Coverage profiling support for integration tests - The Go Programming Language

Back to Go Testing

Table of Contents:

Beginning in Go 1.20, Go supports collection of coverage profiles from applications and from integration tests, larger and more complex tests for Go programs.

Overview

Go provides easy-to-use support for collecting coverage profiles at the level of package unit tests via the "`go test -coverprofile=... <pkg_target>`" command. Starting with Go 1.20, users can now collect coverage profiles for larger integration tests: more heavy-weight, complex tests that perform multiple runs of a given application binary.

For unit tests, collecting a coverage profile and generating a report requires two steps: a `go test -coverprofile=...` run, followed by an invocation of `go tool cover {-func,-html}` to generate a report.

For integration tests, three steps are needed: a build step, a run step (which may involve multiple invocations of the binary from the build step), and finally a reporting step, as described below.

Building a binary for coverage profiling

To build an application for collecting coverage profiles, pass the `-cover` flag when invoking `go build` on your application binary target. See the section below for a sample `go build -cover` invocation. The resulting binary can then be run using an environment variable setting to capture coverage profiles (see the next section on running).

How packages are selected for instrumentation

During a given “`go build -cover`” invocation, the Go command will select packages in the main module for coverage profiling; other packages that feed into the build (dependencies listed in go.mod, or packages that are part of the Go standard library) will not be included by default.

For example, here is a toy program containing a main package, a local main-module package `greetings` and a set of packages imported from outside the module, including (among others) `rsc.io/quote` and `fmt` (link to full program).

```
$ cat go.mod
module mydomain.com

go 1.20

require rsc.io/quote v1.5.2

require (
    golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c // indirect
    rsc.io/sampler v1.3.0 // indirect
)

$ cat myprogram.go
package main

import (
    "fmt"
    "mydomain.com/greetings"
    "rsc.io/quote"
)

func main() {
    fmt.Printf("I say %q and %q\n", quote.Hello(), greetings.Goodbye())
}
$ cat greetings/greetings.go
package greetings

func Goodbye() string {
    return "see ya"
}
```

```
$ go build -cover -o myprogram.exe .
$
```

If you build this program with the "`-cover`" command line flag and run it, exactly two packages will be included in the profile: `main` and `mydomain.com/greetings`; the other dependent packages will be excluded.

Users who want to have more control over which packages are included for coverage can build with the "`-coverpkg`" flag. Example:

```
$ go build -cover -o myprogramMorePkgs.exe
-coverpkg=io,mydomain.com,rsc.io/quote .
$
```

In the build above, the main package from `mydomain.com` as well as the `rsc.io/quote` and `io` packages are selected for profiling; since `mydomain.com/greetings` isn't specifically listed, it will be excluded from the profile, even though it resides in the main module.

Running a coverage-instrumented binary

Binaries built with "`-cover`" write out profile data files at the end of their execution to a directory specified via the environment variable `GOCOVERDIR`. Example:

```
$ go build -cover -o myprogram.exe myprogram.go
$ mkdir somedata
$ GOCOVERDIR=somedata ./myprogram.exe
I say "Hello, world." and "see ya"
$ ls somedata
covcounters.c6de772f99010ef5925877a7b05db4cc.2424989.1670252383678349347
covmeta.c6de772f99010ef5925877a7b05db4cc
$
```

Note the two files that were written to the directory `somedata`: these (binary) files contain the coverage results. See the following section on reporting for more on how to produce human-readable results from these data files.

