



下载APP



## 22 | 应用构建三剑客：Pflag、Viper、Cobra 核心功能介绍

2021-07-15 孔令飞

《Go 语言项目开发实战》

课程介绍 &gt;

**讲述：孔令飞**

时长 18:15 大小 16.72M



你好，我是孔令飞。这一讲我们来聊聊构建应用时常用的 Go 包。

因为 IAM 项目使用了 Pflag、Viper 和 Cobra 包来构建 IAM 的应用框架，为了让你后面学习更加容易，这里简单介绍下这 3 个包的核心功能和使用方式。其实如果单独讲每个包的话，还是有很多功能可讲的，但我们这一讲的目的是减小你后面学习 IAM 源码的难度，所以我会主要介绍跟 IAM 相关的功能。

在正式介绍这三个包之前，我们先来看下如何构建应用的框架。



### 如何构建应用框架

想知道如何构建应用框架，首先你要明白，一个应用框架包含哪些部分。在我看来，一个应用框架需要包含以下 3 个部分：

命令行参数解析：主要用来解析命令行参数，这些命令行参数可以影响命令的运行效果。

配置文件解析：一个大型应用，通常具有很多参数，为了便于管理和配置这些参数，通常会将这些参数放在一个配置文件中，供程序读取并解析。

应用的命令行框架：应用最终是通过命令来启动的。这里有 3 个需求点，一是命令需要具备 Help 功能，这样才能告诉使用者如何去使用；二是命令需要能够解析命令行参数和配置文件；三是命令需要能够初始化业务代码，并最终启动业务进程。也就是说，我们的命令需要具备框架的能力，来纳管这 3 个部分。

这 3 个部分的功能，你可以自己开发，也可以借助业界已有的成熟实现。跟之前的想法一样，我不建议你自己开发，更建议你采用业界已有的成熟实现。命令行参数可以通过 [🔗 Flag](#) 来解析，配置文件可以通过 [🔗 Viper](#) 来解析，应用的命令行框架则可以通过 [🔗 Cobra](#) 来实现。这 3 个包目前也是最受欢迎的包，并且这 3 个包不是割裂的，而是有联系的，我们可以有机地组合这 3 个包，从而实现一个非常强大、优秀的应用命令行框架。

接下来，我们就来详细看下，这 3 个包在 Go 项目开发中是如何使用的。

## 命令行参数解析工具：Pflag 使用介绍

Go 服务开发中，经常需要给开发的组件加上各种启动参数来配置服务进程，影响服务的行为。像 kube-apiserver 就有多达 200 多个启动参数，而且这些参数的类型各不相同（例如：string、int、ip 类型等），使用方式也不相同（例如：需要支持 -- 长选项，- 短选项等），所以我们需要一个强大的命令行参数解析工具。

虽然 Go 源码中提供了一个标准库 Flag 包，用来对命令行参数进行解析，但在大型项目中应用更广泛的是另外一个包：Pflag。Pflag 提供了很多强大的特性，非常适合用来构建大型项目，一些耳熟能详的开源项目都是用 Pflag 来进行命令行参数解析的，例如：Kubernetes、Istio、Helm、Docker、Etcd 等。

接下来，我们就来介绍下如何使用 Pflag。Pflag 主要是通过创建 Flag 和 FlagSet 来使用的。我们先来看下 Flag。

## Pflag 包 Flag 定义

Pflag 可以对命令行参数进行处理，一个命令行参数在 Pflag 包中会解析为一个 Flag 类型的变量。Flag 是一个结构体，定义如下：

[复制代码](#)

```
1 type Flag struct {
2     Name          string // flag长选项的名称
3     Shorthand     string // flag短选项的名称，一个缩写的字符
4     Usage         string // flag的使用文本
5     Value         Value  // flag的值
6     DefValue      string // flag的默认值
7     Changed       bool  // 记录flag的值是否有被设置过
8     NoOptDefVal   string // 当flag出现在命令行，但是没有指定选项值时的默认值
9     Deprecated    string // 记录该flag是否被放弃
10    Hidden        bool  // 如果值为true，则从help/usage输出信息中隐藏该flag
11    ShorthandDeprecated string // 如果flag的短选项被废弃，当使用flag的短选项时打印该作
12    Annotations   map[string][]string // 给flag设置注解
13 }
```

Flag 的值是一个 Value 类型的接口，Value 定义如下：

[复制代码](#)

```
1 type Value interface {
2     String() string // 将flag类型的值转换为string类型的值，并返回string的内容
3     Set(string) error // 将string类型的值转换为flag类型的值，转换失败报错
4     Type() string // 返回flag的类型，例如：string、int、ip等
5 }
```

通过将 Flag 的值抽象成一个 interface 接口，我们就可以自定义 Flag 的类型了。只要实现了 Value 接口的结构体，就是一个新类型。

## Pflag 包 FlagSet 定义

Pflag 除了支持单个的 Flag 之外，还支持 FlagSet。FlagSet 是一些预先定义好的 Flag 的集合，几乎所有的 Pflag 操作，都需要借助 FlagSet 提供的方法来完成。在实际开发中，我们可以使用两种方法来获取并使用 FlagSet：

