Go Tools Explained In Color

REVISION 1

Hawthorne-Press.com

Go Tools Explained in Color

Published by

Hawthorne-Press.com

10310 Moorberry Lane
Houston, Texas 77043, USA

Table of Contents

Go Tools Explained in Color

Introduction

The Go language is a relatively new language. It is a C-Like language with a simple language structure and built-in concurrency features. It is able to easily scale across multiple processors and handle many thousands on independent concurrent routines. It was designed and is now supported by Google.

For a young language, it comes with many built-in tools that enhance the productivity of Golang Developers. This document will give a brief introduction of many of these tools.

LITEIDE

Liteide is Golang specific IDE. While other platforms provide various levels of support, currently only Liteide directly supports Golang.

Liteide supports the following features and as it is under active development, we can expect a ongoing expansion of capabilities.

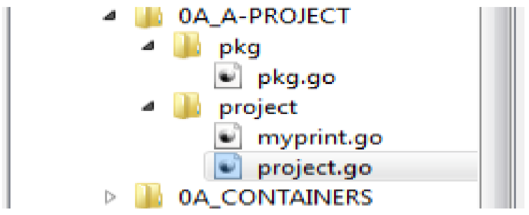
- Projects
- Code Completion
- Code Folding
- Syntax Highlighting
- Debugging
- Automatic Building
- Testing
- Automatic Formatting (Go Fmt)
- Vetting of programs (Go Vet)
- Coverage Statistics

The Liteide Feature Guide is somewhat limited in scope, but as setup document it is very helpful. See: **https://github.com/visualfc/liteide/blob/master/liteidex/deploy/welcome/en/guide.md**

For more information on Liteide: **https://gowalker.org/github.com/visualfc/liteide**

Projects and Packages

Liteide does not provide an explicit "project" management. Projects are created by managing folders and Golang Packages. Each folder contains source files for a particular *Package Name*. To include other packages, even your own, you must import them. Under an organizing folder **0A_A-PROJECT,** we will create a project with multiple files in the main program and a companion package. The following is a portion of the Liteide Folders View:



```
0A_A-PROJECT
  pkg
    pkg.go
  project
    myprint.go
    project.go
0A_CONTAINERS
```

The folder **0A_A-Project** is essentially the "Project Folder". When the developer builds the file containing the main function, the entire project is built. Be aware that the organizing element is the source file with the main function and the folder in which it resides. A folder may only contain files with the same *Package Name* and the source files in this folder determine which other *Packages* are built or used.

The Project Folder may contain other packages, but they are only used if specified by the main package. While this structure provides a project framework, it does not enforce which packages are included. Obviously, it is not good practice for a project structure to contain unrelated packages.

The following is the source code for this example project:

```
// Program project.go is demo of combining local package and two compile units.
package main

import (
    "0A_A-PROJECT/pkg"                          // Import Local Package
    "fmt"
)
// Main function - Demonstrates package and compile unit combination.
func main() {
    myPrint("this is a test!")                  // Call myPrint Function in "myprint.go"
    pkg.CapPrint()                              // Call Pkg Method CapPrint
    fmt.Println("End of Test!")                 // Print local End of Program Message
}
```

```
// Program myprint.go is a part of the "project.go" demo
package main

import (
    "fmt"
)
// myprint method prints the string parameter on standard output
func myPrint(in string) {                       // Function with a "string" paramter
    fmt.Println(in)                             // Print the input parameter
}
```

