Simplifying Repetitive Command Line Flags with viper

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 "Knowing Is Half the Battle: The Cobra Command Line Library of Go" [1], we explored using the `github.com/spf13/cobra` library for creating command line tools. In this article, we're going to expand on that by hooking in multiple ways to handle the flags to those commands by using a sister library to `cobra: github.com/spf13/viper`.

How we handle command configuration changes over the lifetime of the tool. A common evolution for handling command configuration is, in order of precedence:

- command line supplied flag
- environment variable
- configuration file

When you first start to use a tool, you will typically supply the flags on the command line. This allows you to explore and iterate with the flags easily.

After you get comfortable with them, you'll want to avoid having to reenter any common values. For example, `--user` or `--server` become very repetitive if you have to enter them every time you run the command. This is the perfect place for environment variables to come into the picture. Set the environment for your shell session, and you can skip setting it on the command line each time.

Eventually, you're comfortable enough with the overall setup to commit those configurations to a file to preserve them over multiple sessions. These typically end up as part of your dotfiles. You set the file and never have to configure your environment or command line again.

Yes, sometimes you skip steps so this pattern is not exclusive, but it is especially common in tool development.

Since the tool configuration is built up this way, all three layers of configuration methods are available throughout. There are two additional benefits that fall out of these configuration methods:

- You can temporarily override the values from the environment or command line. This allows you to test out new configurations without changing your defaults.
- Different runtime environments and setups prefer different formats. For example, your Puppet setup may prefer configuration files, your Dockerfiles setup may prefer environment variables, and your Kubernetes setup may prefer command line arguments. A flexible binary supports multiple environments since it can support all three mechanisms. This last part is especially apt for 12-factor applications.

We're specifically using the `viper` library because it builds upon the work of the `cobra` library from the previous article. This combination follows the precedence order identified above. This only holds for flags (`--flag`) and not for full command arguments. Arguments are typically specific to each command invocation, and it is unusual to encode this in environment variables or configuration files.

The code for these examples can be found at https://github.com/cmceniry/login in the “viper” directory. Each directory corresponds to a section below and should be executed using `go run $DIR/main.go` to follow along with the article. This uses `go` module support (minimum Go version 1.11), so no prep work is required once the repository is cloned.

Default

To establish a baseline, we’re going to set a default for `viper`-maintained values (this will also help to build up the scaffold around the examples). As usual, we begin with the standard Go intro—setting up our main and imports.

default/main.go: intro.

```
package main

import (
    "fmt"
    "github.com/spf13/cobra"
    "github.com/spf13/viper"
)

func main() {
```

We’re going to be building on top of the existing `cobra` command. In our `Run`, we’re going to just print the output of our flag. Specifically, we get the configuration value of the `Flag` item and we will get it as a string (or nothing).

default/main.go: cobra.

```
rootCmd := &cobra.Command{
    Run: func(c *cobra.Command, args []string) {
        fmt.Println(viper.GetString(“Flag”))
    },
}
```

Setting a default in `viper` is a single function `viper.SetDefault`.

default/main.go: viper.

```
viper.SetDefault("Flag", "default")
```

And to round it out, we execute into our `cobra` command.

default/main.go: execute.

```
rootCmd.Execute()
```

With all of that together, we can run our tool and get our internally set value for `Flag`.

```
$ go run default/main.go
default
```

Command Line

Now let’s add the first pattern by pulling the value in from the command line flag. The code here will be identical to the `default` case, but we’re going to add a couple of lines just before the `Execute`. These set up the command line flag (which comes from the `cobra` command as in the *;login:* article [1]) and then bind it to the `viper` configuration.

commandline/main.go: flag.

```
rootCmd.Flags().String("flag", "", "help for flag")
viper.BindPFlag("Flag", rootCmd.Flags().Lookup("flag"))
```

We can demonstrate that by just adding these lines, we maintain our default compatibility, but we also add support for our command line flag.

```
$ go run commandline/main.go
default
$ go run commandline/main.go  --flag cli
cli
```

Environment Variable

The next step in our flag handling evolution is to set this using an environment variable. As previously, this is done with the addition of a few more items before our `cobra` execute. The first function creates a pseudo-environment namespace so that we don’t accidentally conflict with other applications. The second function connects the environment variables with the `viper` configuration. Make special note that `viper` connects them with the convention of all uppercase with prefix, so in this case, `VF_FLAG`.

```
viper.SetEnvPrefix("VF")
viper.BindEnv("Flag")
```

With these in place, we can now use the default, environment, or command line.

```
$ go run envvar/main.go
default
$ VF_FLAG=env go run envvar/main.go
env
$ VF_FLAG=env go run envvar/main.go  --flag cli
cli
```

Configuration File

`viper` supports a variety of configuration file formats and even has autodetection for them. For simplicity, we’re going to go with the TOML format:

```
Flag = "configfile"
```

As before, we’re building on top of the previous examples by adding a few lines before executing our `cobra` command. First, we tell `viper` where to look for the configuration file. Next, we tell it which configuration file to use (notice that the suffix is ignored since we’re using autodetection). And, finally, we read the config file. This is the first call that can produce an error. To support compatibility with the other three examples, we ignore it if the file is not found and panic otherwise.

```
configfile/main.go: configfile.
        viper.AddConfigPath(".")
        viper.SetConfigName("config")
        if err := viper.ReadInConfig(); err != nil {
            // Only error on errors other than file not found
            if _, ok := err.(viper.ConfigFileNotFoundError);
!ok {
                panic(err)
            }
        }
```

Now we run it again and get our expected output. As before, we can test it with the environment and command line flag options and also still receive the expected outputs. However, short of removing the config file, we will not be able to see the default value (but you can remove the file and try as you want).

```
$ go run configfile/main.go
configfile
$ VF_FLAG=env go run configfile/main.go
env
$ VF_FLAG=env go run configfile/main.go  --flag cli
cli
```

Combining Multiple Configurations

In this, we used viper as a monolithic config. There are times when you want to break this out, and that means creating a viper.Viper struct (using New) instead of the default struct invoked by the package static funcs as we've done here. This allows you to even use it in libraries to combine configuration functionality without having to support multiple formats. To avoid conflicts, you'll want to apply judicious use of the SetEnvPrefix and SetConfigPath or SetConfigName functions for each configuration.

Conclusion

With just a few lines of setup, the viper library has given us fast configuration handling. This supports the regular model of command line flags, environment variables, and configuration files.

I hope this article has provided you with a concrete handle to the viper library and that this helps you in your tool development. Happy Going!

Reference
[1] *;login:* vol. 43, no. 2: https://www.usenix.org/system/files/login/articles/login_summer18_09_mceniry.pdf.