Go 每日一库之 go-flags

Original dj GoUpUp 2020-01-18

收录于话题
#Go 每日一库

48个 >

简介

在上一篇文章中，我们介绍了`flag`库。`flag`库是用于解析命令行选项的。但是`flag`有几个缺点：

- 不显示支持短选项。当然上一篇文章中也提到过可以通过将两个选项共享同一个变量迂回实现，但写起来比较繁琐；
- 选项变量的定义比较繁琐，每个选项都需要根据类型调用对应的`Type`或`TypeVar`函数；
- 默认只支持有限的数据类型，当前只有基本类型`bool/int/uint/string`和`time.Duration`；

为了解决这些问题，出现了不少第三方解析命令行选项的库，今天的主角`go-flags`就是其中一个。第一次看到`go-flags`库是在阅读pgweb源码的时候。

`go-flags`提供了比标准库`flag`更多的选项。它利用结构标签（struct tag）和反射提供了一个方便、简洁的接口。它除了基本的功能，还提供了丰富的特性：

- 支持短选项（-v）和长选项（--verbose）；
- 支持短选项合写，如`-aux`；
- 同一个选项可以设置多个值；
- 支持所有的基础类型和 map 类型，甚至是函数；
- 支持命名空间和选项组；
- 等等。

上面只是粗略介绍了`go-flags`的特性，下面我们依次来介绍。

快速开始

学习从使用开始！我们先来看看`go-flags`的基本使用。

由于是第三方库，使用前需要安装，执行下面的命令安装：

```
$ go get github.com/jessevdk/go-flags
```

代码中使用`import`导入该库：

```
import "github.com/jessevdk/go-flags"
```

完整示例代码如下：

```
package main

import (
  "fmt"

  "github.com/jessevdk/go-flags"
)

type Option struct {
  Verbose []bool `short:"v" long:"verbose" description:"Show verbose debug message"`
}

func main() {
  var opt Option
  flags.Parse(&opt)
```

```
  fmt.Println(opt.Verbose)
}
```

使用`go-flags`的一般步骤：

- 定义选项结构，在结构标签中设置选项信息。通过`short`和`long`设置短、长选项名字，`description`设置帮助信息。命令行传参时，短选项前加`-`，长选项前加`--`；
- 声明选项变量；
- 调用`go-flags`的解析方法解析。

编译、运行代码（我的环境是 Win10 + Git Bash）：

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

短选项：

```
$ ./main.exe -v
[true]
```

长选项：

```
$ ./main.exe --verbose
[true]
```

由于`Verbose`字段是切片类型，每次遇到`-v`或`--verbose`都会追加一个`true`到切片中。

多个短选项：

```
$ ./main.exe -v -v
[true true]
```

多个长选项：

```
$ ./main.exe --verbose --verbose
[true true]
```

短选项 + 长选项：

```
$ ./main.exe -v --verbose -v
[true true true]
```

短选项合写：

```
$ ./main.exe -vvv
[true true true]
```

基本特性

支持丰富的数据类型

`go-flags`相比标准库`flag`支持更丰富的数据类型：

- 所有的基本类型（包括有符号整数`int/int8/int16/int32/int64`，无符号整数`uint/uint8/uint16/uint32/uint64`，浮点数`float32/float64`，布尔类型`bool`和字符串`string`）和**它们的切片**；

- map 类型。只支持键为 `string`，值为基础类型的 map；
- 函数类型。

如果字段是基本类型的切片，基本解析流程与对应的基本类型是一样的。切片类型选项的不同之处在于，遇到相同的选项时，值会被追加到切片中。而非切片类型的选项，后出现的值会覆盖先出现的值。

下面来看一个示例：

```
package main

import (
  "fmt"

  "github.com/jessevdk/go-flags"
)

type Option struct {
  IntFlag         int            `short:"i" long:"int" description:"int flag value"`
  IntSlice        []int          `long:"intslice" description:"int slice flag value"`
  BoolFlag        bool           `long:"bool" description:"bool flag value"`
  BoolSlice       []bool         `long:"boolslice" description:"bool slice flag value"`
  FloatFlag       float64        `long:"float", description:"float64 flag value"`
  FloatSlice      []float64      `long:"floatslice" description:"float64 slice flag value"`
  StringFlag      string         `short:"s" long:"string" description:"string flag value"`
  StringSlice     []string       `long:"strslice" description:"string slice flag value"`
  PtrStringSlice  []*string      `long:"pstrslice" description:"slice of pointer of string flag value"`
  Call            func(string)   `long:"call" description:"callback"`
  IntMap          map[string]int `long:"intmap" description:"A map from string to int"`
}

func main() {
  var opt Option
```

```
  var opt option
  opt.Call = func (value string) {

    fmt.Println("in callback: ", value)
  }

  err := flags.Parse(&opt, os.Args[1:])
  if err != nil {
    fmt.Println("Parse error:", err)
    return
  }

  fmt.Printf("int flag: %v\n", opt.IntFlag)
  fmt.Printf("int slice flag: %v\n", opt.IntSlice)
  fmt.Printf("bool flag: %v\n", opt.BoolFlag)
  fmt.Printf("bool slice flag: %v\n", opt.BoolSlice)
  fmt.Printf("float flag: %v\n", opt.FloatFlag)
  fmt.Printf("float slice flag: %v\n", opt.FloatSlice)
  fmt.Printf("string flag: %v\n", opt.StringFlag)
  fmt.Printf("string slice flag: %v\n", opt.StringSlice)
  fmt.Println("slice of pointer of string flag: ")
  for i := 0; i < len(opt.PtrStringSlice); i++ {
    fmt.Printf("\t%d: %v\n", i, *opt.PtrStringSlice[i])
  }
  fmt.Printf("int map: %v\n", opt.IntMap)
}
```

