Code coverage for Go integration tests - The Go Programming Language

Than McIntosh 8 March 2023

The Go Blog

Code coverage tools help developers determine what fraction of a source code base is executed (covered) when a given test suite is executed.

Go has for some time provided support (introduced in the Go 1.2 release) to measure code coverage at the package level, using the **"-cover"** flag of the “go test” command.

This tooling works well in most cases, but has some weaknesses for larger Go applications. For such applications, developers often write “integration” tests that verify the behavior of an entire program (in addition to package-level unit tests).

This type of test typically involves building a complete application binary, then running the binary on a set of representative inputs (or under production load, if it is a server) to ensure that all of the component packages are working correctly together, as opposed to testing individual packages in isolation.

Because the integration test binaries are built with “go build” and not “go test”, Go’s tooling didn’t provide any easy way to collect a coverage profile for these tests, up until now.

With Go 1.20, you can now build coverage-instrumented programs using “go build -cover”, then feed these instrumented binaries into an integration test to extend the scope of coverage testing.

In this blog post we’ll give an example of how these new features work, and outline some of the use cases and workflow for collecting coverage profiles from integration tests.

Example

Let’s take a very small example program, write a simple integration test for it, and then collect a coverage profile from the integration test.

For this exercise we’ll use the “mdtool” markdown processing tool from `gitlab.com/golang-commonmark/mdtool`. This is a demo program designed to show how clients can use the package `gitlab.com/golang-commonmark/markdown`, a markdown-to-HTML conversion library.

First let’s download a copy of “mdtool” itself (we’re picking a specific version just to make these steps reproducible):

```
$ git clone https://gitlab.com/golang-commonmark/mdtool.git
...
$ cd mdtool
$ git tag example e210a4502a825ef7205691395804eefce536a02f
$ git checkout example
...
$
```

A simple integration test

Now we'll write a simple integration test for "mdtool"; our test will build the "mdtool" binary, then run it on a set of input markdown files. This very simple script runs the "mdtool" binary on each file from a test data directory, checking to make sure that it produces some output and doesn't crash.

```
$ cat integration_test.sh
#!/bin/sh
BUILDARGS="$*"
#
# Terminate the test if any command below does not complete successfully.
#
set -e
#
# Download some test inputs (the 'website' repo contains various *.md
files).
#
if [ ! -d testdata ]; then
  git clone https://go.googlesource.com/website testdata
  git -C testdata tag example 8bb4a56901ae3b427039d490207a99b48245de2c
  git -C testdata checkout example
fi
#
# Build mdtool binary for testing purposes.
#
rm -f mdtool.exe
go build $BUILDARGS -o mdtool.exe .
#
# Run the tool on a set of input files from 'testdata'.
#
```

```
FILES=$(find testdata -name "*.md" -print)
N=$(echo $FILES | wc -w)
for F in $FILES
do
  ./mdtool.exe +x +a $F > /dev/null
done
echo "finished processing $N files, no crashes"
$
```

Here is an example run of our test:

```
$ /bin/sh integration_test.sh
...
finished processing 380 files, no crashes
$
```

Success: we’ve verified that the “mdtool” binary successfully digested a set of input files… but how much of the tool’s source code have we actually exercised? In the next section we’ll collect a coverage profile to find out.

Using the integration test to collect coverage data

Let’s write another wrapper script that invokes the previous script, but builds the tool for coverage and then post-processes the resulting profiles:

```
$ cat wrap_test_for_coverage.sh
#!/bin/sh
set -e
PKGARGS="$*"
#
# Setup
#
rm -rf covdatafiles
mkdir covdatafiles
#
# Pass in "-cover" to the script to build for coverage, then
# run with GOCOVERDIR set.
#
GOCOVERDIR=covdatafiles \
  /bin/sh integration_test.sh -cover $PKGARGS
```

```
#
# Post-process the resulting profiles.
#
go tool covdata percent -i=covdatafiles
$
```

Some key things to note about the wrapper above:

it passes in the “-cover” flag when running `integration_test.sh`, which gives us a coverage-instrumented “mdtool.exe” binary

it sets the GOCOVERDIR environment variable to a directory into which coverage data files will be written

when the test is complete, it runs “go tool covdata percent” to produce a report on percentage of statements covered

Here’s the output when we run this new wrapper script:

```
$ /bin/sh wrap_test_for_coverage.sh
...
    gitlab.com/golang-commonmark/mdtool coverage: 48.1% of statements
$
# Note: covdatafiles now contains 381 files.
```

Voila! We now have some idea of how well our integration tests work in exercising the “mdtool” application’s source code.

If we make changes to enhance the test harness, then do a second coverage collection run, we’ll see the changes reflected in the coverage report. For example, suppose we improve our test by adding these two additional lines to `integration_test.sh`:

```
./mdtool.exe +ty testdata/README.md  > /dev/null
./mdtool.exe +ta < testdata/README.md  > /dev/null
```

Running the coverage testing wrapper again:

```
$ /bin/sh wrap_test_for_coverage.sh
finished processing 380 files, no crashes
    gitlab.com/golang-commonmark/mdtool coverage: 54.6% of statements
$
```

We can see the effects of our change: statement coverage has increased from 48% to 54%.

