



下载APP



11 | 面向接口编程：一切皆服务，服务基于协议(下)

2021-10-11 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 16:47 大小 15.38M



你好，我是轩脉刃。

之前对比面向过程 / 面向对象，讲解了抽象程度更高的面向接口编程理念，先定义接口再梳理如何通过接口协议交互，最后实现具体模块。

根据这一理念，我们框架的核心设计就是：框架主体作为一个服务容器，其他各个服务模块都作为服务提供者，在服务容器中注册自己的服务凭证和服务接口，通过服务凭证来获取具体的服务实例。这样，功能的具体实现交给了各个服务模块，我们只需要规范服务提供者也就是服务容器中的接口协议。



在上节课也已经完成了服务提供方接口的实现。所以今天就接着实现框架的主体逻辑。

服务容器的实现

首先是服务容器，先看它需要具有什么能力。同样的，按照面向接口编程，我们不考虑具体实现，先思考服务容器的接口设计。

将服务容器实现在 `framework/container.go` 文件中。

正如之前讨论的，**一个服务容器主要的功能是：为服务提供注册绑定、提供获取服务实例**，所以服务容器至少有两个方法：注册方法 `Bind`、获取实例方法 `Make`。

对于注册的方法，直接将一个服务提供者注册到容器中，参数是之前定义的服务提供者，返回值则是 `error` 是否注册成功。

 复制代码

```
1 // Bind 绑定一个服务提供者，如果关键字凭证已经存在，会进行替换操作，不返回 error
2 Bind(provider ServiceProvider) error
```

获取实例的方法是 `Make`，它会根据一个关键字凭证，来获取容器中已经实例化好的服务。所以参数是一个关键字凭证 `string`，返回值是实例化好的服务 `interface` 和是否有错误的 `error` 信息。

 复制代码

```
1 // Make 根据关键字凭证获取一个服务，
2 Make(key string) (interface{}, error)
```

有了这两个基础方法，再考虑在注册绑定及获取服务实例过程中，有什么方面可以扩展。

首先，因为有绑定操作，那么需要有一个确认某个关键字凭证是否已经绑定的能力：`IsBind`，参数为关键字凭证，返回为 `bool` 表示是否已经绑定。

其次，`Make` 方法返回值中带 `error` 信息，其实这在易用性上并不友好，因为大部分情况下，我们能确定某个服务容器已经被注册了，并不需要处理这个 `error`。所以可以增加一个 `MustMake` 方法，它的参数和 `Make` 方法一样，为关键字凭证，返回值为实例化服务，但是不返回 `error`。

最后我们考虑 Make 的一种拓展场景，**是否会有在获取服务实例的时候，按照不同参数初始化的需求？**

上节课说服务提供者提供了，初始化服务实例方法的能力 Register 和获取初始化服务实例参数的能力 Params。一旦服务实例被初始化了，它就保存在服务容器中了，下次获取的时候，只需要获取已经实例化好的服务。

但是在某次获取的时候，也会有需求要根据不同参数获取新的实例。比如需要根据不同的配置，获取不同的缓存实例的时候，我们可能需要传递不同的参数。所以可以定义一个 MakeNew 的方法，根据参数获取不同实例。

整理一下服务容器的五个接口能力，在 framework/container.go 中代码如下：

[复制代码](#)

```
1 // Container 是一个服务容器，提供绑定服务和获取服务的功能
2 type Container interface {
3     // Bind 绑定一个服务提供者，如果关键字凭证已经存在，会进行替换操作，返回 error
4     Bind(provider ServiceProvider) error
5     // IsBind 关键字凭证是否已经绑定服务提供者
6     IsBind(key string) bool
7
8     // Make 根据关键字凭证获取一个服务，
9     Make(key string) (interface{}, error)
10    // MustMake 根据关键字凭证获取一个服务，如果这个关键字凭证未绑定服务提供者，那么会 panic
11    // 所以在用这个接口的时候请保证服务容器已经为这个关键字凭证绑定了服务提供者。
12    MustMake(key string) interface{}
13    // MakeNew 根据关键字凭证获取一个服务，只是这个服务并不是单例模式的
14    // 它是根据服务提供者注册的启动函数和传递的 params 参数实例化出来的
15    // 这个函数在需要为不同参数启动不同实例的时候非常有用
16    MakeNew(key string, params []interface{}) (interface{}, error)
17 }
```

具体实现

现在接口设计好了，下面结合交互思考下如何实现这个服务容器。我们就定义一个 HadeContainer 数据结构来实现 Container 接口，因为功能是提供绑定服务和获取服务，需要根据关键字获取一个对象，所以 Hash Map 是符合需求的。

在 HadeContainer 内部应该有一个 map[string]interface{} 结构 instances，其中 key 为关键字，value 为具体的服务实例，这样 instances 结构可以存储每个关键字凭证对应的

服务实例，在 Make 系列的方法中，就可以根据这个结构获取对应的服务实例。

同理服务提供方也需要设计一个 `map[string]ServiceProvider` 来存储它们，这样在 Bind 操作的时候，只需要将服务提供方绑定到某个关键字凭证上即可。

另外还要关注一下数据结构的并发性。因为当 Bind 的时候**会对实例或者服务提供方有一定的变动，需要使用一个机制来保证 HadeContainer 的并发性**，是用读写锁还是互斥锁呢？

这里我们就要关注功能了，这个 HadeContainer 是一个读多于写的数据结构，即 Bind 是一次性的，但是 Make 是频繁的。所以使用读写锁的性能会优于互斥锁。

好整理一下，在 `framework/container.go` 中 HadeContainer 的字段定义如下：

[复制代码](#)

```
1 // HadeContainer 是服务容器的具体实现
2 type HadeContainer struct {
3     Container // 强制要求 HadeContainer 实现 Container 接口
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     // instance 存储具体的实例，key 为字符串凭证
7     instances map[string]interface{}
8     // lock 用于锁住对容器的变更操作
9     lock sync.RWMutex
10 }
```

然后对应 HadeContainer，来实现 Container 定义的几个方法就比较容易了，本质上就是对 providers 和 instances 两个 map 结构的修改和读取。这里最核心的 Bind 方法，我会结合代码讲得比较细致，你可以认真体会。

Bind 方法


首先因为 Bind 方法是一个写操作，会修改 providers 和 instances，所以在函数一开头，先给加上一个写锁，然后我们修改 providers 这个字段，它的 key 为关键字，value 为注册的 ServiceProvider。

接着这里**需要先判断是否实例化**，因为定义的 ServiceProvider 中的 IsDefer 方法，控制了实例化时机。

如果 IsDefer 方法标记这个服务实例要延迟实例化，即等到第一次 make 的时候再实例化，那么在 Bind 操作的时候，就什么都不需要做；而如果 IsDefer 方法为 false，即注册时就要实例化，那么我们就需要在 Bind 函数中增加实例化的方法。

所以接下来实现实例化，方法和参数就是 ServiceProvider 中的 Register 和 Params 方法，分别取出实例化方法和参数进行调用，就获取到了具体的服务实例。

最后还有一点要注意下，之前为 ServiceProvider 定义过一个 Boot 方法，是为了服务实例化前做一些准备工作的。所以在实例化之前，要先调用这个 Boot 方法，同样在 framework/container.go 中进行修改。

 复制代码

```
1 // Bind 将服务容器和关键字做了绑定
2 func (hade *HadeContainer) Bind(provider ServiceProvider) error {
3     hade.lock.Lock()
4     defer hade.lock.Unlock()
5     key := provider.Name()
6
7     hade.providers[key] = provider
8
9     // if provider is not defer
10    if provider.IsDefer() == false {
11        if err := provider.Boot(hade); err != nil {
12            return err
13        }
14        // 实例化方法
15        params := provider.Params(hade)
16        method := provider.Register(hade)
17        instance, err := method(params...)
18        if err != nil {
19            return errors.New(err.Error())
20        }
21        hade.instances[key] = instance
22    }
23    return nil
24 }
```

Make 方法

为服务提供注册绑定的 Bind 方法我们就完成了，再来看服务容器的另一个功能提供获取服务实例，也就是 Make 方法。

先判断某个关键字是否已经注册了服务提供者，如果没有注册则后续不能成功返回错误，如果已经注册了就进行实例化操作。

同时注意下，在上面解释过，也会有扩展需求，按照不同参数再次初始化服务的需求。也就是 MakeNew 方法，它和 Make 方法在内部调用最大的不同是传递了一个强制初始化的标记 forceNew，和初始化需要的参数 params。

所以，在两个函数共同的内部实现 make 方法中，我们要先判断是否需要强制初始化实例，如果需要强制初始化，初始化后直接返回。而不需要强制初始化，那么就需要判断之前是否已经实例化了，如果已经实例化了，则返回。

方法的实现同样在 framework/container.go 中：

[复制代码](#)

```
1 // Make 方式调用内部的 make 实现
2 func (hade *HadeContainer) Make(key string) (interface{}, error) {
3     return hade.make(key, nil, false)
4 }
5
6 // MakeNew 方式使用内部的 make 初始化
7 func (hade *HadeContainer) MakeNew(key string, params []interface{}) (interface{}, error) {
8     return hade.make(key, params, true)
9 }
10
11 // 真正的实例化一个服务
12 func (hade *HadeContainer) make(key string, params []interface{}, forceNew bool) (interface{}, error) {
13     hade.lock.RLock()
14     defer hade.lock.RUnlock()
15     // 查询是否已经注册了这个服务提供者，如果没有注册，则返回错误
16     sp := hade.findServiceProvider(key)
17     if sp == nil {
18         return nil, errors.New("contract " + key + " have not register")
19     }
20
21     if forceNew {
22         return hade.newInstance(sp, params)
23     }
24
25     // 不需要强制重新实例化，如果容器中已经实例化了，那么就直接使用容器中的实例
26     if ins, ok := hade.instances[key]; ok {
```

```
27     return ins, nil
28 }
29
30 // 容器中还未实例化，则进行一次实例化
31 inst, err := hade.NewInstance(sp, nil)
32 if err != nil {
33     return nil, err
34 }
35
36 hade.instances[key] = inst
37 return inst, nil
38 }
```

容器和框架的结合

完成了服务容器的接口和对应具体实现，下面就要思考如何将服务容器融合进入框架中。

回顾下 hade 框架中最核心的两个数据结构 Engine 和 Context。

Engine 就是在 [第一节](#) 中实现的 Core 数据结构，这个数据结构是整个框架的入口，也承担了整个框架最核心的路由、中间件等部分。

Context 数据结构对应 [第二课](#) 中实现的 Context 数据结构，它为每个请求创建一个 Context，其中封装了各种对请求操作的方法。

对应来看我们的服务容器，它提供了两类方法，绑定操作和获取操作。绑定操作是全局的操作，而获取操作是在单个请求中使用的。所以在全局，我们为服务容器绑定了服务提供方，就能在单个请求中获取这个服务。

那么对应到框架中，可以**将服务容器存放在 Engine 中，并且在 Engine 初始化 Context 的时候，将服务容器传递进入 Context**。思路很清晰，接下来就按部就班写。

首先，在框架文件 framework/gin/gin.go 中修改 Engine 的数据结构，增加 container 容器，并且在初始化 Engine 的时候，也同时初始化 container。

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

 复制代码

```
6
7 func New() *Engine {
8     debugPrintWARNINGNew()
9     engine := &Engine{
10         ...
11         // 这里注入了 container
12         container: framework.NewHadeContainer(),
13         ...
14     }
15     ...
16     return engine
17 }
```

接着，在 Engine 创建 context 的时候，我们将 engine 中的 container 容器注入到每个 context 中，并且修改 Context 的数据结构，增加 container 容器。同样修改 framework/gin/gin.go 中的 allocateContext 方法。

[复制代码](#)

```
1 // engine 创建 context
2 func (engine *Engine) allocateContext() *Context {
3     v := make(Params, 0, engine.maxParams)
4     // 在分配新的 Context 的时候，注入了 container
5     return &Context{engine: engine, params: &v, container: engine.container}
6 }
```

和修改 framework/gin/context.go 中的 Context 定义。


[复制代码](#)

```
1 // Context 在每个请求中都有
2 type Context struct {
3     // Context 中保存容器
4     container framework.Container
5     ...
6 }
```

这样就完成了服务容器的创建和传递，接下来完成服务容器方法的封装。

根据上面描述的，Engine 中负责绑定，Context 中负责获取，所以我们将 container 的五个能力拆分到 Engine 和 Context 数据结构中。Engine 封装 Bind 和 IsBind 方法，Context 封装 Make、MakeNew、MustMake 方法。

将这些为 Engine 和 Context 增加的新的方法单独存放在一个新的文件 framework/gin/hade_context.go 中。

 复制代码

```
1 // engine 实现 container 的绑定封装
2 func (engine *Engine) Bind(provider framework.ServiceProvider) error {
3     return engine.container.Bind(provider)
4 }
5
6 // IsBind 关键字凭证是否已经绑定服务提供者
7 func (engine *Engine) IsBind(key string) bool {
8     return engine.container.IsBind(key)
9 }
10
11
12
13 // context 实现 container 的几个封装
14 // 实现 make 的封装
15 func (ctx *Context) Make(key string) (interface{}, error) {
16     return ctx.container.Make(key)
17 }
18
19 // 实现 mustMake 的封装
20 func (ctx *Context) MustMake(key string) interface{} {
21     return ctx.container.MustMake(key)
22 }
23
24 // 实现 makenew 的封装
25 func (ctx *Context) MakeNew(key string, params []interface{}) (interface{}, error) {
26     return ctx.container.MakeNew(key, params)
27 }
```

到这里，我们就将服务容器和框架结合了，还需要完成创建服务提供方并创建服务的逻辑。

如何创建一个服务提供方

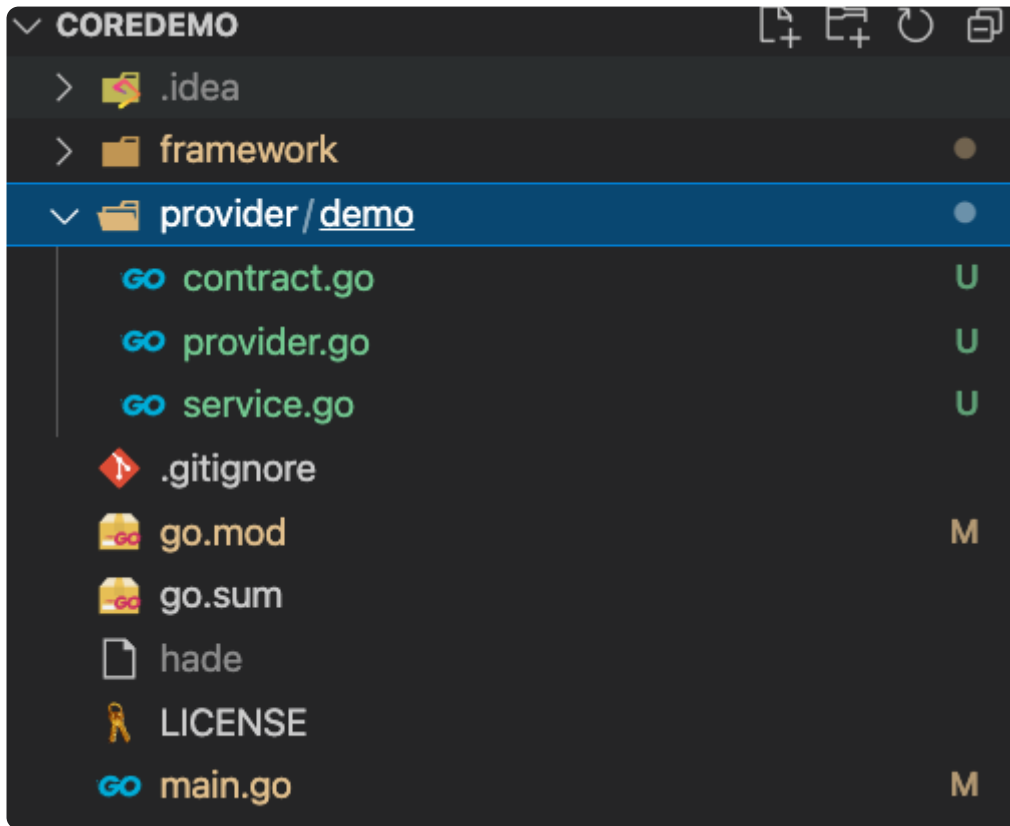
下面我们来创建一个服务 DemoService，为这个服务创建一个服务提供方 DemoServiceProvider，并注入到服务容器中。在业务目录中创建一个目录 provider/demo 存放这个服务。

先搞清楚需要为这个服务设计几个文件。

要有一个服务接口文件 `contract.go`，存放服务的接口文件和服务凭证。

需要设计一个 `provider.go`，这个文件存放服务提供方 `ServiceProvider` 的实现。

最后在 `service.go` 文件中实现具体的服务实例。



所以先来看第一个服务接口说明文件 `contract.go`，在文件中要做两件事情。一是定义一个服务的关键字凭证，这个凭证是用来注册服务到容器中使用的，这里使用 `"hade:demo"` 来作为服务容器的关键字。

另外在这个文件中，我们要设计这个服务的接口，包括接口方法和接口方法使用的对象。比如这里就设计了 `demo.Service` 接口，它有一个 `GetFoo` 方法，返回了 `Foo` 的数据结构。

[复制代码](#)

```
1 package demo
2
3 // Demo 服务的 key
4 const Key = "hade:demo"
5
6 // Demo 服务的接口
7 type Service interface {
8     GetFoo() Foo
9 }
10
```

```
11 // Demo 服务接口定义的一个数据结构
12 type Foo struct {
13     Name string
14 }
```

接下来是服务提供方 `DemoServiceProvider`，在上一节课中，我们描述了服务提供方 `ServiceProvider` 需要实现的五个能力方法，——对照完成定义。

`Name` 方法直接将服务对应的字符串凭证返回，在这个例子中就是 `"hade.demo"`。

`Register` 方法，是注册初始化服务实例的方法，这里先暂定为 `NewDemoService`。

`Params` 方法表示实例化的参数。这里只实例化一个参数：`container`，表示我们在 `NewDemoService` 这个函数中，只有一个参数 `container`。

`IsDefer` 方法表示是否延迟实例化，这里设置为 `true`，将这个服务的实例化延迟到第一次 `make` 的时候。

`Boot` 方法，这里就简单设计为什么逻辑都不执行，只打印一行日志信息。

在 `provider/demo/provider.go` 中定义逻辑：

[复制代码](#)

```
1 // 服务提供方
2 type DemoServiceProvider struct {
3 }
4
5 // Name 方法直接将服务对应的字符串凭证返回，在这个例子中就是"hade.demo"
6 func (sp *DemoServiceProvider) Name() string {
7     return Key
8 }
9
10 // Register 方法是注册初始化服务实例的方法，这里先暂定为 NewDemoService
11 func (sp *DemoServiceProvider) Register(c framework.Container) framework.NewIn
12     return NewDemoService
13 }
14
15 // IsDefer 方法表示是否延迟实例化，我们这里设置为 true，将这个服务的实例化延迟到第一次 ma
16 func (sp *DemoServiceProvider) IsDefer() bool {
17     return true
18 }
19
20 // Params 方法表示实例化的参数。我们这里只实例化一个参数：container，表示我们在 NewDemoS
21 func (sp *DemoServiceProvider) Params(c framework.Container) []interface{} {
22     return []interface{}{c}
23 }
```

```
24
25 // Boot 方法我们这里我们什么逻辑都不执行，只打印一行日志信息
26 func (sp *DemoServiceProvider) Boot(c framework.Container) error {
27     fmt.Println("demo service boot")
28     return nil
29 }
```

在最后的具體 Demo 服务实现文件 service.go 中，我们需要实现 demo 的接口。创建一个 DemoService 的数据结构，来实现 demo 的接口 GetFoo，和正常写一个服务实现某个接口的逻辑是一样的。

[复制代码](#)


```
1 // 具体的接口实例
2 type DemoService struct {
3     // 实现接口
4     Service
5
6     // 参数
7     c framework.Container
8 }
9
10 // 实现接口
11 func (s *DemoService) GetFoo() Foo {
12     return Foo{
13         Name: "i am foo",
14     }
15 }
```

这里不知道你有没有发现，在 DemoService 的数据结构中，**直接嵌套显式地写了 Service 接口，表示这个 DemoService 必须实现 Service 接口**。这是我个人的小习惯，当然这里也可以不用这样显式强调接口实现的，因为在 Golang 中，一个数据结构只需要实现了一个接口的方法，它就隐式地实现了这个接口。

但是我还是习惯这样显式强调的写法，也推荐你可以试一试，有两个好处。

第一对 IDE 友好，有的 IDE 比如 VS Code，在“根据接口查找具体实现的数据结构”的时候，如果没有这么一个显式标记，是寻找不出来的。第二个更重要的原因是对编译友好，一旦接口 Service 变化了，那么这个实现接口的实例，必须要有对应变化，否则在编译期就会出现错误了。

现在有了具体的接口实现结构 `DemoService`，我们只需要最后实现一个初始化这个服务实例的方法 `NewDemoService`。它的参数是一个数组接口，返回值是服务实例。还是把这个方法存放在 `provider/demo/service.go` 中：

 复制代码


```
1 // 初始化实例的方法
2 func NewDemoService(params ...interface{}) (interface{}, error) {
3     // 这里需要将参数展开
4     c := params[0].(framework.Container)
5
6     fmt.Println("new demo service")
7     // 返回实例
8     return &DemoService{c: c}, nil
9 }
```

到这里，就创建了一个示例的服务提供方 `DemoService`。下面再看看如何使用这个服务提供方。

如何通过服务提供方创建服务

这里实现非常简单，我们需要做两个操作，绑定服务提供方、获取服务。

首先是在业务文件夹的 `main.go` 中绑定操作，在 `main` 函数中，完成 `engine` 的创建之后，用在 `engine` 中封装的 `Bind` 方法做一次绑定操作。

 复制代码

```
1 func main() {
2     // 创建 engine 结构
3     core := gin.New()
4     // 绑定具体的服务
5     core.Bind(&demo.DemoServiceProvider{})
6     ...
7 }
```

然后就是服务的获取了。在具体的业务逻辑控制器中，我们选择路由 `/subject/list/all` 对应的控制器 `SubjectListController`，使用为 `context` 封装的 `MustMake` 方法来获取 `demo` 服务实例。

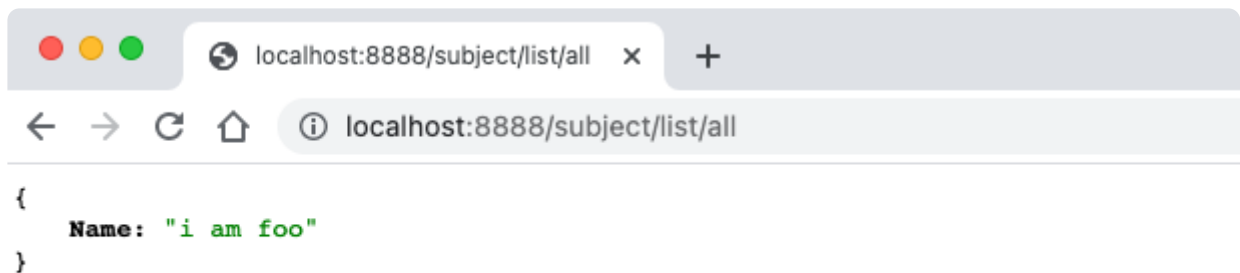
MustMake 的参数为 demo 的服务凭证 demo.Key，返回的是一个 interface 结构，这个 interface 结构实际上是实现了 demo.Service 接口的一个服务实例。

而在接口的具体输出中，输出的是这个接口定义的 GetFoo() 方法的输出，也就是最终会从服务容器中获取到 DemoService 的 GetFoo() 方法的返回值 Foo 结构，带有字段 Name: "i am foo" 输出在页面上。

[复制代码](#)

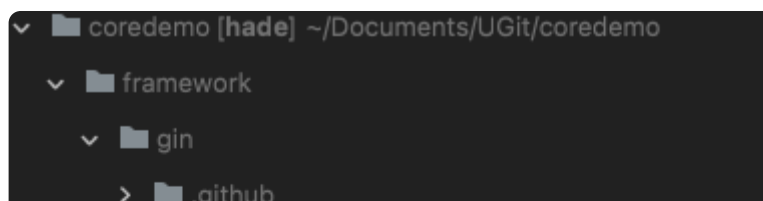
```
1 // 对应路由 /subject/list/all
2 func SubjectListController(c *gin.Context) {
3     // 获取 demo 服务实例
4     demoService := c.MustMake(demo.Key).(demo.Service)
5
6     // 调用服务实例的方法
7     foo := demoService.GetFoo()
8
9     // 输出结果
10    c.JSON(c.StatusCode(), foo)
11 }
```






































最后验证一下，在浏览器中，我们访问这个路由 /subject/list/all，获取到了 Foo 数据结构 Json 化出来的结果，如下图，验证完毕。

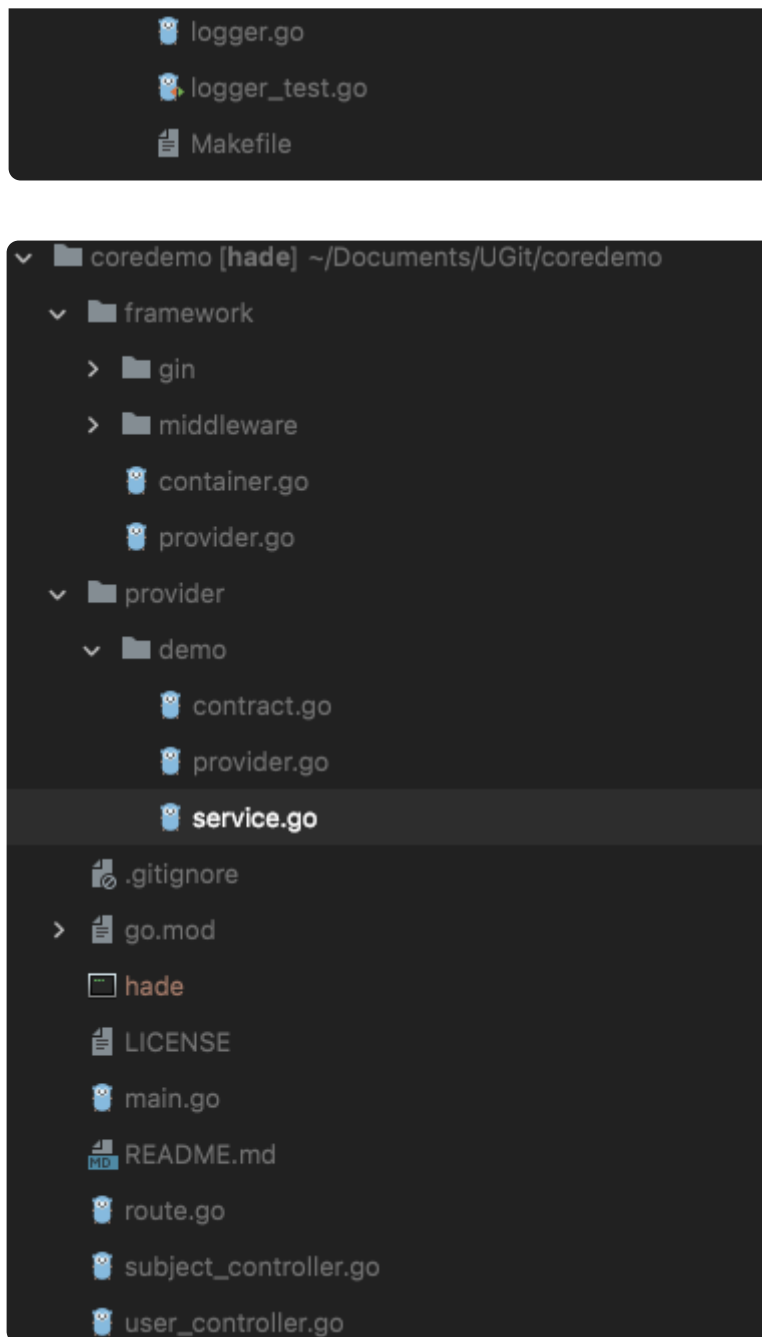


现在服务容器、服务提供者的整个框架主体就搭建完成了。

今天所有的代码都已经上传到 GitHub 的 [@geekbang/11](#) 分支了。目录截图如下：



- >  binding
- >  examples
- >  ginS
- >  internal
- >  render
- >  testdata
-  .gitignore
-  .travis.yml
-  auth.go
-  auth_test.go
-  AUTHORS.md
-  BENCHMARKS.md
-  benchmarks_test.go
-  CHANGELOG.md
-  CODE_OF_CONDUCT.md
-  codecov.yml
-  context.go
-  context_appengine.go
-  context_test.go
-  CONTRIBUTING.md
-  debug.go
-  debug_test.go
-  deprecated.go
-  deprecated_test.go
-  doc.go
-  errors.go
-  errors_1.13_test.go
-  errors_test.go
-  fs.go
-  gin.go
-  gin_integration_test.go
-  gin_test.go
-  githubapi_test.go
-  hade_context.go
-  hade_request.go
-  hade_response.go
-  LICENSE



小结

我们在主体框架中实现了服务容器、服务提供者的逻辑，现在，hade 框架就包含一个服务容器，所有的服务都会在服务容器中注册。当业务需要获取某个服务实例的时候，也就是从服务容器中获取服务。

在这节课中，你是不是能感知到服务容器的方便之处了。只需要往服务容器中注册服务提供者，之后不管任何时候，想要获取某个服务，都能很方便地从服务容器中，获取到符合服务接口的实例，而不需要考虑到服务的具体实现。**这种设计的拓展性非常好，之后在实际业务中我们只要保证服务协议不变，而不用担心具体的某个服务实现进行了变化。**

后续开发的所有服务模块，比如日志、配置等我们都会以服务的形式进行开发，先定义好服务的接口，后定义服务的服务提供者，最后再定义服务的具体实例化方法。

思考题

在实现服务容器时，不知道你会不会有一个疑问：我们将服务容器的 Make 系列的方法在 Context 中实现了，为什么不把 Bind 系列方法也在 Context 中实现呢？这个问题你会怎么思考呢？Context 允许 Bind 方法有什么好处和什么不好的地方呢？

欢迎在留言区分享你的思考。感谢你的收听，如果你觉得有收获，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 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 10 | 面向接口编程：一切皆服务，服务基于协议(上)

下一篇 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 (4)

写留言



芒果少侠

2021-10-12

老师我提个并发场景下的潜在bug：如果serviceProvider是延迟加载的话，多个请求同时调用Make的时候，这时候是读锁，所以在后面新建出来实例后，回写map的时候，可能会出现并发写的情况。<https://github.com/gohade/coredemo/blob/geekbang/11/framework/container.go#L133>

...

展开



芒果少侠

2021-10-12

context是请求层面的，如果每个业务请求都能根据业务属性进行bind，那么对于凭证名相同的serviceProvider，可能会出现每次取出来的serviceProvider实例都不一样。

另一个角度考虑，应该是性能和开销考虑。

展开



qinsi

2021-10-12

在Context里Bind就能允许在运行时对不同的请求使用不同的服务。比如在Bind时读取一下配置，看下要用的服务是配置成pkgA提供的还是pkgB提供的。这样可能上一个请求用的是pkgA的服务，改了配置后下个请求就用上了pkgB的。这样做的好处是切换不同的服务实现时不用重启app，缺点就是每个请求都要读取配置会有额外的消耗。随着容器编排技术的流行，滚动更新容器化的app也变得容易，需要配置热更新的场合也会减少吧。

展开 ∨



Panmax

2021-10-11

思考题：在实现服务容器时，不知道你会不会有一个疑问：我们将服务容器的 Make 系列的方法在 Context 中实现了，为什么不把 Bind 系列方法也在 Context 中实现呢？这个问题你会怎么思考呢？Context 允许 Bind 方法有什么好处和什么不好的地方呢？

因为 Make 是用来获取服务的，获取服务通常是在处理每个请求时获取，每个请求都会...

展开 ∨

