Go 每日一库之 flag

2020-01-10 · Go · 约 4480 字 · 预计阅读 9 分钟 | 阅读 1883

文章目录

缘起

我一直在想，有什么方式可以让人比较轻易地保持每日学习，持续输出的状态。写博客是一种方式，但不是每天都有想写的，值得写的东西。 有时候一个技术比较复杂，写博客的时候经常会写着写着发现自己的理解有偏差，或者细节还没有完全掌握，要去查资料，了解了之后又继续写，如此反复。 这样会导致一篇博客的耗时过长。

我在每天浏览思否、掘金和Github的过程中，发现一些比较好的想法，有**JS 每日一题，NodeJS 每日一库，每天一道面试题**等等等等。 https://github.com/parro-it/awesome-micro-npm-packages这个仓库收集 NodeJS 小型库，一天看一个不是梦！这也是我这个系列的灵感。 我计划每天学习一个 Go 语言的库，输出一篇介绍型的博文。每天一库当然是理想状态，我心中的预期是一周 3-5 个。

今天是第一天，我们从一个基础库聊起————Go 标准库中的`flag`。

简介

`flag`用于解析命令行选项。有过类 Unix 系统使用经验的童鞋对命令行选项应该不陌生。例如命令`ls -al`列出当前目录下所有文件和目录的详细信息，其中`-al`就是命令行选项。

命令行选项在实际开发中很常用，特别是在写工具的时候。

- 指定配置文件的路径，如`redis-server ./redis.conf`以当前目录下的配置文件`redis.conf`启动 Redis 服务器；
- 自定义某些参数，如`python -m SimpleHTTPServer 8080`启动一个 HTTP 服务器，监听 8080 端口。如果不指定，则默认监听 8000 端口。

快速使用

学习一个库的第一步当然是使用它。我们先看看`flag`库的基本使用：

```
package main

import (
  "fmt"
  "flag"
)

var (
  intflag int
  boolflag bool
  stringflag string
)

func init() {
  flag.IntVar(&intflag, "intflag", 0, "int flag value")
  flag.BoolVar(&boolflag, "boolflag", false, "bool flag value")
  flag.StringVar(&stringflag, "stringflag", "default", "string flag value")
}

func main() {
  flag.Parse()
```

```
22
23    fmt.Println("int flag:", intflag)
24    fmt.Println("bool flag:", boolflag)
25    fmt.Println("string flag:", stringflag)
26  }
```

可以先编译程序，然后运行（我使用的是 Win10 + Git Bash）：

```
1  $ go build -o main.exe main.go
2  $ ./main.exe -intflag 12 -boolflag 1 -stringflag test
```

输出：

```
1  int flag: 12
2  bool flag: true
3  string flag: test
```

如果不设置某个选项，相应变量会取默认值：

```
1  $ ./main.exe -intflag 12 -boolflag 1
```

输出：

```
int flag: 12
bool flag: true
string flag: default
```

可以看到没有设置的选项`stringflag`为默认值`default`。

还可以直接使用`go run`，这个命令会先编译程序生成可执行文件，然后执行该文件，将命令行中的其它选项传给这个程序。

```
$ go run main.go -intflag 12 -boolflag 1
```

可以使用`-h`显示选项帮助信息：

```
$ ./main.exe -h
Usage of D:\code\golang\src\github.com\darjun\cmd\flag\main.exe:
  -boolflag
        bool flag value
  -intflag int
        int flag value
  -stringflag string
        string flag value (default "default")
```

总结一下，使用`flag`库的一般步骤：

- 定义一些全局变量存储选项的值，如这里的`intflag/boolflag/stringflag`；
- 在`init`方法中使用`flag.TypeVar`方法定义选项，这里的`Type`可以为基本类型`Int/Uint/Float64/Bool`，还可以是时间间隔`time.Duration`。定义时传入变量的地址、选项名、默认值和帮助信息；
- 在`main`方法中调用`flag.Parse`从`os.Args[1:]`中解析选项。因为`os.Args[0]`为可执行程序路径，会被剔除。

注意点：

`flag.Parse`方法必须在所有选项都定义之后调用，且`flag.Parse`调用之后不能再定义选项。如果按照前面的步骤，基本不会出现问题。 因为`init`在所有代码之前执行，将选项定义都放在`init`中，`main`函数中执行`flag.Parse`时所有选项都已经定义了。

