



下载APP



10 | 面向接口编程：一切皆服务，服务基于协议(上)

2021-10-08 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 18:53 大小 17.30M



你好，我是轩脉刃。

在上一节课我们已经将 Gin 框架迁移到自己的 hade 框架中，但是如果 hade 框架只止步于 Gin 的已有功能：支持 Context、路由、中间件这些框架最简单的功能，显然背离了我们设计这个框架的初衷，毕竟我们的目标是搭建一个生产中可用且具有丰富功能模块的框架。

那么如何组织这些功能模块更好地协作，就是我们今天要讨论的问题。



组织的方法也很简单，之前提到过。如果你还记得我们第五课封装请求和返回结构的时候，先定义了 IRequest 和 IResponse 接口，再一一实现具体的函数方法，**这种先接口后**

实现的方式，其实不仅仅是一种代码优化手段，更是一种编程思想：面向接口编程，这其实就是我们组织功能模块的核心思路。

面向接口编程

面向接口编程的思想到底是什么含义？我们从关键词“接口”开始思考。

不知道你考虑过这个问题没有，现在的高级语言，比如 PHP、Golang、Java 等，除了函数、对象等定义之外，都无一例外地拥有“接口”。但是为什么这些高级语言会需要有“接口”这个定义呢？我们从接口做到了什么的角度来反向思考。

抽象业务

首先，接口实现了对业务逻辑的抽象，设计接口就是抽象业务的过程。

因为在工作、生活中，我们需要把业务转换到代码中。但是一个真实的业务需求往往有很多复杂的描述，有些描述是某个业务特有的，而有些描述是所有同类型业务共有的，我们需要把描述的相同点抽象出来，成为一个个的步骤。而每个步骤实际上都是一个通过输入、产生输出的方法，把这些方法聚合起来，就是一个接口。

这段话说得有点抽象，我们结合一个具体例子来加深理解。在《面向接口设计（Interface Oriented Design）》这本书中，作者举了寻找一个 pizza 店并购买 pizza 的例子，很简单也便于理解，这里我们是为了明白接口实现了什么，所以就不再另外举例，直接看这个例子就好。

在找 pizza 店之前我们首先要知道 pizza 店是什么，所以先来定义一个 pizza 店，并且来寻找所有 pizza 店共性的地方。

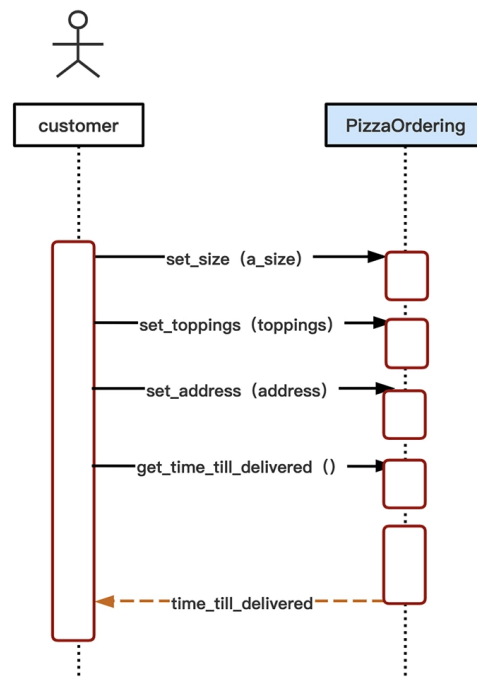
作者发现购买 pizza 的流程基本都是相似的，先确认大小、再确认佐料、再确认收货地址、最后被告知送货时间，**所以可以将“购买 pizza”这个业务逻辑，抽象成一个接口结构：PizzaOrdering**。它包含四个方法：set_size、set_toppings、set_address、get_time_till_delivered。

set_size 的参数是一个枚举，代表 pizza 的预设大小；

set_toppings 的输入是一个数组，代表不同的佐料；

set_address 的输入是一个字符串，代表收货地址；

get_time_till_delivered 的输入为空，但是输出为时间格式，代表预估的送货到达时间。



所以“寻找 pizza 店”这个事情，也可以抽象成为一个接口：PizzaOrderingFinder。它实现的方法有三个，都返回对于 pizza 店的 PizzaOrdering 接口。

PizzaOrdering find_implementation_by_name(String
name_of_pizza_shop) 根据名字查询对应的 pizza 店；

PizzaOrdering find_first_implementation() 查找第一个 pizza 店；

PizzaOrdering find_last_implementation() 查找最后一个 pizza 店。

从上面这个例子我们可以看出，一个业务可以由一个或者多个接口组成。针对去 pizza 店预订 pizza 这个业务，我们定义了两个接口：寻找 pizza 店的接口、pizza 店的接口。

其中寻找 pizza 店的接口，它提供多种筛选能力，比如按名字查询、按正序 / 倒序第一查询；pizza 店的接口拥有四个能力，它们有自己的输入输出，这四个能力聚合组成了 pizza 店的抽象接口。

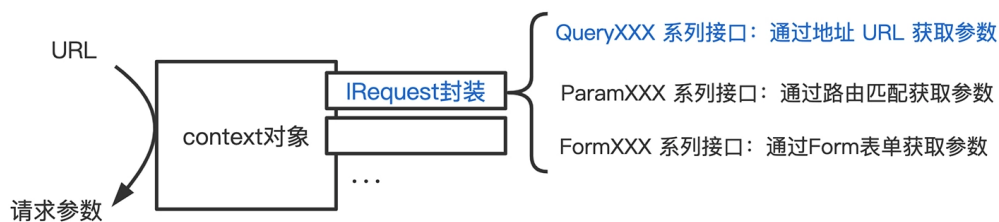
这就是一种业务抽象，用一个接口结构抽象可以购买 pizza 的店铺所具有的能力，用另外一个接口数据抽象查询 pizza 店铺的方法。当设计完一个业务的接口之后，我们实际上就完成了对这个业务从实际生活到代码世界的抽象和建模。

屏蔽具体实现

其次，接口的使用能让调用方对能力敏感，而对具体实现不敏感。

拿我们在第五章设计的 IRequest 接口举例，如果一个对象实现了这个接口，那么不管在什么模块调用这个对象，我们都能知道它拥有的能力有哪些、能通过哪些方式来获取到这些能力。

具体来说在写业务的时候，如果我们要从 URL 中获取请求参数，可以寻找实现了 IRequest 接口的对象，不管是什么对象，只要实现了 IRequest 接口，它就能满足我们的需求。比如在我们这个框架中就是 context 数据结构。从 context 这个对象中，通过 IRequest 中封装的 QueryXXX 系列的接口，就能获取到 URL 中的参数。



当模块之间的调用有了接口，调用方就无需关心被调用模块的具体实现。因为在调用方逻辑中，不会出现被调用者具体实现的数据结构名称，只会出现被调用者承诺能力的一个简单接口。

这对调用方是非常友好的。一方面调用方不用担心被调用方内部的结构调整，只要被调用方按照承诺的能力提供服务；另一方面，调用方可以随时换掉被调用方，只要替换者也提供同等的能力即可。

比如上面那个寻找 Pizza 店的业务，PizzaOrderFinder 这个结构中的方法 find_implementation_by_name 返回的，就直接是接口 PizzaOrdering。意思是，要查询的是满足 PizzaOrdering 定义的一个店铺，只要有定义中的四个能力，顾客就能购买到 Pizza。所以，不管这个店铺是 pizza 店还是超市，只要提供了 pizza 店有的四个能力，就能将店铺放进寻找范围内。

面向接口 / 对象 / 过程

理解了接口的优势，我们来思考下“面向接口编程”的意义，它和“面向对象编程”、“面向过程编程”又有什么区别。

其实，这三个名词描述的都是思维方式，就是我们在抽象业务的时候如何思考问题。

“面向过程编程”是指进行业务抽象的时候，**我们定义一个一个的过程方法，通过这些过程方法的串联完成具体的业务。**

还是拿查询 Pizza 店的业务举例，面向过程编程的思考思路可能是最符合我们平时的思维逻辑：第一步，先查找附近的所有店铺；第二步，过滤出 pizza 店；第三步，按照自定义规则再过滤，查找出我要的 pizza 店。

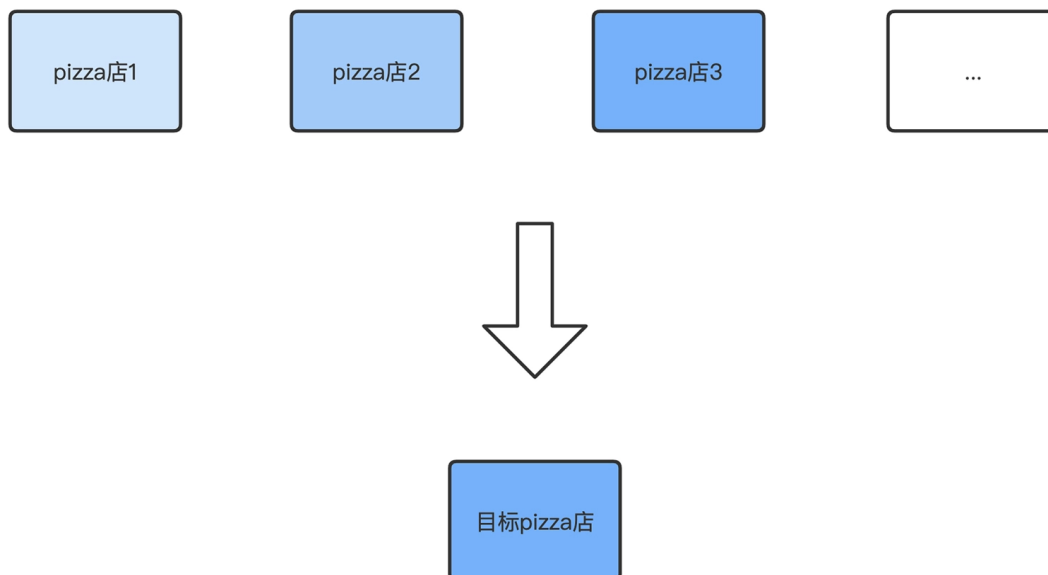


所以面向过程编程的整个思路就是实现思路，它的思维方式比较直接简单，按照目标一步步实现。**但缺点是一旦需求有一些变化，整个链路的改动都会受影响。**

比如最终的目标从按照 pizza 店铺的名字进行搜索，变化成按照店铺中的座位数进行搜索，那么由于这个需求变化，第二步过滤的逻辑，原本只返回 pizza 店铺的名字，就要增加返回 pizza 店铺的座位。这种改动往往涉及到全链路的改动，影响比较大。

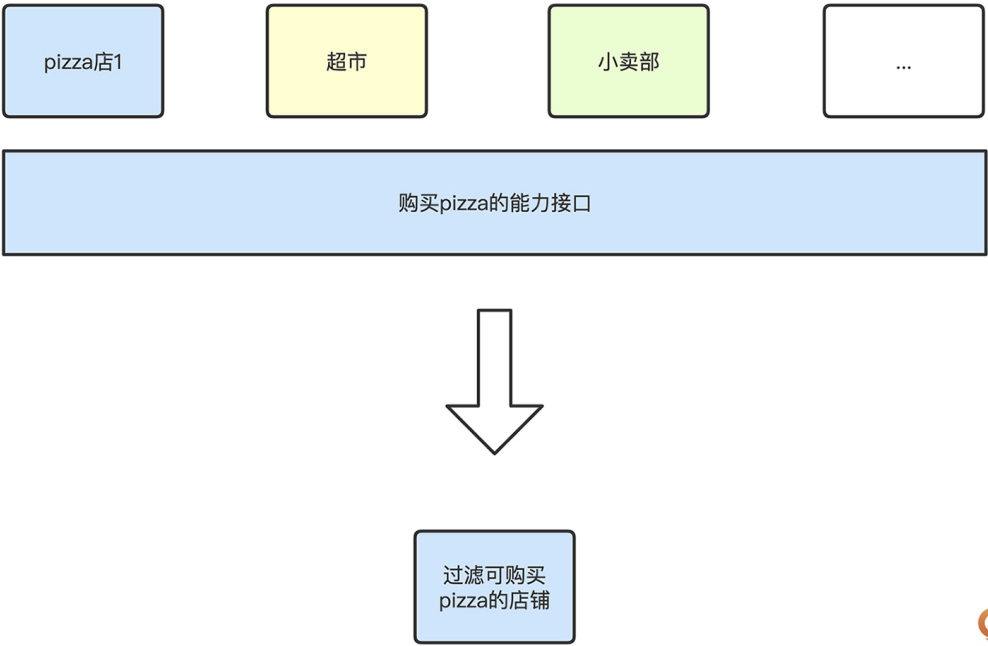
“面向对象编程”表示的是在业务抽象的时候，**我们先定义业务中的对象，通过这些对象之间的关联来表示整个业务。**

对于寻找 Pizza 店的例子，我们会将所有的 pizza 店铺抽象成一个数据结构 Pizza 店，这个 Pizza 店铺提供购买 pizza 的能力和属性。这个时候过滤这些店铺就比较简单了，直接按照某些属性进行过滤。



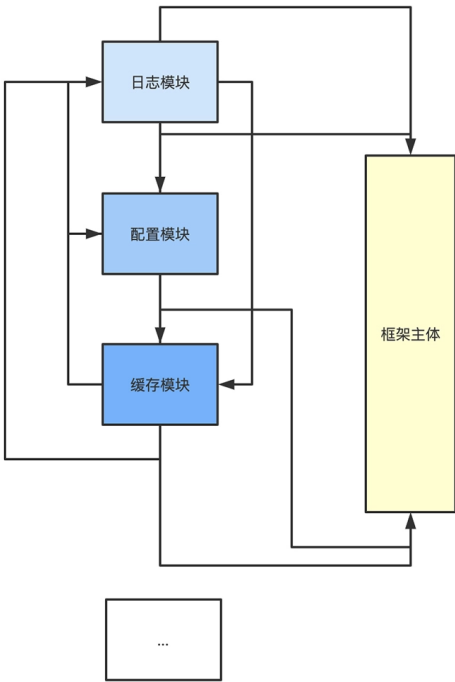
所以面向对象编程抽象性就很强了，但是它的问题就是调用方依赖具体的数据结构 pizza。比如还是按照座位号过滤，调用方会根据 pizza 结构中的座位号字段来过滤，但是 pizza 中的座位号这个字段我定义成什么变量，这个其实调用方并不想知道。所以我们其实还可以再进一步抽象，就是面向接口编程。

面对业务，我们并不先定义具体的对象、思考对象有哪些属性，而是**先思考如何抽象接口，把接口的定义放在第一步，然后多个模块之间梳理如何通过接口进行交互，最后才是实现具体的模块。**



接口服务的理论基础

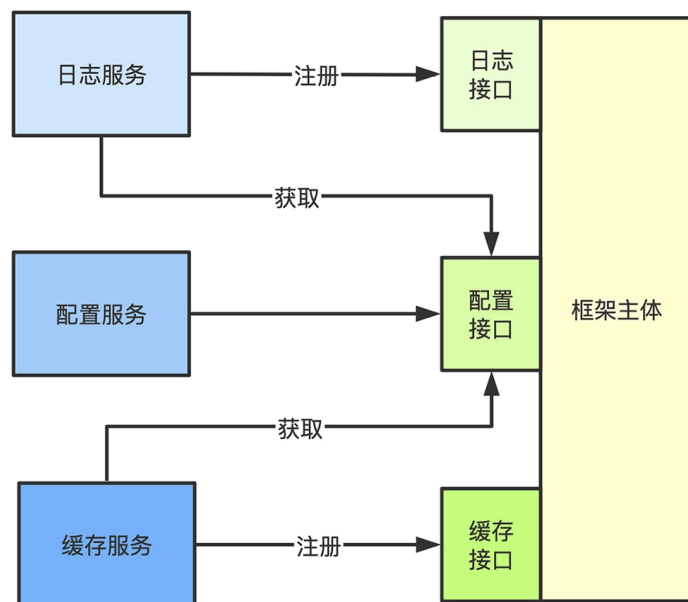
现在理解了面向接口编程的理念，就可以将这个理念运用到我们的 hade 框架中了。在框架中会包含很多模块，这些模块会和框架主体交互，也会互相交互，所以如果从功能的交互上看，整体会是一个非常复杂的网状结构。



如果改变一下思路，按照面向接口编程的理念，将每个模块看成是一个服务，服务的具体实现我们其实并不关心，我们关心的是服务提供的能力，即接口协议。那么框架主体真正要做的事情是什么呢？其实是：**定义好每个模块服务的接口协议，规范服务与服务之间的调用，并且管理每个服务的具体实现。**

所有的服务都去框架主体中注册自身的模块接口协议，其他的服务调用功能模块的时候，并不是直接去这个服务获取实例，而是从框架主体中获取有这个接口协议的服务实例。

这样，所有的模块服务都不和具体的服务进行交互，而是和框架主体进行交互，所有的接口也都注册在框架主体中，非常方便管理。



极客时间

每个模块服务都做两件事情：一是它和自己提供的接口协议做绑定，这样当其他人要使用这个接口协议时能找到自己；二是它使用到其他接口协议的时候，去框架主体中寻找。

所以，这个时候，每个模块服务都是一个“**服务提供者**”（service provider），而我们主体框架需要承担起来的角色叫做“**服务容器**”（service container），服务容器中绑定了多个接口协议，每个接口协议都由一个服务提供者提供服务。

在框架初始化启动的时候，我们可以选择在服务容器中绑定多个服务提供者，每个服务提供者对应一个凭证。当要使用到某个服务的时候，再根据这个凭证去服务容器中，获取这个服务提供者提供的服务。这样就能很方便地获取服务了。

这两个结构的逻辑非常重要，这里我再强调一下。我们的设计是将每个服务，不管是配置、还是日志、还是缓存，都看成是一个服务。

这个服务，通过提供一个服务提供者注册到服务容器中。服务提供者提供的是“创建服务实例的方法”，服务容器提供的是“实例化服务的方法”。至于这个服务实例拥有哪些能力，即符合哪个接口协议，是预先在框架主体中定义好的。

讲完服务提供者和服务容器的理论基础，就要讲具体实现了，今天我们先了解服务提供者如何实现，下一课接着学服务容器。

服务提供者的接口定义

按照面向接口编程的逻辑，一个服务提供者需要有哪些能力呢？一共有五个能力，先简单看一眼做到心中有数，再来详细理解为什么要设计这些能力：

获取服务凭证的能力 Name；

创建服务实例化方法的能力 Register；

获取服务实例化方法参数的能力 Params；

两个与实例化控制相关的方法，控制实例化时机方法 IsDefer、实例化预处理的方法 Boot。

我们将服务提供者的接口定义放在框架目录的 framework/provider.go 中。

基本功能

首先因为要和服务容器做绑定，所以一个服务提供者需要有一个凭证，绑定时作为凭证关联。这里的凭证我们就直接设计为一个字符串结构，即**服务提供者首先有一个获取凭证字符串的方法 Name()**。

 复制代码

```
1 // Name 代表了这个服务提供者的凭证
2 Name() string
```

然后一个服务提供者需要有创建服务实例方法的能力。因为在服务容器中绑定后，如果服务容器要初始化一个服务实例，就需要调用服务提供者中创建服务实例的方法。

按照面向接口编程的思想，每个具体服务“创建服务实例”的方法不一样，比如日志服务初始化的时候可能需要有日志输出地址，但是配置服务初始化的时候需要有配置文件地址，但是我们这里需要规范它们的输入和输出，使用 Golang 中的 function type，也叫函数定义，是可以做这个事情的。

[复制代码](#)

```
1 // NewInstance 定义了如何创建一个新实例，所有服务容器的创建服务
2 type NewInstance func(...interface{}) (interface{}, error)
```

这个 NewInstance 就是一个函数定义，它规定所有创建服务实例的方法必须：有相同的参数 interface{} 数组，并且返回 interface{} 和错误信息这两个数据结构。

interface{} 数组代表实例化一个服务所需要的参数，我们这里设计为可变参数，为的是适配不同数量、不同类型的参数需求；

返回值返回的 interface{} 结构代表了具体的服务实例。

定义好了“创建服务实例的方法”的函数，我们再看服务提供者的创建能力如何实现，也就是 NewInstance 方法，它的返回值就是刚才写的 NewInstance 的函数定义。

[复制代码](#)

```
1 // Register 在服务容器中注册了一个实例化服务的方法，是否在注册的时候就实例化这个服务，需要参
2 Register(Container) NewInstance
```

而对于方法的输入参数，将服务容器传进来是因为，如果后续希望根据一个服务的某个能力，比如配置服务的获取某个配置的能力，返回定义好的不同 NewInstance 函数，那我们就需要先服务容器获取配置服务，才能判断返回哪个 NewInstance。

所以这里我们将服务容器作为传入参数。（这个服务容器的结构我们下节课再具体讨论，这里先用 Container 名称代替。）

“创建服务实例的方法”的能力，除了实现 `NewInstance` 方法之外，还需要注册 `NewInstance` 方法的参数，即可变的 `interface{}` 参数。所以我们的服务提供者还需要提供一个获取服务参数的能力。

[复制代码](#)

```
1 // Params params 定义传递给 NewInstance 的参数，可以自定义多个，建议将 container 作为  
2 Params(Container) []interface{}
```

实例化过程的控制

到这里服务提供者的能力已经基本设计好了。不过我们可以再思考下实例化的过程，看看还有什么讲究。

实例化的时机，可以在服务提供者注册的时候，也可以是第一次获取服务的时候，即是注册的时候就实例化，还是延迟到获取服务的时候实例化。

所以我们需要有一个能力能控制实例化的时机，对应到服务提供者上，要提供告知服务容器是否延迟实例化的方法 `IsDefer`。同样在 `framework/provider.go` 中。

[复制代码](#)

```
1 // IsDefer 决定是否在注册的时候实例化这个服务，如果不是注册的时候实例化，那就是在第一次 ma  
2 // false 表示不需要延迟实例化，在注册的时候就实例化。true 表示延迟实例化  
3 IsDefer() bool
```

实例化之前有可能需要做一些准备工作，比如在每次实例化之前，想记录一下日志，或者想通过确认某些配置，修改一下实例化参数。

所以这里我们需要设计一个在实例化前调用准备工作的函数 `Boot`。它的参数是服务容器，返回值是一个 `error`，在实例化服务的时候，如果准备工作 `Boot` 失败了，那么我们就进行后续的实例化操作了，将这个 `error` 直接返回给获取服务的方法。

[复制代码](#)

```
1 // Boot 在调用实例化服务的时候会调用，可以把一些准备工作：基础配置，初始化参数的操作放在这个  
2 // 如果 Boot 返回 error，整个服务实例化就会实例化失败，返回错误  
3 Boot(Container) error
```

到这里，我们就定义好了服务提供者的接口了。再简单回顾一下，有三个基础能力、两个控制相关的能力：


获取服务凭证的能力 Name；

注册服务实例化的方法的能力 Register；

获取服务实例化方法参数的能力 Params；

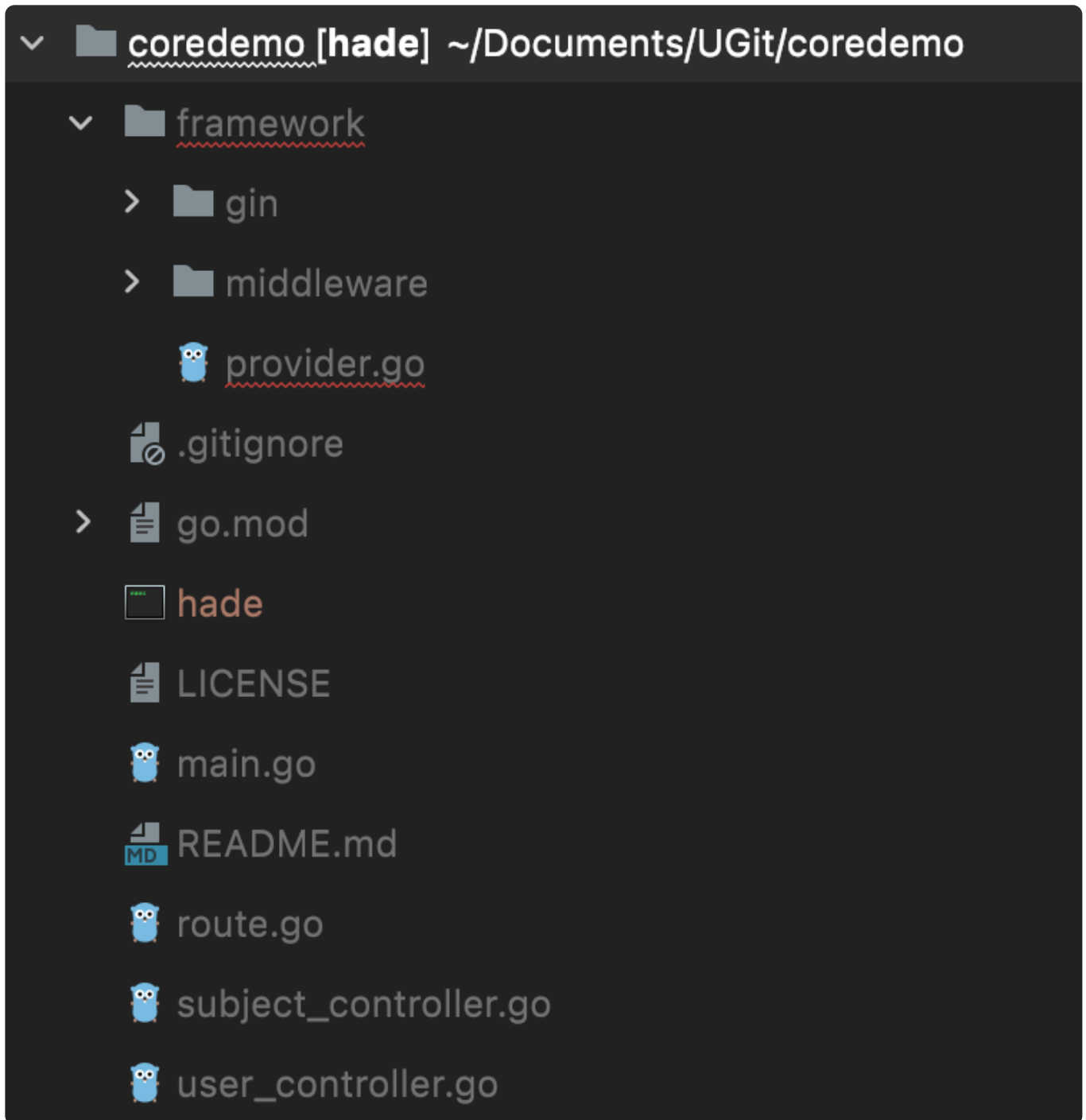
两个与实例化控制相关的方法，控制实例化时机方法 IsDefer、实例化预处理的方法 Boot。

framework/provider.go 中完整代码如下：

 复制代码

```
1 package framework
2
3 // NewInstance 定义了如何创建一个新实例，所有服务容器的创建服务
4 type NewInstance func(...interface{}) (interface{}, error){}
5
6 // ServiceProvider 定义一个服务提供者需要实现的接口
7 type ServiceProvider interface {
8     // Register 在服务容器中注册了一个实例化服务的方法，是否在注册的时候就实例化这个服务，需要
9     Register(Container) NewInstance
10    // Boot 在调用实例化服务的时候会调用，可以把一些准备工作：基础配置，初始化参数的操作放在这
11    // 如果 Boot 返回 error，整个服务实例化就会实例化失败，返回错误
12    Boot(Container) error
13    // IsDefer 决定是否在注册的时候就实例化这个服务，如果不是注册的时候就实例化，那就是在第一次 n
14    // false 表示不需要延迟实例化，在注册的时候就实例化。true 表示延迟实例化
15    IsDefer() bool
16    // Params params 定义传递给 NewInstance 的参数，可以自定义多个，建议将 container 作
17    Params(Container) []interface{}
18    // Name 代表了这个服务提供者的凭证
19    Name() string
20 }
21
```

今天主要说明了面向接口编程的逻辑，以及服务提供者、服务容器、服务之间的关系。代码层面我们只增加了一个 provider.go。代码我已经上传到了 GitHub 上的 [geekbang/10](https://github.com/geekbang/10) 分支。但是在定义 Provider 的时候，我们在参数中使用了下节课需要定义的服务容器结构 Container，所以目前这个分支的代码暂时还是不能运行的。



小结

结合面向接口编程的理念，我们希望设计出的框架是一个服务容器，也就是说，并不是在它的内部实现各种各样的功能模块，而是在框架中，**定义好每个模块服务的接口，规范服务与服务之间的调用，并且管理每个服务的具体实现。**

具体功能模块的实现由绑定的服务提供者进行，我们只需要规范服务提供者的能力，就能获取到具体的服务实例了。

今天的内容比较偏向理论，一直围绕“面向接口编程”的概念讨论。但是希望你不要觉得枯燥，还是那句话，不仅要知其然，还要知其所以然。先搞清楚为什么要这么设计我们的 hade 框架、它的理论基础是什么，才能顺利实现，并且在后期灵活调整和拓展。下一讲我们继续学习服务容器的实现。

思考题

今天花了很大篇幅来解释面向接口的思想，我们会将这个思想贯穿在 Web 框架的整个设计中。其实面向接口思想不仅仅应用在编程分层中，在架构设计、微服务中也都有所体现，你可以想想在工作生活中有遇到使用面向接口思想的一些实例么？

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 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 [加餐 | 阶段答疑：这些代码里的小知识点你都知道吗？](#)

下一篇 [11 | 面向接口编程：一切皆服务，服务基于协议\(下\)](#)

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 (2)

写留言



Geek_5244fa

2021-10-08

Laravel 既视感。

好奇最后怎样在 handler 里获取服务，怎样实现依赖注入。

1

1



拾掇拾掇

2021-10-08

签到，学习了

展开

1

1