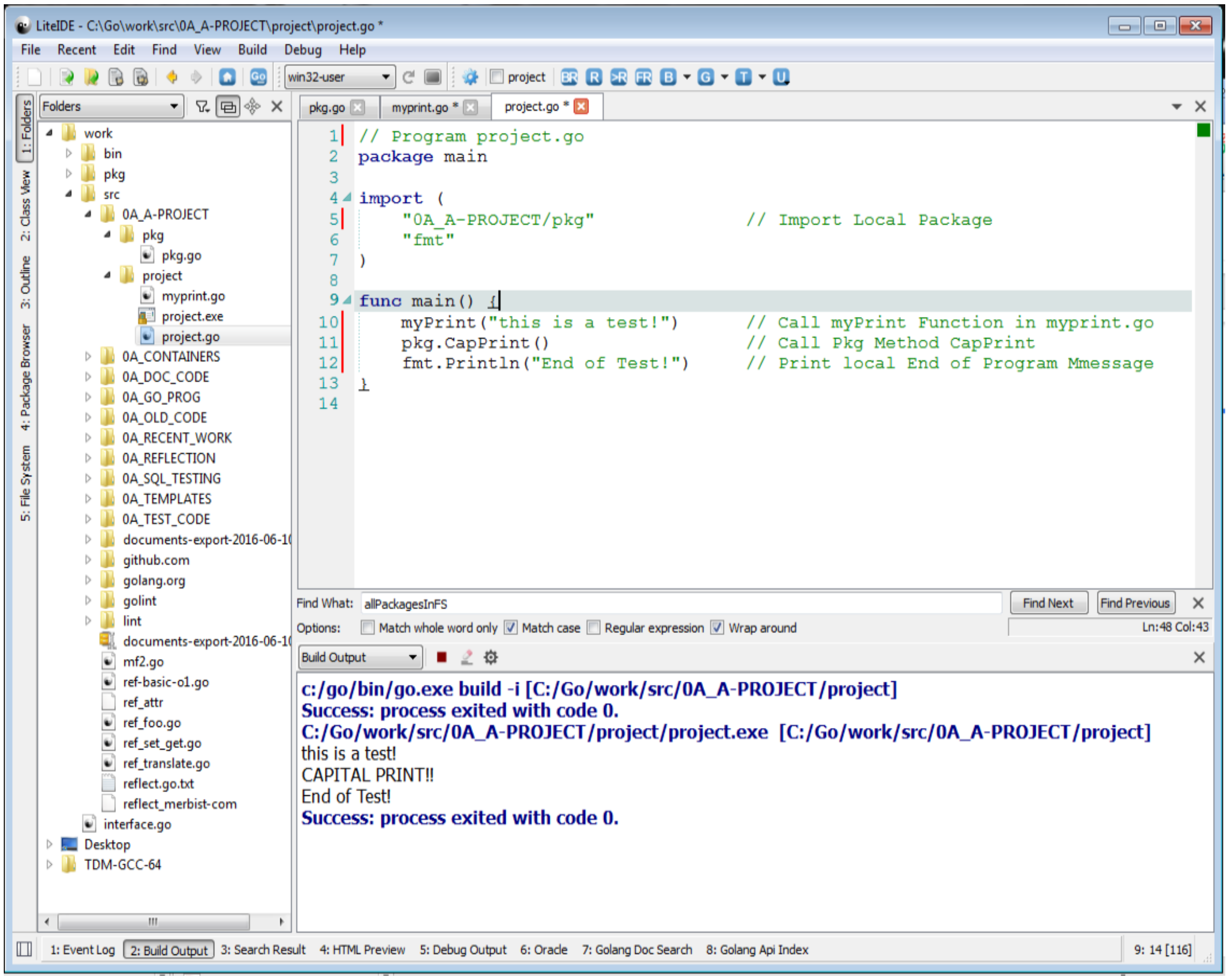
```
// pkg is an example package with one method used in the "project.go" demo
package pkg

import (
    "fmt"
)
// CapPrint is a simple print method that displays a static message for demo
func CapPrint() {                               // Method to print a fixed string
    fmt.Println("CAPITAL PRINT!!")
}
```

The result of running program project.go:

this is a test!
CAPITAL PRINT!!
End of Test!

Example Of Liteide Display

The following is a snapshot of the Liteide Display just before building project.go. *Note the following*, after the build completes the in-line comments will be formatted as defined by the **go.fmt** command. The author reformats the comments in this fashion for clarity.