方法一，调用 NewFlagSet 创建一个 FlagSet。

方法二，使用 Pflag 包定义的全局 FlagSet : CommandLine。实际上 CommandLine 也是由 NewFlagSet 函数创建的。

先来看下第一种方法，自定义 FlagSet。下面是一个自定义 FlagSet 的示例：

[复制代码](#)

```
1 var version bool
2 flagSet := pflag.NewFlagSet("test", pflag.ContinueOnError)
3 flagSet.BoolVar(&version, "version", true, "Print version information and quit")
```

我们可以通过定义一个新的 FlagSet 来定义命令及其子命令的 Flag。

再来看下第二种方法，使用全局 FlagSet。下面是一个使用全局 FlagSet 的示例：

[复制代码](#)

```
1 import (
2     "github.com/spf13/pflag"
3 )
4
5 pflag.BoolVarP(&version, "version", "v", true, "Print version information and quit")
```

这其中，pflag.BoolVarP 函数定义如下：

[复制代码](#)

```
1 func BoolVarP(p *bool, name, shorthand string, value bool, usage string) {
2     flag := CommandLine.VarPF(newBoolValue(value, p), name, shorthand, usage)
3     flag.NoOptDefVal = "true"
4 }
```

可以看到 pflag.BoolVarP 最终调用了 CommandLine，CommandLine 是一个包级别的变量，定义为：

[复制代码](#)

```
1 // CommandLine is the default set of command-line flags, parsed from os.Args.
2 var CommandLine = NewFlagSet(os.Args[0], ExitOnError)
```

在一些不需要定义子命令的命令行工具中，我们可以直接使用全局的 FlagSet，更加简单方便。

## Pflag 使用方法

上面，我们介绍了使用 Pflag 包的两个核心结构体。接下来，我来详细介绍下 Pflag 的常见使用方法。Pflag 有很多强大的功能，我这里介绍 7 个常见的使用方法。

### 1. 支持多种命令行参数定义方式。

Pflag 支持以下 4 种命令行参数定义方式：

支持长选项、默认值和使用文本，并将标志的值存储在指针中。

```
1 var name = pflag.String("name", "colin", "Input Your Name")
```

[复制代码](#)

支持长选项、短选项、默认值和使用文本，并将标志的值存储在指针中。

```
1 var name = pflag.StringP("name", "n", "colin", "Input Your Name")
```

[复制代码](#)

支持长选项、默认值和使用文本，并将标志的值绑定到变量。

```
1 var name string
2 pflag.StringVar(&name, "name", "colin", "Input Your Name")
```

[复制代码](#)

支持长选项、短选项、默认值和使用文本，并将标志的值绑定到变量。

```
1 var name string
2 pflag.StringVarP(&name, "name", "n", "colin", "Input Your Name")
```

[复制代码](#)

上面的函数命名是有规则的：

函数名带Var说明是将标志的值绑定到变量，否则是将标志的值存储在指针中。

函数名带P说明支持短选项，否则不支持短选项。

## 2. 使用Get<Type>获取参数的值。

可以使用Get<Type>来获取标志的值，<Type>代表 Pflag 所支持的类型。例如：有一个 pflag.FlagSet，带有一个名为 flagname 的 int 类型的标志，可以使用GetInt()来获取 int 值。需要注意 flagname 必须存在且必须是 int，例如：

```
1 i, err := flagset.GetInt("flagname")
```

[复制代码](#)


## 3. 获取非选项参数。

代码示例如下：

```
1 package main
2
3 import (
4     "fmt"
5
6     "github.com/spf13/pflag"
7 )
8
9 var (
10     flagvar = pflag.Int("flagname", 1234, "help message for flagname")
11 )
12
13 func main() {
14     pflag.Parse()
15
16     fmt.Printf("argument number is: %v\n", pflag.NArg())
17     fmt.Printf("argument list is: %v\n", pflag.Args())
18     fmt.Printf("the first argument is: %v\n", pflag.Arg(0))
19 }
```

[复制代码](#)

执行上述代码，输出如下：


 复制代码

```
1 $ go run example1.go arg1 arg2
2 argument number is: 2
3 argument list is: [arg1 arg2]
4 the first argument is: arg1
```

在定义完标志之后，可以调用pflag.Parse()来解析定义的标志。解析后，可通过pflag.Args()返回所有的非选项参数，通过pflag.Arg(i)返回第i个非选项参数。参数下标 0 到 pflag.NArg() - 1。

4. 指定了选项但是没有指定选项值时的默认值。

创建一个 Flag 后，可以为这个 Flag 设置pflag.NoOptDefVal。如果一个 Flag 具有NoOptDefVal，并且该 Flag 在命令行上没有设置这个 Flag 的值，则该标志将设置为NoOptDefVal 指定的值。例如：

 复制代码

```
1 var ip = flag.IntP("flagname", "f", 1234, "help message")
2 flag.Lookup("flagname").NoOptDefVal = "4321"
```

上面的代码会产生结果，具体你可以参照下表：

命令行参数	解析结果
--flagname=1357	ip=1357
--flagname	ip=4321
[nothing]	ip=1234

5. 弃用标志或者标志的简写。

Pflag 可以弃用标志或者标志的简写。弃用的标志或标志简写在帮助文本中会被隐藏，并在使用不推荐的标志或简写时打印正确的用法提示。例如，弃用名为 logmode 的标志，并

