GO Function Coverage

Authors: muratekici@ Contributors: icher@, pratyai@

Links: github.com/googleinterns/go-function-coverage

Table of Contents

Overview

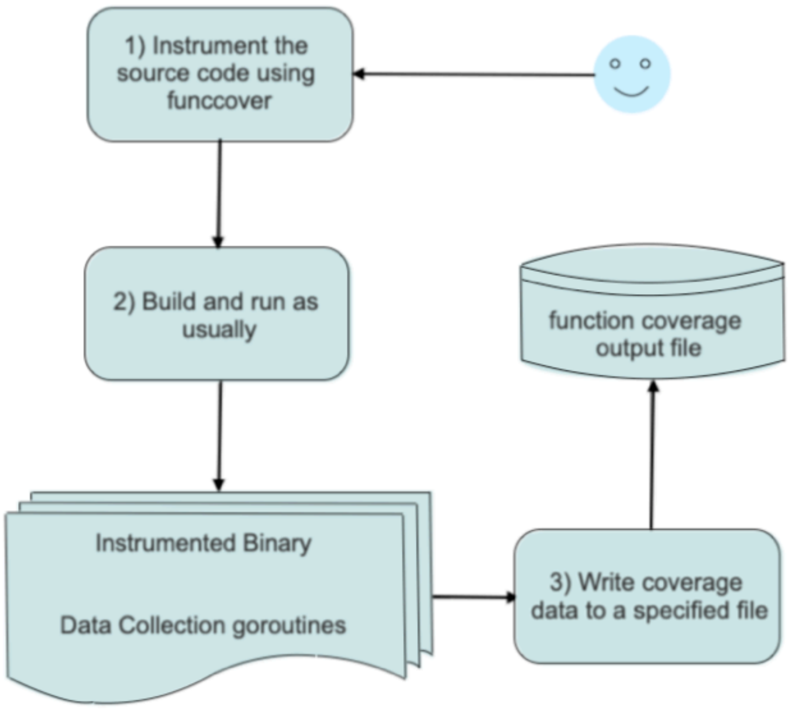
Go Function Coverage tool *funccover* supports generating instrumented source code files so that running binary automatically collects the coverage data for functions.

Function level run-time coverage data can be collected to:

1. Detect dynamically dead code as functions never run in certain production jobs.
2. Detect run-time dependencies of a test or of a production binary.

Compared to the go cover, *funccover* works with any binary and not just unit tests. Since it collects the coverage data for only tests, it adds less overhead. Performance tests will be performed.

Workflow


1) Instrument the source code using funccover
2) Build and run as usually
function coverage output file
Instrumented Binary
Data Collection goroutines
3) Write coverage data to a specified file

GO Function Coverage uses source code instrumentation which is also used in go cover. *funccover* tool injects a global function coverage variable that will keep the coverage data for functions to the given source code. Then it inserts necessary instructions to the top of each function (basic assignment instruction to global coverage variable). This way when a function starts executing, global coverage variable will keep the information that this function started execution some time.

We also have to save that coverage information in the variable somewhere. Initially *funccover* tool writes coverage information to a file (RPC will be more useful in the future). We handled this part by injecting functions that will collect the data to the source code. Currently *funccover* inserts 2 functions to the given source code, one writes coverage data to a file, other calls it periodically. Tool also inserts a defer call to the main to write coverage data also after main function ends. This way it works more general.

Actions:

1. User instruments the source code using *funccover*

```
      $ funccover [flags] [source code]
```

2. User builds and runs the instrumented source code, no additional parameters needed.

```
      $ go build [instrumented source]
      $ ./[instrumented binary]
```

3. *During runtime instrumented binary periodically writes coverage data to a specified file.*

Flags:

f*unccover* tool has 3 flags. Each flag tells how it should instrument the source code.