LiteIDE - C:\Go\work\src\0A_A-PROJECT\project\project.go *
File Recent Edit Find View Build Debug Help
win32-user
project
Folders
1: Folders 2: Class View 3: Outline 4: Package Browser 5: File System
work
bin
pkg
src
0A_A-PROJECT
pkg
pkg.go
project
myprint.go
project.exe
project.go
0A_CONTAINERS
0A_DOC_CODE
0A_GO_PROG
0A_OLD_CODE
0A_RECENT_WORK
0A_REFLECTION
0A_SQL_TESTING
0A_TEMPLATES
0A_TEST_CODE
documents-export-2016-06-1
github.com
golang.org
golint
lint
documents-export-2016-06-1
mf2.go
ref-basic-o1.go
ref_attr
ref_foo.go
ref_set_get.go
ref_translate.go
reflect.go.txt
reflect_merbist-com
interface.go
Desktop
TDM-GCC-64
pkg.go
myprint.go *
project.go *
Find What: allPackagesInFS
Find Next
Find Previous
Options: Match whole word only [✓ Match case] Regular expression [✓ Wrap around]
Ln:48 Col:43
Build Output
1: Event Log 2: Build Output 3: Search Result 4: HTML Preview 5: Debug Output 6: Oracle 7: Golang Doc Search 8: Golang Api Index
9: 14 [116]

```go
// Program project.go
package main

import (
    "0A_A-PROJECT/pkg"              // Import Local Package
    "fmt"
)

func main() {
    myPrint("this is a test!")      // Call myPrint Function in myprint.go
    pkg.CapPrint()                  // Call Pkg Method CapPrint
    fmt.Println("End of Test!")     // Print local End of Program Mmessage
}
```

```
c:/go/bin/go.exe build -i [C:/Go/work/src/0A_A-PROJECT/project]
Success: process exited with code 0.
C:/Go/work/src/0A_A-PROJECT/project/project.exe  [C:/Go/work/src/0A_A-PROJECT/project]
this is a test!
CAPITAL PRINT!!
End of Test!
Success: process exited with code 0.
```

Golint.go

The **Golint** program is different the **go vet.** Vet is used to insure code correctness. Golint is concerned with coding style.

Golint prints suggestions and these are just suggestions. Like all *lint* programs, it will produce both false positives and false negatives. Examine each suggestion and make a considered opinion on whether to follow the suggestion. Style questions generally need a human-in-the-loop to produce better programs.

The following output was obtained by running Golint on balance.go from the document **Go Concurrency and Load Balancing Explained In Color** available at **www.hawthorne-press.com.**

```
balance.go:17:6: exported type Request should have comment or be unexported
balance.go:41:6: exported type Worker should have comment or be unexported
balance.go:55:6: exported type Pool should have comment or be unexported
balance.go:70:1: exported method Pool.Push should have comment or be unexported
balance.go:80:1: exported method Pool.Pop should have comment or be unexported
balance.go:88:6: exported type Balancer should have comment or be unexported
balance.go:93:1: comment on exported function NewBalancer should be of the form
"NewBalancer…S”
```

The reader should compare this with the actual program code and make your own decision. Since that document is already published, I will leave the program unchanged for your review. The program is programmatically “correct”, but may not be stylistically proper.

Command Vet

This command can be called within Liteide or by one of three command line methods:

- go vet package-name/path/name
- go tool vet source/directory/*.go
- go tool vet source/directory

This command checks for the following Golang patterns:

Assembly declarations	**Printf family**
Useless assignments	**Struct tags**
Atomic mistakes	**Shadowed variables**
Boolean conditions	**Shifts**
Build tags	**Unreachable code**
Invalid uses of cgo	**Misuse of unsafe Pointers**
Unkeyed composite literals	**Unused result of certain function calls**
Copying locks	**Other flags**
Documentation examples	**Subdirectories**
Methods	**Nil function comparison**

For more through discussion see: **https://golang.org/cmd/vet**

Command Gofmt

Basic Reformatting is done automatically by the Liteide build process. This command line version has a number of options to perform more complex reformatting.

gofmt [flags] [path…]

```
-d
        Do not print reformatted sources to standard output.
        If a file's formatting is different than gofmt's, print diffs
        to standard output.
-e
        Print all (including spurious) errors.
-l
        Do not print reformatted sources to standard output.
        If a file's formatting is different from gofmt's, print its name
        to standard output.
-r rule
        Apply the rewrite rule to the source before reformatting.
-s
        Try to simplify code (after applying the rewrite rule, if any).
-w
        Do not print reformatted sources to standard output.
        If a file's formatting is different from gofmt's, overwrite it
        with gofmt's version.
```

Most of the commands perform operations with reformatted code. Either listing all the file names that need reformatting, outputting the new code or the differences to standard output, or overwriting the source file with the new code.

However, two of the flags perform more subtle changes. The –s flag simplifies the code after formatting. However, these code changes can cause problems with earlier versions of Golang.

The **'–r'** flag applies a specified rewrite rule before rewriting the code.

For more details see: **https://golang.org/cmd/gofmt/**

Command Cgo

This command is used internally when a file with special constructs are present. These constructs are used to interface **'C'** code with Golang. It can directly compile **'C'** and call functions essentially using the *pkg.method* syntax with the package **'C'**. If a developer wishes to interface with **'C'** please read the reference for Cgo at:

https://golang.org/cmd/cgo/

The following is a short working example. It can be compiled with the standard Liteide **Build** command for go **build <program>** syntax.

```
package main

// typedef int (*intFunc) ();           // 'C' Comments can be added without corrupting 'C' Code
//
```

```
//                                      // No blank Lines:  Each line must be a comment
//
// int bridge_int_func(intFunc f)        /* Original 'C' Comments are allowed */
// {
//      return f();
// }
//
// int fortytwo()
// {
//      return 42;
// }
import "C"
import "fmt"

func main() {
       f := C.intFunc(C.fortytwo)
       fmt.Println(int(C.bridge_int_func(f)))
}
```

Output Result: 42

Command Compile

The Compile command provides a number of specialized ways to compile Golang files and produce various outputs in addition to the normal compiled output. The command is invoked as follows:

go tool compile [flags] file…

This command has numerous flags that are displayed by issuing the command without arguments:

```
$ go tool compile
usage: compile [options] file.go...
  -%    debug non-static initializers
  -+    compiling runtime
  -A    for bootstrapping, allow 'any' type
  -B    disable bounds checking
  -D path
   ::      ::      ::      ::      ::
  -x    debug lexer
  -y    debug declarations in canned imports (with -d)
```

The compiler also accepts directives in the form of comments of the form: **//<directive>.** The directives must not have a space after the comment indicator.

- //line path/to/file:linenumber
- //go:noescape
- //go:nosplit
- //go:linkname localname importpath.name

The purpose of these directives and flags are fully explained in the document

https://golang.org/cmd/compile/.

As of version 1.5, the compiler and runtime no longer require 'C' code, Just Go code and some Assembler.

Command Asm

The Go assembler is loosely based on the **Plan 9** assembler syntax. See **https://9p.io/sys/doc/asm.html** for related documentation. Be aware that some of the documentation refers to Plan 9 specific features. This assembler is used for all support platforms as specified by GOOS and GOARCH environment variables.

The assembler usage display is presented when the command is executed without arguments. The invoking syntax and usage are as follows:

go tool asm [flags] file

```
usage: asm [options] file.s
Flags:
  -D value
        predefined symbol with optional simple value -D=identifer=value; can be set multiple times
  -I value
        include directory; can be set multiple times
  -S    print assembly and machine code
  -debug
        dump instructions as they are parsed
  -dynlink
        support references to Go symbols defined in other shared libraries
  -e    no limit on number of errors reported
  -o string
        output file; default foo.6 for /a/b/c/foo.s on amd64
  -shared
        generate code that can be linked into a shared library
  -trimpath string
        remove prefix from recorded source file paths
```

See **https://golang.org/cmd/asm/** for more complete description of the assembler documentation.

Command Cover

The Cover command usage display is presented when the command is executed without arguments or adding the –help flag. This command can also be invoked from Go Test. The invoking syntax and usage are as follows:

go tool cover	-- Display usage
go tool cover –help	-- Display usage
go tool cover [flags] file	-- Compute coverage for a <file>

```
$ go tool cover
Usage of 'go tool cover':
Given a coverage profile produced by 'go test':
        go test -coverprofile=c.out
```

```
Open a web browser displaying annotated source code:
        go tool cover -html=c.out

Write out an HTML file instead of launching a web browser:
        go tool cover -html=c.out -o coverage.html

Display coverage percentages to stdout for each function:
        go tool cover -func=c.out

Finally, to generate modified source code with coverage annotations
(what go test -cover does):
        go tool cover -mode=set -var=CoverageVariableName program.go

Flags:
  -func string
        output coverage profile information for each function
  -html string
        generate HTML representation of coverage profile
  -mode string
        coverage mode: set, count, atomic
  -o string
        file for output; default: stdout
  -var string
        name of coverage variable to generate (default "GoCover")

  Only one of -html, -func, or -mode may be set.
```

The author believes that in addition to Test Driven Development, the HTML representation of the coverage profile is very useful in visualizing missing coverage. We will build a simple test program for the *project.go* program we built earlier. The coverage program only works on the Go files in a particular folder. Even though the entire program is tested, the coverage output is only for the files in *0A_A-PROJECT/project* folder.

