Knowing Is Half the Battle
The Cobra Command Line Library of Go

CHRIS "MAC" MCENIRY

Chris "Mac" McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

In our previous articles, we built a remote directory listing service. If we wanted to, we could extend this to provide generalized remote file system access by adding a `gcp`, `gmv`, `grm`, `gcat` or any other number of mirroring actions that you can do with a local file system and the command line. In this article, we're going to go through a little bit of an exercise to see what that would look like.

The straightforward extension of the `gls` command is to put the core of the actions into libraries and then copy the interface of those libraries into separate mains that become separate binaries. In working with Go, you may have also noticed that the binaries produced tend to be a little on the large side. A simple "hello world" built with Go 1.9 weighs in at around 2.5 MB. There's a lot in that binary, and it is a tradeoff between maintaining a separate runtime and dependencies versus having the runtime and dependencies come with the binary-space versus ease of distributing the application. While the size consideration is not an issue in most cases with current storage availability, there are various use cases where storage is still limited—embedded and RAM-based just to name a couple.

Several tools provide a way for you to have your cake (ease of distribution) and eat it too (limit overall size needs). We can combine all of the commands into a single binary, and then access it one of two ways. In tools like BusyBox, the same binary is invoked, and it looks at what it was called in order to decide what to do. Other tools, like Git, Vault, the AWS command line tools, and even Go itself, use subcommands to decide what to do.

In addition to deciding which action to take, the binary has to handle all of the arguments passed to it. In handling command line arguments, there are two categories: positional and optional. Positional arguments are ones whose meaning comes from where they show up on the command line. One could argue that subcommands are just a special case of the positional argument, and that the command name as subcommand is a special case on top of that. Optional arguments are ones that are identified by name and an argument token (the poster children have the token being a hyphen – or, in POSIX/LongOpt format, a double hyphen --). One of the added benefits of a combined binary is that you can more easily maintain consistency across your command line arguments, especially the ones that are needed across multiple binaries.

gofs

In this article, we're going to focus on building out the command-line interface to gls into a general tool called gofs. To make this a bit more concrete, we're going to use the following rules:

- It will be a single command line tool with subcommands (instead of examining the process name).
- It will have a general option, which will allow us to select different servers.
- It will implement the interface to `ls` and `cp` to show two patterns for handling positional arguments.
- It will combine the server start interface as well under the `serve` subcommand.

Note: For brevity, the focus here is strictly on the command line interface. Most operations of the actual gofs library and service would look the same as our existing gls-based ones, so I'll leave an actual implementation of the service as an exercise for the reader.

The code for this can be found at https://github.com/cmceniry/login/tree/master/gofs.

Go Command Line Options

Most programming decisions are opinions; the Go command line options are no different. There are several ways to achieve this. We're going to examine three options, and then choose one.

Standard Library: `os.Args`

As seen in the preceding articles, the most basic level is parsing the command line options directly. This involves handling the `os.Args` slice.

For the gls services, we just took the very first argument passed into our program and operated on that. There was no decision process, and it was very simple.

This has the upside of being very straightforward to process. The downside is that it is the application's responsibility to handle the various types of arguments. It has to

1. process and remove the optional arguments and then
2. decode subcommands/positional arguments.

Standard Library: flags

Go comes with an opinionated command line options-parsing library. It is strictly for parsing the command line arguments—i.e., it does not handle subcommands. It needs to handle the remaining arguments, directly or via another library, to be able to achieve the subcommand pattern that we're aiming for.

Probably the most controversial opinionated implementation of the flags library is what format it uses for arguments. It only handles the single hyphen token. Additionally, it does not support short options. Though you can name an option using a short name, there's no library method to connect a short and long option.

The Cobra Library

Steve Francia took the time to extract the subcommand pattern out into a library, and separately took the time to extend (more like a drop-in replacement for) the flags library to support the LongOpt format. These libraries are:

- https://github.com/spf13/cobra
- https://github.com/spf13/pflag

Cobra is billed as "a library providing a simple interface to create powerful modern CLI interfaces similar to git & go tools." pflag handles the the option parsing for Cobra built commands.

Given that Cobra handles the subcommand pattern that we want, and pflag implements the familiar LongOpt format, this seems like a good choice on top of which to build.

Implementation

Cobra works by defining Command structs and then wiring them together with flag arguments and with other commands. Both our primary command and all subcommands use the Command struct. It is the wiring that decides whether a command is a primary command or a subcommand.

To organize our code, we're going to implement our application as the `gofs` package. The primary, or root, command will be in its own root file. Each subcommand will also get its own file.