告知用户应该使用哪个标志代替：

[复制代码](#)

```
1 // deprecate a flag by specifying its name and a usage message
2 pflag.CommandLine.MarkDeprecated("logmode", "please use --log-mode instead")
```

这样隐藏了帮助文本中的 logmode，并且当使用 logmode 时，打印了 Flag --logmode has been deprecated, please use --log-mode instead。

6) 保留名为 port 的标志，但是弃用它的简写形式。

[复制代码](#)

```
1 pflag.IntVarP(&port, "port", "P", 3306, "MySQL service host port.")
2
3 // deprecate a flag shorthand by specifying its flag name and a usage message
4 pflag.CommandLine.MarkShorthandDeprecated("port", "please use --port only")
```

这样隐藏了帮助文本中的简写 P，并且当使用简写 P 时，打印了 Flag shorthand -P has been deprecated, please use --port only。usage message 在此处必不可少，并且不应为空。

7. 隐藏标志。

可以将 Flag 标记为隐藏的，这意味着它仍将正常运行，但不会显示在 usage/help 文本中。例如：隐藏名为 secretFlag 的标志，只在内部使用，并且不希望它显示在帮助文本或者使用文本中。代码如下：

[复制代码](#)

```
1 // hide a flag by specifying its name
2 pflag.CommandLine.MarkHidden("secretFlag")
```

至此，我们介绍了 Pflag 包的重要用法。接下来，我们再来看下如何解析配置文件。

## 配置解析神器：Viper 使用介绍



几乎所有的后端服务，都需要一些配置项来配置我们的服务，一些小型的项目，配置不是很多，可以选择只通过命令行参数来传递配置。但是大型项目配置很多，通过命令行参数传递就变得很麻烦，不好维护。标准的解决方案是将这些配置信息保存在配置文件中，由程序启动时加载和解析。Go 生态中有很多包可以加载并解析配置文件，目前最受欢迎的是 Viper 包。

Viper 是 Go 应用程序现代化的、完整的解决方案，能够处理不同格式的配置文件，让我们在构建现代应用程序时，不必担心配置文件格式。Viper 也能够满足我们对应用配置的各种需求。

Viper 可以从不同的位置读取配置，不同位置的配置具有不同的优先级，高优先级的配置会覆盖低优先级相同的配置，按优先级从高到低排列如下：

1. 通过 viper.Set 函数显示设置的配置
2. 命令行参数
3. 环境变量
4. 配置文件
5. Key/Value 存储
6. 默认值

这里需要注意，Viper 配置键不区分大小写。

Viper 有很多功能，最重要的两类功能是读入配置和读取配置，Viper 提供不同的方式来实现这两类功能。接下来，我们就来详细介绍下 Viper 如何读入配置和读取配置。

## 读入配置

读入配置，就是将配置读入到 Viper 中，有如下读入方式：

设置默认的配置文件名。

读取配置文件。

监听和重新读取配置文件。

从 io.Reader 读取配置。

从环境变量读取。


从命令行标志读取。

从远程 Key/Value 存储读取。

这几个方法的具体读入方式，你可以看下面的展示。

## 1. 设置默认值。

一个好的配置系统应该支持默认值。Viper 支持对 key 设置默认值，当没有通过配置文件、环境变量、远程配置或命令行标志设置 key 时，设置默认值通常是很有用的，可以让程序在没有明确指定配置时也能够正常运行。例如：


 复制代码

```
1 viper.SetDefault("ContentDir", "content")
2 viper.SetDefault("LayoutDir", "layouts")
3 viper.SetDefault("Taxonomies", map[string]string{"tag": "tags", "category": "c
```

## 2. 读取配置文件。

Viper 可以读取配置文件来解析配置，支持 JSON、TOML、YAML、YML、Properties、Props、Prop、HCL、Dotenv、Env 格式的配置文件。Viper 支持搜索多个路径，并且默认不配置任何搜索路径，将默认决策留给应用程序。

以下是如何使用 Viper 搜索和读取配置文件的示例：

 复制代码

```
1 package main
2
3 import (
4     "fmt"
5
6     "github.com/spf13/pflag"
7     "github.com/spf13/viper"
8 )
9
10 var (
```


```
11  cfg = pflag.StringP("config", "c", "", "Configuration file.")
12  help = pflag.BoolP("help", "h", false, "Show this help message.")
13 )
14
15 func main() {
16     pflag.Parse()
17     if *help {
18         pflag.Usage()
19         return
20     }
21
22     // 从配置文件中读取配置
23     if *cfg != "" {
24         viper.SetConfigFile(*cfg) // 指定配置文件名
25         viper.SetConfigType("yaml") // 如果配置文件中没有文件扩展名，则需要指定配置文件的
26     } else {
27         viper.AddConfigPath(".") // 把当前目录加入到配置文件的搜索路径中
28         viper.AddConfigPath("$HOME/.iam") // 配置文件搜索路径，可以设置多个配置文件搜索路
29         viper.SetConfigName("config") // 配置文件名称（没有文件扩展名）
30     }
31
32     if err := viper.ReadInConfig(); err != nil { // 读取配置文件。如果指定了配置文件名
33         panic(fmt.Errorf("Fatal error config file: %s \n", err))
34     }
35
36     fmt.Printf("Used configuration file is: %s\n", viper.ConfigFileUsed())
37 }
```