-period (type duration)

This flag represents the period of the data collection, if it is not given periodical collection will be disabled.

```
$ funccover -period=500ms source.go
```

-dst (type string)

This flag sets the destination file name for the instrumented source code (default "instrumented_$source.go").

```
$ funccover -dst=instrumented_source.go source.go
```

-o (type string)

This flag sets the coverage output file name (default "cover.out").

```
$ funccover -period=1s -dst=instrumented.go -o=function_coverage.out source.go
```

Instrument Package

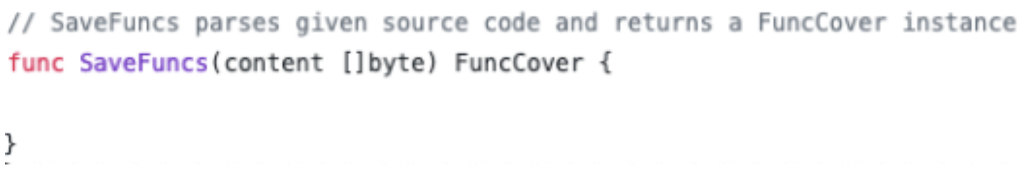
Instrument package is the implemented inside go-function-coverage module. It contains necessary type definitions and functions that will be used to instrument the source code. User gives *funccover* binary necessary flags and arguments, then using the instrument package, *funccover* instruments and writes the new source file.

Exported

```go
// FuncCoverBlock contains tha name and line of a function
// Line contains the first line of the definition in the source code
type FuncCoverBlock struct {
        Name string // Name of the function
        Line uint32 // Line number for block start.
}

// FuncCover contains the FuncCoverBlock informations of a source code
type FuncCover struct {
        FuncBlocks []FuncCoverBlock
}
```

Type FunCoverBlock and FuncCover are used to get the function information for each function in given source code.

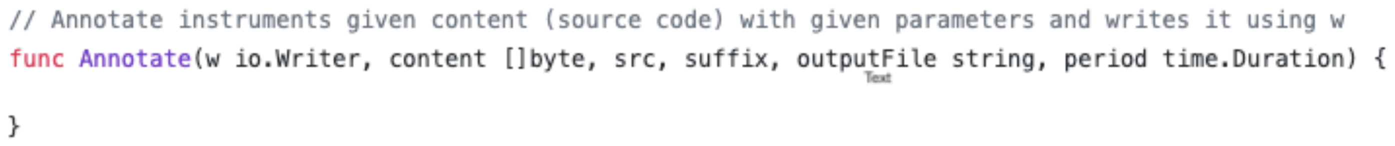


```
// SaveFuncs parses given source code and returns a FuncCover instance
func SaveFuncs(content []byte) FuncCover {

}
```

SaveFuncs function returns the FuncCover instance for given source code.

content: source code content



```
// Annotate instruments given content (source code) with given parameters and writes it using w
func Annotate(w io.Writer, content []byte, src, suffix, outputFile string, period time.Duration) {

}
```

Annotate function writes the instrumented source code using given parameters. Uses internal package functions to instrument.

w: Writer to write instrumented source code

content: source code content

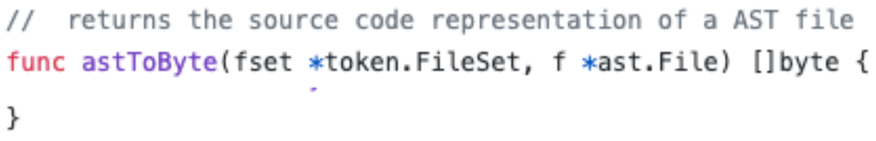
src: name of the source code file

suffix: unique suffix that will be appended to injected functions and variables

outputFile: name of the coverage output file

period: period of data collection

Internal



```
//  returns the source code representation of a AST file
func astToByte(fset *token.FileSet, f *ast.File) []byte {
}
```

astToByte function returns source code representation of given ast.

```go
// writes the declaration of funcCover variable and necessery functions
// to the end of the file using go templates
func declCover(w io.Writer, src, suffix, out string, period time.Duration, funcBlocks []FuncCoverBlock) {

}
```

declCover function writes the declaration of global counter variable, collection functions and init call for periodical collection call using given parameters.

w: Writer to write the declarations

src: name of the source code file

suffix: unique suffix that will be appended to injected functions and variables

out: name of the coverage output file

period: period of data collection

funcBlocks: funcCover data of the content

```go
// writes necessary counters for instrumentation using w
// suffix is the suffix string that will be added to the end of variables and functions
func addCounters(w io.Writer, content []byte, suffix string) {

}
```

addCounters function writes the source code with injected instructions. These instructions are assignment operations at each function and a defer collection call to the main function.

w: Writer to write instrumented source code (without declarations)

content: source code content

suffix: unique suffix that will be appended to injected functions and variables