



下载APP



16 | 配置和环境：配置服务中的设计思路(下)

2021-10-22 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 16:41 大小 15.29M



你好，我是轩脉刃。

上一节课，我们已经定义好了配置文件服务的接口，这节课就来实现这些接口。先来规划配置文件服务目录，按照上一节课分析的，多个配置文件按类别放在不同配置文件夹中，在框架文件夹中，我们将配置文件接口代码写在框架文件夹下的 `contract/config.go` 文件中，将具体实现放在 `provider/config/` 目录中。

配置服务的设计



不过设计优于实现，动手之前我们先思考下实现这个接口要如何设计。

首先，要读取一下配置文件夹中的文件。上节课说了，最终的配置文件夹地址为，应用服务的 ConfigFolder 下的环境变量对应的文件夹，比如 ConfigFolder/development。但是还有一个问题，就是配置文件的格式的选择。

目前市面上的配置文件格式非常多，但是很难说哪种配置文件比较好，完全是不同平台、不同时代下的产物。比如 Windows 开发的配置常用 INI、Java 开发配置常用 Properties，我这里选择了使用 YAML 格式。

配置文件的读取

YAML 格式是在 Golang 的项目中比较通用的一种格式，比如 Kubernetes、Docker、Swagger 等项目，都是使用 YAML 作为其配置文件的。YAML 配置文件除了能表达基础类型比如 string、int、float 之外，也能表达复杂的数组、结构等数据类型。


目前最新的 YAML 版本为 1.2 版本，配置的说明文档在 [官网](#)上。它提供多种语言的解析库，其中 [go-yaml](#) 就是非常通用的一个 Go 解析库，这个库的封装性非常好。

我们通过第一节课讲的快速阅读一个库的命令 `go doc github.com/go-yaml/yaml |grep '^func'`，可以看出来这个库对外提供的方法非常明确，一共三个方法：

Marshal 表示序列化一个结构成为 YAML 格式；

Unmarshal 表示反序列化一个 YAML 格式文本成为一个结构；

还有一个 UnmarshalStrict 函数，表示严格反序列化，比如如果 YAML 格式文件中包含重复 key 的字段，那么使用 UnmarshalStrict 函数反序列化会出现错误。

 复制代码

```
1 // 序列化
2 func Marshal(in interface{}) (out []byte, err error)
3 // 反序列化
4 func Unmarshal(in []byte, out interface{}) (err error)
5 // 严格反序列化
6 func UnmarshalStrict(in []byte, out interface{}) (err error)
```

我们选择 Unmarshal 的函数进行反序列化，因为这样能提高框架对配置文件的容错性和易用性。好，读取配置文件的格式和对应工具搞定，下一步就是想清楚怎么替换了。

配置文件的替换

在上一节课说的环境变量服务中，存放了包括.env 中设置的环境变量，那么我们自然会希望使用上这些环境变量，把配置文件中有的字段使用环境变量替换掉。那么这里在配置文件中就需要有一个“占位符”。这个占位符表示当前这个字段去环境变量中进行阅读。

这个占位符的设计只有一个要求：够特别。只要这个占位符能和其他配置文件字符区分开就行，所以这里设计占位符为比较有语义的“env(XXXX)”。比如 app/config/development/database.yaml 文件中的数据库密码，使用占位符表示如下：

[复制代码](#)

```
1 mysql:
2   hostname: 127.0.0.1
3   username: yejianfeng
4   password: env(DB_PASSWORD)
5   timeout: 1
6   readtime: 2.3
```

要实现这个功能，其实也很简单，可以在读取 YAML 配置文件内容之后，进行完整的文本匹配，将所有环境变量 env(xxx) 的字符替换为环境变量。我们应该能设计出替换文本的函数。