If the `GOCOVERDIR` environment variable is not set, a coverage-instrumented binary will still execute correctly, but will issue a warning. Example:

```
$ ./myprogram.exe
warning: GOCOVERDIR not set, no coverage data emitted
I say "Hello, world." and "see ya"
```

```
$
```

Tests involving multiple runs

Integration tests can in many cases involve multiple program runs; when the program is built with “`-cover`”, each run will produce a new data file. Example

```
$ mkdir somedata2
$ GOCOVERDIR=somedata2 ./myprogram.exe          // first run
I say "Hello, world." and "see ya"
$ GOCOVERDIR=somedata2 ./myprogram.exe -flag    // second run
I say "Hello, world." and "see ya"
$ ls somedata2
covcounters.890814fca98ac3a4d41b9bd2a7ec9f7f.2456041.1670259309405583534
covcounters.890814fca98ac3a4d41b9bd2a7ec9f7f.2456047.1670259309410891043
covmeta.890814fca98ac3a4d41b9bd2a7ec9f7f
$
```

Coverage data output files come in two flavors: meta-data files (containing the items that are invariant from run to run, such as source file names and function names), and counter data files (which record the parts of the program that executed).

In the example above, the first run produced two files (counter and meta), whereas the second run generated only a counter data file: since meta-data doesn’t change from run to run, it only needs to be written once.

Working with coverage data files

Go 1.20 introduces a new tool, ‘`covdata`’, that can be used to read and manipulate coverage data files from a `GOCOVERDIR` directory.

Go’s `covdata` tool runs in a variety of modes. The general form of a `covdata` tool invocation takes the form

```
$ go tool covdata <mode> -i=<dir1,dir2,...> ...flags...
```

where the “`-i`” flag provides a list of directories to read, where each directories is derived from an execution of a coverage-instrumented binary (via `GOCOVERDIR`).

Creating coverage profile reports

This section discusses how to use “`go tool covdata`” to produce human-readable reports from coverage data files.

Reporting percent statements covered

To report a “percent statements covered” metric for each instrumented package, use the command “`go tool covdata percent -i=<directory>`”. Using the example from the running section above:

```
$ ls somedata
covcounters.c6de772f99010ef5925877a7b05db4cc.2424989.1670252383678349347
covmeta.c6de772f99010ef5925877a7b05db4cc
$ go tool covdata percent -i=somedata
    main    coverage: 100.0% of statements
    mydomain.com/greetings  coverage: 100.0% of statements
$
```

The “statements covered” percentages here correspond directly to those reported by `go test -cover`.

Converting to legacy text format

You can convert binary coverage data files into the legacy textual format generated by “`go test -coverprofile=<outfile>`” using the covdata `textfmt` selector. The resulting text file can then be used with “`go tool cover -func`” or “`go tool cover -html`” to create additional reports. Example:

```
$ ls somedata
covcounters.c6de772f99010ef5925877a7b05db4cc.2424989.1670252383678349347
covmeta.c6de772f99010ef5925877a7b05db4cc
$ go tool covdata textfmt -i=somedata -o profile.txt
$ cat profile.txt
mode: set
mydomain.com/myprogram.go:10.13,12.2 1 1
mydomain.com/greetings/greetings.go:3.23,5.2 1 1
$ go tool cover -func=profile.txt
mydomain.com/greetings/greetings.go:3:  Goodbye     100.0%
mydomain.com/myprogram.go:10:       main        100.0%
total:                  (statements)    100.0%
$
```

Merging

The `merge` subcommand of “`go tool covdata`” can be used to merge together profiles from multiple data directories.

For example, consider a program that runs on both macOS and on Windows. The author of this program might want to combine coverage profiles from separate runs on each operating system into a single profile corpus, so as to produce a cross-platform coverage summary. For example:

```
$ ls windows_datadir
covcounters.f3833f80c91d8229544b25a855285890.1025623.1667481441036838252
covcounters.f3833f80c91d8229544b25a855285890.1025628.1667481441042785007
covmeta.f3833f80c91d8229544b25a855285890
$ ls macos_datadir
covcounters.b245ad845b5068d116a4e25033b429fb.1025358.1667481440551734165
covcounters.b245ad845b5068d116a4e25033b429fb.1025364.1667481440557770197
covmeta.b245ad845b5068d116a4e25033b429fb
$ ls macos_datadir
$ mkdir merged
$ go tool covdata merge -i=windows_datadir,macos_datadir -o merged
$
```

The merge operation above will combine the data from the specified input directories and write a new set of merged data files to the directory “merged”.