The `main.main` func will be in its own `cmd` directory. This is to reduce any confusion with a `main` package file being in the `gofs` package directory—some tools and IDEs will see this as an error.

Our directory structure looks like:

```
gofs/cmd/main.go # main.main calls into root.go
gofs/ls.go
gofs/mv.go
gofs/put.go
gofs/root.go     # primary command
gofs/serve.go
```

To build this, using the default `GOPATH`:

```
go get -u github.com/cmceniry/login/gofs
cd ~/go/src/github.com/cmceniry/login/gofs
go build -o gofs ./cmd/main.go
```

Root Command

Cobra builds commands off each other. At the start of it is the root command:

`root.go : rootcmd.`

```
var rootCmd = &cobra.Command{
```

Common convention is that each command tends to be a package-level variable. You could define these inside of a setup function (and we'll see something like that with the arguments), but the common method is to do this at the package level.

For each command, we first need to define its usage. We'll see additional usage options shortly, but to start, we need to describe what our application is:

root.go : rootusage.

```
    Long: `A simple interface to a remote file system.
It provides a remote interface similar to the standard tools of
ls, cp, mv, etc.`,
}
```

Cobra provides a built-in help system. `Long` is displayed whenever `help` or `--help` or invalid options are used.

```
$ go run cmd/main.go
A simple interface to a remote file system.
It provides a remote interface similar to the standard tools of
ls, cp, mv, etc.
```

Now that we have a root command, we need to wire it into the `main.main`. Since the root command is in the `gofs` package, we need to export that in some manner. The common way is to define an `Execute` func which `main.main` invokes:

cmd/main.go.

```
package main

import "github.com/cmceniry/login/gofs"

func main() {
    gofs.Execute()
}
```

The `gofs.Execute` func is really just a wrapper around `rootCmd.Execute`:

root.go : execute.

```
func Execute() {
    if err := rootCmd.Execute(); err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}
```

We cannot invoke `rootCmd.Execute` from `main.main` directly because `rootCmd` is a `gofs` package-level variable. While this may present extra hoops to jump through, it does keep the code cleanly separated. It's unlikely that any `main` using this will want to do anything else, but this does make it very clear that this command line library is meant to only operate in an opinionated way.

`--server` Optional

Since all of our subcommands are going to depend on being able to connect to the server, we have to put the configuration for that at the highest level. This will be an optional variable `--server`, which takes the Go network string (e.g., `localhost:4270`). This can be passed to every subcommand so that they are consistently connecting to the correct server.

Options are added to Cobra commands using the `*Flags()` methods. Cobra has `PersistentFlags`, which carry from commands to subcommands, and local `Flags`, which only apply to the subcommands. There are special rules for applying local `Flags` to parent commands and passing those along, but those aren't needed for this exercise.

To add an option, we first have to define a variable for where to keep the flag's value inside of our program. Like the commands, these are commonly found as package-level variables since they may be used in various parts of the package's code. Since it attaches at our `rootCmd`, we can include this in our `root.go` file:

root.go : serveraddress.

```
var (
    serverAddress string
)
```

We could choose to add sections to our `Execute` command to do some initialization, but it would be nice to have this happen a bit automatically. Conveniently, Go has a facility for this.

When Go instantiates a package inside of the runtime, it performs a few initializations. The first that we're concerned with is to set up package-level variables (e.g., `rootCmd`, `serverAddress`). Following that, Go invokes the package's `init` func. This order is critical to us because we need to be able to reference the package-level variables inside of our `init` func.

Our `init` func starts like this—we'll add more to it in a bit:

root.go : flag.

```
func init() {
    rootCmd.PersistentFlags().StringVarP(
        &serverAddress,
        "server",
        "s",
        "localhost:4270",
        "address:port of the server to connect to",
    )
```

Since we're binding only one option, we only need to invoke it once. `PersistentFlags` accesses the flags for the `gofs` root command. `StringVarP` binds a pointer for a string to a variable, so that the variable can be modified. We use the address of our `serverAddress` variable to act as our pointer. The next two arguments, `"server"` and `"s"`, declare the name of the argument. Following that is the default value. The very last part is our usage information, which shows up in `help` much like the `command.Long` field.

The `serve` Subcommand: No Arguments

Now that we've built out our primary command, we can build out and attach a child to it.

The simplest of these is the command that would start up our gofs server: `serve`. We construct it very much like our `rootCmd`, although we're going to add more usage information to it:

server.go : servercmd.

```
    var serveCmd = &cobra.Command{
        Use:   "serve",
        Short: "Start the gofs server side",
        Long: `This will run the gofs server.
    The gofs server provides a remote file system management
interface.`,
```

There are two new fields here. `Use` describes the name of our subcommand. `Short` shows up when help is invoked on our `rootCmd` (as opposed to `Long` which shows up when help is invoked on this command).

