Go 每日一库之 cobra

dj GoUpUp 2020-01-21

简介

cobra是一个命令行程序库，可以用来编写命令行程序。同时，它也提供了一个脚手架， 用于生成基于 cobra 的应用程序框架。非常多知名的开源项目使用了 cobra 库构建命令行，如Kubernetes、Hugo、etcd等等等等。本文介绍 cobra 库的基本使用和一些有趣的特性。

关于作者spf13，这里多说两句。spf13 开源不少项目，而且他的开源项目质量都比较高。相信使用过 vim 的都知道spf13-vim，号称 vim 终极配置。可以一键配置，对于我这样的懒人来说绝对是福音。他的viper是一个完整的配置解决方案。完美支持 JSON/TOML/YAML/HCL/envfile/Java properties 配置文件等格式，还有一些比较实用的特性，如配置热更新、多查找目录、配置保存等。还有非常火的静态网站生成器hugo也是他的作品。

快速使用

第三方库都需要先安装，后使用。下面命令安装了 `cobra` 生成器程序和 cobra 库：

```
$ go get github.com/spf13/cobra/cobra
```

如果出现了 `golang.org/x/text` 库找不到之类的错误，需要手动从 GitHub 上下载该库，再执行上面的安装命令。我以前写过一篇博客搭建 Go 开发环境提到了这个方法。

我们实现一个简单的命令行程序 git，当然这不是真的 git，只是模拟其命令行。最终还是通过 `os/exec` 库调用外部程序执行真实的 git 命令，返回结果。所以我们的系统上要安装 git，且 git 在可执行路径中。目前我们只添加一个子命令 `version` 。目录结构如下：

```
▾ get-started/
    ▾ cmd/
        helper.go
```

```
        root.go
        version.go
    main.go
```

`root.go` :

```
package cmd

import (
  "errors"

  "github.com/spf13/cobra"
)

var rootCmd = &cobra.Command {
  Use: "git",
  Short: "Git is a distributed version control system.",
  Long: `Git is a free and open source distributed version control system
designed to handle everything from small to very large projects
with speed and efficiency.`,
  Run: func(cmd *cobra.Command, args []string) {
    Error(cmd, args, errors.New("unrecognized command"))
  },
}

func Execute() {
  rootCmd.Execute()
}
```

`version.go` :

```go
package cmd

import (
  "fmt"
  "os"

  "github.com/spf13/cobra"
)

var versionCmd = &cobra.Command {
  Use: "version",
  Short: "version subcommand show git version info.",

  Run: func(cmd *cobra.Command, args []string) {
    output, err := ExecuteCommand("git", "version", args...)
    if err != nil {
      Error(cmd, args, err)
    }

    fmt.Fprint(os.Stdout, output)
  },
}

func init() {
  rootCmd.AddCommand(versionCmd)
}
```

`main.go` 文件中只是调用命令入口：

```go
package main

import (
  "github.com/darjun/go-daily-lib/cobra/get-started/cmd"
)

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

为了编码方便，在`helpers.go`中封装了调用外部程序和错误处理函数：

```go
package cmd

import (
  "fmt"
  "os"
  "os/exec"

  "github.com/spf13/cobra"
)


func ExecuteCommand(name string, subname string, args ...string) (string, error) {
  args = append([]string{subname}, args...)

  cmd := exec.Command(name, args...)
  bytes, err := cmd.CombinedOutput()

  return string(bytes), err
```

```
}

func Error(cmd *cobra.Command, args []string, err error) {
  fmt.Fprintf(os.Stderr, "execute %s args:%v error:%v\n", cmd.Name(), args, err)
  os.Exit(1)
}
```

每个 cobra 程序都有一个根命令，可以给它添加任意多个子命令。我们在`version.go`的`init`函数中将子命令添加到根命令中。

编译程序。注意，不能直接`go run main.go`，这已经不是单文件程序了。如果强行要用，请使用`go run .`：

```
$ go build -o main.exe
```

cobra 自动生成的帮助信息，very cool：

```
$ ./main.exe -h
Git is a free and open source distributed version control system
designed to handle everything from small to very large projects
with speed and efficiency.

