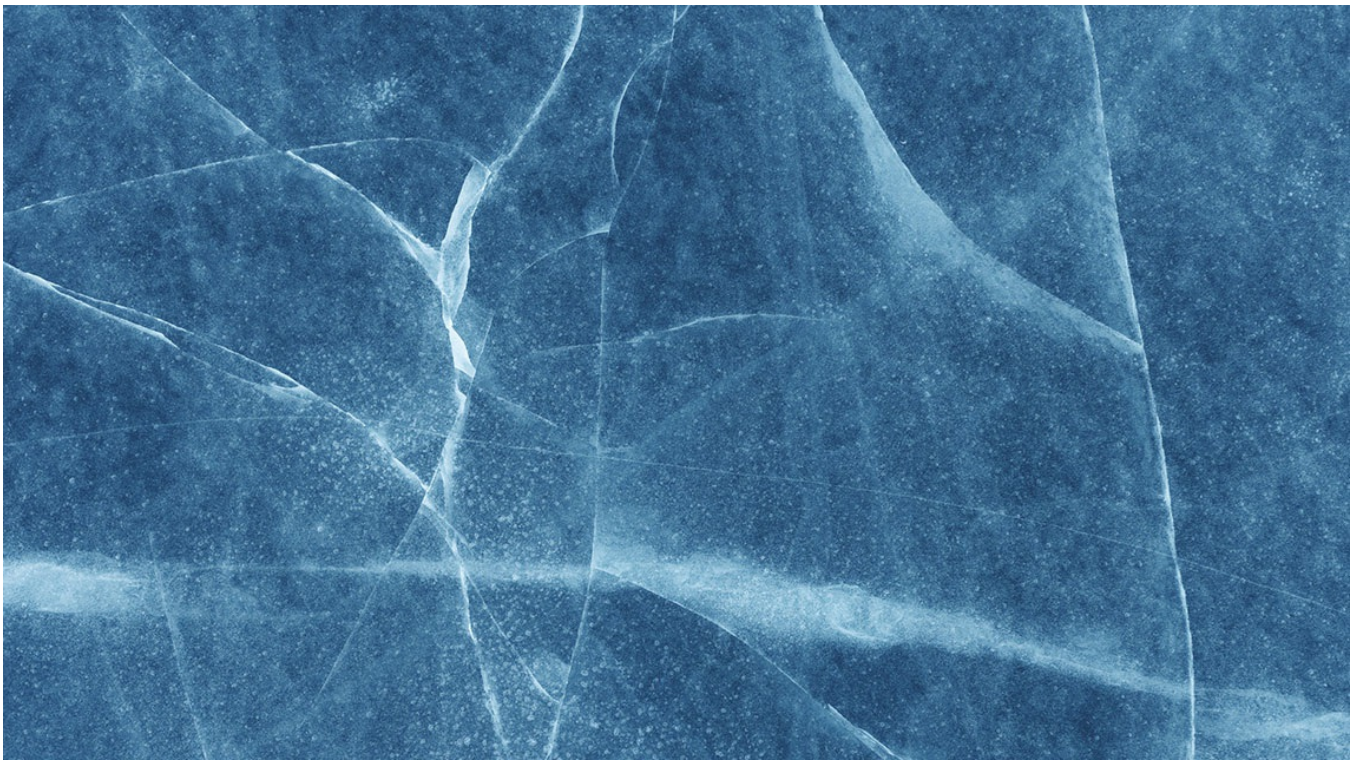
# 02 | 命令源码文件

2018-08-13 郝林



**【Go语言代码较多，建议配合文章收听音频。】**

我们已经知道，环境变量GOPATH指向的是一个或多个工作区，每个工作区中都会有以代码包为基本组织形式的源码文件。

**这里的源码文件又分为三种，即：命令源码文件、库源码文件和测试源码文件，它们都有着不同的用途和编写规则。** 我在“预习篇”的基础知识图介绍过这三种文件的基本情况。


（长按保持大图查看）

今天，我们就沿着**命令源码文件**的知识点，展开更深层级的学习。

---

一旦开始学习用编程语言编写程序，我们就一定希望在编码的过程中及时地得到反馈，只有这样才能清楚对错。实际上，我们的有效学习和进步，都是通过不断地接受反馈和执行修正实现的。

对于Go语言学习者来说，你在学习阶段中，也一定会经常编写可以直接运行的程序。这样的程序肯定会涉及命令源码文件的编写，而且，命令源码文件也可以很方便地用`go run`命令启动。

那么，**我今天的问题就是：命令源码文件的用途是什么，怎样编写它？**

这里，我给出你一个**参考的回答**：命令源码文件是程序的运行入口，是每个可独立运行的程序必须拥有的。我们可以通过构建或安装，生成与其对应的可执行文件，后者一般会与该命令源码文件的直接父目录同名。