The test program **project_test.go** is shown below:

```go
package main

import (
    "testing"
)

func TestAll(t *testing.T) {
    main()
}
```

Running test produces the following display and the coverage file **cover.out**.

```
$ go test -coverprofile=cover.out
this is a test!
CAPITAL PRINT!!
End of Test!
PASS
coverage: 100.0% of statements
ok      0A_A-PROJECT/project    0.115s
```

The following is the contents of file cover.out:

```
$ cat cover.out
mode: set
0A_A-PROJECT\project\myprint.go:8.25,12.2 1 1
0A_A-PROJECT\project\project.go:9.13,13.2 3 0
```

To covert the **cover.out** to a displayable format and display in browser window:

Go tool cover –html=cover.out

The following are the displayed output for the two project files:

```
// Program project.go
package main

import (
    "0A_A-PROJECT/pkg" // Import Local Package
    "fmt"
)

func main() {
    myPrint("this is a test!")  // Call myPrint Function in myprint.go
    pkg.CapPrint()              // Call Pkg Method CapPrint
    fmt.Println("End of Test!") // Print local End of Program Mmessage
}
```

```
// Program myprint.go
package main

import (
    "fmt"
)

func myPrint(in string) {
    fmt.Println(in)
}
```

As reported originally, this simple program has 100% coverage. Only the statements that generate code are displayed in green for covered, or red for uncovered. The code that does not directly generate code is shown in grey.

In Larger programs there will generally be code that cannot be tested by the Go's testing facilities. For well-designed test programs, this will normally be a small percentage of the code. This is where the HTML output of coverage statistics is very helpful to visualizing untested code.

For an example places where code is unreachable download **GO_Testing_Source_and_Examples.tgz** from **www.hawthorne-press.com** under Downloads. The file **coverage.html** shows the HTML output from the test program. The unreachable code consisted of two error checks and the main function since only the support functions were tested. The error checks were manually tested to insure they functioned.

The documentation at **https://golang.org/cmd/cover/** is sparse and the usage display will be more helpful.

Command Doc

This command is used to extract documentation from properly structured source files.

The **doc** command basic format is:

```
go doc <pkg>
go doc <sym>[.<method>]
go doc [<pkg>.]<sym>[.<method>]
go doc [<pkg>.][<sym>.]<method>
```

Some examples from the documentation:

```
go doc
        Show documentation for current package.
go doc Foo
        Show documentation for Foo in the current package.
        (Foo starts with a capital letter so it cannot match
        a package path.)
go doc encoding/json
        Show documentation for the encoding/json package.
go doc json
        Shorthand for encoding/json.
go doc json.Number (or go doc json.number)
        Show documentation and method summary for json.Number.
go doc json.Number.Int64 (or go doc json.number.int64)
        Show documentation for json.Number's Int64 method.
```

Unless the '-c' flag indicating that "case" should be respected, the following are equivalent:

```
go doc json.Decoder.Decode
go doc json.decoder.decode
go doc json.decode
```

Command doc flags:

```
-c
        Respect case when matching symbols.
-cmd
        Treat a command (package main) like a regular package.
        Otherwise package main's exported symbols are hidden
        when showing the package's top-level documentation.
-u
        Show documentation for unexported as well as exported symbols and methods.
-<anything else>
        Displays usage information
```

For a more through discussion of parameters, and the documentation selection process, see the following:

https://golang.org/src/cmd/go/doc.go?m=text

The options for the **doc** command allow selecting a *package, const, func, type, var, or method*. Any documentation associated with an element, and of the proper form, will be displayed on standard output.

Source Code Documentation Guidelines

In order for source documentation to be found by the **doc** command, it must be in the right place. Beyond this there are guidelines for content depending on the element being documented.

The documentation for an element must immediately precede the element without spaces.

Package Element Documentation

The first sentence of the package documentation will be shown on package lists. It is important clearly describe the purpose of the package. While spaces are not allowed between the element and the documentation, this author puts a blank comment line for esthetics. This is a personal trait and is not required by any standard.

```
// The driveLib package provides Sqlite3 Direct Driver Methods.
//
package driveLib
```

The following command will display the basic information for the package "drive_basic"

$ go doc drive_basic

```

package drive_basic // import "0A_SQL_TESTING/drive_basic"

The drive_basic package provides Sqlite3 Direct Driver Methods.

func CreateTable(db *sqlite3.Conn)
func InitDB(filepath string) *sqlite3.Conn
func ReadItem(db *sqlite3.Conn) []TestItem
func StoreItem(db *sqlite3.Conn, items []TestItem)
type TestItem struct { ... }
```

Other Elements

Documenting other elements follows the same rules. Comments are place directly preceding the element.

```
// A Structure for holding Data Record Values
//
type TestItem struct {
    Id    string
    Name  string
    Phone string
}

// Unexported value Documentation
//
var unexported int

// A Method to Open and Initialize Database
//
func InitDB(filepath string) *sqlite3.Conn {
   ::     ::     ::     ::     ::     ::
}
```

The following doc commands produce the displayed results. The authors comments are in red.

```
$ go doc TestItem
type TestItem struct {
        Id    string
        Name  string
        Phone string
}
    A Structure for holding Data Record Values

$ go doc -u unexported        // Unexported values only displayed when '-u' flag is present
var unexported int
    Unexported value Documentation

$ go doc initDB
func InitDB(filepath string) *sqlite3.Conn
    A Method to Open and Initialize Database
```

Things to notice, since we know the elements we want the package name does not need to be specified. Additionally, using the '-u' flag allows the display of un-exported elements.

The basic display structure is the *go element* followed by the *element documentation*.

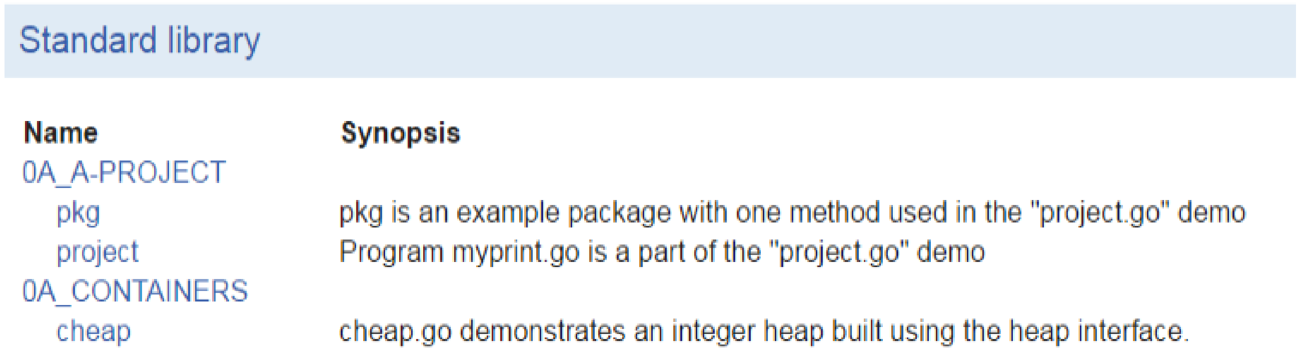
Godoc Program

There is another program in the Go family that also displays documentation. See documentation at:

http://godoc.org/golang.org/x/tools/cmd/godoc

This program has many features, but the most important one is the browser interface. Setup your browser for **localhost:6060** and issue the command **godoc –http=:6060.**

This will display the same page as **https://golang.org/doc/** with one important difference. When you list "Packages", it will list your packages before listing the standard library packages. Below is an excerpt from my Go packages list. 0A_A-PROJECT contains the code presented at the top of this document.



Standard library

Name	Synopsis
0A_A-PROJECT	
pkg	pkg is an example package with one method used in the "project.go" demo
project	Program myprint.go is a part of the "project.go" demo
0A_CONTAINERS	
cheap	cheap.go demonstrates an integer heap built using the heap interface.

Always document your source as described above. Good documentation is one of the corner stones of good program management.

Command Fix

The command fix tool examines source code for old API's and rewrites them to use newer ones. It has several modes as shown below:

go tool fix [-r fixname,...] [-force fixname] [-diff] [path ...]