Selecting packages to cover

By default, “go build -cover” will instrument just the packages that are part of the Go module being built, which in this case is the `gitlab.com/golang-commonmark/mdtool` package. In some cases however it is useful to extend coverage instrumentation to other packages; this can be accomplished by passing “-coverpkg” to “go build -cover”.

For our example program, “mdtool” is in fact largely just a wrapper around the package `gitlab.com/golang-commonmark/markdown`, so it is interesting to include `markdown` in the set of packages that are instrumented.

Here’s the `go.mod` file for “mdtool”:

```
$ head go.mod
module gitlab.com/golang-commonmark/mdtool

go 1.17

require (
    github.com/pkg/browser v0.0.0-20210911075715-681adbf594b8
    gitlab.com/golang-commonmark/markdown v0.0.0-20211110145824-
bf3e522c626a
)
```

We can use the “-coverpkg” flag to control which packages are selected for inclusion in the coverage analysis to include one of the deps above. Here’s an example:

```
$ /bin/sh wrap_test_for_coverage.sh -coverpkg=gitlab.com/golang-commonmark
/markdown,gitlab.com/golang-commonmark/mdtool
...
    gitlab.com/golang-commonmark/markdown   coverage: 70.6% of statements
    gitlab.com/golang-commonmark/mdtool coverage: 54.6% of statements
$
```

Working with coverage data files

When a coverage integration test has completed and written out a set of raw data files (in our case, the contents of the `covdatafiles` directory), we can post-process these files in various ways.

Converting profiles to ‘-coverprofile’ text format

When working with unit tests, you can run `go test -coverprofile=abc.txt` to write a text-format coverage profile for a given coverage test run.

With binaries built with `go build -cover`, you can generate a text-format profile after the fact by running `go tool covdata textfmt` on the files emitted into the GOCOVERDIR directory.

Once this step is complete, you can use `go tool cover -func=<file>` or `go tool cover -html=<file>` to interpret/visualize the data just as you would with `go test -coverprofile`.

Example:

```
$ /bin/sh wrap_test_for_coverage.sh
...
$ go tool covdata textfmt -i=covdatafiles -o=cov.txt
$ go tool cover -func=cov.txt
gitlab.com/golang-commonmark/mdtool/main.go:40:     readFromStdin   100.0%
gitlab.com/golang-commonmark/mdtool/main.go:44:     readFromFile    80.0%
gitlab.com/golang-commonmark/mdtool/main.go:54:     readFromWeb 0.0%
gitlab.com/golang-commonmark/mdtool/main.go:64:     readInput   80.0%
gitlab.com/golang-commonmark/mdtool/main.go:74:     extractText 100.0%
gitlab.com/golang-commonmark/mdtool/main.go:88:     writePreamble   100.0%
gitlab.com/golang-commonmark/mdtool/main.go:111:    writePostamble  100.0%
gitlab.com/golang-commonmark/mdtool/main.go:118:    handler     0.0%
gitlab.com/golang-commonmark/mdtool/main.go:139:    main        51.6%
total:                          (statements)    54.6%
$
```

Each execution of a "-cover" built application will write out one or more data files to the directory specified in the GOCOVERDIR environment variable. If an integration test performs N program executions, you'll wind up with O(N) files in your output directory. There is typically a lot of duplicated content in the data files, so to compact the data and/or combine data sets from different integration test runs, you can use the `go tool covdata merge` command to merge profiles together. Example:

```
$ /bin/sh wrap_test_for_coverage.sh
finished processing 380 files, no crashes
    gitlab.com/golang-commonmark/mdtool coverage: 54.6% of statements
$ ls covdatafiles
covcounters.13326b42c2a107249da22f6e0d35b638.772307.1677775306041466651
covcounters.13326b42c2a107249da22f6e0d35b638.772314.1677775306053066987
...
covcounters.13326b42c2a107249da22f6e0d35b638.774973.1677775310032569308
covmeta.13326b42c2a107249da22f6e0d35b638
$ ls covdatafiles | wc
```

```
    381     381   27401
$ rm -rf merged ; mkdir merged ; go tool covdata merge -i=covdatafiles
-o=merged
$ ls merged
covcounters.13326b42c2a107249da22f6e0d35b638.0.1677775331350024014
covmeta.13326b42c2a107249da22f6e0d35b638
$
```

The `go tool covdata merge` operation also accepts a `-pkg` flag that can be used to select out a specific package or set of packages, if that is desired.

This merge capability is also useful to combine results from different types of test runs, including runs generated by other test harnesses.

Wrap-up

That covers it: with the 1.20 release, Go's coverage tooling is no longer limited to package tests, but supports collecting profiles from larger integration tests. We hope you will make good use of the new features to help understand how well your larger and more complicated tests are working, and which parts of your source code they are exercising.

Please try out these new features, and as always if you encounter problems, file issues on our GitHub issue tracker. Thanks.