RX-M / go Private

Branch: master go / modules / program-construction / rx-m-go-program-construction.md Find file Copy path

ronaldpetty updated program construction lab, normalized file names largely ffbad02 on Sep 8

3 contributors

698 lines (494 sloc) 20.4 KB

RX-M Cloud Native Consulting

Go

Go program structure

A Go program consists of various parts, in particular:

- Package declarations
- Package imports
- Function declarations
- Variable declarations
- Statements & expressions
- Comments

In this lab we will explore additional Go programming basics.

1. The GOPATH

The `go` tool requires you to organize your code in a specific way. Go programmers typically keep all of their Go code in a single workspace. A workspace contains many version control repositories (managed by Git, for example). Each repository contains one or more packages.

Each package consists of one or more Go source files in a single directory. The path to a package's directory determines its import path. This differs from other programming environments in which every project has a separate workspace and workspaces are closely tied to version control repositories.

A Go workspace is a directory hierarchy with three directories at its root.

- `$GOPATH/src` - contains Go source files,
- `$GOPATH/pkg` - contains package objects
- `$GOPATH/bin` - contains executable commands

The `go` tool builds source packages and installs the resulting binaries in the `pkg` (for libraries) and `bin` (for binaries) directories. The `src` subdirectory typically contains multiple version control repositories that track the development of one or more source packages.

List the files in your `$HOME/go/src` directory:

```
user@ubuntu:~/go/src/lab-data-types$ cd

user@ubuntu:~$



user@ubuntu:~$ ls ~user/go/src/

golang.org  lab-data-types  lab-overview  lab-syntax
user@ubuntu:~$
```

Now we know why we placed `lab-overview` in the `~user/go/src/` directory. We currently have not created `~/go/pkg` or `~/go/bin` directly, we will see more on this later.

For now lets create a new directory for our lab-program-construction code and run a test program:

```
user@ubuntu:~$ mkdir ~user/go/src/lab-program-construction

user@ubuntu:~$
```

```
user@ubuntu:~$ cd ~user/go/src/lab-program-construction

user@ubuntu:~/go/src/lab-program-construction$
```

```
user@ubuntu:~/go/src/lab-program-construction$ vim program-construction.go
user@ubuntu:~/go/src/lab-program-construction$ cat program-construction.go

package main

import "fmt"

func main() {
        fmt.Println("hello world")
}
user@ubuntu:~/go/src/lab-program-construction$
```

```
user@ubuntu:~/go/src/lab-program-construction$ go run program-construction.go

hello world
user@ubuntu:~/go/src/lab-program-construction$
```

This program is the same as the hello program created in the overview lab, the goal here is to understand how `go` is placing and/or finding things.

There are two key environment variables that allow our Go operations to succeed:

- GOPATH - the location of your Go workspace
- GOROOT - the install directory of the Go binaries

Display the environment variables on your lab system that have the word Go in them:

```
user@ubuntu:~/go/src/lab-program-construction$ env | grep -i go

PATH=/usr/local/go/bin:/usr/local/go/bin:/home/user/bin:/home/user/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbi

PWD=/home/user/go/src/lab-program-construction
user@ubuntu:~/go/src/lab-program-construction$
```

As you can see neither of the critical Go variables is set. In our case, Go is using the default values for both.

Display the environment variables used by Go:

```
user@ubuntu:~/go/src/lab-program-construction$ go env | grep -i -e goroot -e gopath

GOPATH="/home/user/go"
GOROOT="/usr/local/go"
user@ubuntu:~/go/src/lab-program-construction$
```

- Use `go env` to review other settings.

While all of these settings can be useful under certain circumstances, the GOPATH and GOROOT are always important. You can display just the GOROOT with `go env` :

```
user@ubuntu:~/go/src/lab-program-construction$ go env GOROOT

/usr/local/go
user@ubuntu:~/go/src/lab-program-construction$
```

In the example the GOROOT is `/usr/local/go` . This is where we moved our uncompressed Go distribution (now as you see, not by chance).