Usage:
  git [flags]
  git [command]

Available Commands:
  help        Help about any command
  version     version subcommand show git version info.

Flags:
  -h, --help   help for git
```

```
Use "git [command] --help" for more information about a command.
```

单个子命令的帮助信息：

```
$ ./main.exe version -h
version subcommand show git version info.

Usage:
  git version [flags]

Flags:
  -h, --help   help for version
```

调用子命令：

```
$ ./main.exe version
git version 2.19.1.windows.1
```

未识别的子命令：

```
$ ./main.exe clone
Error: unknown command "clone" for "git"
Run 'git --help' for usage.
```

编译时可以将`main.exe`改成`git`，用起来会更有感觉😀。

```
$ go build -o git
$ ./git version
```

```
git version 2.19.1.windows.1
```

使用 cobra 构建命令行时，程序的目录结构一般比较简单，推荐使用下面这种结构：

```
▾ appName/
    ▾ cmd/
        cmd1.go
        cmd2.go
        cmd3.go
        root.go
    main.go
```

每个命令实现一个文件，所有命令文件存放在`cmd`目录下。外层的`main.go`仅初始化 cobra。

特性

cobra 提供非常丰富的功能：

- 轻松支持子命令，如`app server`，`app fetch`等；
- 完全兼容 POSIX 选项（包括短、长选项）；
- 嵌套子命令；
- 全局、本地层级选项。可以在多处设置选项，按照一定的顺序取用；
- 使用脚手架轻松生成程序框架和命令。

首先需要明确 3 个基本概念：

- 命令（Command）：就是需要执行的操作；
- 参数（Arg）：命令的参数，即要操作的对象；
- 选项（Flag）：命令选项可以调整命令的行为。

下面示例中，`server`是一个（子）命令，`--port`是选项：

```
hugo server --port=1313
```

下面示例中，`clone`是一个（子）命令，`URL`是参数，`--bare`是选项：

```
git clone URL --bare
```

命令

在 cobra 中，命令和子命令都是用`Command`结构表示的。`Command`有非常多的字段，用来定制命令的行为。在实际中，最常用的就那么几个。我们在前面示例中已经看到了`Use/Short/Long/Run`。

`Use`指定使用信息，即命令怎么被调用，格式为`name arg1 [arg2]`。`name`为命令名，后面的`arg1`为必填参数，`arg3`为可选参数，参数可以多个。

`Short/Long`都是指定命令的帮助信息，只是前者简短，后者详尽而已。

`Run`是实际执行操作的函数。

定义新的子命令很简单，就是创建一个`cobra.Command`变量，设置一些字段，然后添加到根命令中。例如我们要添加一个`clone`子命令：

```
package cmd

import (
  "fmt"
  "os"

  "github.com/spf13/cobra"
)

var cloneCmd = &cobra.Command {
```

```
  Use: "clone url [destination]",
  Short: "Clone a repository into a new directory",
  Run: func(cmd *cobra.Command, args []string) {
    output, err := ExecuteCommand("git", "clone", args...)
    if err != nil {
      Error(cmd, args, err)
    }

    fmt.Fprintf(os.Stdout, output)
  },
}

func init() {
  rootCmd.AddCommand(cloneCmd)
}
```

其中`Use`字段`clone url [destination]`表示子命令名为`clone`，参数`url`是必须的，目标路径`destination`可选。

我们将程序编译为`mygit`可执行文件，然后将它放到`$GOPATH/bin`中。我喜欢将`$GOPATH/bin`放到`$PATH`中，所以可以直接调用`mygit`命令了：

```
$ go build -o mygit
$ mv mygit $GOPATH/bin
$ mygit clone https://github.com/darjun/leetcode
Cloning into 'leetcode'...
```

大家可以继续添加命令。但是我这边只是偷了个懒，将操作都转发到实际的 git 去执行了。这确实没什么实际的用处。有这个思路，试想一下，我们可以结合多个命令实现很多有用的工具，例如打包工具☺。

选项

cobra 中选项分为两种，一种是**永久选项**，定义它的命令和其子命令都可以使用。通过给根命令添加一个选项定义全局选项。另一种是**本地选项**，只能在定义它的命令中使用。

cobra 使用pflag解析命令行选项。`pflag`使用上基本与`flag`相同，该系列文章有一篇介绍`flag`库的，Go 每日一库之 flag。

与`flag`一样，存储选项的变量也需要提前定义好：