在框架目录的 provider/config/service.go 中，可以先实现这个方法。

[复制代码](#)

```
1 // replace 表示使用环境变量maps替换context中的env(xxx)的环境变量
2 func replace(content []byte, maps map[string]string) []byte {
3     if maps == nil {
4         return content
5     }
6     // 直接使用ReplaceAll替换。这个性能可能不是最优，但是配置文件加载，频率是比较低的，可以:
7     for key, val := range maps {
8         reKey := "env(" + key + ")"
9         content = bytes.ReplaceAll(content, []byte(reKey), []byte(val))
10    }
11    return content
12 }
```

配置项的解析


读取并解析完配置文件内容，接下来就要根据 path 来解析某个配置项了。上一节课说，我们使用点号分割的路径读取方式，比如 database.mysql.password 表示在配置文件夹中的 database.yaml 文件，其中的 mysql 配置，对应的是数据结构中的 password 字段。

那这种根据 path 来读取字段应该怎么实现呢？

在获取配置项的时候，我们已经通过 go-yaml 库将配置文件解析到一个 map 数据结构中了，而这个 map 数据结构的子项，明显也有可能是一个 map 数据结构。所以按照 path 路径查找，这明显应该是一个**函数递归逻辑**。

还是用刚才的 database.mysql.password 举例，可以拆分为 3 个结构。database 去根 map 中寻找；如果有这个 key，就拿着 mysql.password 的 path，去 database 这个 key 对应的 value 中进行寻找；而递归寻找到了最后一级 path 为 password，发现这个 path 没有下一级了，就停止递归。

详细的代码方法如下，同样存放在框架目录的 provider/config/service.go 中。

 复制代码


```
1 // 查找某个路径的配置项
2 func searchMap(source map[string]interface{}, path []string) interface{} {
3     if len(path) == 0 {
4         return source
5     }
6
7     // 判断是否有下个路径
8     next, ok := source[path[0]]
9     if ok {
10        // 判断这个路径是否为1
11        if len(path) == 1 {
12            return next
13        }
14
15        // 判断下一个路径的类型
16        switch next.(type) {
17            case map[interface{}]interface{}:
18                // 如果是interface的map，使用cast进行下value转换
19                return searchMap(cast.ToStringMap(next), path[1:])
20            case map[string]interface{}:
21                // 如果是map[string]，直接循环调用
22                return searchMap(next.(map[string]interface{}), path[1:])
23            default:
24                // 否则的话，返回nil
25                return nil
26        }
27    }
28    return nil
29 }
```

```
26     }
27 }
28 return nil
29 }
30
31 // 通过path获取某个元素
32 func (conf *HadeConfig) find(key string) interface{} {
33     ...
34     return searchMap(conf.confMaps, strings.Split(key, conf.keyDelim))
35 }
36
```

想通了以上三个核心实现难点，我们就可以着手整体代码实现了。

配置服务的代码实现


首先，在框架文件夹的 provider/config/service.go 中，创建一个配置文件服务 HadeConfig。它有几个属性：folder 代表配置本地配置文件所在的文件夹；keyDelim 代表路径中的分割符号，也就是点；envMaps 存放所有的环境变量；而 confMaps 存放每个配置解析后的结构，confRaws 存放每个配置的原始文件信息。

 复制代码

```
1 // HadeConfig 表示hade框架的配置文件服务
2 type HadeConfig struct {
3     c          framework.Container // 容器
4     folder     string             // 文件夹
5     keyDelim   string             // 路径的分隔符，默认为点
6     ...
7     envMaps   map[string]string // 所有的环境变量
8     confMaps  map[string]interface{} // 配置文件结构，key为文件名
9     confRaws  map[string][]byte  // 配置文件的原始信息
10 }
```

我们初始化这个 HadeConfig 的函数，它从服务提供者 provider/config/provider.go 中获取到三个参数，除了容器之外，另外两个是文件夹地址和所有的环境变量。

我们这里对 provider.go 只列一下参数函数，其他的四个服务提供者函数 (Register、Boot、IsDefer、Name) 可以参考 [🔗 GitHub 上的代码](#)。

 复制代码


```

1 // Paramas 服务提供者实例化的时候参数
2 func (provider *HadeConfigProvider) Params(c framework.Container) []interface{
3     appService := c.MustMake(contract.AppKey).(contract.App)
4     envService := c.MustMake(contract.EnvKey).(contract.Env)
5     env := envService.AppEnv()
6     // 配置文件夹地址
7     configFolder := appService.ConfigFolder()
8     envFolder := filepath.Join(configFolder, env)
9     // 传递容器，配置文件夹地址，所有环境变量
10    return []interface{}{c, envFolder, envService.All()}
11 }

```

那么在 provider/config/service.go 中，实例化的函数逻辑如下：

 复制代码

```

1 // NewHadeConfig 初始化Config方法
2 func NewHadeConfig(params ...interface{}) (interface{}, error) {
3     container := params[0].(framework.Container)
4     envFolder := params[1].(string)
5     envMaps := params[2].(map[string]string)
6
7     // 检查文件夹是否存在
8     if _, err := os.Stat(envFolder); os.IsNotExist(err) {
9         return nil, errors.New("folder " + envFolder + " not exist: " + err.Error)
10    }
11    // 实例化
12    hadeConf := &HadeConfig{
13        c:        container,
14        folder:    envFolder,
15        envMaps:   envMaps,
16        confMaps:  map[string]interface{}{},
17        confRaws:  map[string][]byte{},
18        keyDelim:  ".",
19        lock:      sync.RWMutex{},
20    }
21    // 读取每个文件
22    files, err := ioutil.ReadDir(envFolder)
23    if err != nil {
24        return nil, errors.WithStack(err)
25    }
26    for _, file := range files {
27        fileName := file.Name()
28        err := hadeConf.loadConfigFile(envFolder, fileName)
29        if err != nil {
30            log.Println(err)
31            continue
32        }
33    }
34    ...

```

```
35     return hadeConf, nil
36 }
37
38 // 读取某个配置文件
39 func (conf *HadeConfig) loadConfigFile(folder string, file string) error {
40     conf.lock.Lock()
41     defer conf.lock.Unlock()
42     // 判断文件是否以yaml或者yml作为后缀
43     s := strings.Split(file, ".")
44     if len(s) == 2 && (s[1] == "yaml" || s[1] == "yml") {
45         name := s[0]
46         // 读取文件内容
47         bf, err := ioutil.ReadFile(filepath.Join(folder, file))
48         if err != nil {
49             return err
50         }
51         // 直接针对文本做环境变量的替换
52         bf = replace(bf, conf.envMaps)
53         // 解析对应的文件
54         c := map[string]interface{}{}
55         if err := yaml.Unmarshal(bf, &c); err != nil {
56             return err
57         }
58         conf.confMaps[name] = c
59         conf.confRaws[name] = bf
60     }
61     return nil
62 }
```

逻辑非常清晰。先检查配置文件夹是否存在，然后读取文件夹中的每个以 yaml 或者 yml 后缀的文件；读取之后，先用 replace 对环境变量进行一次替换；替换之后使用 go-yaml，对文件进行解析。

初始化实例就是一个完整的 解析文件的过程，解析结束之后，confMaps 里存放的就是解析之后的结果。

配置文件的获取接口上节课已经写好了，定义了接口的系列方法，这里我们就详细实现 Get/GetBool/GetInt，其他方法大同小异，就不贴出来了，你可以直接参考 [GitHub 上的代码](#)。

前面已经想好了，用方法 find，通过 path，从一个嵌套 map confMaps 中获取数据。所以 Get 方法就是调用一下 find 方法而已，同样也在 service.go 中：

```
1 // Get 获取某个配置项
2 func (conf *HadeConfig) Get(key string) interface{} {
3     return conf.find(key)
4 }
```

[复制代码](#)

