



下载APP



## 13 | 交互：可以执行命令行的框架才是好框架

2021-10-15 叶剑峰

《手把手带你写一个Web框架》

课程介绍 &gt;

**讲述：叶剑峰**

时长 17:47 大小 16.30M



你好，我是轩脉刃。

上一节课，我们开始把框架向工业级迭代，重新规划了目录，这一节课将对框架做更大的改动，让框架支持命令行工具。

一个拥有命令行工具的框架会非常受欢迎。比如 Beego 框架提供了一个命令行工具 Bee、Vue 框架提供了 Vue-CLI，这些工具无疑给框架的使用者提供了不少便利。在使用框架过程中，命令行工具能将很多编码行为自动化。



而且退一步说，在实际工作中你会发现，即使用的框架没有提供任何命令行工具，在业务运营的过程中，我们也需要开发各种大大小小的命令行运营工具，作为业务运营的必要辅

助。所以一个自带命令行工具，且很方便让用户自行开发命令行的框架，是非常有必要的。

这节课我们就研究一下如何将 hade 框架改造成为支持命令行交互的框架。

## 第三方命令行工具库 cobra

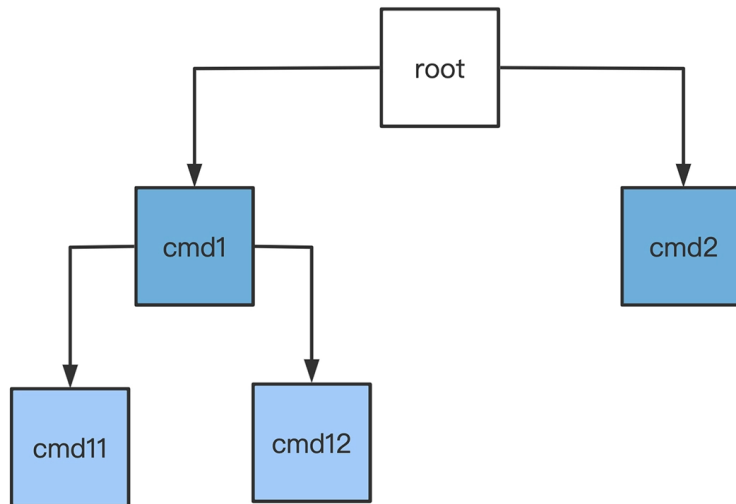
要让一个程序支持命令行，那么它的核心功能就是要能解析参数，比如 `./hade app start --address=:8888` 其中的 `./hade` 是我们要运行的程序，而后面的 `app` 和 `start` 两个字段以及 `--address=:8888` 就是这个程序对应的参数了。

那么如何解析参数呢？

Golang 标准库提供了 `flag` 包能对参数进行解析。但是 `flag` 包**只是一个命令行解析的类库，不支持组织，所以如果想基于 `flag` 包实现父子命令行工具，显然就不够了**。出于不重复造轮子，站在巨人肩膀上的想法，我们将视线移向开源社区一个最出名的命令行工具库 [cobra](#)。

`cobra` 是由大名鼎鼎的谷歌工程师 Steve Francia ( `spf13` ) 开发并开源的一个项目。Steve Francia 是 Golang 开源界比较出名的一名程序员，是 Golang、Doctor、MongoDB 的开源贡献者，同时开源的 `hugo`、`viper` 等项目应用也非常广泛。而由他开发开源的 `cobra` 目前在 GitHub 上已经有了 23k 的 star。

`cobra` 不仅仅能让我们快速构建一个命令行，它更大的优势是能更快地组织起有许多命令行工具，因为从根命令行工具开始，`cobra` 把所有的命令按照树形结构组织起来了。



要使用 cobra 就要从源码上了解这个库。按照第一节课说的，按照 **库函数 > 结构定义 > 结构函数** 的顺序读，你会发现，cobra 这个库最核心的内容是一个数据结构 [@Command](#)。

一个 Command 代表一个执行命令。这个 Command 包含很多可设置的字段，如何使用这个 Command，就取决于我们如何设置这些属性。下面是源码片段，我在注释中列出了这些属性的意义。

[📄 复制代码](#)

```
1 // Command代表执行命令的结构
2 type Command struct {
3     // 代表当前命令的，如何执行，root 最好和生成的命令工具名称一致
4     Use string
5
6     // 代表这个工具的别名，在 subCommand 中 useful，比如 root cmd1 和 root cmd_1 想
7     Aliases []string
8
9     // 由于不强制设置，用于输入错误的时候建议字段
10    SuggestFor []string
11
12    // 这个就是在 help 的时候一句话描述这个命令的功能
13    Short string
14
15    // 详细描述这个命令的功能
16    Long string
17
18    // 例子
```

```
19     Example string
20
21     // 需要验证的参数
22     ValidArgs []string
23
24     // 有多少个参数，这里放了一个验证函数，可以是 ExactArgs, MaximumNArgs 等，验证
25     Args PositionalArgs
26
27     // 参数别名
28     ArgAliases []string
29
30     // 自动生成的命令设置
31     BashCompletionFunction string
32
33     // 如果这个命令已经废弃了，那么就这里写上废弃信息
34     Deprecated string
35
36     // 如果这个命令要被隐藏，设置这个字段
37     Hidden bool
38
39     // Annotations are key/value pairs that can be used by applications to
40     // group commands.
41     Annotations map[string]string
42
43     // 这个命令的版本
44     Version string
45
46     // 是否要打印错误信息
47     SilenceErrors bool
48
49     // 是否要打印如何使用
50     SilenceUsage bool
51
52     // 是否有 flag，如果这个命令没有 flag，设置为 true，那么所有的命令后面的参数都会
53     DisableFlagParsing bool
54
55     // 是否打印自动生成字样：("Auto generated by spf13/cobra...")
56     DisableAutoGenTag bool
57
58     // 是否显示[flags]字样
59     DisableFlagsInUseLine bool
60
61     // 是否打印建议
62     DisableSuggestions bool
63
64     // 两个字符串的差距多少会进入 suggest
65     SuggestionsMinimumDistance int
66
67     // 是否使用 Traverse 的方式来解析参数
68     TraverseChildren bool
69
70     // 解析错误白名单，比如像未知参数
```

```
71     FParseErrWhitelist FParseErrWhitelist
72
73     // The *Run 函数运行顺序：
74     //     * PersistentPreRun()
75     //     * PreRun()
76     //     * Run()
77     //     * PostRun()
78     //     * PersistentPostRun()
79     // 会被继承的前置 Run
80     PersistentPreRun func(cmd *Command, args []string)
81
82     // 会被继承的前置 Run, 带 error
83     PersistentPreRunE func(cmd *Command, args []string) error
84
85     // 当前这个命令的前置 Run
86     PreRun func(cmd *Command, args []string)
87     // 当前这个命令的前置 Run, 带 Error
88     PreRunE func(cmd *Command, args []string) error
89     // zh: 实际跑的时候运行的函数
90     Run func(cmd *Command, args []string)
91     // zh: Run 执行错误了之后
92     RunE func(cmd *Command, args []string) error
93     // 后置运行
94     PostRun func(cmd *Command, args []string)
95     // 后置运行, 带 error
96     PostRunE func(cmd *Command, args []string) error
97     // 会被继承的后置运行
98     PersistentPostRun func(cmd *Command, args []string)
99     // 会被继承的后置运行, 带 error
100    PersistentPostRunE func(cmd *Command, args []string) error
101
102
103 }
```

这里属性非常多，你也不需要都记住是啥。来看一些常用属性，我们用一个设置好的输出结果图就能很好理解。

```
~/Documents/UGit/coredemo  geekbang/13  ./hade help foo

foo的长说明

Usage:
  hade foo [flags]
  hade foo [command]

Aliases:
  foo, fo, f

Examples:
foo命令的例子

Available Commands:
  foo1      foo1的简要说明

Flags:
  -h, --help  help for foo

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

它对应的代码如下，后面会解释每一行都是怎么实现的：

```
1 // InitFoo 初始化 Foo 命令
2 func InitFoo() *cobra.Command {
3     FooCommand.AddCommand(Foo1Command)
4     return FooCommand
5 }
6 // FooCommand 代表 Foo 命令
7 var FooCommand = &cobra.Command{
8     Use:      "foo",
9     Short:    "foo 的简要说明",
10    Long:     "foo 的长说明",
11    Aliases: []string{"fo", "f"},
12    Example:  "foo 命令的例子",
13    RunE: func(c *cobra.Command, args []string) error {
14        container := c.GetContainer()
15        log.Println(container)
16        return nil
17    },
18 }
```

[复制代码](#)

```
19 // Foo1Command 代表 Foo 命令的子命令 Foo1
20 var Foo1Command = &cobra.Command{
21     Use:      "foo1",
22     Short:    "foo1 的简要说明",
23     Long:     "foo1 的长说明",
24     Aliases: []string{"fo1", "f1"},
25     Example:  "foo1 命令的例子",
26     RunE: func(c *cobra.Command, args []string) error {
27         container := c.GetContainer()
28         log.Println(container)
29         return nil
30     },
31 }
```

对照代码和输出结果图，能看出 Command 中最常用的一些字段设置。

Use 代表这个命令的调用关键字，比如要调用 Foo1 命令，我们就要用 `./hade foo foo1`。Short 代表这个命令的简短说明，它会出现在上级命令的使用文档中。

Long 代表这个命令的长说明，它会出现在当前命令的使用文档中。

Aliases 是当前命令的别名，等同于 Use 字段；

Example 是当前命令的例子，也是显示在当前命令的使用文档中。

而 **RunE** 代表当前命令的真正执行函数：

 复制代码

```
1 RunE: func(c *cobra.Command, args []string) error
```

这个执行函数的参数有两个：一个是 `cobra.Command`，表示当前的这个命令；而第二个参数是 `args`，表示当前这个命令的参数，返回值是一个 `error`，代表命令的执行成功或者失败。

## 如何使用命令行 cobra

现在大致了解 cobra 这个库的使用方法和最核心的 Command 结构，就要想想接下来我们要用它来做些什么事情了。



**首先，要把 cobra 库引入到框架中。**由于希望后续能修改 Command 的数据，并且在后面的章节中会在 Command 结构中，继续加入一些字段来支持定时的命令行，所以和 Gin 框架的引入一样，我们采用源码引入的方式。

引入后要对 Command 结构进行修改。我们希望把服务容器嵌入到 Command 结构中，让 Command 在调用执行函数 RunE 时，能从参数中获取到服务容器，这样就能从服务容器中使用之前定义的 Make 系列方法获取出具体的服务实例了。

那服务容器嵌到哪里合适呢？因为刚才说，在 cobra 中 Command 结构是一个树形结构，所有的命令都是由一个根 Command 衍生来的。所以我们可以根 Command 中设置服务容器，让所有的子 Command 都可以根据 Root 方法来找到树的根 Command，最终找到服务容器。

不要忘记了，最终目的是完善 Web 框架，所以**之前存放在 main 函数中的启动 Web 服务的一些方法我们也要做修改**，让它们能通过一个命令启动。main 函数不再是启动一个 Web 服务了，而是启动一个 cobra 的命令。

也就是说，我们将 Web 服务的启动逻辑封装为一个 Command 命令，将这个 Command 挂载到跟 Command 中，然后根据参数获取到这个 Command 节点，执行这个节点中的 RunE 方法，就能启动 Web 服务了。

**但是在调用 Web 服务所在节点的 RunE 方法的时候，存在一个 Engine 结构的传递问题。**

在 main 函数中，我们使用 gin.New 创建了一个 Engine 结构，在业务中对这个 Engine 结构进行路由设置，这些都应该在业务代码中。而后，我们就进入了框架代码中，调用 Web 服务所在 Command 节点的 RunE 方法，在这个方法里进行初始化 http.Server，并且启动 Goroutine 进行监听：

 复制代码

```
1 func main() {  
2     // 创建engine结构  
3     core := gin.New()  
4     ...  
5  
6     hadeHttp.Routes(core)  
7 }
```



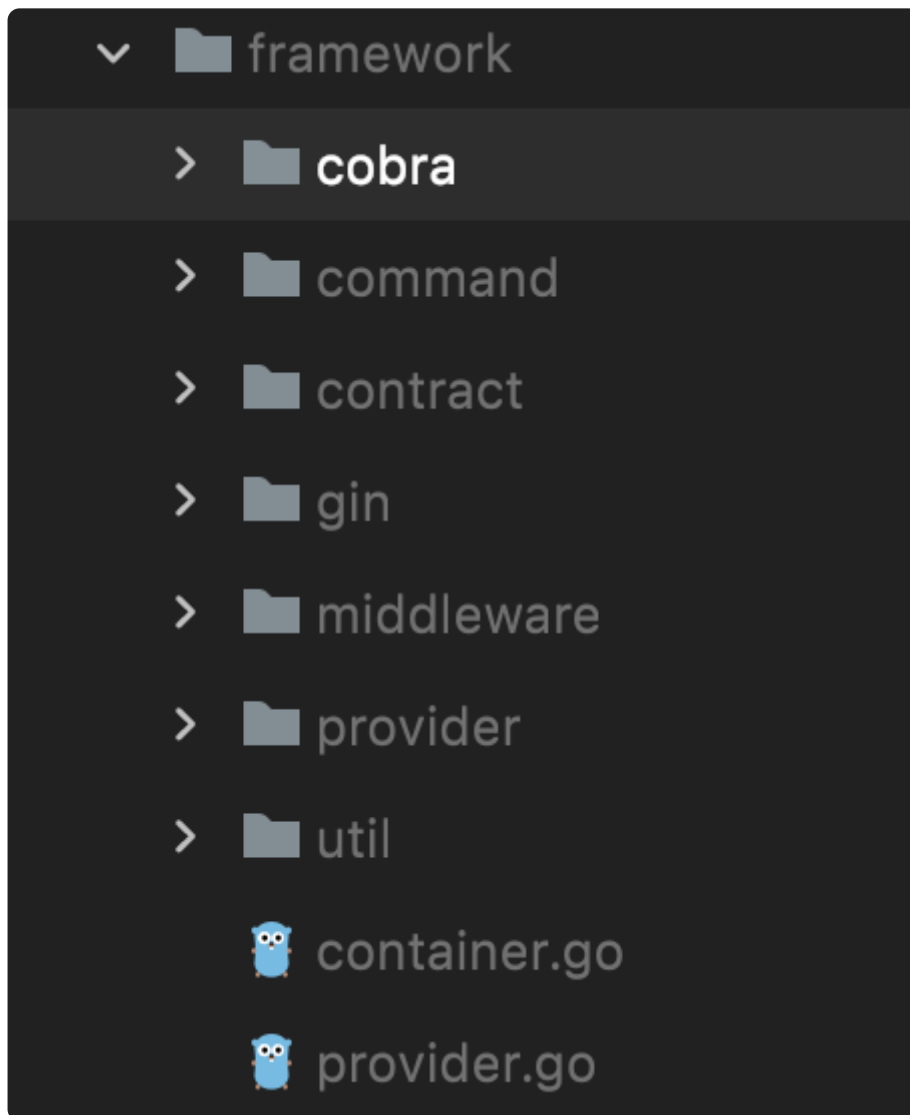
```
8     server := &http.Server{
9         Handler: core,
10        Addr:    ":8888",
11    }
12
13    // 这个goroutine是启动服务的goroutine
14    go func() {
15        server.ListenAndServe()
16    }()
17    ...
18 }
```

也就是说，我们只能根据 Command 拿到服务容器，那怎么拿到 Gin 函数创建的 Engine 结构呢？这个问题我提供一个解决思路，是否可以将“提供服务引擎”作为一个接口，通过服务提供者注入进服务容器？这样就能在命令行中就能获取服务容器了。

## 使用 cobra 增加框架的交互性

现在思路有了，可能发生的问题也想到了，下面进入实操。

首先是源码引入 cobra 库。引入的方式基本上和 Gin 框架引入的方式一样，先看下 cobra 源码的许可证，是 Apache License。这种许可证允许修改、商用、私有化等，只要求保留著作声明。所以我们直接拷贝最新的 cobra 源码，用 cobra v1.1.3 版本，将它放在 framework/cobra 目录下。




然后，对 Command 结构进行修改。要在 Command 结构中加入服务容器，由于刚才是源码引入的，很容易为 Command 增加一个 container 字段，在 framework/cobra/command.go 中修改 Command 结构：

```
1 type Command struct {  
2     // 服务容器  
3     container framework.Container  
4     ...  
5 }
```

[复制代码](#)

**再为 Command 提供两个方法：设置服务容器、获取服务容器。**设置服务容器的方法是为了在创建根 Command 之后，能将服务容器设置到里面去；而获取服务容器的方法，是为了在执行命令的 RunE 函数的时候，能从参数 Command 中获取到服务容器。

将定义的方法放在单独的一个文件 framework/cobra/hade\_command.go 中。

 复制代码

```
1 // SetContainer 设置服务容器
2 func (c *Command) SetContainer(container framework.Container) {
3     c.container = container
4 }
5 // GetContainer 获取容器
6 func (c *Command) GetContainer() framework.Container {
7     return c.Root().container
8 }
```

做到这里，前面两步 cobra 的引入和 Command 结构的修改就都完成了。

## 将 Web 启动改成一个命令

第三步，如何改造 Web 启动服务是最繁琐的，先简单梳理一下。

把创建 Web 服务引擎的方法作为一个服务封装在服务容器中，完成准备工作。

开始 main 函数的改造。首先要做的必然是初始化一个服务容器，然后将各个服务绑定到这个服务容器中，有一个就是刚才定义的提供 Web 引擎的服务。

在业务代码中将业务需要的路由绑定到 Web 引擎中去。

完成服务的绑定之后，最后要创建一个根 Command，并且创建一个 Web 启动的 Command，这两个 Command 会形成一个树形结构。

我们先要将创建 Web 服务引擎的方法作为一个服务封装在服务容器中，按照 [🔗 第十节课](#) 封装服务的三个步骤：封装接口协议、定义一个服务提供者、初始化服务实例。

在 framework/contract/kernel.go 中，把创建 Engine 的过程封装为一个服务接口协议：

 复制代码

```
1 // KernelKey 提供 kernel 服务凭证
2 const KernelKey = "hade:kernel"
3
4 // Kernel 接口提供框架最核心的结构
5 type Kernel interface {
6     // HttpEngine http.Handler结构，作为net/http框架使用，实际上是gin.Engine
7     HttpEngine() http.Handler
8 }
```

在定义的 Kernel 接口，提供了 HttpEngine 的方法，返回了 net/http 启动的时候需要的 http.Handler 接口，并且设置它在服务容器中的字符串凭证为"hade:kernel"。

然后为这个服务定义一个服务提供者。这个服务提供者可以在初始化服务的时候传递 Web 引擎，如果初始化的时候没有传递，则需要在启动的时候默认初始化。

在对应的 Kernel 的服务提供者代码 framework/provider/kernel/provider.go 中，我们实现了服务提供者需要实现的五个函数 Register、Boot、isDefer、Params、Name。

 复制代码

```
1 package kernel
2 import (
3     "github.com/gohade/hade/framework"
4     "github.com/gohade/hade/framework/contract"
5     "github.com/gohade/hade/framework/gin"
6 )
7
8 // HadeKernelProvider 提供web引擎
9 type HadeKernelProvider struct {
10     HttpEngine *gin.Engine
11 }
12
13 // Register 注册服务提供者
14 func (provider *HadeKernelProvider) Register(c framework.Container) framework.
15     return NewHadeKernelService
16 }
17
18 // Boot 启动的时候判断是否由外界注入了Engine，如果注入的化，用注入的，如果没有，重新实例化
19 func (provider *HadeKernelProvider) Boot(c framework.Container) error {
20     if provider.HttpEngine == nil {
21         provider.HttpEngine = gin.Default()
22     }
23     provider.HttpEngine.SetContainer(c)
24     return nil
25 }
26
27 // IsDefer 引擎的初始化我们希望开始就进行初始化
28 func (provider *HadeKernelProvider) IsDefer() bool {
29     return false
30 }
31
32 // Params 参数就是一个HttpEngine
33 func (provider *HadeKernelProvider) Params(c framework.Container) []interface{
34     return []interface{}{provider.HttpEngine}
35 }
```

```
36 // Name 提供凭证
37 func (provider *HadeKernelProvider) Name() string {
38     return contract.KernelKey
39 }
40 }
```

创建服务的第三步就是初始化实例了。这个服务实例比较简单，就是一个包含着 Web 引擎的服务结构。在刚才实现的 `HttpEngine()` 接口中，把服务结构中包含的 Web 引擎返回即可。

[复制代码](#)

```
1 // 引擎服务
2 type HadeKernelService struct {
3     engine *gin.Engine
4 }
5
6 // 初始化 web 引擎服务实例
7 func NewHadeKernelService(params ...interface{}) (interface{}, error) {
8     httpEngine := params[0].(*gin.Engine)
9     return &HadeKernelService{engine: httpEngine}, nil
10 }
11
12 // 返回 web 引擎
13 func (s *HadeKernelService) HttpEngine() http.Handler {
14     return s.engine
15 }
```

现在我们完成了 Web 服务 Kernel 的设计，转而我们改造一下入口函数。main 函数是我们的入口，但是现在，入口函数就不再是启动一个 HTTP 服务了，而是执行一个命令。那么这个 main 函数要做些什么呢？

整个框架目前都是围绕服务容器进行设计的了。所以在业务目录的 `main.go` 的 main 函数中，我们第一步要做的，必然是初始化一个服务容器。

[复制代码](#)

```
1 // 初始化服务容器
2 container := framework.NewHadeContainer()
```

接着，要将各个服务绑定到这个服务容器中。目前要绑定的服务容器有两个，一个是上一节课我们定义的目录结构服务 HadeAppProvider，第二个是这节课定义的提供 Web 引擎的服务。

[复制代码](#)

```
1 // 绑定 App 服务提供者
2 container.Bind(&app.HadeAppProvider{})
3
4 // 后续初始化需要绑定的服务提供者...
5 // 将 HTTP 引擎初始化,并且作为服务提供者绑定到服务容器中
6 if engine, err := http.NewHttpEngine(); err == nil {
7     container.Bind(&kernel.HadeKernelProvider{HttpEngine: engine})
8 }
```

http.NewHttpEngine 这个创建 Web 引擎的方法必须放在业务层，因为这个 Web 引擎不仅仅是调用了 Gin 创建 Web 引擎的方法，更重要的是调用了业务需要的绑定路由的功能。

将业务需要的路由绑定到 Web 引擎中去。因为这个是业务逻辑，我们放在业务目录的 app/kernel.go 文件中：

[复制代码](#)

```
1 // NewHttpEngine 创建了一个绑定了路由的 Web 引擎
2 func NewHttpEngine() (*gin.Engine, error) {
3     // 设置为 Release, 为的是默认在启动中不输出调试信息
4     gin.SetMode(gin.ReleaseMode)
5     // 默认启动一个 Web 引擎
6     r := gin.Default()
7     // 业务绑定路由操作
8     Routes(r)
9     // 返回绑定路由后的 Web 引擎
10    return r, nil
11 }
```

而对应的业务绑定路由操作，还是放在业务代码的 app/http/route.go 中：

[复制代码](#)

```
1
2 // Routes 绑定业务层路由
3 func Routes(r *gin.Engine) {
```

```
4     r.Static("/dist/", "./dist/")
5
6     demo.Register(r)
7 }
8
```

完成服务提供者的绑定和路由设置之后，**最后要创建一个根 Command，并且将业务的 Command 和框架定义的 Command 都加载到根 Command 中，形成一个树形结构。**

在 main 中，我们用 console.RunCommand 来创建和运行根 Command。

[复制代码](#)

```
1 // 运行 root 命令
2 console.RunCommand(container)
```

而这里 RunCommand 的方法简要来说做了三个事情：

1. 创建根 Command，并且将容器设置进根 Command 中。
2. 绑定框架和业务的 Command 命令。
3. 调用 Execute 启动命令结构。

具体的代码实现放在业务目录的 app/console/kernel.go 文件中，如下：

[复制代码](#)

```
1 // RunCommand 初始化根 Command 并运行
2 func RunCommand(container framework.Container) error {
3     // 根 Command
4     var rootCmd = &cobra.Command{
5         // 定义根命令的关键字
6         Use: "hade",
7         // 简短介绍
8         Short: "hade 命令",
9         // 根命令的详细介绍
10        Long: "hade 框架提供的命令行工具，使用这个命令行工具能很方便执行框架自带命令，也能很
11        // 根命令的执行函数
12        RunE: func(cmd *cobra.Command, args []string) error {
13            cmd.InitDefaultHelpFlag()
14            return cmd.Help()
15        },
16        // 不需要出现 cobra 默认的 completion 子命令
```



```
17     CompletionOptions: cobra.CompletionOptions{DisableDefaultCmd: true},
18 }
19 // 为根 Command 设置服务容器
20 rootCmd.SetContainer(container)
21 // 绑定框架的命令
22 command.AddKernelCommands(rootCmd)
23 // 绑定业务的命令
24 AddAppCommand(rootCmd)
25 // 执行 RootCommand
26 return rootCmd.Execute()
```

仔细看这段代码，我们这一节课前面说的内容都在这里得到了体现。

首先，根 Command 的各个属性设置是基于我们对 cobra 的 Command 结构比较熟悉才能进行的；而为根 Command 设置服务容器，我们用之前为服务容器扩展的 SetContainer 方法设置的；最后运行 cobra 的命令是调用 Execute 方法来实现的。

这里额外注意下，这里有**两个函数 AddKernelCommands 和 AddAppCommand**，**分别是将框架定义的命令和业务定义的命令挂载到根 Command 下。**


框架定义的命令我们使用 framework/command/kernel.go 中的 AddKernelCommands 进行挂载。而业务定义的命令我们使用 app/console/kernel.go 中的 AddAppCommand 进行挂载。比如下面要定义的启动服务的命令 appCommand 是所有业务通用的一个框架命令，最终会在 framework/command/kernel.go 的 AddKernelCommands 中进行挂载。

## 启动服务

现在已经将 main 函数改造成根据命令行参数定位 Command 树并执行，且在执行函数的参数 Command 中已经放入了服务容器，在服务容器中我们也已经注入了 Web 引擎。那么下面就来创建一个命令 ./hade app start 启动 Web 服务。

这个命令和业务无关，是框架自带的，所以它的实现应该放在 frame/command 下，而启动 Web 服务的命令是一个二级命令，其一级命令关键字为 app，二级命令关键字为 start。


那么我们先创建一级命令，这个一级命令 app 没有具体的功能，只是打印帮助信息。在 framework/command/app.go 中定义 appCommand：

 复制代码

```
1 // AppCommand 是命令行参数第一级为 app 的命令，它没有实际功能，只是打印帮助文档
2 var appCommand = &cobra.Command{
3     Use:     "app",
4     Short:   "业务应用控制命令",
5     RunE:    func(c *cobra.Command, args []string) error {
6         // 打印帮助文档
7         c.Help()
8         return nil
9     },
10 }
```


而二级命令关键字为 **start**，它是真正启动 Web 服务的命令。这个命令的启动执行函数有哪些逻辑呢？

首先，它需要获取 Web 引擎。具体方法根据前面讲的，要从参数 Command 中获取服务容器，从服务容器中获取引擎服务实例，从引擎服务实例中获取 Web 引擎：

 复制代码

```
1 // 从 Command 中获取服务容器
2 container := c.GetContainer()
3 // 从服务容器中获取 kernel 的服务实例
4 kernelService := container.MustMake(contract.KernelKey).(contract.Kernel)
5 // 从 kernel 服务实例中获取引擎
6 core := kernelService.HttpEngine()
```

获取到了 Web 引擎之后如何启动 Web 服务，就和第一节课描述的一样，通过创建 `http.Server`，并且调用其 `ListenAndServe` 方法。这里贴一下具体的 `appStartCommand` 命令的实现，供你参考思路，在 `framework/command/app.go` 中：

 复制代码

```
1 // appStartCommand 启动一个Web服务
2 var appStartCommand = &cobra.Command{
3     Use:     "start",
4     Short:   "启动一个Web服务",
5     RunE:    func(c *cobra.Command, args []string) error {
6         // 从Command中获取服务容器
7         container := c.GetContainer()
8         // 从服务容器中获取kernel的服务实例
9         kernelService := container.MustMake(contract.KernelKey).(contract.Kernel)
10        // 从kernel服务实例中获取引擎
11        core := kernelService.HttpEngine()
```

```
12
13 // 创建一个Server服务
14 server := &http.Server{
15     Handler: core,
16     Addr:    ":8888",
17 }
18
19 // 这个goroutine是启动服务的goroutine
20 go func() {
21     server.ListenAndServe()
22 }()
23
24 // 当前的goroutine等待信号量
25 quit := make(chan os.Signal)
26 // 监控信号：SIGINT, SIGTERM, SIGQUIT
27 signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM, syscall.SIGQUIT)
28 // 这里会阻塞当前goroutine等待信号
29 <-quit
30
31 // 调用Server.Shutdown graceful结束
32 timeoutCtx, cancel := context.WithTimeout(context.Background(), 5*time.S
33 defer cancel()
34
35 if err := server.Shutdown(timeoutCtx); err != nil {
36     log.Fatal("Server Shutdown:", err)
37 }
38
39 return nil
40 },
41 }
```

**最后将 RootCommand 和 AppCommand 进行关联。**在 framework/command/app.go 中定义 initAppCommand() 方法，将 appStartCommand 作为 appCommand 的子命令：

[复制代码](#)

```
1 // initAppCommand 初始化app命令和其子命令
2 func initAppCommand() *cobra.Command {
3     appCommand.AddCommand(appStartCommand)
4     return appCommand
5 }
```

在 framework/command/kernel.go 中，挂载对应的 appCommand 的命令：

```
1 // AddKernelCommands will add all command/* to root command
2 func AddKernelCommands(root *cobra.Command) {
3     // 挂载AppCommand命令
4     root.AddCommand(initAppCommand())
5 }
```

[复制代码](#)

我们就完成了 Web 启动的改造工作了。

## 验证

好了到这里，整个命令行工具就引入成功，并且 Web 框架也改造完成了。下面做一下验证。编译后调用./hade，我们获取到根 Command 命令行工具的帮助信息：

```
~/Documents/UGit/coredemo geekbang/13 ➤ ./hade
hade 框架提供的命令行工具，使用这个命令行工具能很方便执行框架自带命令，也能很方便编写业务命令

Usage:
  hade [flags]
  hade [command]

Available Commands:
  app      业务应用控制命令
  demo     demo for framework
  foo      foo的简要说明
  help     Help about any command

Flags:
  -h, --help  help for hade

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

提示可以通过一级关键字 app 获取下一级命令：

```
~/Documents/UGit/coredemo geekbang/13 ●+ ./hade app
业务应用控制命令，其包含业务启动，关闭，重启，查询等功能

Usage:
  hade app [flags]
  hade app [command]

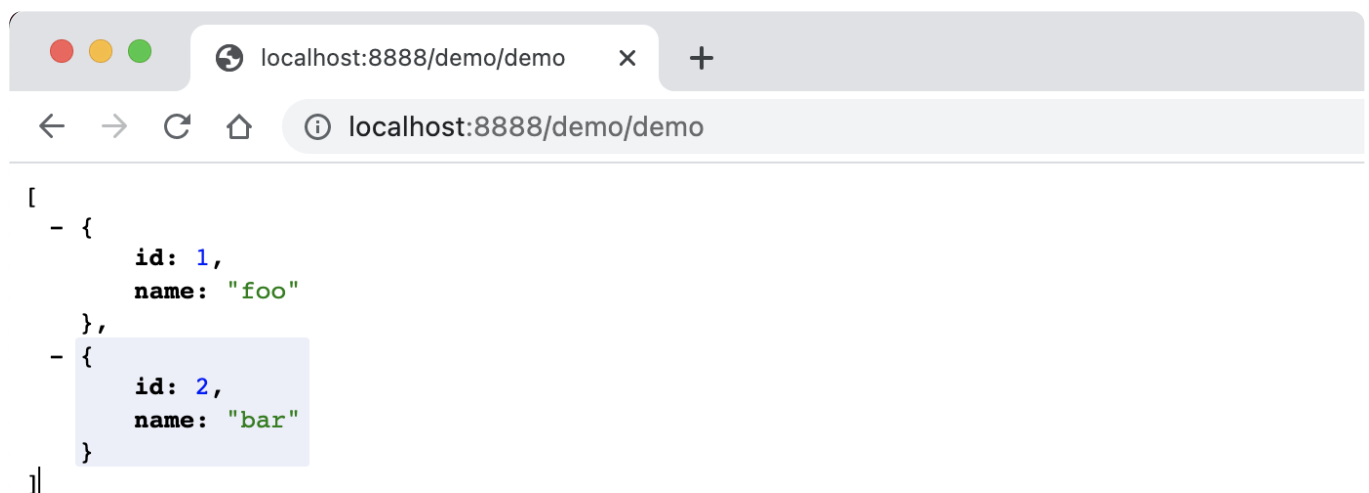
Available Commands:
  start      启动一个Web服务

Flags:
  -h, --help  help for app

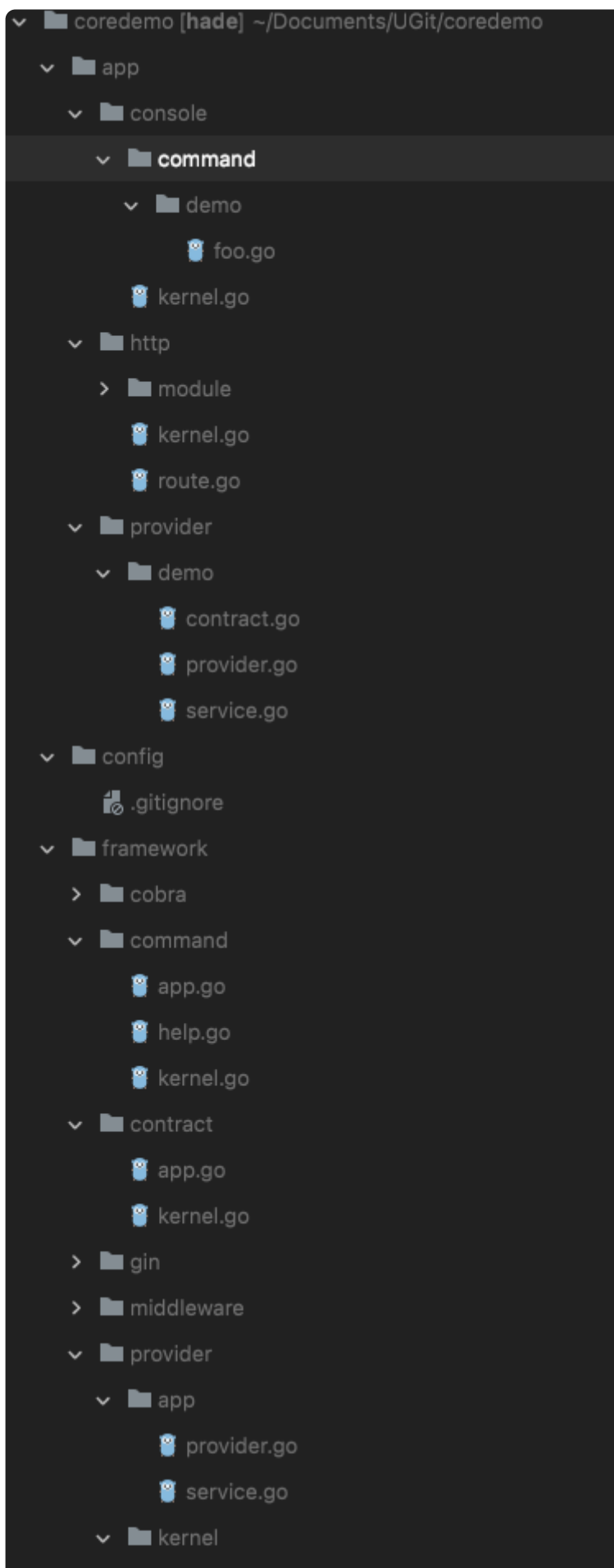
Use "hade app [command] --help" for more information about a command.
```

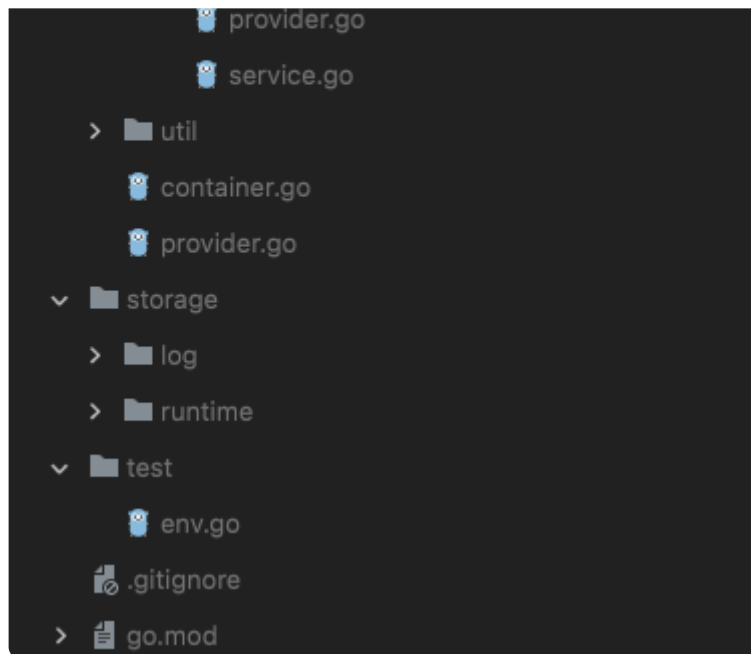
而./hade app 提醒我们可以通过二级关键字 start 来启动一个 Web 服务，调用 ./hade app start。

Web 服务启动成功，通过浏览器可以访问到业务定义的 /demo/demo 路径。



今天所有代码都存放在 GitHub 的 [@geekbang/13](#) 分支了，文中未展示的代码直接参考这个分支。本节课结束对应的目录结构如下：






## 总结

今天我们把之前的 Web 框架改造成了一个命令行工具，引入了 cobra 库，并且将原本的进程启动，也就是启动 Web 服务的方式，改成了调用一个命令来启动 Web 服务。

不知道你有没有感觉，将框架的入口改造成命令行，这个设计**不仅仅是简单换了一种 Web 服务的启动方式，而且是扩展了框架的另外一种可能性——设计命令行工具**。改造后，这个框架可以用来开发业务需要的各种命令行工具，同时也允许我们后续为框架增加多种多样易用性高的工具。

## 思考题

其实在之前的版本，我在 framework/contract/kernel.go 是这么设计 kernel 服务接口的：

 复制代码

```
1 package contract
2
3 const KernelKey = "hade:kernel"
4
5 // Kernel 接口提供框架最核心的结构
6 type Kernel interface {
7     // HttpEngine 提供gin的Engine结构
8     HttpEngine() *gin.Engine
9 }
```



在 provider/kernel/service.go 中是这么实现接口的：

[复制代码](#)

```
1 // 返回web引擎
2 func (s *HadeKernelService) HttpEngine() *gin.Engine {
3     return s.engine
4 }
```


和现在实现最大的不同是返回值。之前的返回值是返回了 \*gin.Engine。而现在的返回值是返回了 http.Handler，其他的实现没有任何变化。你能看出这样的改动相较之前有什么好处么？为什么这么改？

欢迎在留言区分享你的思考。感谢你的阅读，如果觉得有收获，也欢迎你把今天的内容分享给你身边的朋友，邀他一起学习。我们下节课见。

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 0

 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 12 | 结构：如何系统设计框架的整体目录？

# 技术管理案例课

踩坑复盘+案例分析+精进攻略=高效管理

许健

eBay 基础架构工程研发总监



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

## 精选留言 (1)

💬 写留言



qinsi

2021-10-16

似乎是想让app能够使用被集成到核心框架里的不同的web框架，而使用了不同web框架的app可以共享一些核心框架提供的服务，比如可以自定义命令行子命令生成不同类型项目的脚手架等。有些好奇这个方案有没有实际的使用场景，因为看上去不管是整合已有的web框架还是已有的app都存在一定的工程量，不确定整合带来的好处会大于这些工作量。

展开 ∨