```
var Verbose bool
var Source string
```

设置永久选项：

```
rootCmd.PersistentFlags().BoolVarP(&Verbose, "verbose", "v", false, "verbose output")
```

设置本地选项：

```
localCmd.Flags().StringVarP(&Source, "source", "s", "", "Source directory to read from")
```

两种参数都是相同的，长选项/短选项名、默认值和帮助信息。

下面，我们通过一个案例来演示选项的使用。

假设我们要做一个简单的计算器，支持加、减、乘、除操作。并且可以通过选项设置是否忽略非数字参数，设置除 0 是否报错。显然，前一个选项应该放在全局选项中，后一个应该放在除法命令中。程序结构如下：

```
▾ math/
    ▾ cmd/
        add.go
        divide.go
        minus.go
```

```
        multiply.go
        root.go
    main.go
```

这里展示`divide.go`和`root.go`，其它命令文件都类似。完整代码我放在GitHub上了。

`divide.go`：

```
var (
  dividedByZeroHanding int // 除 0 如何处理
)
var divideCmd = &cobra.Command {
  Use: "divide",
  Short: "Divide subcommand divide all passed args.",
  Run: func(cmd *cobra.Command, args []string) {
    values := ConvertArgsToFloat64Slice(args, ErrorHandling(parseHandling))
    result := calc(values, DIVIDE)
    fmt.Printf("%s = %.2f\n", strings.Join(args, "/"), result)
  },
}

func init() {
  divideCmd.Flags().IntVarP(&dividedByZeroHanding, "divide_by_zero", "d", int(PanicOnDividedByZero), "do what wh

  rootCmd.AddCommand(divideCmd)
}
```

`root.go`：

```
var (
  parseHandling int
)

var rootCmd = &cobra.Command {
  Use: "math",
  Short: "Math calc the accumulative result.",
  Run: func(cmd *cobra.Command, args []string) {
    Error(cmd, args, errors.New("unrecognized subcommand"))
  },
}

func init() {
  rootCmd.PersistentFlags().IntVarP(&parseHandling, "parse_error", "p", int(ContinueOnParseError), "do what when
}

func Execute() {
  rootCmd.Execute()
}
```

在`divide.go`中定义了如何处理除 0 错误的选项，在`root.go`中定义了如何处理解析错误的选项。选项枚举如下：

```
const (
  ContinueOnParseError  ErrorHandling = 1 // 解析错误尝试继续处理
  ExitOnParseError      ErrorHandling = 2 // 解析错误程序停止
  PanicOnParseError     ErrorHandling = 3 // 解析错误 panic
  ReturnOnDividedByZero ErrorHandling = 4 // 除0返回
  PanicOnDividedByZero  ErrorHandling = 5 // 除0 painc
)
```

其实命令的执行逻辑并不复杂，就是将参数转为`float64`。然后执行相应的运算，输出结果。

测试程序：

```
$ go build -o math
$ ./math add 1 2 3 4
1+2+3+4 = 10.00

$ ./math minus 1 2 3 4
1-2-3-4 = -8.00

$ ./math multiply 1 2 3 4
1*2*3*4 = 24.00

$ ./math divide 1 2 3 4
1/2/3/4 = 0.04
```

默认情况，解析错误被忽略，只计算格式正确的参数的结果：

```
$ ./math add 1 2a 3b 4
1+2a+3b+4 = 5.00

$ ./math divide 1 2a 3b 4
1/2a/3b/4 = 0.25
```

设置解析失败的处理，2 表示退出程序，3 表示 panic（看上面的枚举）：

```
$ ./math add 1 2a 3b 4 -p 2
invalid number: 2a
```

```
$ ./math add 1 2a 3b 4 -p 3
panic: strconv.ParseFloat: parsing "2a": invalid syntax

goroutine 1 [running]:
github.com/darjun/go-daily-lib/cobra/math/cmd.ConvertArgsToFloat64Slice(0xc00004e300, 0x4, 0x6, 0x3, 0xc00008bd7
    D:/code/golang/src/github.com/darjun/go-daily-lib/cobra/math/cmd/helper.go:58 +0x2c3
github.com/darjun/go-daily-lib/cobra/math/cmd.glob..func1(0x74c620, 0xc00004e300, 0x4, 0x6)
    D:/code/golang/src/github.com/darjun/go-daily-lib/cobra/math/cmd/add.go:14 +0x6d
github.com/spf13/cobra.(*Command).execute(0x74c620, 0xc00004e1e0, 0x6, 0x6, 0x74c620, 0xc00004e1e0)
    D:/code/golang/src/github.com/spf13/cobra/command.go:835 +0x2b1
github.com/spf13/cobra.(*Command).ExecuteC(0x74d020, 0x0, 0x599ee0, 0xc000056058)
    D:/code/golang/src/github.com/spf13/cobra/command.go:919 +0x302
github.com/spf13/cobra.(*Command).Execute(...)
    D:/code/golang/src/github.com/spf13/cobra/command.go:869
github.com/darjun/go-daily-lib/cobra/math/cmd.Execute(...)
    D:/code/golang/src/github.com/darjun/go-daily-lib/cobra/math/cmd/root.go:45
main.main()
    D:/code/golang/src/github.com/darjun/go-daily-lib/cobra/math/main.go:8 +0x35
```

至于除 0 选项大家自己试试。

细心的朋友应该都注意到了，该程序还有一些不完善的地方。例如这里如果输入非数字参数，该参数也会显示在结果中：

```
$ ./math add 1 2 3d cc
1+2+3d+cc = 3.00
```

感兴趣可以自己完善一下~

脚手架

通过前面的介绍，我们也看到了其实 cobra 命令的框架还是比较固定的。这就有了工具的用武之地了，可极大地提高我们的开发效率。

前面安装 cobra 库的时候也将脚手架程序安装好了。下面我们介绍如何使用这个生成器。

使用`cobra init`命令创建一个 cobra 应用程序：

```
$ cobra init scaffold --pkg-name github.com/darjun/go-daily-lib/cobra/scaffold
```

其中`scaffold`为应用程序名，后面通过`pkg-name`选项指定包路径。生成的程序目录结构如下：

```
▾ scaffold/
    ▾ cmd/
        root.go
    LICENSE
    main.go
```

这个项目结构与之前介绍的完全相同，也是 cobra 推荐使用的结构。同样地，`main.go`也仅仅是入口。

在`root.go`中，工具额外帮我们生成了一些代码。

在根命令中添加了配置文件选项，大部分应用程序都需要配置文件：

```
func init() {
  cobra.OnInitialize(initConfig)

  rootCmd.PersistentFlags().StringVar(&cfgFile, "config", "", "config file (default is $HOME/.scaffold.yaml)")
  rootCmd.Flags().BoolP("toggle", "t", false, "Help message for toggle")
}
```

在初始化完成的回调中，如果发现该选项为空，则默认使用主目录下的`.scaffold.yaml`文件：

```
func initConfig() {
  if cfgFile != "" {
    viper.SetConfigFile(cfgFile)
  } else {
    home, err := homedir.Dir()
    if err != nil {
      fmt.Println(err)
      os.Exit(1)
    }

    viper.AddConfigPath(home)
    viper.SetConfigName(".scaffold")
  }

  viper.AutomaticEnv()

  if err := viper.ReadInConfig(); err == nil {
    fmt.Println("Using config file:", viper.ConfigFileUsed())
  }
}
```

这里用到了我前几天介绍的go-homedir库。配置文件的读取使用了 spf13 自己的开源项目viper（毒龙？真是起名天才）。

除了代码文件，cobra 还生成了一个 LICENSE 文件。

现在这个程序还不能做任何事情，我们需要给它添加子命令，使用`cobra add`命令：

```
$ cobra add date
```

该命令在`cmd`目录下新增了`date.go`文件。基本结构已经搭好了，剩下的就是修改一些描述，添加一些选项了。

我们现在实现这样一个功能，根据传入的年、月，打印这个月的日历。如果没有传入选项，使用当前的年、月。

选项定义：

```
func init() {
  rootCmd.AddCommand(dateCmd)

  dateCmd.PersistentFlags().IntVarP(&year, "year", "y", 0, "year to show (should in [1000, 9999]")
  dateCmd.PersistentFlags().IntVarP(&month, "month", "m", 0, "month to show (should in [1, 12]")
}
```

修改`dateCmd`的`Run`函数：

```
Run: func(cmd *cobra.Command, args []string) {
  if year < 1000 && year > 9999 {
    fmt.Fprintln(os.Stderr, "invalid year should in [1000, 9999], actual:%d", year)
    os.Exit(1)
  }

  if month < 1 && year > 12 {
    fmt.Fprintln(os.Stderr, "invalid month should in [1, 12], actual:%d", month)
    os.Exit(1)
  }

  showCalendar()
}
```

`showCalendar` 函数就是利用 `time` 提供的方法实现的，这里就不赘述了。感兴趣可以去我的 GitHub 上查看实现。

看看程序运行效果：

```
$ go build -o main.exe
$ ./main.exe date
  Sun  Mon  Tue  Wed  Thu  Fri  Sat
                    1    2    3    4
    5    6    7    8    9   10   11
   12   13   14   15   16   17   18
   19   20   21   22   23   24   25
   26   27   28   29   30   31

$ ./main.exe date --year 2019 --month 12
  Sun  Mon  Tue  Wed  Thu  Fri  Sat
    1    2    3    4    5    6    7
    8    9   10   11   12   13   14
   15   16   17   18   19   20   21
   22   23   24   25   26   27   28
   29   30   31
```

可以再为这个程序添加其他功能，试一试吧~

其他

cobra 提供了非常丰富的特性和定制化接口，例如：

- 设置钩子函数，在命令执行前、后执行某些操作；
- 生成 Markdown/ReStructed Text/Man Page 格式的文档；

- 等等等等。

由于篇幅限制，就不一一介绍了。有兴趣可自行研究。cobra 库的使用非常广泛，很多知名项目都有用到，前面也提到过这些项目。学习这些项目是如何使用 cobra 的，可以从中学习 cobra 的特性和最佳实践。这也是学习开源项目的一个很好的途径。

文中所有示例代码都已上传至我的 GitHub，Go 每日一库，https://github.com/darjun/go-daily-lib/tree/master/cobra。

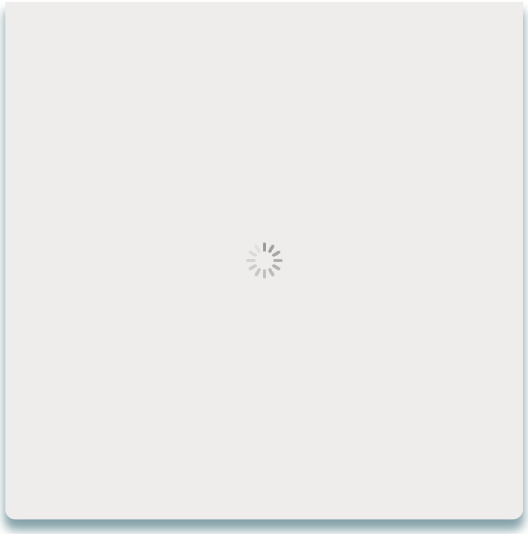
参考

1. cobra GitHub 仓库

我

我的博客

欢迎关注我的微信公众号【GoUpUp】，共同学习，一起进步~



People who liked this content also liked

Go 每日一库之 reflect

GoUpUp

又传噩耗！知名歌手凌晨跳楼自杀：他为何走上了绝路？

拾读

三胎政策引发热议！那些真生了仨的妈妈，有话要说

年糕妈妈