基本类型和其切片比较简单，就不过多介绍了。值得留意的是基本类型指针的切片，即上面的`PtrStringSlice`字段，类型为`[]*string`。由于结构中存储的是字符串指针，`go-flags`在解析过程中遇到该选项会自动创建字符串，将指针追加到切片中。

运行程序，传入`--pstrslice`选项：

```
$ ./main.exe --pstrslice test1 --pstrslice test2
slice of pointer of string flag:
    0: test1
    1: test2
```

另外，我们可以在选项中定义函数类型。该函数的唯一要求是有一个字符串类型的参数。解析中每次遇到该选项就会以选项值为参数调用这个函数。上面代码中，`Call`函数只是简单的打印传入的选项值。运行代码，传入`--call`选项：

```
$ ./main.exe --call test1 --call test2
in callback:  test1
in callback:  test2
```

最后，`go-flags`还支持 map 类型。虽然限制键必须是`string`类型，值必须是基本类型，也能实现比较灵活的配置。`map`类型的选项值中键-值通过`:`分隔，如`key:value`，可设置多个。运行代码，传入`--intmap`选项：

```
$ ./main.exe --intmap key1:12 --intmap key2:58
int map: map[key1:12 key2:58]
```

常用设置

`go-flags`提供了非常多的设置选项，具体可参见文档。这里重点介绍两个`required`和`default`。

`required`非空时，表示对应的选项必须设置值，否则解析时返回`ErrRequired`错误。

`default`用于设置选项的默认值。如果已经设置了默认值，那么`required`是否设置并不影响，也就是说命令行参数中该选项可以没有。

看下面示例：

```
package main

import (
  "fmt"
  "log"

  "github.com/jessevdk/go-flags"
)

type Option struct {
  Required    string  `short:"r" long:"required" required:"true"`
  Default     string  `short:"d" long:"default" default:"default"`
}

func main() {
  var opt Option
  _, err := flags.Parse(&opt)
  if err != nil {
    log.Fatal("Parse error:", err)
  }

  fmt.Println("required: ", opt.Required)
  fmt.Println("default: ", opt.Default)
}
```

运行程序，不传入`default`选项，`Default`字段取默认值，不传入`required`选项，执行报错：

```
$ ./main.exe -r required-data
required:  required-data
default:  default
```

```
$ ./main.exe -d default-data -r required-data
required:  required-data
default:  default-data

$ ./main.exe
the required flag `/r, /required' was not specified
2020/01/09 18:07:39 Parse error:the required flag `/r, /required' was not specified
```

高级特性

选项分组

```
package main

import (
  "fmt"
  "log"
  "os"

  "github.com/jessevdk/go-flags"
)

type Option struct {
  Basic GroupBasicOption `description:"basic type" group:"basic"`
  Slice GroupSliceOption `description:"slice of basic type" group:"slice"`
}

type GroupBasicOption struct {
```

```
  IntFlag    int     `short:"i" long:"intflag" description:"int flag"`
  BoolFlag   bool    `short:"b" long:"boolflag" description:"bool flag"`
  FloatFlag  float64 `short:"f" long:"floatflag" description:"float flag"`
  StringFlag string  `short:"s" long:"stringflag" description:"string flag"`
}

type GroupSliceOption struct {
  IntSlice        int            `long:"intslice" description:"int slice"`
  BoolSlice        bool        `long:"boolslice" description:"bool slice"`
  FloatSlice    float64    `long:"floatslice" description:"float slice"`
  StringSlice    string    `long:"stringslice" description:"string slice"`
}

func main() {
  var opt Option
  p := flags.NewParser(&opt, flags.Default)
  _, err := p.ParseArgs(os.Args[1:])
  if err != nil {
    log.Fatal("Parse error:", err)
  }

  basicGroup := p.Command.Group.Find("basic")
  for _, option := range basicGroup.Options() {
    fmt.Printf("name:%s value:%v\n", option.LongNameWithNamespace(), option.Value())
  }


  sliceGroup := p.Command.Group.Find("slice")
  for _, option := range sliceGroup.Options() {
    fmt.Printf("name:%s value:%v\n", option.LongNameWithNamespace(), option.Value())
  }
```

```
}
```