Viper 支持设置多个配置文件搜索路径，需要注意添加搜索路径的顺序，Viper 会根据添加的路径顺序搜索配置文件，如果找到则停止搜索。如果调用 `SetConfigFile` 直接指定了配置文件名，并且配置文件名没有文件扩展名时，需要显式指定配置文件的格式，以使 Viper 能够正确解析配置文件。

如果通过搜索的方式查找配置文件，则需要注意，`SetConfigName` 设置的配置文件名是不带扩展名的，在搜索时 Viper 会在文件名之后追加文件扩展名，并尝试搜索所有支持的扩展类型。

### 3. 监听和重新读取配置文件。

Viper 支持在运行时让应用程序实时读取配置文件，也就是热加载配置。可以通过 `WatchConfig` 函数热加载配置。在调用 `WatchConfig` 函数之前，需要确保已经添加了配置文件的搜索路径。另外，还可以为 Viper 提供一个回调函数，以便在每次发生更改时运行。这里我也给你个示例：


 复制代码

```
1 viper.WatchConfig()
2 viper.OnConfigChange(func(e fsnotify.Event) {
3     // 配置文件发生变更之后会调用的回调函数
4     fmt.Println("Config file changed:", e.Name)
5 })
```

我不建议在实际开发中使用热加载功能，因为即使配置热加载了，程序中的代码也不一定会热加载。例如：修改了服务监听端口，但是服务没有重启，这时候服务还是监听在老的端口上，会造成不一致。

#### 4) 设置配置值。

我们可以通过 viper.Set() 函数来显式设置配置：

 复制代码

```
1 viper.Set("user.username", "colin")
```

#### 5. 使用环境变量。

Viper 还支持环境变量，通过如下 5 个函数来支持环境变量：

AutomaticEnv()

BindEnv(input ...string) error

SetEnvPrefix(in string)

SetEnvKeyReplacer(r \*strings.Replacer)

AllowEmptyEnv(allowEmptyEnv bool)

这里要注意：Viper 读取环境变量是区分大小写的。Viper 提供了一种机制来确保 Env 变量是唯一的。通过使用 SetEnvPrefix，可以告诉 Viper 在读取环境变量时使用前缀。

BindEnv 和 AutomaticEnv 都将使用此前缀。比如，我们设置了

viper.SetEnvPrefix("VIPER")，当使用 viper.Get("apiversion") 时，实际读取的环境变量是VIPER\_APIVERSION。

BindEnv 需要一个或两个参数。第一个参数是键名，第二个是环境变量的名称，环境变量的名称区分大小写。如果未提供 Env 变量名，则 Viper 将假定 Env 变量名为：环境变量前缀\_键名全大写。例如：前缀为 VIPER，key 为 username，则 Env 变量名为 VIPER\_USERNAME。当显示提供 Env 变量名（第二个参数）时，它不会自动添加前缀。例如，如果第二个参数是 ID，Viper 将查找环境变量 ID。

在使用 Env 变量时，需要注意的一件重要的事情是：每次访问该值时都将读取它。Viper 在调用 BindEnv 时不固定该值。

还有一个魔法函数 SetEnvKeyReplacer，SetEnvKeyReplacer 允许你使用 strings.Replacer 对象来重写 Env 键。如果你想在 Get() 调用中使用-或者.，但希望你的环境变量使用\_分隔符，可以通过 SetEnvKeyReplacer 来实现。比如，我们设置了环境变量 USER\_SECRET\_KEY=bVix2WBv0VPfrDrvLLWrhEdzjLpPCNYb，但我们想用 viper.Get("user.secret-key")，那我们就调用函数：

```
1 viper.SetEnvKeyReplacer(strings.NewReplacer(".", "_", "-", "_"))
```

[复制代码](#)

上面的代码，在调用 viper.Get() 函数时，会用 \_ 替换.和-。默认情况下，空环境变量被认为是未设置的，并将返回到下一个配置源。若要将空环境变量视为已设置，可以使用 AllowEmptyEnv 方法。使用环境变量示例如下：

```
1 // 使用环境变量
2 os.Setenv("VIPER_USER_SECRET_ID", "QLdywI2MrmDVjSSv6e95weNRvmteRj fKAuNV")
3 os.Setenv("VIPER_USER_SECRET_KEY", "bVix2WBv0VPfrDrvLLWrhEdzjLpPCNYb")
4
5 viper.AutomaticEnv() // 读取环境变量
6 viper.SetEnvPrefix("VIPER") // 设置环境变量
7 viper.SetEnvKeyReplacer(strings.NewReplacer(".", "_", "-", "_")) // 将viper.Ge
8 viper.BindEnv("user.secret-key")
9 viper.BindEnv("user.secret-id", "USER_SECRET_ID") // 绑定环境变量名到key
```

[复制代码](#)

