44｜一个程序多种功能：构建子命令与flags

2023-01-19 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

课程介绍 >

讲述：郑建勋

时长 06:25　大小 5.86M

你好，我是郑建勋。

之前，我们介绍了 Worker 的开发以及代码的测试，但之前的程序其实还是单机执行的。接下来让我们打开分布式开发的大门，一起看看如何开发 Master 服务，实现任务的调度与故障容错。

考虑到 Worker 和 Master 有许多可以共用的代码，并且关系紧密，所以我们可以将 Worker 与 Master 放到同一个代码仓库里。

Cobra 实现命令行工具

代码放置在同一个仓库后，我们遇到了一个新的问题。代码中只有一个 main 函数，该如何构建两个程序呢？其实，我们可以参考 Linux 中的一些命令行工具，或者 Go 这个二进制文件的

处理方式。例如，执行 go fmt 代表执行代码格式化程序，执行 go doc 代表执行文档注释程序。

在本项目中，我们使用 github.com/spf13/cobra 库提供的能力构建命令行应用程序。命令行应用程序通常接受各种输入作为参数，这些参数也被称为子命令，例如 go fmt 中的 fmt 和 go doc 中的 doc。同时，命令行应用程序也提供了一些选项或运行参数来控制程序的不同行为，这些选项通常被称为 flags。

Cobra 实例代码

怎么用 Cobra 来实现命令行工具呢？我们先来看一个简单的例子。在下面这个例子中，cmdPrint、cmdEcho、cmdTimes 表示我们将向程序加入的 3 个子命令。

复制代码

```go
package main

import (
  "fmt"
  "strings"

  "github.com/spf13/cobra"
)

func main() {
  var echoTimes int

  var cmdPrint = &cobra.Command{
    Use:   "c [string to print]",
    Short: "Print anything to the screen",
    Long: `print is for printing anything back to the screen.
For many years people have printed back to the screen.`,
    Args: cobra.MinimumNArgs(1),
    Run: func(cmd *cobra.Command, args []string) {
      fmt.Println("Print: " + strings.Join(args, " "))
    },
  }

  var cmdEcho = &cobra.Command{
    Use:   "echo [string to echo]",
    Short: "Echo anything to the screen",
    Long: `echo is for echoing anything back.
Echo works a lot like print, except it has a child command.`,
    Args: cobra.MinimumNArgs(1),
    Run: func(cmd *cobra.Command, args []string) {
      fmt.Println("Echo: " + strings.Join(args, " "))
    },
```

```
  }

  var cmdTimes = &cobra.Command{
    Use:   "times [string to echo]",
    Short: "Echo anything to the screen more times",
    Long: `echo things multiple times back to the user by providing
a count and a string.`,
    Args: cobra.MinimumNArgs(1),
    Run: func(cmd *cobra.Command, args []string) {
      for i := 0; i < echoTimes; i++ {
        fmt.Println("Echo: " + strings.Join(args, " "))
      }
    },
  }

  cmdTimes.Flags().IntVarP(&echoTimes, "times", "t", 1, "times to echo the inpu

  var rootCmd = &cobra.Command{Use: "app"}
  rootCmd.AddCommand(cmdPrint, cmdEcho)
  cmdEcho.AddCommand(cmdTimes)
  rootCmd.Execute()
}
```


以 cmdPrint 变量为例，它定义了一个子命令。cobra.Command 中的第一个字段 Use 定义了子命令的名字为 print；Short 和 Long 描述了子命令的使用方法；Args 为子命令需要传入的参数，在这里 `cobra.MinimumNArgs(1)` 表示至少需要传入一个参数；Run 为该子命令要执行的入口函数。

rootCmd 为程序的根命令，在这里命名为 app。AddCommand 方法会为命令添加子命令。 例如，rootCmd.AddCommand(cmdPrint, cmdEcho) 表示为根命令添加了两个子命令 cmdPrint 与 cmdEcho。而 cmdTimes 命令为 cmdEcho 的子命令。