而对应的 Get 系列的方法我们使用 cast 库进行类型转换，比如：

```
1 // GetBool 获取bool类型配置
2 func (conf *HadeConfig) GetBool(key string) bool {
3     return cast.ToBool(conf.find(key))
4 }
5 // GetInt 获取int类型配置
6 func (conf *HadeConfig) GetInt(key string) int {
7     return cast.ToInt(conf.find(key))
8 }
```

[复制代码](#)

到这里，配置服务的代码已经基本成型了。但是实际上还有两个细节我们需要认真思考。

首先，因为之前我们设置过 App 服务，将一个 App 服务的目录都安排好了，但是如果之后有需求要改变这些目录的配置呢？如果有的话，是否可以通过配置来进行修改呢？所以第一个问题就是，我们要思考配置文件更新 App 服务的操作。

其次，假设现在配置服务能从文件中获取配置了，但是如果文件修改了，我们是否需要重新启动应用呢？是否有能不启动应用的方法呢？

下面我们来一一解决这两个问题。

配置文件更新 App 服务

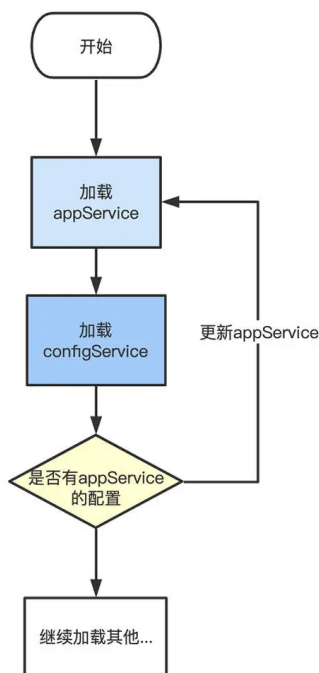
现在有了配置文件服务，但在没有配置文件服务之前，我们启动服务的 appService，也是有可能要修改这个服务的配置的。回忆 [第十二课](#)，appService 中存放了启动这个业务实例默认设置的文件夹目录和地址。

```
1 //BaseFolder 定义项目基础地址
2 BaseFolder() string
3 // ConfigFolder 定义了配置文件的路径
```

[复制代码](#)


```
4 ConfigFolder() string
5 // LogFolder 定义了日志所在路径
6 LogFolder() string
7 // ProviderFolder 定义业务自己的服务提供者地址
8 ProviderFolder() string
9 // MiddlewareFolder 定义业务自己定义的中间件
10 MiddlewareFolder() string
11 // CommandFolder 定义业务定义的命令
12 CommandFolder() string
13 // RuntimeFolder 定义业务的运行中间态信息
14 RuntimeFolder() string
15 // TestFolder 存放测试所需要的信息
16 TestFolder() string
```

现在需求将这些文件夹目录，在配置文件中进行配置并修改。所以应该在加载到配置服务时，再更新下 appService。加载逻辑如下：



可以把设定 App 的这些配置文件，存放在配置文件夹的 app.yaml 文件的 path 设置项下，其中每个配置项的 key，对应 appService 中每个对应的服务。比如 log_folder 对应 LogFolder 目录：

```
1 path:
2   log_folder: "/home/jianfengye/hade/log/"
3   runtime_folder: "/home/jianfengye/hade/runtime/"
```

[复制代码](#)

现在加载配置服务的时候，当读取到配置服务 app.path 下有内容，就需要更新 appService 的配置。首先需要修改 appService，修改框架目录下的 provider/app/service.go 文件。

将 HadeApp 增加一个 configMap 字段：

[复制代码](#)

```
1 // HadeApp 代表hade框架的App实现
2 type HadeApp struct {
3     ...
4     configMap map[string]string // 配置加载
5 }
```

同时为 HadeApp 增加 LoadAppConfig 方法，用于读取配置文件中的信息：

[复制代码](#)

```
1 // LoadAppConfig 加载配置map
2 func (app *HadeApp) LoadAppConfig(kv map[string]string) {
3     for key, val := range kv {
4         app.configMap[key] = val
5     }
6 }
```

再修改对应的 LogFolder 等一系列 XXXFolder 的方法，先读取 configMap 中的值，如果有的话，先用 configMap 中的值：

[复制代码](#)

```
1 // LogFolder 表示日志存放地址
2 func (app HadeApp) LogFolder() string {
3     if val, ok := app.configMap["log_folder"]; ok {
4         return val
5     }
6     return filepath.Join(app.StorageFolder(), "log")
7 }
8
```

这样，对 appService 的修改就完成了。

在 configService，读取配置文件 loadConfigFile 的时候，要注意，如果当前的配置文件是 app.yaml，我们需要调用 appService 的 LoadAppConfig 方法：

[复制代码](#)

```
1 // 读取某个配置文件
2 func (conf *HadeConfig) loadConfigFile(folder string, file string) error {
3     ...
4
5     // 判断文件是否以yaml或者yml作为后缀
6     s := strings.Split(file, ".")
7     if len(s) == 2 && (s[1] == "yaml" || s[1] == "yml") {
8         name := s[0]
9
10        ...
11
12        // 读取app.path中的信息，更新app对应的folder
13        if name == "app" && conf.c.IsBind(contract.AppKey) {
14            if p, ok := c["path"]; ok {
15                appService := conf.c.MustMake(contract.AppKey).(contract.App)
16                appService.LoadAppConfig(cast.ToStringMapString(p))
17            }
18        }
19    }
20    return nil
21 }
```

这样在加载 app.yaml 的配置文件的时候，就同时更新了 appService 里面的配置。


配置文件热更新

正常来说，在程序启动的时候会读取一次配置文件，但是在程序运行过程中，我们难免会遇到需要修改配置文件的操作。也就是之前思考的第二个问题。

这个时候，是否需要重新启动一次程序再加载一次配置文件呢？这当然是没有问题的，但是更为强大的是，**我们可以自动监控配置文件目录下的所有文件，当配置文件有修改和更新的时候，能自动更新程序中的配置文件信息，也就是实现配置文件热更新。**

这个热更新看起来很麻烦，其实在 Golang 中是非常简单的事情。我们使用 [fsnotify](#) 库能很方便对一个文件夹进行监控，当文件夹中有文件增 / 删 / 改的时候，会通过 channel 进行事件回调。

这个库的使用方式很简单。大致思路就是先使用 NewWatcher 创建一个监控器 watcher，然后使用 Add 来监控某个文件夹，通过 watcher 设置的 events 来判断文件是否有变化，如果有变化，就进行对应的操作，比如更新内存中配置服务存储的 map 结构。

 复制代码

```
1 // NewHadeConfig 初始化Config方法
2 func NewHadeConfig(params ...interface{}) (interface{}, error) {
3     ...
4
5     // 监控文件夹文件
6     watch, err := fsnotify.NewWatcher()
7     if err != nil {
8         return nil, err
9     }
10    err = watch.Add(envFolder)
11    if err != nil {
12        return nil, err
13    }
14    go func() {
15        defer func() {
16            if err := recover(); err != nil {
17                fmt.Println(err)
18            }
19        }()
20
21        for {
22            select {
23            case ev := <-watch.Events:
24                {
25                    //判断事件发生的类型，如下5种
26                    // Create 创建
27                    // Write 写入
28                    // Remove 删除
29                    path, _ := filepath.Abs(ev.Name)
30                    index := strings.LastIndex(path, string(os.PathSeparator))
31                    folder := path[:index]
32                    fileName := path[index+1:]
33
34                    if ev.Op&fsnotify.Create == fsnotify.Create {
35                        log.Println("创建文件 :", ev.Name)
36                        hadeConf.loadConfigFile(folder, fileName)
37                    }
38                    if ev.Op&fsnotify.Write == fsnotify.Write {
39                        log.Println("写入文件 :", ev.Name)
40                        hadeConf.loadConfigFile(folder, fileName)
41                    }
42                    if ev.Op&fsnotify.Remove == fsnotify.Remove {
43                        log.Println("删除文件 :", ev.Name)
44                        hadeConf.removeConfigFile(folder, fileName)
```

```

45         }
46     }
47     case err := <-watch.Errors:
48     {
49         log.Println("error : ", err)
50         return
51     }
52 }
53 }
54 }()
55
56 return hadeConf, nil
57 }

```

代码如下，我们使用 NewWatcher 创建一个监听器，监听配置文件目录，接着启动一个新的 Goroutine 作为监听协程。在监听协程中，监听配置文件的创建、更新、删除操作。创建和更新对应 LoadConfigFile 操作。

而删除，对应的是 removeConfigFile 操作，这个操作的内容就是删除配置服务中的 confMaps 中对应的 key。

[复制代码](#)

```

1 // 删除文件的操作
2 func (conf *HadeConfig) removeConfigFile(folder string, file string) error {
3     conf.lock.Lock()
4     defer conf.lock.Unlock()
5     s := strings.Split(file, ".")
6     // 只有yaml或者yml后缀才执行
7     if len(s) == 2 && (s[1] == "yaml" || s[1] == "yml") {
8         name := s[0]
9         // 删除内存中对应的key
10        delete(conf.confRaws, name)
11        delete(conf.confMaps, name)
12    }
13    return nil
14 }

```

这里注意下，由于在运行时增加了对 confMaps 的写操作，所以需要对 confMaps 进行锁设置，以防止在写 confMaps 的时候，读操作进入读取了错误信息。

分析目前的这个场景，读明显多于写。所以我们的锁应该是一个读写锁，读写锁可以让多个读并发读，但是只要有一个写操作，读和写都需要等待。这个很符合当前这个场景。

所以在框架目录的 `provider/config/service.go` 中的 `HadeConfig`，我们增加了一个读写锁 `lock`。

[复制代码](#)

```
1 // HadeConfig 表示hade框架的配置文件服务
2 type HadeConfig struct {
3     ...
4     lock      sync.RWMutex      // 配置文件读写锁
5     ...
6 }
```

而在 `loadConfigFile` 和 `removeConfigFile` 这两个对配置有修改的情况，使用写锁锁住 `HadeConfig`。

[复制代码](#)

```
1 // 读取某个配置文件
2 func (conf *HadeConfig) loadConfigFile(folder string, file string) error {
3     conf.lock.Lock()
4     defer conf.lock.Unlock()
5
6     ...
7 }
```

在 `Get` 系列方法调用的 `find` 函数中，使用读锁来进行读操作。

[复制代码](#)

```
1 // 通过path来获取某个配置项
2 func (conf *HadeConfig) find(key string) interface{} {
3     conf.lock.RLock()
4     defer conf.lock.RUnlock()
5     ...
6 }
```

这样，配置服务就开发完成了。

验证

我们先测试环境变量注入配置文件的功能。将业务目录下的 `config/development/database.yaml` 中的 `mysql.password`，使用环境变量进行替换。

[复制代码](#)

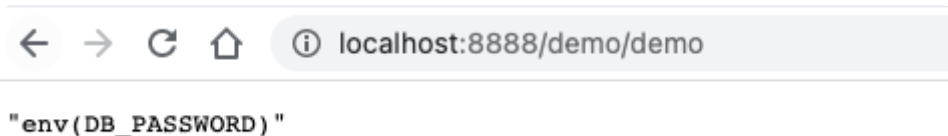
```
1 mysql:
2   hostname: 127.0.0.1
3   username: yejianfeng
4   password: env(DB_PASSWORD)
5   timeout: 1
6   readtime: 2.3
```

然后修改业务目录下的 `module/demo/api.go`，替换其中 `/demo/demo` 对应的路由方法。

[复制代码](#)

```
1 func (api *DemoApi) Demo(c *gin.Context) {
2   // 获取password
3   configService := c.MustMake(contract.ConfigKey).(contract.Config)
4   password := configService.GetString("database.mysql.password")
5   // 打印出来
6   c.JSON(200, password)
7 }
```

最后使用命令行 `./hade app start` 启动服务。打开浏览器，看到输出：



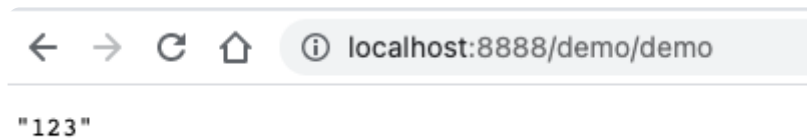
说明此时还没注入环境变量。下面使用命令行：

[复制代码](#)

```
1 DB_PASSWORD=123 ./hade app start
```


启动服务。这个命令注入了 `DB_PASSWORD` 这个环境变量。

重启打开浏览器看到输出。



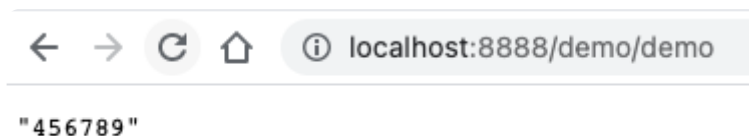
环境变量注入成功！

这个时候我们不停止进程，直接修改配置文件 database.yaml 中的 mysql.password：

 复制代码

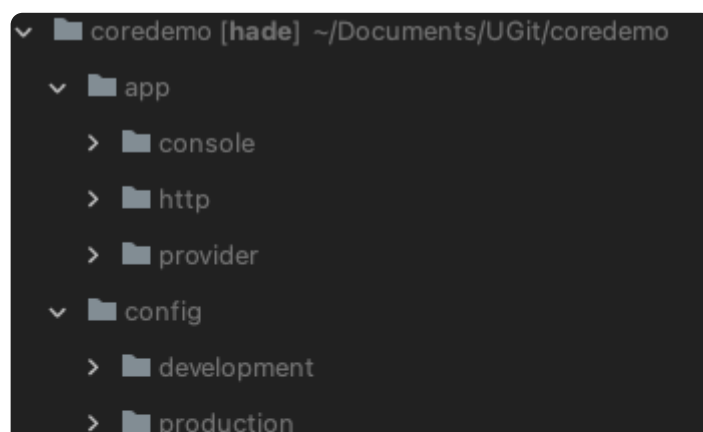
```
1 mysql:
2   hostname: 127.0.0.1
3   username: yejianfeng
4   password: 456789
5   timeout: 1
6   readtime: 2.3
```

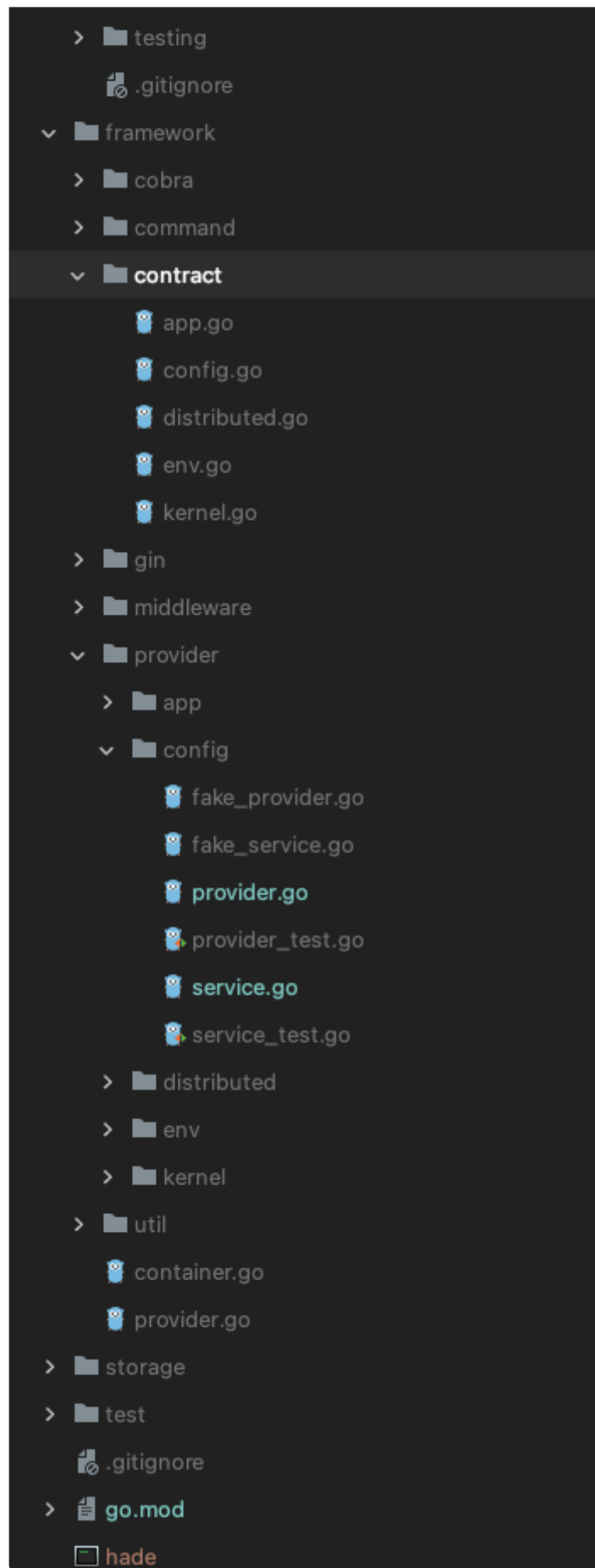
打开浏览器，输出已经变化了。



说明热更新已经生效了，测试成功。

今天所有代码的目录结构截图，也贴在这里供你对比检查，代码放在 GitHub 上的 [16 分支](#) 里。





小结

配置服务在框架中是一个非常基础且重要的服务。

我们考虑了整个配置服务的实现，先读取配置文件，再替换环境变量，最后再根据路径获取配置项，这样三步走完成了基本的配置服务。在配置服务的基础上，我们又补充了配置服务加载时对 App 服务的更新，并且为配置服务增加了热更新的机制。

我个人认为，配置服务是一个 App 中最常用到的服务了，有非常方便的配置服务接口，能为业务代码节省不少的代码量。**提供多种设置配置的方式，是真实从业务需求出发的。**

比如在实际工作中，有的需求要求数据库密码不能进入 git 库，必须通过环境变量获取，我们就可以通过环境变量获取配置；而有的需求要求在一个服务器上调试测试和预发布环境，我们可以通过.env 切换不同环境。所以，有个多层次的环境配置机制，对于一个框架来说是非常必要的。

思考题

现在有配置文件服务了，但是根据路径、获取某个配置却只能在代码中获取。这里我们希望有一个命令行工具 `./hade config get "database.mysql"` 能获取到这个 path 路径对应的配置。你可以尝试实现么？

欢迎在留言区分享你的思考。感谢你的收听，如果觉得有收获，也欢迎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 15 | 配置和环境：配置服务中的设计思路(上)

下一篇 17 | 日志：如何设计多输出的日志服务？

1024 活动特惠

VIP 年卡直降 ¥2000

新课上线即解锁，享 365 天畅看全场

超值拿下 ¥999

极客时间VIP年卡

365天畅学

1024 超值福利

精选留言 (2)

写留言



qinsi
2021-10-23

配置服务和app服务强耦合了，或许它们本来就是一体的？

作者回复：并没有强耦合吧，配置服务获取app都是从容器中获取的



Panmax
2021-10-22

password 是通过环境变量传入的，优先级高于配置文件，我认为不应该被配置文件覆盖掉。