## 6. 使用标志。

Viper 支持 Pflag 包，能够绑定 key 到 Flag。与 BindEnv 类似，在调用绑定方法时，不会设置该值，但在访问它时会设置。对于单个标志，可以调用 BindPFlag() 进行绑定：

```
1 viper.BindPFlag("token", pflag.Lookup("token")) // 绑定单个标志
```

[复制代码](#)

还可以绑定一组现有的 pflags ( pflag.FlagSet )：

```
1 viper.BindPFlags(pflag.CommandLine) //绑定标志集
```

[复制代码](#)

## 读取配置

Viper 提供了如下方法来读取配置：

Get(key string) interface{}

Get<Type>(key string) <Type>

AllSettings() map[string]interface{}

IsSet(key string) : bool

每一个 Get 方法在找不到值的时候都会返回零值。为了检查给定的键是否存在，可以使用 IsSet() 方法。<Type>可以是 Viper 支持的类型，首字母大写：Bool、Float64、Int、IntSlice、String、StringMap、StringMapString、StringSlice、Time、Duration。例如：GetInt()。

常见的读取配置方法有以下几种。

### 1. 访问嵌套的键。

例如，加载下面的 JSON 文件：

```
1 {
```

[复制代码](#)

```
2     "host": {
3         "address": "localhost",
4         "port": 5799
5     },
6     "datastore": {
7         "metric": {
8             "host": "127.0.0.1",
9             "port": 3099
10        },
11        "warehouse": {
12            "host": "198.0.0.1",
13            "port": 2112
14        }
15    }
16 }
```

Viper 可以通过传入 . 分隔的路径来访问嵌套字段：

[复制代码](#)

```
1 viper.GetString("datastore.metric.host") // (返回 "127.0.0.1")
```

如果datastore.metric被直接赋值覆盖（被 Flag、环境变量、set() 方法等等），那么datastore.metric的所有子键都将变为未定义状态，它们被高优先级配置级别覆盖了。

如果存在与分隔的键路径匹配的键，则直接返回其值。例如：

[复制代码](#)

```
1 {
2     "datastore.metric.host": "0.0.0.0",
3     "host": {
4         "address": "localhost",
5         "port": 5799
6     },
7     "datastore": {
8         "metric": {
9             "host": "127.0.0.1",
10            "port": 3099
11        },
12        "warehouse": {
13            "host": "198.0.0.1",
14            "port": 2112
15        }
16    }
17 }
```

通过 viper.GetString 获取值：

```
1 viper.GetString("datastore.metric.host") // 返回 "0.0.0.0"
```

[复制代码](#)

## 2. 反序列化。

Viper 可以支持将所有或特定的值解析到结构体、map 等。可以通过两个函数来实现：

Unmarshal(rawVal interface{}) error

UnmarshalKey(key string, rawVal interface{}) error

一个示例：

```
1 type config struct {
2     Port int
3     Name string
4     PathMap string `mapstructure:"path_map"`
5 }
6
7 var C config
8
9 err := viper.Unmarshal(&C)
10 if err != nil {
11     t.Fatalf("unable to decode into struct, %v", err)
12 }
```

[复制代码](#)

如果想要解析那些键本身就包含.(默认的键分隔符)的配置,则需要修改分隔符：

```
1 v := viper.NewWithOptions(viper.KeyDelimiter("::"))
2
3 v.SetDefault("chart::values", map[string]interface{}{
4     "ingress": map[string]interface{}{
5         "annotations": map[string]interface{}{
6             "traefik.frontend.rule.type": "PathPrefix",
```

[复制代码](#)



```
7         "traefik.ingress.kubernetes.io/ssl-redirect": "true",
8     },
9 },
10 })
11
12 type config struct {
13     Chart struct{
14         Values map[string]interface{}
15     }
16 }
17
18 var C config
19
20 v.Unmarshal(&C)
```

Viper 在后台使用 [github.com/mitchellh/mapstructure](https://github.com/mitchellh/mapstructure) 来解析值，其默认情况下使用 `mapstructure tags`。当我们需要将 Viper 读取的配置反序列到我们定义的结构体变量中时，一定要使用 `mapstructure tags`。

### 3. 序列化成字符串。

有时候我们需要将 Viper 中保存的所有设置序列化到一个字符串中，而不是将它们写入到一个文件中，示例如下：

```
1 import (
2     yaml "gopkg.in/yaml.v2"
3     // ...
4 )
5
6 func yamlStringSettings() string {
7     c := viper.AllSettings()
8     bs, err := yaml.Marshal(c)
9     if err != nil {
10         log.Fatalf("unable to marshal config to YAML: %v", err)
11     }
12     return string(bs)
13 }
```

[复制代码](#)

## 现代化的命令行框架：Cobra 全解

Cobra 既是一个可以创建强大的现代 CLI 应用程序的库，也是一个可以生成应用和命令文件的程序。有许多大型项目都是用 Cobra 来构建应用程序的，例如 Kubernetes、

Docker、etcd、Rkt、Hugo 等。

Cobra 建立在 commands、arguments 和 flags 结构之上。commands 代表命令，arguments 代表非选项参数，flags 代表选项参数（也叫标志）。一个好的应用程序应该是易懂的，用户可以清晰地知道如何去使用这个应用程序。应用程序通常遵循如下模式：APPNAME VERB NOUN --ADJECTIVE 或者 APPNAME COMMAND ARG --FLAG，例如：