接下来，我们执行上面的程序，会发现出现了一连串的文字。这是 Cobra 自动为我们生成的帮助文档，非常清晰。帮助文档中显示了我们当前程序有 3 个子命令 echo、help 与 print。


```
» go build app.go
» ./app -h
Usage:
  app [command]

Available Commands:
  echo        Echo anything to the screen
  help        Help about any command
```

```
  print       Print anything to the screen

Flags:
  -h, --help   help for app

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

```


接下来，我们输入子命令 echo，发现依然无法正确地执行并打印出新的帮助文档。帮助文档中提示，我们 echo 必须要传递一个启动参数。

复制代码

```
» ./app echo
Error: requires at least 1 arg(s), only received 0
Usage:
  app echo [string to echo] [flags]
  app echo [command]

Available Commands:
  times       Echo anything to the screen more times

Flags:
  -h, --help   help for echo

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

正确的执行方式如下。在这里，我们的 echo 子命令模拟了 Linux 中的 echo 指令，打印出了我们输入的信息。

复制代码

```
» ./app echo hello world
Echo: hello world
```

由于我们还为 echo 添加了一个子命令 times，因此我们可以方便地使用它。另外我们会看到子命令 times 绑定了一个 flags，名字是 times，缩写为 t。

复制代码

```
  cmdTimes.Flags().IntVarP(&echoTimes, "times", "t", 1, "times to echo the inpu
```

因此，我们可以用下面的方式执行 times 子命令，-t 这个 flag 则可以控制打印文本的次数。


```
» ./app echo times hello-world  -t=3
Echo: hello-world
Echo: hello-world
Echo: hello-world
```

接下来，让我们在项目中使用 Cobra。

在这里，我们遵循 Cobra 给出的组织代码的推荐目录结构。在最外层 main.go 的 main 函数中，只包含一个简单清晰的 cmd.Execute() 函数调用。实际的 Worker 与 Master 子命令则放置到了 cmd 包中。


```
package main

import (
  "github.com/dreamerjackson/crawler/cmd"
)

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

Worker 子命令

在 cmd.go 中，Execute 函数添加了 Worker、Master、Version 这三个子命令，他们都不需要添加运行参数。Worker 子命令最终会调用 worker.Run(), 和之前一样运行 GRPC 与 HTTP 服务。我们只是将之前 main.go 中的 Worker 代码迁移到了 cmd/worker 下。


```
// cmd.go
package cmd

import (
  "github.com/dreamerjackson/crawler/cmd/master"
  "github.com/dreamerjackson/crawler/cmd/worker"
  "github.com/dreamerjackson/crawler/version"
  "github.com/spf13/cobra"
)
```

```go
var workerCmd = &cobra.Command{
  Use:   "worker",
  Short: "run worker service.",
  Long:  "run worker service.",
  Args:  cobra.NoArgs,
  Run: func(cmd *cobra.Command, args []string) {
    worker.Run()
  },
}

var masterCmd = &cobra.Command{
  Use:   "master",
  Short: "run master service.",
  Long:  "run master service.",
  Args:  cobra.NoArgs,
  Run: func(cmd *cobra.Command, args []string) {
    master.Run()
  },
}

var versionCmd = &cobra.Command{
  Use:   "version",
  Short: "print version.",
  Long:  "print version.",
  Args:  cobra.NoArgs,
  Run: func(cmd *cobra.Command, args []string) {
    version.Printer()
  },
}

func Execute() {
  var rootCmd = &cobra.Command{Use: "crawler"}
  rootCmd.AddCommand(masterCmd, workerCmd, versionCmd)
  rootCmd.Execute()
}
```