选项格式

`flag`库支持三种命令行选项格式。

```
-flag
-flag=x
-flag x
```

`-`和`--`都可以使用，它们的作用是一样的。有些库使用`-`表示短选项，`--`表示长选项。相对而言，`flag`使用起来更简单。

第一种形式只支持布尔类型的选项，出现即为 `true` ，不出现为默认值。 第三种形式不支持布尔类型的选项。因为这种形式的布尔选项在类 Unix 系统中可能会出现意想不到的行为。看下面的命令：

```
cmd -x *
```

其中， `*` 是 shell 通配符。如果有名字为 0、false的文件，布尔选项 `-x` 将会取 `false` 。反之，布尔选项 `-x` 将会取 `true` 。而且这个选项消耗了一个参数。 如果要显示设置一个布尔选项为 `false` ，只能使用 `-flag=false` 这种形式。

遇到第一个非选项参数（即不是以 `-` 和 `--` 开头的）或终止符 `--` ，解析停止。运行下面程序：

```
$ ./main.exe noflag -intflag 12
```

将会输出：

```
int flag: 0
bool flag: false
string flag: default
```

因为解析遇到 `noflag` 就停止了，后面的选项 `-intflag` 没有被解析到。所以所有选项都取的默认值。

运行下面的程序：

```
$ ./main.exe -intflag 12 -- -boolflag=true
```

将会输出：

```
int flag: 12
bool flag: false
string flag: default
```

首先解析了选项`intflag`，设置其值为 12。遇到`--`后解析终止了，后面的`--boolflag=true`没有被解析到，所以`boolflag`选项取默认值`false`。

解析终止之后如果还有命令行参数，`flag`库会存储下来，通过`flag.Args`方法返回这些参数的切片。 可以通过`flag.NArg`方法获取未解析的参数数量，`flag.Arg(i)`访问位置`i`（从 0 开始）上的参数。 选项个数也可以通过调用`flag.NFlag`方法获取。

稍稍修改一下上面的程序：

```
func main() {
  flag.Parse()

  fmt.Println(flag.Args())
  fmt.Println("Non-Flag Argument Count:", flag.NArg())
  for i := 0; i < flag.NArg(); i++ {
    fmt.Printf("Argument %d: %s\n", i, flag.Arg(i))
```

```
    }

    fmt.Println("Flag Count:", flag.NFlag())
}
```

编译运行该程序：

```
$ go build -o main.exe main.go
$ ./main.exe -intflag 12 -- -stringflag test
```

输出：

```
[-stringflag test]
Non-Flag Argument Count: 2
Argument 0: -stringflag
Argument 1: test
```

解析遇到 `--` 终止后，剩余参数 `-stringflag test` 保存在 `flag` 中，可以通过 `Args/NArg/Arg` 等方法访问。

整数选项值可以接受 1234（十进制）、0664（八进制）和 0x1234（十六进制）的形式，并且可以是负数。实际上 `flag` 在内部使用 `strconv.ParseInt` 方法将字符串解析成 `int`。 所以理论上， `ParseInt` 接受的格式都可以。

布尔类型的选项值可以为：

- 取值为`true`的：1、t、T、true、TRUE、True；
- 取值为`false`的：0、f、F、false、FALSE、False。

另一种定义选项的方式

上面我们介绍了使用`flag.TypeVar`定义选项，这种方式需要我们先定义变量，然后变量的地址。 还有一种方式，调用`flag.Type`（其中`Type`可以为`Int/Uint/Bool/Float64/String/Duration`等）会自动为我们分配变量，返回该变量的地址。用法与前一种方式类似：

```
1  package main
2
3  import (
4    "fmt"
5    "flag"
6  )
7
8  var (
9    intflag *int
10   boolflag *bool
11   stringflag *string
12 )
13
14 func init() {
```

```
  intflag = flag.Int("intflag", 0, "int flag value")
  boolflag = flag.Bool("boolflag", false, "bool flag value")
  stringflag = flag.String("stringflag", "default", "string flag value")
}

func main() {
  flag.Parse()

  fmt.Println("int flag:", *intflag)
  fmt.Println("bool flag:", *boolflag)
  fmt.Println("string flag:", *stringflag)
}
```