**如果一个源码文件声明属于`main`包，并且包含一个无参数声明且无结果声明的`main`函**

**数，那么它就是命令源码文件。** 就像下面这段代码：

```
package main

import "fmt"

func main() {
 fmt.Println("Hello, world!")
}
```

如果你把这段代码存成demo1.go文件，那么运行`go run demo1.go`命令后就会在屏幕（标准输出）中看到`Hello, world!`

> 当需要模块化编程时，我们往往会将代码拆分到多个文件，甚至拆分到不同的代码包中。但无论怎样，对于一个独立的程序来说，命令源码文件永远只会也只能有一个。如果有与命令源码文件同包的源码文件，那么它们也应该声明属于`main`包。

## 问题解析

命令源码文件如此重要，以至于它毫无疑问地成为了我们学习Go语言的第一助手。不过，只会打印`Hello, world`是远远不够的，咱们千万不要成为“Hello, world”党。既然决定学习Go语言，你就应该从每一个知识点深入下去。

无论是Linux还是Windows，如果你用过命令行（command line）的话，肯定就会知道几乎所有命令（command）都是可以接收参数（argument）的。通过构建或安装命令源码文件，生成的可执行文件就可以被视为“命令”，既然是命令，那么就应该具备接收参数的能力。

下面，我就带你深入了解一下与命令参数的接收和解析有关的一系列问题。

## 知识精讲

### 1. 命令源码文件怎样接收参数

我们先看一段不完整的代码：

```
package main

import (
 // 需在此处添加代码。[1]
 "fmt"
)

var name string

func init() {
 // 需在此处添加代码。[2]
}

func main() {
 // 需在此处添加代码。[3]
 fmt.Printf("Hello, %s!\n", name)
}
```

**如果邀请你帮助我，在注释处添加相应的代码，并让程序实现”根据运行程序时给定的参数问候某人”的功能，你会打算怎样做？**

如果你知道做法，请现在就动手实现它。如果不知道也不要着急，咱们一起来搞定。

首先，Go语言标准库中有一个代码包专门用于接收和解析命令参数。这个代码包的名字叫`flag`。

我之前说过，如果想要在代码中使用某个包中的程序实体，那么应该先导入这个包。因此，我们需要在`[1]`处添加代码`"flag"`。注意，这里应该在代码包导入路径的前后加上英文半角的引号。如此一来，上述代码导入了`flag`和`fmt`这两个包。

其次，人名肯定是由字符串代表的。所以我们要在`[2]`处添加调用`flag`包的`StringVar`函数的代码。就像这样：

```
flag.StringVar(&name, "name", "everyone", "The greeting object.")
```

函数`flag.StringVar`接受4个参数。

第1个参数是用于存储该命令参数值的地址，具体到这里就是在前面声明的变量name的地址了，由表达式&name表示。

第2个参数是为了指定该命令参数的名称，这里是name。

第3个参数是为了指定在未追加该命令参数时的默认值，这里是everyone。

至于第4个函数参数，即是该命令参数的简短说明了，这在打印命令说明时会用到。

顺便说一下，还有一个与flag.StringVar函数类似的函数，叫flag.String。这两个函数的区别是，后者会直接返回一个已经分配好的用于存储命令参数值的地址。如果使用它的话，我们就需要把

```
var name string
```

改为

```
var name = flag.String("name", "everyone", "The greeting object.")
```

所以，如果我们使用flag.String函数就需要改动原有的代码。这样并不符合上述问题的要求。

再说最后一个填空。我们需要在[3]处添加代码flag.Parse()。函数flag.Parse用于真正解析命令参数，并把它们的值赋给相应的变量。

对该函数的调用必须在所有命令参数存储载体的声明（这里是对变量name的声明）和设置（这里是在[2]处对flag.StringVar函数的调用）之后，并且在读取任何命令参数值之前进行。

正因为如此，我们最好把flag.Parse()放在main函数的函数体的第一行。

**2. 怎样在运行命令源码文件的时候传入参数，又怎样查看参数的使用说明**

如果我们把上述代码存成名为demo2.go的文件，那么运行如下命令就可以为参数name传值：

```
go run demo2.go -name="Robert"
```

运行后，打印到标准输出（stdout）的内容会是：

```
Hello, Robert!
```

另外，如果想查看该命令源码文件的参数说明，可以这样做：

```
$ go run demo2.go --help
```

其中的$表示我们是在命令提示符后运行go run命令的。运行后输出的内容会类似：

```
Usage of /var/folders/ts/7lg_tl_x2gd_k1lm5g_48c7w0000gn/T/go-build155438482/b001/exe/demo2:
 -name string
    The greeting object. (default "everyone")
exit status 2
```

你可能不明白下面这段输出代码的意思。

```
/var/folders/ts/7lg_tl_x2gd_k1lm5g_48c7w0000gn/T/go-build155438482/b001/exe/demo2
```

这其实是go run命令构建上述命令源码文件时临时生成的可执行文件的完整路径。

如果我们先构建这个命令源码文件再运行生成的可执行文件，像这样：

```
$ go build demo2.go
$ ./demo2 --help
```

那么输出就会是

```
Usage of ./demo2:
 -name string
    The greeting object. (default "everyone")
```

