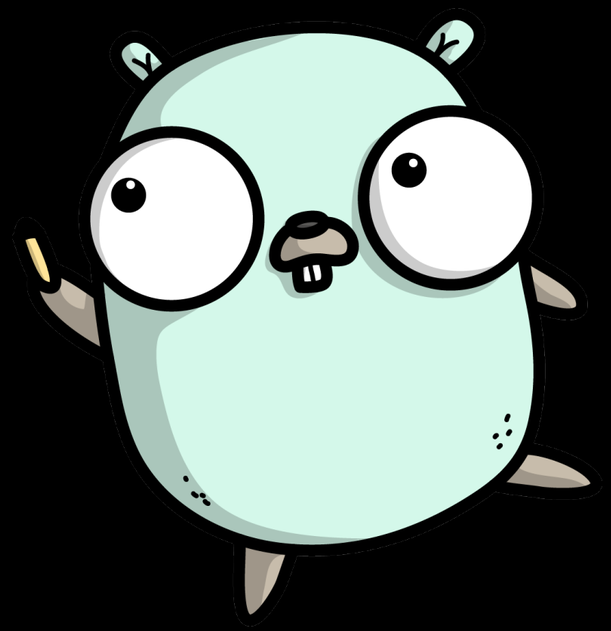
Testing Go programs with `go-internal/testscript`

Giuseppe Maxia



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.

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.

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.

A first example

testdata/1/hello.txtar

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

hello_test.go

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

A modified first example (1)

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

A modified first example (2)

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

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
```

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)

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)
```

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
```

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?

custom commands definition

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

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

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,
    }
}
```

commands implementation (2)

```
// 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)
}
```

commands implementation (3)

```
// 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)
        }
    }
}
```

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?

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
        },
    })
}
```

custom conditions creation (1)

The `Condition` parameter points to a single function:

* receiving a string as input
* returning a boolean and error

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)
    }
}
```

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)
}
```

Condition implementation (3)

```
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
}
```

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