Package selection

Most “`go tool covdata`” commands support a “`-pkg`” flag to perform package selection as part of the operation; the argument to “`-pkg`” takes the same form as that used by the Go command’s “`-coverpkg`” flag. Example:

```

$ ls somedata
covcounters.c6de772f99010ef5925877a7b05db4cc.2424989.1670252383678349347
covmeta.c6de772f99010ef5925877a7b05db4cc
$ go tool covdata percent -i=somedata -pkg=mydomain.com/greetings
    mydomain.com/greetings  coverage: 100.0% of statements
$ go tool covdata percent -i=somedata -pkg=nonexistentpackage
$
```

The “`-pkg`” flag can be used to select the specific subset of packages of interest for a given report.

Frequently Asked Questions

How can I request coverage instrumentation for all imported packages mentioned in my `go.mod` file

Can I use `go build -cover` in GOPATH/GO111MODULE=off mode?

If my program panics, will coverage data be written?

Will `-coverpkg=main` select my main package for profiling?

How can I request coverage instrumentation for all imported packages mentioned in my `go.mod` file

By default, `go build -cover` will instrument all main module packages for coverage, but will not instrument imports outside the main module (e.g. standard library packages or imports listed in `go.mod`). One way to request instrumentation for all non-stdlib dependencies is to feed the output of `go list` into `-coverpkg`. Here is an example, again using the example program cited above:

```
$ go list -f '{{if not .Standard}}{{.ImportPath}}{{end}}' -deps . | paste -sd "," > pkgs.txt
$ go build -o myprogram.exe -coverpkg=`cat pkgs.txt` .
$ mkdir somedata
$ GOCOVERDIR=somedata ./myprogram.exe
$ go tool covdata percent -i=somedata
    golang.org/x/text/internal/tag  coverage: 78.4% of statements
    golang.org/x/text/language  coverage: 35.5% of statements
    mydomain.com    coverage: 100.0% of statements
    mydomain.com/greetings  coverage: 100.0% of statements
    rsc.io/quote    coverage: 25.0% of statements
    rsc.io/sampler  coverage: 86.7% of statements
$
```

Can I use `go build -cover` in GO111MODULE=off mode?

Yes, `go build -cover` does work with `GO111MODULE=off`. When building a program in GO111MODULE=off mode, only the package specifically named as the target on the command line will be instrumented for profiling. Use the `-coverpkg` flag to include additional packages in the profile.

If my program panics, will coverage data be written?

Programs built with `go build -cover` will only write out complete profile data at the end of execution if the program invokes `os.Exit()` or returns normally from `main.main`. If a program terminates in an unrecovered panic, or if the program hits a fatal exception (such as a segmentation violation, divide by zero, etc), profile data from statements executed during the run will be lost.

Will `-coverpkg=main` select my main package for profiling?

The `-coverpkg` flag accepts a list of import paths, not a list of package names. If you want to select your `main` package for coverage instrumention, please identify it by import path, not by name. Example (using this example program):

```
$ go list -m
mydomain.com
$ go build -coverpkg=main -o oops.exe .
warning: no packages being built depend on matches for pattern main
$ go build -coverpkg=mydomain.com -o myprogram.exe .
$ mkdir somedata
$ GOCOVERDIR=somedata ./myprogram.exe
I say "Hello, world." and "see ya"
$ go tool covdata percent -i=somedata
    mydomain.com    coverage: 100.0% of statements
$
```

Resources

Blog post introducing unit test coverage in Go 1.2:
Coverage profiling for unit tests was introduced as part of the Go 1.2 release; see this blog post for details.

Documentation:
The `cmd/go` package docs describe the build and test flags associated with coverage.

Technical details:
Design draft

Proposal

Glossary

unit test: Tests within a `*_test.go` file associated with a specific Go package, utilizing Go's `testing` package.

integration test: A more comprehensive, heavier weight test for a given application or binary. Integration tests typically involve building a program or set of programs, then performing a series of runs of the programs using multiple inputs and scenarios, under control of a test harness that may or may not be based on Go's `testing` package.