While the Go language goes to great lengths to stay backwardly compatible, the language is still young and growing. Old API's generally will work, even though depreciated, but new API's with expanded capacities take their place. The fix tool finds API's that can be replaced and rewrites the code in place.

This tool does not make backup copies of the files it rewrites. A developer should use the **'–diff '** flag to inspect the changes before committing to rewriting files. When rewriting files, fix prints a line to standard error indicating the file name and the rewrite applied.

The list of possible available rewrites is listed by the following command:

```
$ go tool fix -help
usage: go tool fix [-diff] [-r fixname,...] [-force fixname,...] [path ...]
  -diff
        display diffs instead of rewriting files
  -force string
        force these fixes to run even if the code looks updated
  -r string
        restrict the rewrites to this comma-separated list

Available rewrites are:

gotypes
        Change imports of golang.org/x/tools/go/{exact,types} to go/{constant,types}

netipv6zone
        Adapt element key to IPAddr, UDPAddr or TCPAddr composite literals.

        https://codereview.appspot.com/6849045/

printerconfig
        Add element keys to Config composite literals.
```

Options

- Without a specified **path**, the tool reads standard input and writes the modified code to standard output.
- If the path is a file name, fix rewrites to file in place.
- If the path is a directory, fix rewrites all the files in that directory tree.
- The **'–diff'** flag uses the "diff" functionality to display the proposed rewrites.
- The **'-r'** restricts rewrites to the listed rewrite names. Normally, all rewrite types are considered.

The documentation for this command is available at the following wed address.

https://golang.org/cmd/fix/

Command Objdump

This program extracts and prints a disassembly of all code information from Go object files. This command has two modes:

Go tool objdump [-s symregexp] binary

Go tool objdump binary start end

The first mode prints all the disassembly information in the object file unless the '-s' flag is present. If present is limits the disassembly to text symbols matching the specified regular expression.

The second form prints all disassembly information between the specified starting an ending address. See documentation for this mode.

For complete documentation see:

https://golang.org/cmd/objdump/

Program Objdump

There is a *program* named **objdump** that has many more options. This is only related to the **command objdump** by the fact that is also extracts information from Go object files. The author has not found any documentation other than the usage display presented by issuing the following command:

objdump

A developer with an urge to explore may find the program interesting. Some of the major options are:

- Dump of symbol table information
- Disassembly
- Display of Debugging information

Command Pack

This command is a simplified version of an archive tool, such as the Unix command **ar**. This version is very simple, as it only supports five commands. The command sequence is defined as follows:

go tool pack <cmd> <archive_name> [filename...]

The commands consist of a single character.

```
c       append files (from the file system) to a new archive
p       print files from the archive
r       append files (from the file system) to the archive
t       list files from the archive
x       extract files from the archive
```

The **'c'** archive must be a valid archive file or non-existent in the case of archive creation.

The **'p'**, **'t'**, and **'x'** commands that do not specify filenames will apply the operation to the entire archive.

The **'r'** command always appends to the archive, even if the file of the same name is already present.

Adding the letter **'v'** to a command enables "verbose" mode.

For a more complete description of this command see:

https://golang.org/cmd/pack/

COMMANDS PPROF, TRACE, AND YACC

The commands below are beyond the scope of this document. They provide the following services:

- Pprof – provides program profiling visualization
  See: **https://golang.org/pkg/net/http/pprof/**
- Trace – a tool for viewing trace files
  See: **https://golang.org/cmd/trace/**
- Yacc – A version of yacc written in Go and generates parsers written in Go.
  See: **https://golang.org/cmd/yacc/**

A Final Word

Strive for simplicity in your code. The *Occam's Razor principle* applies as strongly for software as it does for many other walks of life. In most situations, giving up a little in efficiency to simplify your code is often the right decision. Code Simplicity reduces the chances of subtle errors.

As always, comments are welcome at hawthornepresscom@gmail.com.

As to why I have an email address on **gmail** instead of on my website, has to do with AT&T and their inability to handle the concept that not everyone is running a Microsoft OS and my stubborn insistence on running Linux.

Published by

Hawthorne-Press.com

10310 Moorberry Lane
Houston, Texas 77043, USA

Revision History

Date	By	Section	Changes
07/14/2016	C.E. Thornton	All	Initial Document