### 3. 怎样自定义命令源码文件的参数使用说明

这有很多种方式，最简单的一种方式就是对变量flag.Usage重新赋值。flag.Usage的类型是func()，即一种无参数声明且无结果声明的函数类型。

flag.Usage变量在声明时就已经被赋值了，所以我们才能够在运行命令go run demo2.go --help时看到正确的结果。

注意，对flag.Usage的赋值必须在调用flag.Parse函数之前。

现在，我们把demo2.go另存为demo3.go，然后在main函数体的开始处加入如下代码。

```
flag.Usage = func() {
 fmt.Fprintf(os.Stderr, "Usage of %s:\n", "question")
 flag.PrintDefaults()
}
```

那么当运行

```
$ go run demo3.go --help
```

后，就会看到

```
Usage of question:
 -name string
    The greeting object. (default "everyone")
exit status 2
```

现在再深入一层，我们在调用flag包中的一些函数（比如StringVar、Parse等等）的时候，实际上是在调用flag.CommandLine变量的对应方法。

flag.CommandLine相当于默认情况下的命令参数容器。所以，通过对flag.CommandLine重新赋值，我们可以更深层次地定制当前命令源码文件的参数使用说明。

现在我们把main函数体中的那条对flag.Usage变量的赋值语句注销掉，然后在init函数体的开始处添加如下代码：

```
flag.CommandLine = flag.NewFlagSet("", flag.ExitOnError)
flag.CommandLine.Usage = func() {
 fmt.Fprintf(os.Stderr, "Usage of %s:\n", "question")
 flag.PrintDefaults()
}
```

再运行命令go run demo3.go --help后，其输出会与上一次的输出的一致。不过后面这种

定制的方法更加灵活。比如，当我们把为flag.CommandLine赋值的那条语句改为

```
flag.CommandLine = flag.NewFlagSet("", flag.PanicOnError)
```

后，再运行go run demo3.go --help命令就会产生另一种输出效果。这是由于我们在这里传给flag.NewFlagSet函数的第二个参数值是flag.PanicOnError。flag.PanicOnError和flag.ExitOnError都是预定义在flag包中的常量。

flag.ExitOnError的含义是，告诉命令参数容器，当命令后跟--help或者参数设置的不正确的时候，在打印命令参数使用说明后以状态码2结束当前程序。

状态码2代表用户错误地使用了命令，而flag.PanicOnError与之的区别是在最后抛出“运行时恐慌（panic）”。

上述两种情况都会在我们调用flag.Parse函数时被触发。顺便提一句，“运行时恐慌”是Go程序错误处理方面的概念。关于它的抛出和恢复方法，我在本专栏的后续部分中会讲到。

下面再进一步，我们索性不用全局的flag.CommandLine变量，转而自己创建一个私有的命令参数容器。我们在函数外再添加一个变量声明：

```
var cmdLine = flag.NewFlagSet("question", flag.ExitOnError)
```

然后，我们把对flag.StringVar的调用替换为对cmdLine.StringVar调用，再把flag.Parse()替换为cmdLine.Parse(os.Args[1:])。

其中的os.Args[1:]指的就是我们给定的那些命令参数。这样做就完全脱离了flag.CommandLine。*flag.FlagSet类型的变量cmdLine拥有很多有意思的方法。你可以去探索一下。我就不在这里一一讲述了。

这样做的好处依然是更灵活地定制命令参数容器。但更重要的是，你的定制完全不会影响到那个全局变量flag.CommandLine。

**总结**

恭喜你！你现在已经走出了Go语言编程的第一步。你可以用Go编写命令，并可以让它们像众多操作系统命令那样被使用，甚至可以把它们嵌入到各种脚本中。

虽然我为你讲解了命令源码文件的基本编写方法，并且也谈到了为了让它接受参数而需要做的各种准备工作，但这并不是全部。

别担心，我在后面会经常提到它的。另外，如果你想详细了解flag包的用法，可以到这个网址查看文档。或者直接使用godoc命令在本地启动一个Go语言文档服务器。怎样使用godoc命令？你可以参看这里。

## 思考题

我们已经见识过为命令源码文件传入字符串类型的参数值的方法，那还可以传入别的吗？这就是今天我留下的思考题。

1. 默认情况下，我们可以让命令源码文件接受哪些类型的参数值？
2. 我们可以把自定义的数据类型作为参数值的类型吗？如果可以，怎样做？

你可以通过查阅文档获得第一个问题的答案。记住，快速查看和理解文档是一项必备的技能。

至于第二个问题，你回答起来可能会有些困难，因为这涉及了另一个问题：“怎样声明自己的数据类型？”这个问题我在专栏的后续部分中也会讲到。如果是这样，我希望你记下它和这里说的另一问题，并在能解决后者之后再来回答前者。

戳此查看Go语言专栏文章配套详细代码。