接着运行 go run main.go worker，可以看到 Worker 程序已经正常地运行了。

复制代码

```
» go run main.go worker
{"level":"INFO","ts":"2022-12-10T18:07:20.615+0800","caller":"worker/worker.go:
{"level":"INFO","ts":"2022-12-10T18:07:20.615+0800","caller":"worker/worker.go:
{"level":"ERROR","ts":"2022-12-10T18:07:21.050+0800","caller":"engine/schedule.
{"level":"DEBUG","ts":"2022-12-10T18:07:21.050+0800","caller":"worker/worker.go
{"level":"DEBUG","ts":"2022-12-10T18:07:21.052+0800","caller":"worker/worker.go
2022-12-10 18:07:21  file=worker/worker.go:161 level=info Starting [service] go
2022-12-10 18:07:21  file=v4@v4.9.0/service.go:96 level=info Server [grpc] List
2022-12-10 18:07:21  file=grpc@v1.2.0/grpc.go:913 level=info Registry [etcd] Re
```

Master 子命令

我们再来看看怎么书写 Master 程序。cmd/master 包用于启动 Master 程序。和 Worker 代码非常类似，Master 也需要启动 GRPC 服务和 HTTP 服务，但是和 Worker 不同的是，Master 服务的配置文件参数需要做相应的修改。如下，我们增加了 Master 的服务配置。

复制代码

```
// config.toml
[MasterServer]
HTTPListenAddress = ":8081"
GRPCListenAddress = ":9091"
ID = "1"
RegistryAddress = ":2379"
RegisterTTL = 60
RegisterInterval = 15
ClientTimeOut   = 10
Name = "go.micro.server.master"
```

接着执行 go run main.go master，可以看到 Master 服务已经正常地运行了。

复制代码

```
» go run main.go master
{"level":"INFO","ts":"2022-12-10T18:03:21.986+0800","caller":"master/master.go:
hello master
{"level":"DEBUG","ts":"2022-12-10T18:03:21.986+0800","caller":"master/master.go
{"level":"DEBUG","ts":"2022-12-10T18:03:21.988+0800","caller":"master/master.go
2022-12-10 18:03:21  file=master/master.go:114 level=info Starting [service] go
2022-12-10 18:03:21  file=v4@v4.9.0/service.go:96 level=info Server [grpc] List
2022-12-10 18:03:21  file=grpc@v1.2.0/grpc.go:913 level=info Registry [etcd] Re
```

Version 子命令

接下来我们来看看 Version 子命令，该命令主要用于打印程序的版本号。我们将打印版本的功能从 main.go 迁移到 version/version.go 中。同时，我们在 Makefile 中构建程序时的编译时选项 `ldflags` 也需要进行一些调整。如下所示，我们将版本信息注入到了 version 包的全局变量中。

复制代码

```
// Makefile
```

```
LDFLAGS = -X "github.com/dreamerjackson/crawler/version.BuildTS=$(shell date -u
LDFLAGS += -X "github.com/dreamerjackson/crawler/version.GitHash=$(shell git re
LDFLAGS += -X "github.com/dreamerjackson/crawler/version.GitBranch=$(shell git
LDFLAGS += -X "github.com/dreamerjackson/crawler/version.Version=${VERSION}"

build:
  go build -ldflags '$(LDFLAGS)' $(BUILD_FLAGS) main.go
```


执行 make build 构建程序，然后运行./main version 即可打印出程序的详细版本信息。

复制代码

```
» make build
go build -ldflags '-X "github.com/dreamerjackson/crawler/version.BuildTS=2022-1
» ./main version
Version:           v1.0.0-c841af5
Git Branch:        HEAD
Git Commit:        c841af5deb497745d1ae39d3f565579344950777
Build Time (UTC):  2022-12-10 10:25:17
```

此外，运行./main -h 还可以看到 Cobra 自动生成的帮助文档。

复制代码

```
» ./main -h
Usage:
  crawler [command]

Available Commands:
  completion  Generate the autocompletion script for the specified shell
  help        Help about any command
  master      run master service.
  version     print version.
  worker      run worker service.

Flags:
  -h, --help   help for crawler

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

这节课我们先把框架搭建起来，后续我们还会具体实现 Master 的功能。这节课的代码我放在了v0.3.4分支，你可以打开链接查看。

flags 控制程序行为

刚才，我们都是将一些通用的配置写到配置文件中的。不过很快我们会发现一个问题，如果我们想在同一台机器上运行多个 Worker 或 Master 程序，就会发生端口冲突，导致程序异常退出。

复制代码