[复制代码](#)

```
1 git clone URL --bare # clone 是一个命令，URL是一个非选项参数，bare是一个选项参数
```

这里，VERB 代表动词，NOUN 代表名词，ADJECTIVE 代表形容词。

Cobra 提供了两种方式来创建命令：Cobra 命令和 Cobra 库。Cobra 命令可以生成一个 Cobra 命令模板，而命令模板也是通过引用 Cobra 库来构建命令的。所以，这里我直接介绍如何使用 Cobra 库来创建命令。

## 使用 Cobra 库创建命令

如果要用 Cobra 库编码实现一个应用程序，需要首先创建一个空的 main.go 文件和一个 rootCmd 文件，之后可以根据需要添加其他命令。具体步骤如下：

### 1. 创建 rootCmd。

[复制代码](#)

```
1 $ mkdir -p newApp2 && cd newApp2
```

通常情况下，我们会将 rootCmd 放在文件 cmd/root.go 中。

[复制代码](#)

```
1 var rootCmd = &cobra.Command{
2     Use:     "hugo",
3     Short:   "Hugo is a very fast static site generator",
4     Long:    `A Fast and Flexible Static Site Generator built with
5             love by spf13 and friends in Go.
6             Complete documentation is available at http://hugo.spf13.com`,
7     Run:     func(cmd *cobra.Command, args []string) {
```

```
8     // Do Stuff Here
9 },
10 }
11
12 func Execute() {
13     if err := rootCmd.Execute(); err != nil {
14         fmt.Println(err)
15         os.Exit(1)
16     }
17 }
```

还可以在 `init()` 函数中定义标志和处理配置，例如 `cmd/root.go`。

[📄 复制代码](#)

```
1 import (
2     "fmt"
3     "os"
4
5     homedir "github.com/mitchellh/go-homedir"
6     "github.com/spf13/cobra"
7     "github.com/spf13/viper"
8 )
9
10 var (
11     cfgFile      string
12     projectBase  string
13     userLicense  string
14 )
15
16 func init() {
17     cobra.OnInitialize(initConfig)
18     rootCmd.PersistentFlags().StringVar(&cfgFile, "config", "", "config file (default is "+homedir.Expand("~/.cobra.conf")+"")
19     rootCmd.PersistentFlags().StringVarP(&projectBase, "projectbase", "b", "", "project base directory")
20     rootCmd.PersistentFlags().StringP("author", "a", "YOUR NAME", "Author name for copyright")
21     rootCmd.PersistentFlags().StringVarP(&userLicense, "license", "l", "", "Name of license")
22     rootCmd.PersistentFlags().Bool("viper", true, "Use Viper for configuration")
23     viper.BindPFlag("author", rootCmd.PersistentFlags().Lookup("author"))
24     viper.BindPFlag("projectbase", rootCmd.PersistentFlags().Lookup("projectbase"))
25     viper.BindPFlag("useViper", rootCmd.PersistentFlags().Lookup("viper"))
26     viper.SetDefault("author", "NAME HERE <EMAIL ADDRESS>")
27     viper.SetDefault("license", "apache")
28 }
29
30 func initConfig() {
31     // Don't forget to read config either from cfgFile or from home directory!
32     if cfgFile != "" {
33         // Use config file from the flag.
34         viper.SetConfigFile(cfgFile)
35     } else {
```

```
36 // Find home directory.
37 home, err := homedir.Dir()
38 if err != nil {
39     fmt.Println(err)
40     os.Exit(1)
41 }
42
43 // Search config in home directory with name ".cobra" (without extension).
44 viper.AddConfigPath(home)
45 viper.SetConfigName(".cobra")
46 }
47
48 if err := viper.ReadInConfig(); err != nil {
49     fmt.Println("Can't read config:", err)
50     os.Exit(1)
51 }
52 }
```

## 2. 创建 main.go。

我们还需要一个 main 函数来调用 rootCmd，通常我们会创建一个 main.go 文件，在 main.go 中调用 rootCmd.Execute() 来执行命令：


[复制代码](#)

```
1 package main
2
3 import (
4     "{pathToYourApp}/cmd"
5 )
6
7 func main() {
8     cmd.Execute()
9 }
```

需要注意，main.go 中不建议放很多代码，通常只需要调用 cmd.Execute() 即可。

## 3. 添加命令。

除了 rootCmd，我们还可以调用 AddCommand 添加其他命令，通常情况下，我们会把其他命令的源码文件放在 cmd/ 目录下，例如，我们添加一个 version 命令，可以创建 cmd/version.go 文件，内容为：

 复制代码

```
1 package cmd
2
3 import (
4     "fmt"
5
6     "github.com/spf13/cobra"
7 )
8
9 func init() {
10     rootCmd.AddCommand(versionCmd)
11 }
12
13 var versionCmd = &cobra.Command{
14     Use:     "version",
15     Short:   "Print the version number of Hugo",
16     Long:    `All software has versions. This is Hugo's`,
17     Run:     func(cmd *cobra.Command, args []string) {
18         fmt.Println("Hugo Static Site Generator v0.9 -- HEAD")
19     },
20 }
```

本示例中，我们通过调用`rootCmd.AddCommand(versionCmd)`给 `rootCmd` 命令添加了一个 `versionCmd` 命令。

#### 4. 编译并运行。

将 `main.go` 中`{pathToYourApp}`替换为对应的路径，例如本示例中 `pathToYourApp` 为 `github.com/marmotedu/gopractise-demo/cobra/newApp2`。

 复制代码

```
1 $ go mod init github.com/marmotedu/gopractise-demo/cobra/newApp2
2 $ go build -v .
3 $ ./newApp2 -h
4 A Fast and Flexible Static Site Generator built with
5 love by spf13 and friends in Go.
6 Complete documentation is available at http://hugo.spf13.com
7
8 Usage:
9 hugo [flags]
10 hugo [command]
11
12 Available Commands:
13 help Help about any command
14 version Print the version number of Hugo
```

```
15
16 Flags:
17 -a, --author string Author name for copyright attribution (default "YOUR NAME"
18 --config string config file (default is $HOME/.cobra.yaml)
19 -h, --help help for hugo
20 -l, --license licensetext Name of license for the project (can provide license
21 -b, --projectbase string base project directory eg. github.com/spf13/
22 --viper Use Viper for configuration (default true)
23
24 Use "hugo [command] --help" for more information about a command.
```

通过步骤一、步骤二、步骤三，我们就成功创建和添加了 Cobra 应用程序及其命令。

接下来，我再来详细介绍下 Cobra 的核心特性。

## 使用标志

Cobra 可以跟 Pflag 结合使用，实现强大的标志功能。使用步骤如下：

### 1. 使用持久化的标志。


标志可以是“持久的”，这意味着该标志可用于它所分配的命令以及该命令下的每个子命令。可以在 rootCmd 上定义持久标志：

 复制代码

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

### 2. 使用本地标志。

也可以分配一个本地标志，本地标志只能在它所绑定的命令上使用：


 复制代码

```
1 rootCmd.Flags().StringVarP(&Source, "source", "s", "", "Source directory to re
```

--source 标志只能在 rootCmd 上引用，而不能在 rootCmd 的子命令上引用。

### 3. 将标志绑定到 Viper。


我们可以将标志绑定到 Viper，这样就可以使用 viper.Get() 获取标志的值。

 复制代码

```
1 var author string
2
3 func init() {
4     rootCmd.PersistentFlags().StringVar(&author, "author", "YOUR NAME", "Author
5     viper.BindPFlag("author", rootCmd.PersistentFlags().Lookup("author"))
6 }
```

#### 4. 设置标志为必选。

默认情况下，标志是可选的，我们也可以设置标志为必选，当设置标志为必选，但是没有提供标志时，Cobra 会报错。

 复制代码

```
1 rootCmd.Flags().StringVarP(&Region, "region", "r", "", "AWS region (required)"
2 rootCmd.MarkFlagRequired("region")
```

## 非选项参数验证

在命令的过程中，经常会传入非选项参数，并且需要对这些非选项参数进行验证，Cobra 提供了机制来对非选项参数进行验证。可以使用 Command 的 Args 字段来验证非选项参数。Cobra 也内置了一些验证函数：

NoArgs：如果存在任何非选项参数，该命令将报错。

ArbitraryArgs：该命令将接受任何非选项参数。

OnlyValidArgs：如果有任何非选项参数不在 Command 的 ValidArgs 字段中，该命令将报错。

MinimumNArgs(int)：如果没有至少 N 个非选项参数，该命令将报错。

MaximumNArgs(int)：如果有多于 N 个非选项参数，该命令将报错。

ExactArgs(int)：如果非选项参数个数不为 N，该命令将报错。

ExactValidArgs(int)：如果非选项参数的个数不为 N，或者非选项参数不在 Command 的 ValidArgs 字段中，该命令将报错。

RangeArgs(min, max) : 如果非选项参数的个数不在 min 和 max 之间, 该命令将报错。

使用预定义验证函数, 示例如下:

[复制代码](#)

```
1 var cmd = &cobra.Command{
2     Short: "hello",
3     Args: cobra.MinimumNArgs(1), // 使用内置的验证函数
4     Run: func(cmd *cobra.Command, args []string) {
5         fmt.Println("Hello, World!")
6     },
7 }
```

当然你也可以自定义验证函数, 示例如下:

[复制代码](#)

```
1 var cmd = &cobra.Command{
2     Short: "hello",
3     // Args: cobra.MinimumNArgs(10), // 使用内置的验证函数
4     Args: func(cmd *cobra.Command, args []string) error { // 自定义验证函数
5         if len(args) < 1 {
6             return errors.New("requires at least one arg")
7         }
8         if myapp.IsValidColor(args[0]) {
9             return nil
10        }
11        return fmt.Errorf("invalid color specified: %s", args[0])
12    },
13    Run: func(cmd *cobra.Command, args []string) {
14        fmt.Println("Hello, World!")
15    },
16 }
```