List the contents of the GOROOT directory:

```
user@ubuntu:~/go/src/lab-program-construction$ ls -l /usr/local/go/

total 176
drwxr-xr-x  2 user user  4096 Aug 24 14:50 api
-rw-r--r--  1 user user 41258 Aug 24 14:50 AUTHORS
drwxr-xr-x  2 user user  4096 Aug 24 14:51 bin
drwxr-xr-x  4 user user  4096 Aug 24 14:51 blog
-rw-r--r--  1 user user  1576 Aug 24 14:50 CONTRIBUTING.md
-rw-r--r--  1 user user 55577 Aug 24 14:50 CONTRIBUTORS
drwxr-xr-x  9 user user  4096 Aug 24 14:50 doc
-rw-r--r--  1 user user  5686 Aug 24 14:50 favicon.ico
drwxr-xr-x  3 user user  4096 Aug 24 14:50 lib
-rw-r--r--  1 user user  1479 Aug 24 14:50 LICENSE
drwxr-xr-x 14 user user  4096 Aug 24 14:51 misc
-rw-r--r--  1 user user  1303 Aug 24 14:50 PATENTS
drwxr-xr-x  7 user user  4096 Aug 24 14:51 pkg
-rw-r--r--  1 user user  1601 Aug 24 14:50 README.md
-rw-r--r--  1 user user    26 Aug 24 14:50 robots.txt
drwxr-xr-x 46 user user  4096 Aug 24 14:50 src
drwxr-xr-x 19 user user 12288 Aug 24 14:50 test
-rw-r--r--  1 user user     5 Aug 24 14:50 VERSION
user@ubuntu:~/go/src/lab-program-construction$
```

Now list the contents of the Go `bin` directory:

```
user@ubuntu:~/go/src/lab-program-construction$ ls -l $(go env GOROOT)/bin

total 28280
-rwxr-xr-x 1 user user 10369401 Aug 24 14:51 go
-rwxr-xr-x 1 user user 15325248 Aug 24 14:51 godoc
-rwxr-xr-x 1 user user  3257829 Aug 24 14:51 gofmt
user@ubuntu:~/go/src/lab-program-construction$
```

List the Go `pkg` (package) directory:

```
user@ubuntu:~/go/src/lab-program-construction$ ls -l $(go env GOROOT)/pkg

total 20
drwxr-xr-x  2 user user 4096 Aug 24 14:50 include
drwxr-xr-x 30 user user 4096 Aug 24 14:51 linux_amd64
drwxr-xr-x 29 user user 4096 Aug 24 14:51 linux_amd64_race
drwxr-xr-x  3 user user 4096 Aug 24 14:50 obj
drwxr-xr-x  3 user user 4096 Aug 24 14:50 tool
user@ubuntu:~/go/src/lab-program-construction$
```

Now display everything under the package `tool` directory:

```
user@ubuntu:~/go/src/lab-program-construction$ ls -l $(go env GOROOT)/pkg/tool

total 4
drwxr-xr-x 2 user user 4096 Aug 24 14:51 linux_amd64
user@ubuntu:~/go/src/lab-program-construction$
```

Drill down:

```
user@ubuntu:~/go/src/lab-program-construction$ ls -l $(go env GOROOT)/pkg/tool/linux_amd64

total 92364
-rwxr-xr-x 1 user user  3521516 Aug 24 14:51 addr2line
-rwxr-xr-x 1 user user  3825570 Aug 24 14:51 asm
-rwxr-xr-x 1 user user  4015761 Aug 24 14:51 cgo
-rwxr-xr-x 1 user user 15715999 Aug 24 14:51 compile
-rwxr-xr-x 1 user user  4992472 Aug 24 14:51 cover
-rwxr-xr-x 1 user user  3292713 Aug 24 14:51 dist
-rwxr-xr-x 1 user user  3945072 Aug 24 14:51 doc
-rwxr-xr-x 1 user user  2990733 Aug 24 14:51 fix
-rwxr-xr-x 1 user user  4488971 Aug 24 14:51 link
-rwxr-xr-x 1 user user  3486821 Aug 24 14:51 nm
```

```
-rwxr-xr-x 1 user user  3836195 Aug 24 14:51 objdump
-rwxr-xr-x 1 user user  2085023 Aug 24 14:51 pack
-rwxr-xr-x 1 user user 11229753 Aug 24 14:51 pprof
-rwxr-xr-x 1 user user 11382976 Aug 24 14:51 tour
-rwxr-xr-x 1 user user  9334367 Aug 24 14:51 trace
-rwxr-xr-x 1 user user  6409401 Aug 24 14:51 vet
user@ubuntu:~/go/src/lab-program-construction$
```

The `go` program and its tools are installed in the paths listed above.

Our GOPATH is using the default `$HOME/go`, the default as of Go 1.8.

When we build and install our user binaries, the GOPATH will be used as the base directory. The `go install` command is used to build and install binaries from your sources.

Create a binary for your `program-construction.go` source and then list the contents of the `bin` directory under your GOPATH:

```
user@ubuntu:~/go/src/lab-program-construction$ go install

user@ubuntu:~/go/src/lab-program-construction$
```

```
user@ubuntu:~/go/src/lab-program-construction$ ls -l ~/go/bin

total 6640
-rwxrwxr-x 1 user user 4939051 Sep  8 14:17 goimports
-rwxrwxr-x 1 user user 1855827 Sep  8 17:07 lab-program-construction
user@ubuntu:~/go/src/lab-program-construction$
```

Notice the `bin` directory was created for us. Go places executables in `GOPATH/bin`. The lab-program-construction executable was generated and placed by `go install`.

Confirm no obj or binary is in the source directory.

```
user@ubuntu:~/go/src/lab-program-construction$ ls

program-construction.go
user@ubuntu:~/go/src/lab-program-construction$
```

Using `go help` we can learn more about the `go install` subcommand.

```
user@ubuntu:~/go/src/lab-program-construction$ go help install

usage: go install [build flags] [packages]

Install compiles and installs the packages named by the import paths,
along with their dependencies.

For more about the build flags, see 'go help build'.
For more about specifying packages, see 'go help packages'.

See also: go build, go get, go clean.
user@ubuntu:~/go/src/lab-program-construction$
```

You can use the `go help gopath` command to get more help on the GOPATH:

```
user@ubuntu:~/go/src/lab-program-construction$ go help gopath

The Go path is used to resolve import statements.
It is implemented by and documented in the go/build package.

The GOPATH environment variable lists places to look for Go code.
On Unix, the value is a colon-separated string.
On Windows, the value is a semicolon-separated string.
On Plan 9, the value is a list.

If the environment variable is unset, GOPATH defaults
to a subdirectory named "go" in the user's home directory
($HOME/go on Unix, %USERPROFILE%\go on Windows),
unless that directory holds a Go distribution.
```

```
Run "go env GOPATH" to see the current GOPATH.
...
user@ubuntu:~/go/src/lab-program-construction$
```

2. go get

Like many languages, the Go language has a package manager. The `go get` subcommand is used to retrieve packages from various repositories.

Browse to the following IRI https://github.com/RX-M/godemo.

This is a typical Go package archive.

Use `go get` to install the godemo package.

```
user@ubuntu:~/go/src/lab-program-construction$ go get github.com/rx-m/godemo

user@ubuntu:~/go/src/lab-program-construction$
```

Now examine the result of the installation.

```
user@ubuntu:~/go/src/lab-program-construction$ ls -l $(go env GOPATH)

total 12
drwxrwxr-x 2 user user 4096 Sep  8 17:07 bin
drwxrwxr-x 3 user user 4096 Sep  8 14:17 pkg
drwxrwxr-x 8 user user 4096 Sep  8 17:09 src
user@ubuntu:~/go/src/lab-program-construction$
```

The `go get` operation created the pkg directory for the new package. To see the entire directory structure let's use the `tree` command (installing it first!):

```
user@ubuntu:~/go/src/lab-program-construction$ sudo apt install tree -y
```

```
user@ubuntu:~/go/src/lab-program-construction$



user@ubuntu:~/go/src/lab-program-construction$ tree -I "golang.org" ~/go

/home/user/go
├── bin
│   ├── goimports
│   └── lab-program-construction
├── pkg
│   └── linux_amd64
│       └── github.com
│           └── rx-m
│               └── godemo.a
└── src
    ├── github.com
    │   └── rx-m
    │       └── godemo
    │           ├── double.go
    │           └── README.md
    ├── lab-data-types
    │   ├── data-types-1.go
    │   ├── data-types-2.go
    │   ├── data-types-3b.go
    │   ├── data-types-3c.go
    │   └── data-types-3.go
    ├── lab-overview
    │   ├── hello
    │   ├── hello.go
    │   ├── op.go
    │   └── ws.go
    ├── lab-program-construction
    │   └── program-construction.go
    └── lab-syntax
        └── syntax.go

13 directories, 16 files
user@ubuntu:~/go/src/lab-program-construction$
```

The pkg directory holds user built libraries.