```
» go run main.go master
{"level":"INFO","ts":"2022-12-10T18:37:26.318+0800","caller":"master/master.go:
{"level":"DEBUG","ts":"2022-12-10T18:37:26.318+0800","caller":"master/master.go
{"level":"DEBUG","ts":"2022-12-10T18:37:26.320+0800","caller":"master/master.go
{"level":"FATAL","ts":"2022-12-10T18:37:26.320+0800","caller":"master/master.go
```

要解决这一问题，我们可以为不同的程序指定不同的配置文件，或者我们也可以先修改我们的配置文件再运行，但这些做法都非常繁琐。这时我们就可以借助 flags 来解决这类问题了。

如下所示，我们将 Master 的 ID、监听的 HTTP 地址与 GRPC 地址作为 flags，并将 flags 与子命令 master 绑定在一起。这时，我们可以手动传递运行程序时的 flags，并将 flags 的值设置到全局变量 masterID、HTTPListenAddress 与 GRPCListenAddress 中。这样，我们就能够比较方便地为不同的程序设置不同的运行参数了。

复制代码

```
var MasterCmd = &cobra.Command{
  Use:   "master",
  Short: "run master service.",
  Long:  "run master service.",
  Args:  cobra.NoArgs,
  Run: func(cmd *cobra.Command, args []string) {
    Run()
  },
}

func init() {
  MasterCmd.Flags().StringVar(
    &masterID, "id", "1", "set master id")
  MasterCmd.Flags().StringVar(
    &HTTPListenAddress, "http", ":8081", "set HTTP listen address")

  MasterCmd.Flags().StringVar(
    &GRPCListenAddress, "grpc", ":9091", "set GRPC listen address")
}

var masterID string
var HTTPListenAddress string
var GRPCListenAddress string
```

接下来，通过 flags 中，我们为不同的 Master 服务设置不同的 HTTP 监听地址与 GRPC 监听地址。

现在，我们就可以轻松地运行多个 Master 服务，不必担心端口冲突了。

```
// master 2
» ./main master --id=2 --http=:8081  --grpc=:9091
//master 3
» ./main master --id=3 --http=:8082  --grpc=:9092
```

总结

总结一下。这节课，为了灵活地运行不同的程序与功能，我们使用了 Cobra 包构建命令行程序。

Cobra 提供了推荐的项目组织结构，在 main 函数中有一个清晰的 cmd.Execute() 函数调用，并把相关子命令放置到了 cmd 包中。通过 Cobra，我们灵活地构建了子命令和 flags。子命令帮助我们将 Worker 与 Master 放置到了同一个仓库中，快速地搭建起了 Master 的框架。而 flags 帮助我们设置了程序不同的运行参数，避免了在本地的端口冲突。

下一节课，我们还将看到如何书写 Master 服务，完成服务的监听与选主。

课后题

学完这节课，给你留一道课后题。

你认为，应该在什么场景下使用子命令，什么场景下使用 flags，又在什么场景下使用环境变量呢？

欢迎你在留言区与我交流讨论，我们下节课见。

生成海报并分享

赞 2　提建议

上一篇　43丨分布式协调：etcd读写、MVCC原理与监听机制

下一篇　45丨Master高可用：怎样借助etcd实现服务选主？

精选留言 (2)

写留言

shuff1e

2023-01-19 来自北京

这是想到哪讲到哪么？课程大纲上44节不是讲微服务框架与协议的么？怎么又忽然来讲cobra？pflag？这种基础的工具放在前面讲会不会更好一些？

3

陈卧虫

2023-01-19 来自浙江

正好在写一个命令行工具，今天就用上了，但是遇到了一个问题，我需要实现交互式的，能多次用户输入，但是cobra好像只能在启动时指定参数，无法在运行中输入向Yes 或No这样的参数，有其它的方案吗（除了直接读取标准输入，我现在就这么做的）

1