编译并运行程序：

```
$ go build -o main.exe main.go
$ ./main.exe -intflag 12
```

将输出：

```
int flag: 12
bool flag: false
string flag: default
```

除了使用时需要解引用，其它与前一种方式基本相同。

高级用法

定义短选项

`flag` 库并没有显示支持短选项，但是可以通过给某个相同的变量设置不同的选项来实现。即两个选项共享同一个变量。 由于初始化顺序不确定，必须保证它们拥有相同的默认值。否则不传该选项时，行为是不确定的。

```
1  package main
2
3  import (
4    "fmt"
5    "flag"
6  )
7
8  var logLevel string
9
10 func init() {
11   const (
12     defaultLogLevel = "DEBUG"
13     usage = "set log level value"
```

```
  )

  flag.StringVar(&logLevel, "log_type", defaultLogLevel, usage)
  flag.StringVar(&logLevel, "l", defaultLogLevel, usage + "(shorthand)")
}

func main() {
  flag.Parse()

  fmt.Println("log level:", logLevel)
}
```

编译、运行程序：

```
$ go build -o main.exe main.go
$ ./main.exe -log_type WARNING
$ ./main.exe -l WARNING
```

使用长、短选项均输出：

```
log level: WARNING
```

不传入该选项，输出默认值：

```
$ ./main.exe
log level: DEBUG
```

解析时间间隔

除了能使用基本类型作为选项，`flag`库还支持`time.Duration`类型，即时间间隔。时间间隔支持的格式非常之多，例如"300ms"、"-1.5h"、“2h45m"等等等等。 时间单位可以是 ns/us/ms/s/m/h/day 等。实际上`flag`内部会调用`time.ParseDuration`。具体支持的格式可以参见time（需fq）库的文档。

```
package main

import (
  "flag"
  "fmt"
  "time"
)

var (
  period time.Duration
)

func init() {
  flag.DurationVar(&period, "period", 1*time.Second, "sleep period")
}
```

```
16
17  func main() {
18    flag.Parse()
19    fmt.Printf("Sleeping for %v...", period)
20    time.Sleep(period)
21    fmt.Println()
22  }
```

根据传入的命令行选项 `period`，程序睡眠相应的时间，默认 1 秒。编译、运行程序：

```
1  $ go build -o main.exe main.go
2  $ ./main.exe
3  Sleeping for 1s...
4
5  $ ./main.exe -period 1m30s
6  Sleeping for 1m30s...
```

自定义选项

除了使用 `flag` 库提供的选项类型，我们还可以自定义选项类型。我们分析一下标准库中提供的案例：

```
1  package main
2
```

```
import (
  "errors"
  "flag"
  "fmt"
  "strings"
  "time"
)

type interval []time.Duration

func (i *interval) String() string {
  return fmt.Sprint(*i)
}

func (i *interval) Set(value string) error {
  if len(*i) > 0 {
    return errors.New("interval flag already set")
  }
  for _, dt := range strings.Split(value, ",") {
    duration, err := time.ParseDuration(dt)
    if err != nil {
      return err
    }
    *i = append(*i, duration)
  }
  return nil
}
```

```
30
31  var (
32    intervalFlag interval
33  )
34
35  func init() {
36    flag.Var(&intervalFlag, "deltaT", "comma-seperated list of intervals to use between events")
37  }
38
39  func main() {
40    flag.Parse()
41
42    fmt.Println(intervalFlag)
43  }
```

首先定义一个新类型，这里定义类型 `interval` 。

新类型必须实现 `flag.Value` 接口：

```
1  // src/flag/flag.go
2  type Value interface {
3    String() string
4    Set(string) error
5  }
```

其中`String`方法格式化该类型的值，`flag.Parse`方法在执行时遇到自定义类型的选项会将选项值作为参数调用该类型变量的`Set`方法。 这里将以`,`分隔的时间间隔解析出来存入一个切片中。

自定义类型选项的定义必须使用`flag.Var`方法。

编译、执行程序：

```
$ go build -o main.exe main.go
$ ./main.exe -deltaT 30s
[30s]
$ ./main.exe -deltaT 30s,1m,1m30s
[30s 1m0s 1m30s]
```

如果指定的选项值非法，`Set`方法返回一个`error`类型的值，`Parse`执行终止，打印错误和使用帮助。

```
$ ./main.exe -deltaT 30x
invalid value "30x" for flag -deltaT: time: unknown unit x in duration 30x
Usage of D:\code\golang\src\github.com\darjun\go-daily-lib\flag\self-defined\main.exe:
  -deltaT value
        comma-seperated list of intervals to use between events
```