Next, we declare that this command should have no position arguments.

server.go : args.

```
        Args: cobra.NoArgs,
```

While it's not clear here, `cobra.NoArgs` is a func, not a data value. You can supply your own argument validation function, or use the built-in ones that come with the Cobra library.

We can now provide the func for our command to run. As mentioned, since the focus of this article is on the command line interface, the command just returns output back to the user.

server.go : run.

```
        Run: func(cmd *cobra.Command, args []string) {
            fmt.Printf("Starting server on '%s'\n", serverAddress)
        },
```

We'll see shortly how the func arguments can be used.

The last part to do is to wire `serveCmd` to our `rootCmd`. Like the `--server` argument, this is done in the `init` func:

root.go : wireserve.

```
        rootCmd.AddCommand(serveCmd)
```

`ls`/`mv`: Handling Arguments

Next let's look at one argument with the `ls` command (one here to mirror the use of `ls` from previous articles). At this point, most of the `ls` command should be inferable:

ls.go : cmd.

```
    var lsCmd = &cobra.Command{
        Use:   "ls [path to ls]",
        Short: "Shows the directory contents of a path",
        Long:  `Shows the directory contents of a path. If given
no path, uses the running path for the gofs server.`,
        Args:  cobra.MaximumNArgs(1),
        Run: func(cmd *cobra.Command, args []string) {
            fmt.Printf("ls server='%s' path='%s'\n", serverAddress,
args[0])
        },
    }
```

The new item here is the `Args` value. `cobra.MaximumNArgs` is a built-in func that is used to indicate a cap to the number of positional arguments. Of special note to remember is that `Args` requires a func, so `cobra.MaximumNArgs` is a func that returns a func.

We can see a similar approach to `Args` with the `mv` command:

mv.go : cmd.

```
    var mvCmd = &cobra.Command{
        Use:   "mv source ... target",
        Short: "Moves a file to a destination file, or moves
multiple files into a directory",
        Long: `If given two arguments, moves the first file to
the second file. If that second file is a directory, the first is
moved into it.
    If more than two arguments are given, it moves all but the
last file into the last one which it expects to be a directory.`,
        Args: cobra.MinimumNArgs(2),
```

Note that `Args` is used to validate the arguments, not to decide how to use them. That happens in the `Run` field:

mv.go : run.

```
        Run: func(cmd *cobra.Command, args []string) {
            if len(args) == 2 {
                fmt.Printf("Moving file '%s' to '%s'", args[0],
args[1])
            } else {
                fmt.Printf(
                    "Moving files '%s' into directory '%s'\n",
                    strings.Join(args[0:len(args)-1], ","),
                    args[len(args)-1],
                )
            }
        }
    }
```

In this case, we want to behave differently depending on the number of arguments. With two arguments, we're looking largely at a rename or move. With three or more arguments, we can only do a move. Since this is done at the same time as our normal execution, it happens all at once, unlike the separate validation step.

Before we finish, we have to wire these commands to our `rootCmd` in the package's `init` func:

root.go : wirelsmv.

```
        rootCmd.AddCommand(lsCmd)
        rootCmd.AddCommand(mvCmd)
```

Conclusion

This article has presented just the tip of the iceberg in handling command line arguments in Go. While there are several options, one of the most powerful ones is the github.com/spf13/cobra library. In addition to the work it performs for you, it also demonstrates some good aspects for the developer experience:

- Cobra allows you to keep all of your related code with itself. Each command is largely self-contained except where there are logical overlaps (PersistentFlags).
- It has a built-in help system that automatically generates help and usage results. The developer doesn't have to spend extra time managing a separate usage function, which means it's less work and less likely to go out of date.

I hope that you feel confident using the Cobra library in your own code. As mentioned, implementing the legwork of this code is left as an exercise for the user. Even if you don't implement the actual function, you can consider how other file system utilities might be implemented to work through some of the odd use cases of command line parameters.

Whatever path you choose, good luck and happy Going!

Save the Date!

usenix

LISA18

October 29–31, 2018
Nashville, TN, USA

LISA: Where systems engineering and operations professionals share real-world knowledge about designing, building, and maintaining the critical systems of our interconnected world.

The full program will be available in August.

Program Co-Chairs

Rikki Endsley
Opensource.com

Brendan Gregg
Netflix

www.usenix.org/lisa18

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION