



下载APP

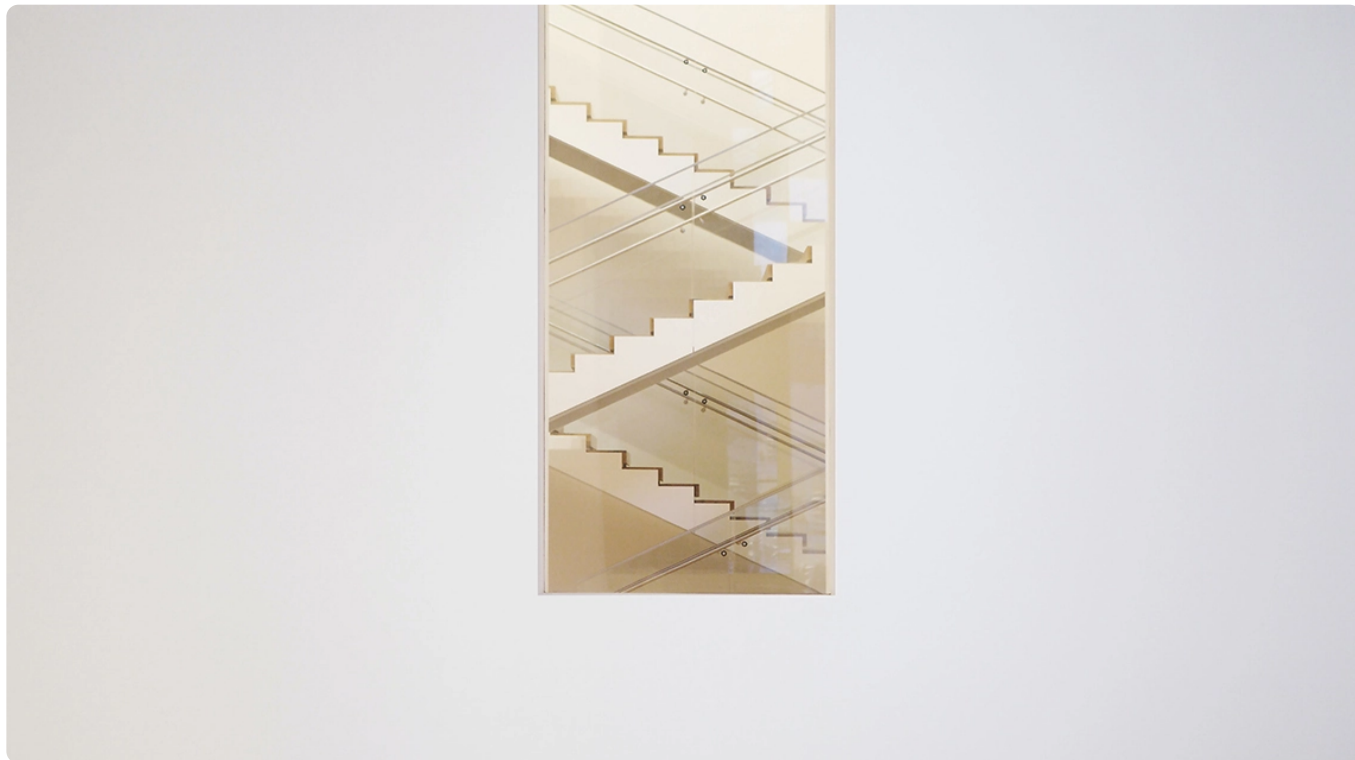


21 | 自动化：DRY，如何自动化一切重复性劳动？（上）

2021-11-03 叶剑峰

《手把手带你写一个Web框架》

课程介绍 >



讲述：叶剑峰

时长 18:07 大小 16.61M



你好，我是轩脉刃。

不知道你有没有听过这种说法，优秀程序员应该有三大大美德：懒惰、急躁和傲慢，这句话是 Perl 语言的发明者 Larry Wall 说的。其中懒惰这一点指的就是，程序员为了懒惰，不重复做同样的事情，会思考是否能把一切重复性的劳动自动化（don't repeat yourself）。

而框架开发到这里，我们也需要思考，有哪些重复性劳动可以自动化么？

领资料



从第十章到现在我们一直在说，框架核心是服务提供者，在开发具体应用时，一定会有很多需求要创建各种各样的服务，毕竟“一切皆服务”；而每次创建服务的时候，我们都需要至少编写三个文件，服务接口、服务提供者、服务实例。如果能自动生成三个文件，提供一个“自动化创建服务的工具”，应该能节省不少的操作。



说到创建工具，我们经常需要为了一个事情而创建一个命令行工具，而每次创建命令行工具，也都需要创建固定的 Command.go 文件，其中有固定的 Command 结构，这些代码我们能不能偷个懒，“**自动化创建命令行工具**”呢？

另外之前我们做过几次中间件的迁移，先将源码拷贝复制，再修改对应的 Gin 路径，这个操作也是颇为繁琐的。那么，我们是否可以写一个“**自动化中间件迁移工具**”，一个命令自动复制和替换呢？

这些命令都是可以实现的，这节课我们就来尝试完成这三项自动化，“自动化创建服务工具”，“自动化创建命令行工具”，以及“自动化中间件迁移工具”。

自动化创建服务工具

在创建各种各样的服务时，“自动化创建服务工具”能帮我们节省不少开发时间。我们先思考下这个工具应该如何实现。

既然之前已经引入 cobra，将框架修改为可以支持命令行工具，创建命令并不是一个难事，我们来定义一套创建服务的 provider 命令即可。照旧先设计好要创建的命令，再一一实现。

命令创建

“自动化创建服务工具”如何设计命令层级呢？我们设计一个一级命令和两个二级命令：

`./hade provider` 一级命令，provider，打印帮助信息；

`./hade provider new` 二级命令，创建一个服务；

`./hade provider list` 二级命令，列出容器内的所有服务，列出它们的字符串凭证。

首先将 provider 的这两个二级命令，都存放在 `command/provider.go` 中。而对应的一级命令 `providerCommand` 是一个打印帮助信息的空实现。

 复制代码

```
1 // providerCommand 一级命令
2 var providerCommand = &cobra.Command{
3
```

```

4   Use:   "provider",
5   Short: "服务提供相关命令",
6   RunE: func(c *cobra.Command, args []string) error {
7       if len(args) == 0 {
8           c.Help()
9       }
10      return nil
11  },

```

预先将两个二级命令挂载到这个一级命令中，在 framework/command/provider.go：

```

1 // 初始化provider相关服务
2 func initProviderCommand() *cobra.Command {
3     providerCommand.AddCommand(providerCreateCommand)
4     providerCommand.AddCommand(providerListCommand)
5     return providerCommand
6 }

```

[复制代码](#)

并且在 framework/command/kernel.go，将这个一级命令挂载到一级命令 rootCommand 中：

```

1 func AddKernelCommands(root *cobra.Command) {
2     // provider一级命令
3     root.AddCommand(initProviderCommand())
4 }

```

[复制代码](#)

下面来实现这两个二级命令 new 和 list。

List 命令的实现

先说 ./hade provider list 这个命令，因为列出容器内的所有服务是比较简单的。还记得吗，在十一章实现服务容器的时候，其中有一个 providers，它存储所有的服务容器提供者，放在文件 framework/container.go 中：

```

1 // HadeContainer 是服务容器的具体实现
2 type HadeContainer struct {


```

[复制代码](#)

```
3     ...
4     // providers 存储注册的服务提供者, key 为字符串凭证
5     providers map[string]ServiceProvider
6     ...
7 }
```

我们只需要将这个 providers 进行遍历, 根据其中每个 ServiceProvider 的 Name() 方法, 获取字符串凭证列表即可。

所以, 在 framework/container.go 的 HadeContainer 中, 增加一个 NameList 方法, 返回所有提供服务者的字符串凭证, 方法也很简单, 直接遍历这个 providers 字段。

 复制代码

```
1 // NameList 列出容器中所有服务提供者的字符串凭证
2 func (hade *HadeContainer) NameList() []string {
3     ret := []string{}
4     for _, provider := range hade.providers {
5         name := provider.Name()
6         ret = append(ret, name)
7     }
8     return ret
9 }
```

而在 framework/command/provider.go 中的 providerListCommand 命令中, 我们调用这个命令并且打印出来。

 复制代码

```
1 // providerListCommand 列出容器内的所有服务
2 var providerListCommand = &cobra.Command{
3     Use:     "list",
4     Short:   "列出容器内的所有服务",
5     RunE:    func(c *cobra.Command, args []string) error {
6         container := c.GetContainer()
7         hadeContainer := container.(*framework.HadeContainer)
8         // 获取字符串凭证
9         list := hadeContainer.NameList()
10        // 打印
11        for _, line := range list {
12            println(line)
13        }
14        return nil
15    },
16 }
```

可以验证一下。编译 `./hade build self` 并且执行 `./hade provider list`，可以看到如下信息：

```
~/Documents/UGit/coredemo  geekbang/21  ./hade provider list
hade:app
hade:env
hade:distributed
hade:config
hade:id
hade:trace
hade:log
hade:kernel
```

你可以很清晰看到容器中绑定了哪些服务提供者，它们的字符串凭证是什么。这样我们在定义一个新的服务的时候，可以很方便看到哪些服务提供者的关键字已经被使用了，避免使用已有的服务关键字。

下面我们来说稍微复杂一点的创建服务的命令 `./hade provider new`。

new 命令的实现

在实际业务开发过程中，我们一想到一个服务，比如去某个用户系统获取信息，一定会想到创建服务的三步骤：创建一个用户系统的交互协议 `contract.go`、再创建一个提供协议的用户服务提供者 `provider.go`、最后才实现具体的用户服务实例 `service.go`。

每次都需要创建这三个文件，且这三个文件的文件大框架都有套路可言。那我们如何将这些重复的套路性的代码自动化生成呢？


首先这里有一个增加参数的过程，我们需要知道要创建服务的服务名是什么？创建这个服务的文件夹名字是什么？当然了，这些参数也可以使用在命令后面增加 `flag` 参数的方式来表示。但是其实还有一种更便捷的方式：交互。

交互的表现形式如：

```
1 输入：./hade provider new (我想创建一个服务)
3 输出：请输入服务名称(服务凭证)：
4 输入：demo
5 输出：请求输入服务目录名称(默认和服务名称相同)：
6 输入：demo
7 输出：创建服务成功，文件夹地址：xxxxxx
    输出：请不要忘记挂载新创建的服务
```

这种命令行交互的方式是不是更智能化？但是如何实现呢？

这里我们借助一个第三方库 [survey](#)。这个库目前在 GitHub 上已经有 2.7k 个 star，最新版本是 v2 版本，使用的是 MIT License 协议，可以放心使用。这个 survey 库支持多种交互模式：单行输入、多行输入、单选、多选、y/n 确认选择，在 [项目 GitHub 首页](#)上就能很清晰看到这个库的使用方式。

 复制代码


```
1 name := false
2 // 使用survey.XXX 的方式来选择交互形式
3 prompt := &survey.Confirm{
4     Message: "Do you like pie?",
5 }
6 // 使用&将最终的选择存储进入变量
7 survey.AskOne(prompt, &name)
```

在 provider new 命令中，我们也可以用 survey 来增加交互性。通过交互，我们可以确认用户想创建的服务凭证，以及想把这个服务创建在 app/provider/ 下的哪个目录中。

当然，**在用户通过交互输入了服务凭证和服务目录之后，是需要进行参数判断的**。服务凭证需要和容器中已注册服务的字符串凭证进行比较，如果已经存在了，应该报错；而服务目录如果已经存在，也应该直接报错。

如果都验证 ok 了，最后一步就是在 app/provider/ 下创建对应的服务目录，在目录下创建 contract.go、provider.go、service.go 三个文件，并且在三个文件中根据预先定义好的模版填充内容。这里我们如何实现呢？使用模版、变更模版中的某些字段、形成新的文本，这个你应该能联想到 Golang 标准库中的 [text/template](#) 库。

这个库的使用方法比较多，我这里把我们用得到的方法解说一下，解析 contract.go 文件的生成过程，就可以了解其使用方法了。


 复制代码

```
1 // 创建title这个模版方法
2 func := template.FuncMap{"title": strings.Title}
3 {
4     // 创建contract.go
5     file := filepath.Join(pFolder, folder, "contract.go")
6     f, err := os.Create(file)
7     if err != nil {
8         return errors.Cause(err)
9     }
10    // 使用contractTmp模版来初始化template, 并且让这个模版支持title方法, 即支持{{.|titl
11    t := template.Must(template.New("contract").Funcs(funcs).Parse(contractTmp))
12    // 将name传递进入到template中渲染, 并且输出到contract.go 中
13    if err := t.Execute(f, name); err != nil {
14        return errors.Cause(err)
15    }
16 }
```

上面代码的逻辑最核心的就是创建模版的 `template.Must` 和渲染模版的 `t.Execute` 方法。

但是在创建模版之前, 我们使用了一个 **`template.FuncMap`** 方法, 它比较不好理解, 主要作用就是在模版中, 让我们可以使用定义的模版方法来控制渲染效果。这个 `FuncMap` 结构定义了模版中支持的模版方法, 比如我支持 `title` 这个方法, 这个方法实际调用的是 `string.Title` 函数, 把字符串首字母大写。

在刚才的代码中, 我们使用 `contractTmp` 来创建模版, 在渲染 `contractTmp` 的时候, 传递了一个 `name` 变量。假设这个 `name` 变量代表的是字符串 `user`, 而我希望创建一个字符串 `"NameKey"` 的变量, 可以这么定义 `contractTmp` :

 复制代码

```
1 var contractTmp string = `package {{.}}
2
3 const {{.|title}}Key = "{{.}}"
4
5 type Service interface {
6     // 请在这里定义你的方法
7     Foo() string
8 }
9 `
```


注意到了么，其中的`{{.|title}}` 实际上是相当于调用了 `strings.Title(name)` 的方法填充，能将字符串 `name` 替换为字符串 `Name`。

而定义好了 `FuncMap` 之后，我们随后使用了 `os.Create` 创建 `contract.go` 文件，然后初始化 `template`：

```
1 t := template.Must(template.New("contract").Funcs(funcs).Parse(contractTmp))
```

[复制代码](#)

这行代码的几个函数我们来看看。

template.Must 表示后面的 **template** 创建必须成功，否则会 **panic**。这种 `Must` 的方法来简化代码的 `error` 处理逻辑，在标准库中经常使用。我们的 `hade` 框架的 `MustMake` 也是同样的原理。

`template.New()` 方法，创建一个 `text/template` 的 `Template` 结构，其中的参数 `contract` 字符串是为这个 `Template` 结构命名的，后面的 `Funcs()` 方法是将签名定义的模版函数注册到这个 `Template` 结构中，最后的 `Parse()` 是使用这个 `Template` 结构解析具体的模版文本。

定义好了模版 `t` 之后，使用代码：

```
1 t.Execute(f, name)
```

[复制代码](#)

来将变量 `name` 注册进入模版 `t`，并且输出到 `f`。这里的 `f`，是我们之前创建的 `contract.go` 文件。也就是使用变量 `name` 解析模版 `t`，输出到 `contract.go` 文件中。

这里的变量可以是一个 `struct` 结构，也可以是基础变量，比如我们这里定义的字符串。在模版中`{{.}}` 就代表这个结构。所以再回顾前面定义的 `contractTmp` 模版，你会看出其中变量 `name` 为字符串 `user` 的时候，最终的显示是什么吗？

好, 创建服务命令的所有思路我们就梳理清楚了, 最后也贴出完整的代码供你参考, 关键步骤都在注释中详细说明了, 实现并不难:

 复制代码

```
1 // providerCreateCommand 创建一个新的服务, 包括服务提供者, 服务接口协议, 服务实例
2 var providerCreateCommand = &cobra.Command{
3     Use:      "new",
4     Aliases: []string{"create", "init"},
5     Short:    "创建一个服务",
6     RunE: func(c *cobra.Command, args []string) error {
7         container := c.GetContainer()
8         fmt.Println("创建一个服务")
9         var name string
10        var folder string
11        {
12            prompt := &survey.Input{
13                Message: "请输入服务名称(服务凭证): ",
14            }
15            err := survey.AskOne(prompt, &name)
16            if err != nil {
17                return err
18            }
19        }
20        {
21            prompt := &survey.Input{
22                Message: "请输入服务所在目录名称(默认: 同服务名称): ",
23            }
24            err := survey.AskOne(prompt, &folder)
25            if err != nil {
26                return err
27            }
28        }
29        // 检查服务是否存在
30        providers := container.(*framework.HadeContainer).NameList()
31        providerColl := collection.NewStrCollection(providers)
32        if providerColl.Contains(name) {
33            fmt.Println("服务名称已经存在")
34            return nil
35        }
36        if folder == "" {
37            folder = name
38        }
39        app := container.MustMake(contract.AppKey).(contract.App)
40        pFolder := app.ProviderFolder()
41        subFolders, err := util.SubDir(pFolder)
42        if err != nil {
43            return err
44        }
45        subColl := collection.NewStrCollection(subFolders)
46        if subColl.Contains(folder) {
```

```
47         fmt.Println("目录名称已经存在")
48         return nil
49     }
50     // 开始创建文件
51     if err := os.Mkdir(filepath.Join(pFolder, folder), 0700); err != nil {
52         return err
53     }
54     // 创建title这个模版方法
55     funcs := template.FuncMap{"title": strings.Title}
56     {
57         // 创建contract.go
58         file := filepath.Join(pFolder, folder, "contract.go")
59         f, err := os.Create(file)
60         if err != nil {
61             return errors.Cause(err)
62         }
63         // 使用contractTmp模版来初始化template, 并且让这个模版支持title方法, 即支持{{
64         t := template.Must(template.New("contract").Funcs(funcs).Parse(contra
65         // 将name传递进入到template中渲染, 并且输出到contract.go 中
66         if err := t.Execute(f, name); err != nil {
67             return errors.Cause(err)
68         }
69     }
70     {
71         // 创建provider.go
72         file := filepath.Join(pFolder, folder, "provider.go")
73         f, err := os.Create(file)
74         if err != nil {
75             return err
76         }
77         t := template.Must(template.New("provider").Funcs(funcs).Parse(provid
78         if err := t.Execute(f, name); err != nil {
79             return err
80         }
81     }
82     {
83         // 创建service.go
84         file := filepath.Join(pFolder, folder, "service.go")
85         f, err := os.Create(file)
86         if err != nil {
87             return err
88         }
89         t := template.Must(template.New("service").Funcs(funcs).Parse(service
90         if err := t.Execute(f, name); err != nil {
91             return err
92         }
93     }
94     fmt.Println("创建服务成功, 文件夹地址:", filepath.Join(pFolder, folder))
95     fmt.Println("请不要忘记挂载新创建的服务")
96     return nil
97 },
98 }
```

```
99 var contractTmp string = `package {{.}}
100 const {{.|title}}Key = "{{.}}"
101 type Service interface {
102     // 请在这里定义你的方法
103     Foo() string
104 }
105 `
106 var providerTmp string = `package {{.}}
107 import (
108     "github.com/gohade/hade/framework"
109 )
110 type {{.|title}}Provider struct {
111     framework.ServiceProvider
112     c framework.Container
113 }
114 func (sp *{{.|title}}Provider) Name() string {
115     return {{.|title}}Key
116 }
117 func (sp *{{.|title}}Provider) Register(c framework.Container) framework.NewIn
118     return New{{.|title}}Service
119 }
120 func (sp *{{.|title}}Provider) IsDefer() bool {
121     return false
122 }
123 func (sp *{{.|title}}Provider) Params(c framework.Container) []interface{} {
124     return []interface{}{c}
125 }
126 func (sp *{{.|title}}Provider) Boot(c framework.Container) error {
127     return nil
128 }
129 `
130 var serviceTmp string = `package {{.}}
131 import "github.com/gohade/hade/framework"
132 type {{.|title}}Service struct {
133     container framework.Container
134 }
135 func New{{.|title}}Service(params ...interface{}) (interface{}, error) {
136     container := params[0].(framework.Container)
137     return &{{.|title}}Service{container: container}, nil
138 }
139 func (s *{{.|title}}Service) Foo() string {
140     return ""
141 }
142 `
143
```

最后我们验证一下这个创建服务命令。同样编译./hade 命令之后, 执行 ./hade provider new, 定义服务凭证为 user, 目录名称同样为 user。

```
~/Documents/UGit/coredemo geekbang/21 ➤ ./hade build self
编译hade成功
~/Documents/UGit/coredemo geekbang/21 ➤ ./hade provider new
创建一个服务
? 请输入服务名称(服务凭证): user
? 请输入服务所在目录名称(默认: 同服务名称):
创建服务成功, 文件夹地址: /Users/yejianfeng/Documents/UGit/coredemo/app/provider/user
请不要忘记挂载新创建的服务
```

能看到 app/provider/ 目录下创建了 user 文件夹, 其中有 contract.go、provider.go、service.go 三个文件:

```
▼ coredemo [hade] ~/Documents/UGit/coredemo
  ▼ app
    > console
    > http
    ▼ provider
      > demo
      ▼ user
        🐙 contract.go
        🐙 provider.go
        🐙 service.go
```

其中每个文件的定义都完整, 且可以直接再次编译通过, 验证完成!

自动化创建命令行工具

到这里我们就完成了创建服务工具的自动化。开头提到具体运营一个应用的时候, 我们也会经常需要创建一个自定义的命令行。比如运营一个网站, 可能会创建一个命令来统计网站注册人数, 也可能要创建一个命令来定期检查是否有违禁的文章需要封禁等。所以自动创建命令行工具在实际工作中是非常有必要的。

同服务命令一样, 我们可以有一套创建命令行工具的命令。

`./hade command` 一级命令, 显示帮助信息

`./hade command list` 二级命令, 列出所有控制台命令

`./hade command new` 二级命令, 创建一个控制台命令


`command` 相关的命令和 `provider` 的命令的实现基本是一致的。这里我们简要解说下重点, 具体对应的代码详情可以参考 GitHub 上的 [framework/command/cmd.go](#) 文件。

一级命令 `./hade command` 我们就不说了, 是简单地显示帮助信息。

二级命令 `./hade command list`。功能是列出所有的控制台命令。这个功能实际上和直接调用 `./hade` 显示的帮助信息差不多, 把一级根命令全部列了出来, 只不过我们使用了一个更为语义化的 `./hade command list` 来显示。

```
~/Documents/UGit/coredemo ➤ geekbang/21 ➤ ./hade command list
app          业务应用控制命令
build        编译相关命令
command      控制台命令相关
config       获取配置相关信息
cron         定时任务相关命令
dev          调试模式
env          获取当前的App环境
foo          foo
go           运行path/go程序, 要求go 必须安装
help         Help about any command
middleware   中间件相关命令
new          创建一个新的应用
npm          运行 PATH/npm 的命令
provider     服务提供相关命令
```

它的实现也并不复杂, 具体就是使用 `Root().Commands()` 方法遍历一级跟命令的所有一级命令。

 复制代码

```
1 // cmdListCommand 列出所有的控制台命令
2 var cmdListCommand = &cobra.Command{
```

```
3   Use:   "list",
4   Short: "列出所有控制台命令",
5   RunE: func(c *cobra.Command, args []string) error {
6       cmds := c.Root().Commands()
7       ps := [][]string{}
8       for _, cmd := range cmds {
9           line := []string{cmd.Name(), cmd.Short}
10          ps = append(ps, line)
11      }
12      util.PrettyPrint(ps)
13      return nil
14  },
15 }
```

二级命令 `./hade command new` 创建命令行工具，就是在 `app/console/command/` 文件夹下增加一个目录，然后在这个目录中存放命令的相关代码。

比如要创建一个 `foo` 命令，就是要在 `app/console/command/` 目录下创建一个 `foo` 目录，其中创建一个 `foo.go` 文件名，这个文件名可以随意起，这里我们就和目录名保持一致。然后在 `app/console/command/foo.go` 文件中输入模版：

[复制代码](#)

```
1 // 命令行工具模版
2 var cmdTpl string = `package {{.}}
3
4 import (
5     "fmt"
6
7     "github.com/gohade/hade/framework/cobra"
8 )
9
10 var {{.|title}}Command = &cobra.Command{
11     Use:   "{{.}}",
12     Short: "{{.}}",
13     RunE: func(c *cobra.Command, args []string) error {
14         container := c.GetContainer()
15         fmt.Println(container)
16         return nil
17     },
18 }
```

实现步骤也很简单：survery 交互先要求用户输入命令名称；然后要求用户输入文件夹名称，记得检查命令名称和文件夹名称是否合理；之后创建文件夹

app/console/command/xxx 和文件 app/console/command/xxx/xxx.go；最后使用 template 将模版写入文件中。

自动化中间件迁移工具

除了服务工具和命令行工具的创建，对于中间件，我们在开发过程中也是经常会使用创建的，同样的，可以为中间件定义一系列的命令来自动化。

```
./hade middleware 一级命令，显示帮助信息  
./hade middleware list 二级命令，列出所有的业务中间件  
./hade middleware new 二级命令，创建一个新的业务中间件  
./hade middleware migrate 二级命令，迁移 Gin 已有的中间件
```


其中的前面三个命令基本上和 provider、command 命令如出一辙，我们就不赘述了，同样你可以通过 GitHub 上的 [framework/command/middleware.go](#) 文件参考其具体实现，相信你可以顺利写出来。

这里重点说一下 ./hade middleware migrate 命令。

不知道你有没有好奇，为什么迁移也要写一个命令？当时在将 Gin 迁移进入 hade 框架的时候我们说，Gin 作为一个成熟的开源作品，有丰富的中间件库，存放 GitHub 的一个项目 [gin-contrib](#) 中。那么在开发过程中，我们一定会经常需要使用到这些中间件。


但是由于这些中间件使用到的 Gin 框架的地址为：

```
1 github.com/gin-gonic/gin
```

 复制代码

而我们的 Gin 框架地址为：

```
1 github.com/gohade/hade/framework/gin
```

 复制代码


所以我们不能使用 import 直接使用这些中间件, 那么有没有一个办法, 能直接一键迁移 gin-contrib 下的某个中间件呢? 比如 `git@github.com:gin-contrib/cors.git`, 直接拷贝并且自动修改好 Gin 框架引用地址, 放到我们的 `app/http/middleware/` 目录中。

于是就有了这个 `./hade middleware migrate` 命令。下面就梳理一下这个命令的逻辑步骤。以下载 cors 中间件为例, 我们的思路是从 GitHub 上将这个 [cors 项目](#) 复制下来, 并且删除这个项目的一些不必要的文件。

什么是不必要的文件呢? `.git` 目录、`go.mod`、`go.sum`, 这些都是作为一个“项目”才会需要的, 而我们要把项目中的这些删掉, 让它成为一个文件, 存放在我们的 `app/http/middleware/cors` 目录下。最后再遍历这个目录的所有文件, 将所有出现 `"github.com/gin-gonic/gin"` 的地方替换为 `"github.com/gohade/hade/framework/gin"` 就可以了。

从 git 上复制一个项目, 在 Golang 中可以使用一个第三方库 [go-git](#), 这个第三方库已经有 2.7k 个 star, 且基于 Apache 的 Licence, 是可以直接 import 使用的。目前这个库最新的版本为 v5。

它的使用方式如下:

 复制代码

```
1 _, err := git.PlainClone("/tmp/foo", false, &git.CloneOptions{
2     URL:      "https://github.com/go-git/go-git",
3     Progress: os.Stdout,
4 })
```


将某个 Git 的 URL 地址使用 `gitclone`, 下载到 `/tmp/foo` 目录, 并且把输出也输出到控制台。

我们也可以使用这样的方式进行复制。具体的代码逻辑也不难, 归纳一下, `migrate` 的实现步骤如下:

1. 参数中获取中间件名称;

2. 使用 go-git , 将对应的 gin-contrib 的项目 clone 到目录 /app/http/middleware ;
3. 删除不必要的文件 go.mod、go.sum、.git ;
4. 替换关键字 "github.com/gin-gonic/gin" 。

在 framework/command/middleware.go 中 , 对应的代码如下 :

 复制代码

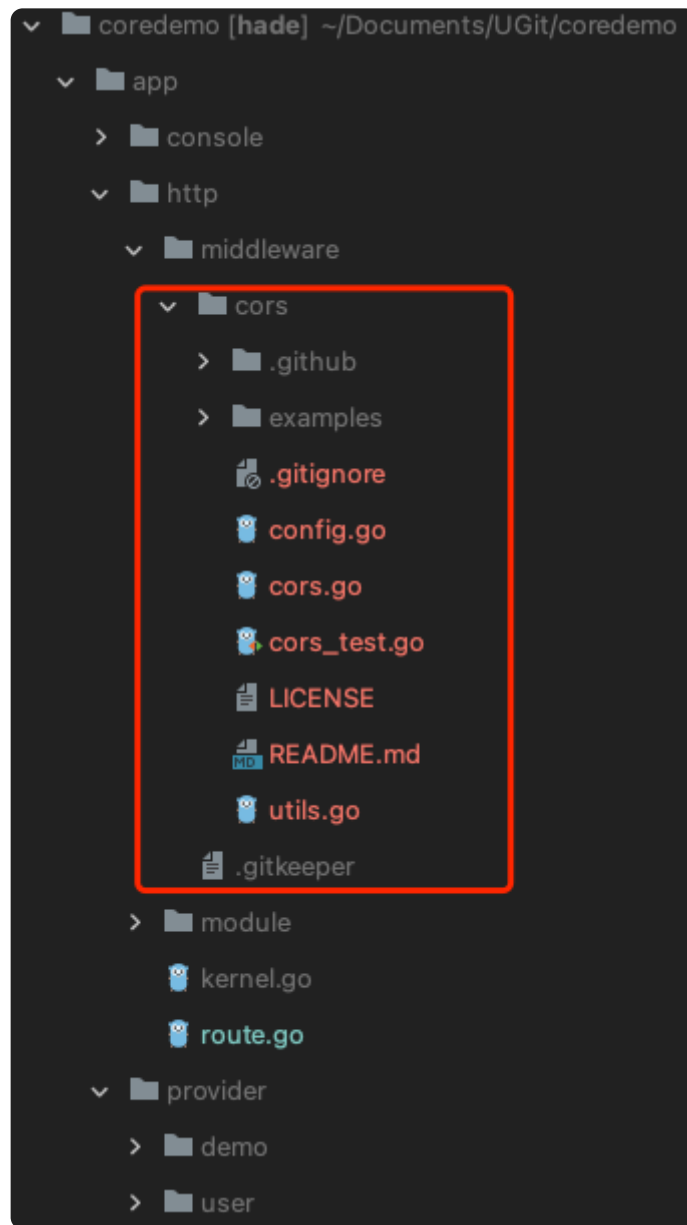
```
1 // 从gin-contrib中迁移中间件
2 var middlewareMigrateCommand = &cobra.Command{
3     Use:     "migrate",
4     Short:   "迁移gin-contrib中间件, 迁移地址: https://github.com/gin-contrib/[middlew",
5     RunE: func(c *cobra.Command, args []string) error {
6         container := c.GetContainer()
7         fmt.Println("迁移一个Gin中间件")
8         // step1: 获取参数
9         var repo string
10        {
11            prompt := &survey.Input{
12                Message: "请输入中间件名称: ",
13            }
14            err := survey.AskOne(prompt, &repo)
15            if err != nil {
16                return err
17            }
18        }
19        // step2 : 下载git到一个目录中
20        appService := container.MustMake(contract.AppKey).(contract.App)
21
22        middlewarePath := appService.MiddlewareFolder()
23        url := "https://github.com/gin-contrib/" + repo + ".git"
24        fmt.Println("下载中间件 gin-contrib:")
25        fmt.Println(url)
26        _, err := git.PlainClone(path.Join(middlewarePath, repo), false, &git.Cl
27            URL:      url,
28            Progress: os.Stdout,
29        })
30        if err != nil {
31            return err
32        }
33
34        // step3:删除不必要的文件 go.mod, go.sum, .git
35        repoFolder := path.Join(middlewarePath, repo)
36        fmt.Println("remove " + path.Join(repoFolder, "go.mod"))
37        os.Remove(path.Join(repoFolder, "go.mod"))
38        fmt.Println("remove " + path.Join(repoFolder, "go.sum"))
39        os.Remove(path.Join(repoFolder, "go.sum"))
40        fmt.Println("remove " + path.Join(repoFolder, ".git"))
```

```
41     os.RemoveAll(path.Join(repoFolder, ".git"))
42
43     // step4 : 替换关键词
44     filepath.Walk(repoFolder, func(path string, info os.FileInfo, err error)
45         if info.IsDir() {
46             return nil
47         }
48
49         if filepath.Ext(path) != ".go" {
50             return nil
51         }
52
53         c, err := ioutil.ReadFile(path)
54         if err != nil {
55             return err
56         }
57         isContain := bytes.Contains(c, []byte("github.com/gin-gonic/gin"))
58         if isContain {
59             fmt.Println("更新文件:" + path)
60             c = bytes.ReplaceAll(c, []byte("github.com/gin-gonic/gin"), []byte
61             err = ioutil.WriteFile(path, c, 0644)
62             if err != nil {
63                 return err
64             }
65         }
66
67         return nil
68     })
69     return nil
70 },
71 }
```


我们可以下载 cors 项目做一下验证, 运行 `./hade middleware migrate` 命令, 并且输入 cors。你会在控制台看到这些信息:

```
~/Documents/UGit/coredemo > geekbang/21 ●+ ./hade middleware migrate
迁移一个Gin中间件
? 请输入中间件名称: cors
下载中间件 gin-contrib:
https://github.com/gin-contrib/cors.git
Enumerating objects: 319, done.
Counting objects: 100% (36/36), done.
Compressing objects: 100% (25/25), done.
Total 319 (delta 12), reused 22 (delta 7), pack-reused 283
remove /Users/yejianfeng/Documents/UGit/coredemo/app/http/middleware/cors/go.mod
remove /Users/yejianfeng/Documents/UGit/coredemo/app/http/middleware/cors/go.sum
remove /Users/yejianfeng/Documents/UGit/coredemo/app/http/middleware/cors/.git
更新文件:/Users/yejianfeng/Documents/UGit/coredemo/app/http/middleware/cors/config.go
更新文件:/Users/yejianfeng/Documents/UGit/coredemo/app/http/middleware/cors/cors.go
更新文件:/Users/yejianfeng/Documents/UGit/coredemo/app/http/middleware/cors/cors_test.go
更新文件:/Users/yejianfeng/Documents/UGit/coredemo/app/http/middleware/cors/examples/example.go
```

并且在目录中看到 cors 中间件已经完整下载下来了。



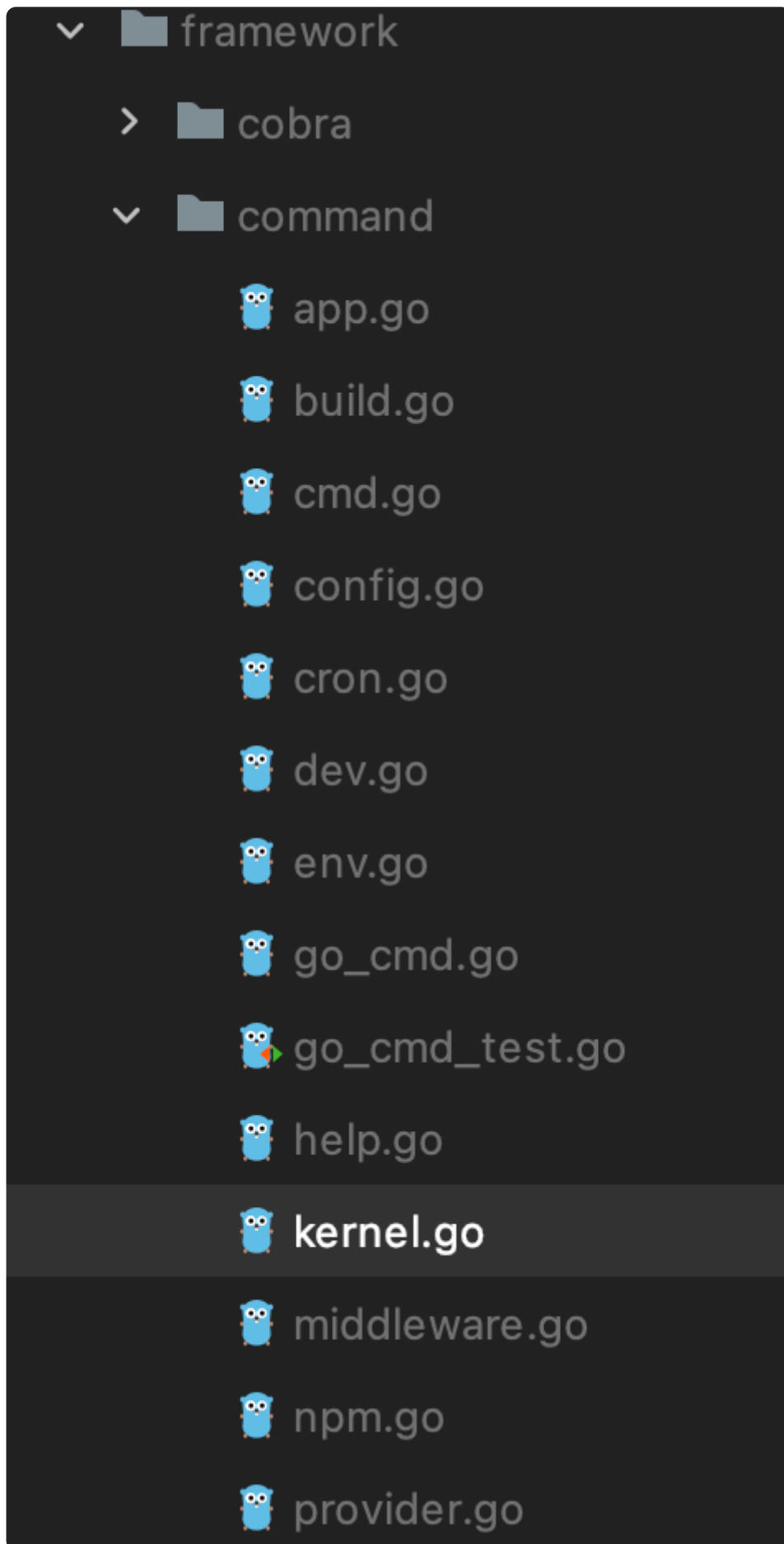
然后，可以直接在 `app/http/route.go` 中直接使用这个 `cors` 中间件：

 复制代码

```
1 ...
2
3 // Routes 绑定业务层路由
4 func Routes(r *gin.Engine) {
5     ...
6     // 使用cors中间件
7     r.Use(cors.Default())
8     ...
9 }
```

验证完成！

今天所有代码都保存在 GitHub 上的 [@geekbang/21](#) 分支了。附上目录结构供你对比查看，只修改了 framework/command/ 目录下的 cmd.go、provider.go、middleware.go 文件。



小结

今天增加的命令不少，自动化创建服务工具、命令行工具，以及中间件迁移工具，这些命令都为我们后续开发应用提供了不少便利。

其实每个自动化命令行工具实现的思路都是差不多的，先思考清楚对于这个工具我们要自动化生成什么，然后使用代码和对应的模版生成对应的文件，并且替换其中特有的单词。原理不复杂，但是对于实际的工作，是非常有帮助的。

这一节课你应该可以感受到之前将 cobra 引入我们的框架是一个多么正确的决定，在 cobra 之上，我们才能实现这些方便的自动化工具。

思考题

我们实现的自动化服务 ./hade command list 命令，目前只展示了一级命令，在写这篇文章的时候我反思了一下，其实可以扩展成为树形结构展示，同时展示一级 / 二级 / 三级 / 命令。你可以想想如何实现，如果可以的话，可以去 github.com/gohade/hade 项目中提交一个 merge request 来补充这个功能吧！

欢迎在留言区分享你的思考。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 20 | 提效：实现调试模式加速开发效率（下）

下一篇 22 | 自动化：DRY，如何自动化一切重复性劳动？（下）

11.11 全年底价

VIP 年卡限定 3 折

畅学 200 门课程 & 新课上线即解锁

超值拿下 ¥999


极客时间 VIP 年卡

365 天畅学

11.11 超值福

精选留言 (2)

写留言



kngxue


2021-11-05

课程设计的好棒

展开

...

👍



qinsi

2021-11-05

中间件也要拷贝代码吗...是否可以安装以后在go.mod里replace呢？

...

👍