## PreRun and PostRun Hooks

在运行 Run 函数时, 我们可以运行一些钩子函数, 比如 PersistentPreRun 和 PreRun 函数在 Run 函数之前执行, PersistentPostRun 和 PostRun 在 Run 函数之后执行。如果子命令没有指定 Persistent\*Run 函数, 则子命令将会继承父命令的 Persistent\*Run 函数。这些函数的运行顺序如下:



1. PersistentPreRun
2. PreRun
3. Run
4. PostRun
5. PersistentPostRun

注意，父级的 PreRun 只会在父级命令运行时调用，子命令是不会调用的。

Cobra 还支持很多其他有用的特性，比如：自定义 Help 命令；可以自动添加 `--version` 标志，输出程序版本信息；当用户提供无效标志或无效命令时，Cobra 可以打印出 usage 信息；当我们输入的命令有误时，Cobra 会根据注册的命令，推算出可能的命令，等等。

## 总结

在开发 Go 项目时，我们可以通过 Pflag 来解析命令行参数，通过 Viper 来解析配置文件，用 Cobra 来实现命令行框架。你可以通过 `pflag.String()`、`pflag.StringP()`、`pflag.StringVar()`、`pflag.StringVarP()` 方法来设置命令行参数，并使用 `Get<Type>` 来获取参数的值。

同时，你也可以使用 Viper 从命令行参数、环境变量、配置文件等位置读取配置项。最常用的是从配置文件中读取，可以通过 `viper.AddConfigPath` 来设置配置文件搜索路径，通过 `viper.SetConfigFile` 和 `viper.SetConfigType` 来设置配置文件名，通过 `viper.ReadInConfig` 来读取配置文件。读取完配置文件，然后在程序中使用 `Get/Get<Type>` 来读取配置项的值。

最后，你可以使用 Cobra 来构建一个命令行框架，Cobra 可以很好地集成 Pflag 和 Viper。

## 课后练习

1. 研究下 Cobra 的代码，看下 Cobra 是如何跟 Pflag 和 Viper 进行集成的。
2. 思考下，除了 Pflag、Viper、Cobra，你在开发过程中还遇到哪些优秀的包，来处理命令行参数、配置文件和启动命令行框架的呢？欢迎在留言区分享。

欢迎你在留言区与我交流讨论，我们下一讲见！

分享给需要的人，Ta订阅后你可得 **24** 元现金奖励

👍 赞 1

💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | 日志处理（下）：手把手教你从 0 编写一个日志包

下一篇 23 | 应用构建实战：如何构建一个优秀的企业应用框架？

## 更多课程推荐

# 容器实战高手课

在实战中深入理解容器技术的本质

李程远

eBay 总监级工程师  
云平台架构师



涨价倒计时 🕒

今日订阅 **¥69**，7月20日涨价至 **¥129**

## 精选留言 (3)

💬 写留言



龍捲風

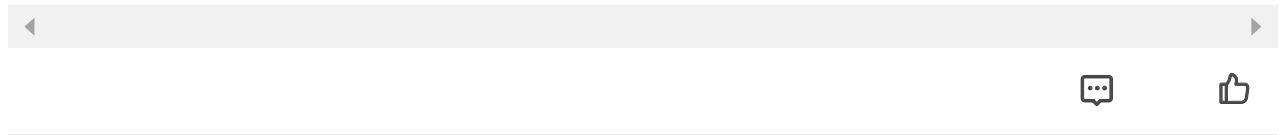
2021-07-16

想问下，关于golang的项目怎么实现优雅下线？注册中心可以将服务节点剔除，但关闭服务时怎么让已接收到请求可以做完呢？

展开 ∨

作者回复: net/http包有提供ShutDown方法。

第24讲：Web 服务：Web 服务核心功能有哪些，如何实现？有讲如何实现优雅关停。



**helloworld**

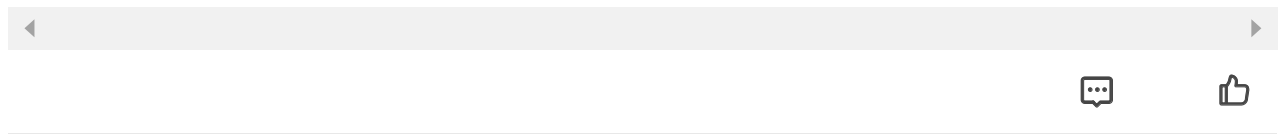
2021-07-16

开发环境，测试环境，生产环境的配置文件是不同的，关于不同环境下读取配置文件，这块有什么好的最佳实践吗，是通过命令行参数指定不同环境的配置文件的方式好呢，还是通过系统环境变量来区分环境并读取对应配置文件好呢

作者回复: 如果参数少，比如 < 5个，可以在命令行参数指定。

如果参数  $\geq 5$ 个，最好在配置文件指定，好维护。

注意：5 不是标准答案，需要你来定。



**Neroldy**

2021-07-15

标志可以是“持久的”，这意味着该标志可用于它所分配的命令以及该命令下的每个子命令。

老师能再具体讲讲这个标志的持久化是什么意思吗？

展开 ∨

作者回复: 就是如果一个标志是“持久的”，那么子命令也可以使用父命令的标志。