Review the code in the repo we just cloned.

```
user@ubuntu:~/go/src/lab-program-construction$ cat ~/go/src/github.com/rx-m/godemo/double.go

//Package godemo is a simple package exporting the Double function
package godemo

//Double returns two times the number passed
func Double(x float64) float64 {
    return x + x
}
user@ubuntu:~/go/src/lab-program-construction$
```

Go has cloned the repo specified into the `src` directory using the full IRI path and compiled the package, placing the object file in the `pkg/[platform]` directory for the target platform.

Let's try using the newly install package:

```
user@ubuntu:~/go/src/lab-program-construction$ vim program-constructionb.go
user@ubuntu:~/go/src/lab-program-construction$ cat program-constructionb.go

package main

import "fmt"
import "github.com/rx-m/godemo"

func main() {
        x := 6.23
        fmt.Println(godemo.Double(x))
}
user@ubuntu:~/go/src/lab-program-construction$
```

Run the above program:

```
user@ubuntu:~/go/src/lab-program-construction$ go run program-constructionb.go

12.46
user@ubuntu:~/go/src/lab-program-construction$
```

By using `import "github.com/rx-m/godemo"` we are able to access publicly accessible members of the godemo package.

You can use the `go get -u` switch to update existing packages.

Properly commented packages allow us to retrieve help text with `go doc` .

Try it:

```
user@ubuntu:~/go/src/lab-program-construction$ go doc godemo

package godemo // import "github.com/rx-m/godemo"

Package godemo is a simple package exporting the Double function

func Double(x float64) float64
user@ubuntu:~/go/src/lab-program-construction$
```

You can use dot (".") notation to get help on specific elements of the package.

Try it:

```
user@ubuntu:~/go/src/lab-program-construction$ go doc godemo.Double

package godemo // import "github.com/rx-m/godemo"

func Double(x float64) float64
    Double returns two times the number passed

user@ubuntu:~/go/src/lab-program-construction$
```

Use the help system to learn more about `go get`.

```
user@ubuntu:~/go/src/lab-program-construction$ go help get

usage: go get [-d] [-f] [-fix] [-insecure] [-t] [-u] [build flags] [packages]

Get downloads the packages named by the import paths, along with their
dependencies. It then installs the named packages, like 'go install'.
...
user@ubuntu:~/go/src/lab-program-construction$
```

3. fmt, bufio, and os packages

Go includes a large standard library of packages. Much of the work of mastering Go is learning your way around the included system packages. To get some experience with Go packages and the standard library, we will explore the fmt package in this step.

To begin, look up the online help topic for the fmt package in a browser at https://golang.org/pkg/fmt/.

Notice the path of the IRI. Go is all about being simple.

- What do you think the documentation path would be for the Go `os` package?

Run `go doc` on the fmt package:

```
user@ubuntu:~/go/src/lab-program-construction$ go doc fmt

package fmt // import "fmt"

Package fmt implements formatted I/O with functions analogous to C's printf
and scanf. The format 'verbs' are derived from C's but are simpler.


Printing

The verbs:
```

```
General:

    %v  the value in a default format
        when printing structs, the plus flag (%+v) adds field names
    %#v a Go-syntax representation of the value
    %T  a Go-syntax representation of the type of the value
    %%  a literal percent sign; consumes no value

Boolean:
...
user@ubuntu:~/go/src/lab-program-construction$
```

Compare the web page to the `go doc` output.

- Are there any differences between the web and the go doc output?
- Which would be more likely to display the exact details of the version of Go you are using?

There are three print functions that output to the STDOUT stream:

- func Print(a ...interface{}) (n int, err error)
- func Printf(format string, a ...interface{}) (n int, err error)
- func Println(a ...interface{}) (n int, err error)

Print and Println are identical except that Println adds a line feed to the end of the output and Print does not place spaces between strings in the argument list.

Try both in your `program-constructionc.go` source:

```
user@ubuntu:~/go/src/lab-program-construction$ vim program-constructionc.go
user@ubuntu:~/go/src/lab-program-construction$ cat program-constructionc.go

package main

import "fmt"
```

```
func main() {
  fmt.Print("Hi", "there")
  fmt.Println("Hi", "there")
}
user@ubuntu:~/go/src/lab-program-construction$
```

Run your program.

You should get output like: "HithereHi there"

Per the `fmt` documentation we can use the `Scanln` method to read lines of text from stdin.

Try it:

```
user@ubuntu:~/go/src/lab-program-construction$ vim program-constructiond.go
user@ubuntu:~/go/src/lab-program-construction$ cat program-constructiond.go

package main

import "fmt"

func main() {
        var name string
        fmt.Print("Enter your name: ")
        fmt.Scanln(&name)
        fmt.Println("Hi", name)
}
user@ubuntu:~/go/src/lab-program-construction$
```

Run your program:

```
user@ubuntu:~/go/src/lab-program-construction$ go run program-constructiond.go

Enter your name: Bob
Hi Bob
user@ubuntu:~/go/src/lab-program-construction$
```

Run the program again, this time enter a first and last name.

- What happens?

Display to `go doc` for the bufio package. This package allows us to read an entire line, spaces and all, through the Reader struct.

Display the `go doc` for bufio.Reader:

```
user@ubuntu:~/go/src/lab-program-construction$ go doc bufio.Reader

type Reader struct {
        // Has unexported fields.
}
    Reader implements buffering for an io.Reader object.


func NewReader(rd io.Reader) *Reader
func NewReaderSize(rd io.Reader, size int) *Reader
func (b *Reader) Buffered() int
func (b *Reader) Discard(n int) (discarded int, err error)
func (b *Reader) Peek(n int) ([]byte, error)
func (b *Reader) Read(p []byte) (n int, err error)
func (b *Reader) ReadByte() (byte, error)
func (b *Reader) ReadBytes(delim byte) ([]byte, error)
func (b *Reader) ReadLine() (line []byte, isPrefix bool, err error)
func (b *Reader) ReadRune() (r rune, size int, err error)
func (b *Reader) ReadSlice(delim byte) (line []byte, err error)
func (b *Reader) ReadString(delim byte) (string, error)
func (b *Reader) Reset(r io.Reader)
func (b *Reader) UnreadByte() error
func (b *Reader) UnreadRune() error
func (b *Reader) WriteTo(w io.Writer) (n int64, err error)
user@ubuntu:~/go/src/lab-program-construction$
```

The bufio package offers a NewReader function which creates a Reader we can then use to call ReadString. The NewReader function requires an io.Reader object. To read from STDIN we can pass the standard os.Stdin object. Display the `go doc` for the os package.

As you can see the Stdin object is an exported variable.

```
user@ubuntu:~/go/src/lab-program-construction$ go doc os.Stdin

var (
        Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
        Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
        Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
    Stdin, Stdout, and Stderr are open Files pointing to the standard input,
    standard output, and standard error file descriptors.

    Note that the Go runtime writes to standard error for panics and crashes;
    closing Stderr may cause those messages to go elsewhere, perhaps to a file
    opened later.

user@ubuntu:~/go/src/lab-program-construction$
```

Update your sample program to import the `fmt, os, and bufio` packages, then add code to read a full line from Stdin.

```
user@ubuntu:~/go/src/lab-program-construction$ vim program-constructione.go
user@ubuntu:~/go/src/lab-program-construction$ cat program-constructione.go

package main

import (
        "bufio"
        "fmt"
        "os"
)

func main() {
        fmt.Print("Enter your name: ")
        reader := bufio.NewReader(os.Stdin)
        name, _ := reader.ReadString('\n')
        fmt.Println("Hi", name)
}
user@ubuntu:~/go/src/lab-program-construction$
```

In this simple program we prompt the user, create a buffered stdin reader and then read a string, stopping at newline. The ReadString() function returns the string read and an error. If you are not interested in one of the return values of a function you can discard it using the "blank identifier", "_". We capture the string entered, ignore any errors, and display the input string back.

Now run the program and enter a first and last name:

```
user@ubuntu:~/go/src/lab-program-construction$ go run program-constructione.go

Enter your name: Bob Smith
Hi Bob Smith

user@ubuntu:~/go/src/lab-program-construction$
```

4. Challenge

Using the `go doc` for os, write a program that displays all of the environment variables. Now, move the code that displays the environment variables into a function in a separate package and use this package from an independent main program to display the environment variables.

Congratulations you have completed the lab!!

Copyright (c) 2013-2017 RX-M LLC, Cloud Native Consulting, all rights reserved