上面代码中我们将基本类型和它们的切片类型选项拆分到两个结构体中，这样可以使代码看起来更清晰自然，特别是在代码量很大的情况下。这样做还有一个好处，我们试试用`--help`运行该程序：

```
$ ./main.exe --help
Usage:
  D:\code\golang\src\github.com\darjun\go-daily-lib\go-flags\group\main.exe [OPTIONS]

basic:
  /i, /intflag:      int flag
  /b, /boolflag      bool flag
  /f, /floatflag:    float flag
  /s, /stringflag:   string flag

slice:
  /intslice:     int slice
  /boolslice     bool slice
  /floatslice:   float slice
  /stringslice:  string slice

Help Options:
  /?                 Show this help message
  /h, /help          Show this help message
```

输出的帮助信息中，也是按照我们设定的分组显示了，便于查看。

子命令

`go-flags`支持子命令。我们经常使用的 Go 和 Git 命令行程序就有大量的子命令。例如`go version`、`go build`、`go run`、`git status`、`git commit`这些命令中`version/build/run/status/commit`就是子命令。使用`go-flags`定义子命令比较简单：

```
package main

import (
  "errors"
  "fmt"
  "log"
  "strconv"
  "strings"

  "github.com/jessevdk/go-flags"
)

type MathCommand struct {
  Op string `long:"op" description:"operation to execute"`
  Args []string
  Result int64
}

func (this *MathCommand) Execute(args []string) error {
  if this.Op != "+" && this.Op != "-" && this.Op != "x" && this.Op != "/" {
    return errors.New("invalid op")
  }

  for _, arg := range args {
    num, err := strconv.ParseInt(arg, 10, 64)
    if err != nil {
      return err
```

```
    }

    this.Result += num
  }

  this.Args = args

  return nil
}

type Option struct {
    Math MathCommand `command:"math"`
}

func main() {
    var opt Option
    _, err := flags.Parse(&opt)

    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("The result of %s is %d", strings.Join(opt.Math.Args, opt.Math.Op), opt.Math.Result)
}
```

子命令必须实现`go-flags`定义的`Commander`接口：

```
type Commander interface {
    Execute(args []string) error
}
```

解析命令行时，如果遇到不是以`-`或`--`开头的参数，`go-flags`会尝试将其解释为子命令名。子命令的名字通过在结构标签中使用`command`指定。子命令后面的参数都将作为子命令的参数，子命令也可以有选项。

上面代码中，我们实现了一个可以计算任意个整数的加、减、乘、除子命令`math`。

接下来看看如何使用：

```
$ ./main.exe math --op + 1 2 3 4 5
The result of 1+2+3+4+5 is 15

$ ./main.exe math --op - 1 2 3 4 5
The result of 1-2-3-4-5 is -13

$ ./main.exe math --op x 1 2 3 4 5
The result of 1x2x3x4x5 is 120

$ ./main.exe math --op ÷ 120 2 3 4 5
The result of 120÷2÷3÷4÷5 is 1
```

注意，不能使用乘法符号`*`和除法符号`/`，它们都不可识别。

其他

`go-flags`库还有很多有意思的特性，例如支持 Windows 选项格式（`/v`和`/verbose`）、从环境变量中读取默认值、从 ini 文件中读取默认设置等等。大家有兴趣可以自行去研究~

参考

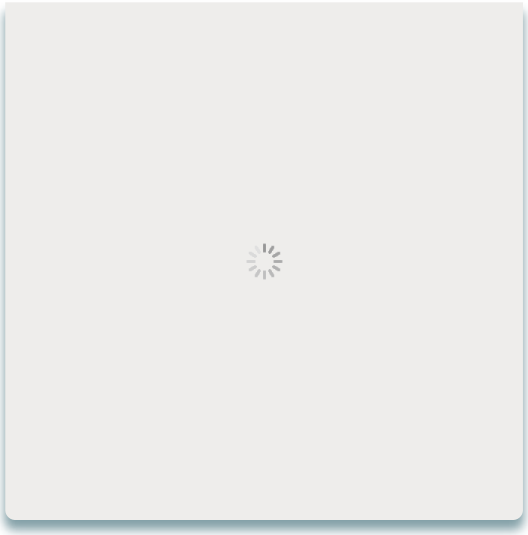
1. go-flagsGithub 仓库

2. go-flagsGoDoc 文档

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~



火车站夜景大PK，看看有没有你的家乡？

中国铁路

论躺平，我只服印度打工人

beebee星球