解析程序中的字符串

有时候选项并不是通过命令行传递的。例如，从配置表中读取或程序生成的。这时候可以使用 `flag.FlagSet` 结构的相关方法来解析这些选项。

实际上，我们前面调用的 `flag` 库的方法，都会间接调用 `FlagSet` 结构的方法。 `flag` 库中定义了一个 `FlagSet` 类型的全局变量 `CommandLine` 专门用于解析命令行选项。 前面调用的 `flag` 库的方法只是为了提供便利，它们内部都是调用的 `CommandLine` 的相应方法：

```
// src/flag/flag.go
var CommandLine = NewFlagSet(os.Args[0], ExitOnError)

func Parse() {
  CommandLine.Parse(os.Args[1:])
}

func IntVar(p *int, name string, value int, usage string) {
  CommandLine.Var(newIntValue(value, p), name, usage)
}

func Int(name string, value int, usage string) *int {
  return CommandLine.Int(name, value, usage)
}

func NFlag() int { return len(CommandLine.actual) }

func Arg(i int) string {
```

```
19    return CommandLine.Arg(i)
20  }
21
22  func NArg() int { return len(CommandLine.args) }
```

同样的，我们也可以自己创建`FlagSet`类型变量来解析选项。

```
1   package main
2
3   import (
4     "flag"
5     "fmt"
6   )
7
8   func main() {
9     args := []string{"-intflag", "12", "-stringflag", "test"}
10
11    var intflag int
12    var boolflag bool
13    var stringflag string
14
15    fs := flag.NewFlagSet("MyFlagSet", flag.ContinueOnError)
16    fs.IntVar(&intflag, "intflag", 0, "int flag value")
17    fs.BoolVar(&boolflag, "boolflag", false, "bool flag value")
18    fs.StringVar(&stringflag, "stringflag", "default", "string flag value")
19
```

```
20    fs.Parse(args)
21
22    fmt.Println("int flag:", intflag)
23    fmt.Println("bool flag:", boolflag)
24    fmt.Println("string flag:", stringflag)
25  }
```

`NewFlagSet`方法有两个参数，第一个参数是程序名称，输出帮助或出错时会显示该信息。第二个参数是解析出错时如何处理，有几个选项：

- `ContinueOnError`：发生错误后继续解析，`CommandLine`就是使用这个选项；
- `ExitOnError`：出错时调用`os.Exit(2)`退出程序；
- `PanicOnError`：出错时产生 panic。

随便看一眼`flag`库中的相关代码：

```
1  // src/flag/flag.go
2  func (f *FlagSet) Parse(arguments []string) error {
3    f.parsed = true
4    f.args = arguments
5    for {
6      seen, err := f.parseOne()
7      if seen {
8        continue
9      }
```

```
    if err == nil {
      break
    }
    switch f.errorHandling {
    case ContinueOnError:
      return err
    case ExitOnError:
      os.Exit(2)
    case PanicOnError:
      panic(err)
    }
  }
  return nil
}
```

与直接使用`flag`库的方法有一点不同，`FlagSet`调用`Parse`方法时需要显示传入字符串切片作为参数。因为`flag.Parse`在内部调用了`CommandLine.Parse(os.Args[1:])`。 示例代码都放在GitHub上了。

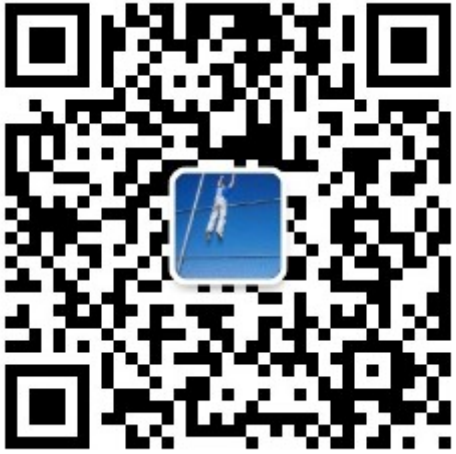
参考

1. flag库文档

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~



上一篇

下一篇

0 Comments - *powered by utteranc.es*

Write | Preview

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub