31 | 通用模块（上）：用户模块开发

2021-12-01 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 18:26 大小 16.89M

你好，我是轩脉刃。

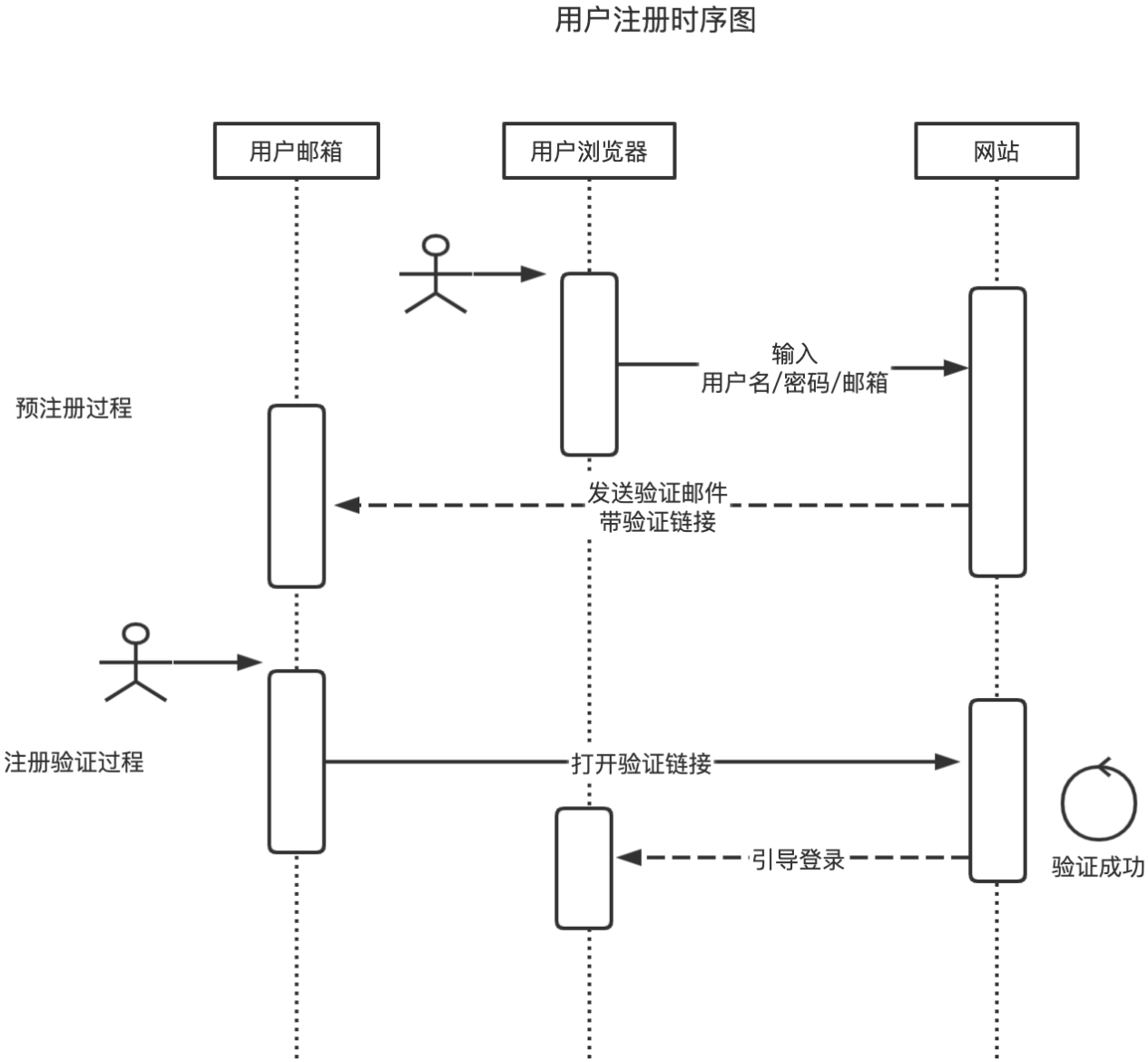
上一节课分析了一下问答网站的需求，并且搭建起来了前端和后端框架，我们就来填充这个网站的具体内容。今天要做的是用户模块的开发。用户模块基本是所有系统的基础模块，所以如何开发设计用户模块，希望你一定好好掌握。

模块设计

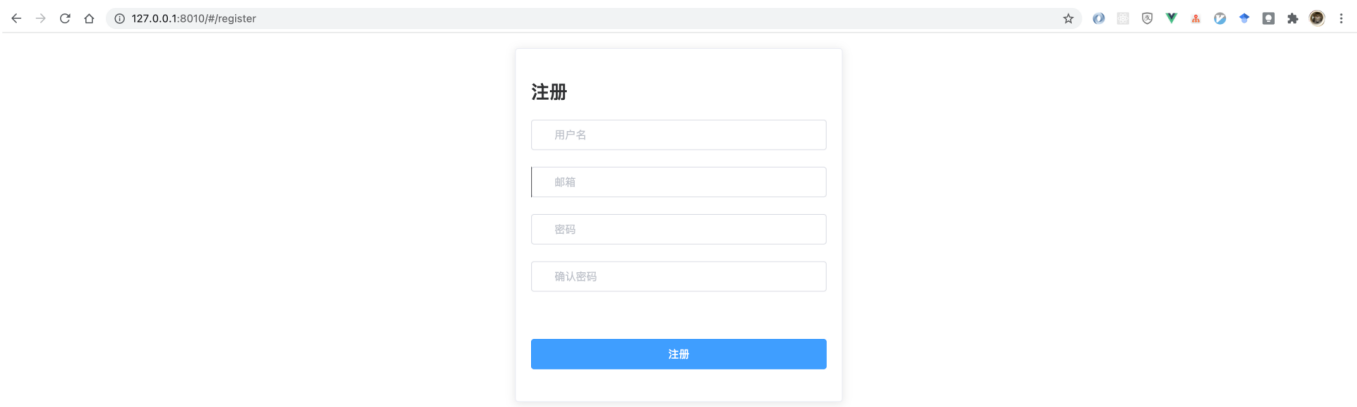
简单回顾一下用户模块的需求分析，分为两个部分，用户注册和用户登录。我们先细化下每个部分并且定义好它们的接口。

用户注册

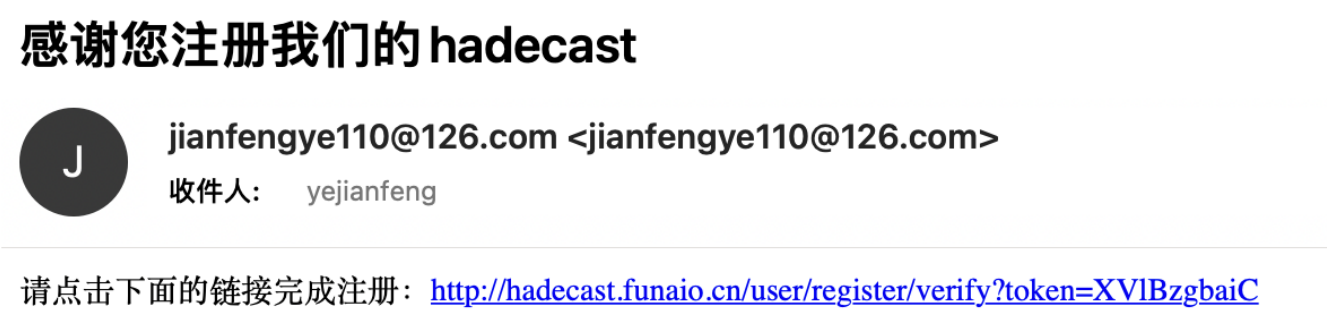
首先是用户注册，它的时序图再放一下：



其中包含两个页面，第一个页面是预注册过程页面，用户在这个页面中输入用户名、密码、邮箱。这个页面的路径我们设置为 /register。



输入用户名密码之后，它会发送一个邮件到用户的邮箱，在这个邮件中会带着一个确认注册的链接，只有通过点击这个链接，你才算完成验证。发送邮件的邮箱，我使用自己注册的一个 126 邮箱，最终邮件内容效果是这样的：



用户点击 /user/register/verify 链接之后，才算正式注册完毕，接着会引导用户进入登录流程。

所以我们梳理一下，预注册这个过程前端和后端一共进行了两次交互，也就是说需要两个接口。

/user/register 预注册接口

用户在这个接口中带着用户名、密码和邮箱到后端。一般这种注册类接口我们会选择使用 POST 方法，它的参数为 username、password、email 三个字段。

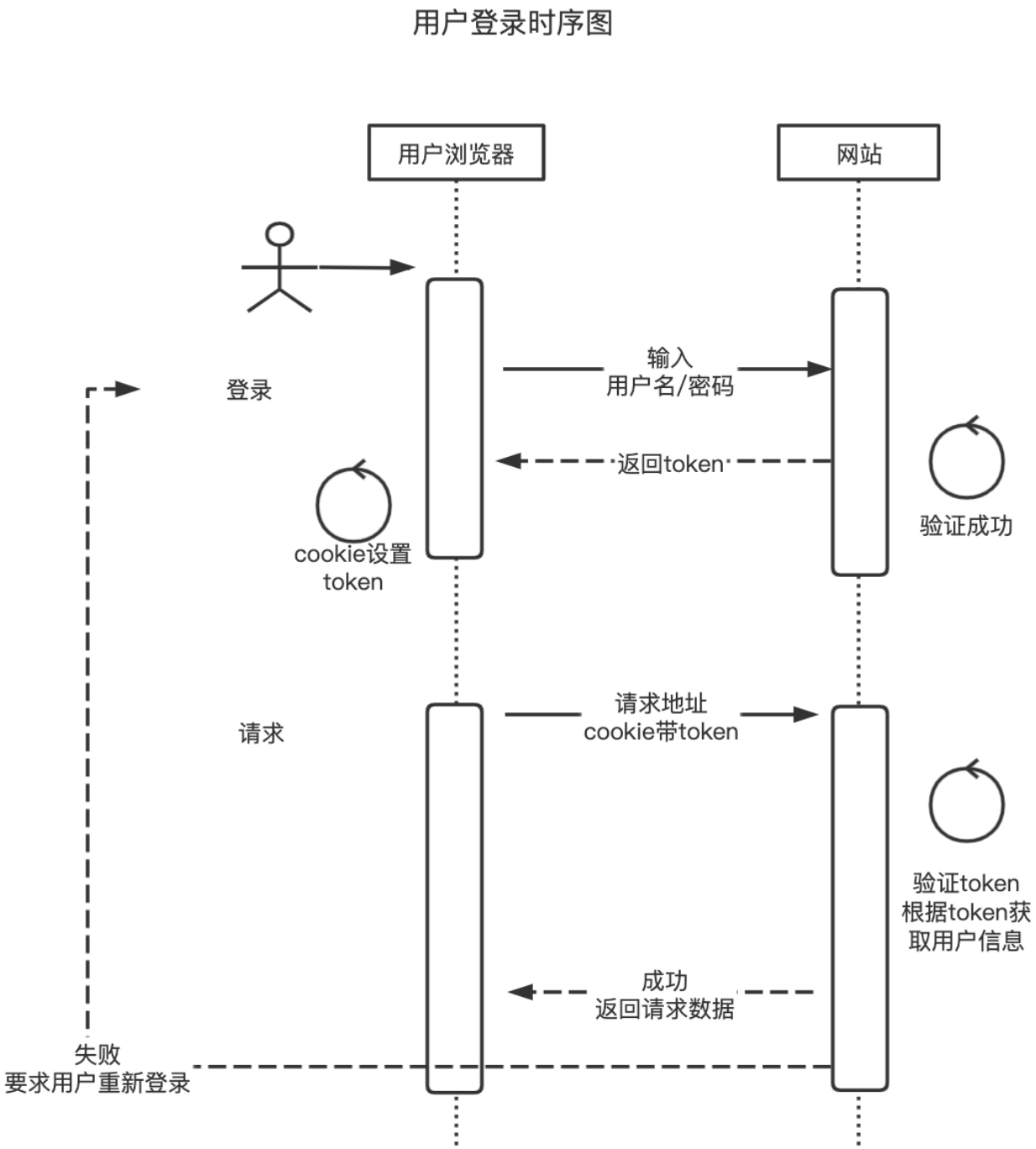
前端请求这个接口后，后端的行为应该是将用户的信息存储在临时缓存中，并且向用户注册的邮箱发送一封邮件。如果接口请求正确，返回值直接可以是一个文本字符，提示“注册成功，请前往邮箱查看邮件”。

/user/register/verify 注册验证接口

这个接口是一个在邮件中的链接，它的参数为 token、请求方法为 GET，用户点击链接就访问了这个接口。一旦验证成功，就引导用户跳转到登录页面，所以它返回的状态码为重定向 301。

用户登录

梳理完用户注册的页面和接口，我们来明确下用户登录的逻辑和接口。



还是看着用户登录时序图分析，在登录页面，用户输入注册时填写的用户名和密码，登录进入问题列表页。

而在问题列表页的头部，右上角有个登出按钮，这个登出按钮能控制用户进行登出操作。

所以用户模块的登录部分实际上要提供两个接口，登录接口和登出接口。

> /user/login 登录接口
> 在这个登录接口，参数为用户登录所需要的用户名 username、密码 password。同注册一样，这种对服务端请求的接口我们使用 POST 方法。

登录接口成功之后，服务端会生成一个代表用户登录态的 token，在缓存中将这个 token 和用户信息进行关联，然后将 token 返回给前端。所以登录接口的返回值为字符串 token。

之后就是上一节课分析过的，前端获取到 token 之后，将 token 种到 cookie 中，后续所有需要登录的请求都要带上这个 cookie；而服务端对每个请求都会验证这个带有 token 的 cookie。

> /user/logout 登出接口
> 登出接口并不需要任何的参数，但是它需要用户以登录态来进行这个请求操作。

怎么判断是登录态呢？服务端可以从 cookie 中获取这个用户的 token，并且根据这个 token 获取到用户信息，验证用户信息正确，就说明处于登录中。**所以可以在验证后将缓存中的 token 信息删除，让这个 token 实际上失效**，后续用户必须再次进行登录操作获取一个新的 token 了，这样就完成了登出的效果。

所以这个接口并没有请求参数，由于触发它的是一个链接，所以选用 GET 方法，返回值就是一个字符串，告诉用户登出成功 / 失败即可。

好，梳理一下关于用户模块的需求和接口。我们一共要实现这么四个接口：/user/register 预注册接口、/user/register/verify 注册验证接口、/user/login 登录接口、/user/logout 登出接口。

后端开发

明确了要实现的四个接口，也明确了各个接口的参数和返回值。下面正式开始我们的后端开发之路。后端开发这块，我们按照这样的步骤：

1. 接口 swagger 化
2. 定义用户服务协议
3. 开发模块接口
4. 实现用户服务协议

这种开发流程每一步都有具体意义。

由于在实际工作中，前端和后端一般是并行开发的，所以前后端都是通过接口来进行交互的。而前面分析明确的所有接口和参数都是在我们的“脑海”中，需要有一个界面将这些接口和参数都展示出来。这个时候就自然想到了 hade 框架集成的 swagger，所以第一步，我们先将接口 swagger 化，让前后端并行开发成为可行。

其次，一切皆服务，我们的用户模块也对应一个用户服务。这个用户服务其实不仅仅给用户模块使用，后续 qa 模块也会使用到。所以这个用户服务提供哪些服务协议给上层业务模块呢？这个是第二步要思考。

定义好用户服务提供的协议之后，我们如何确定这个用户服务协议是能满足要求的呢？这就是第三步。如果可以满足我们的模块接口需求，那么这个用户服务协议是 ok 的，就可以直接使用这些用户协议来开发具体模块接口了。

最后一步才是最终实现，只需要使用各种其他服务比如 ORM、cache 等，来实现我们的用户协议，最终就完成了整个后端开发。

接口 swagger

按照步骤，先实现接口的 swagger 化，回忆一下第 23 章讲集成 swagger 自动生成文件来管理接口，我们着手编写可以生成 swagger-UI 的 swagger 注释。

首先修改 app/http/swagger.go 文件，这个文件以注释形式说明 swagger 概览，包括基础调用地址、接口版本等。所以这个 swagger.go 文件，我们只需要增加关于这些信息的注释即可，格式你可以参考 swaggo 的官方文档。

复制代码

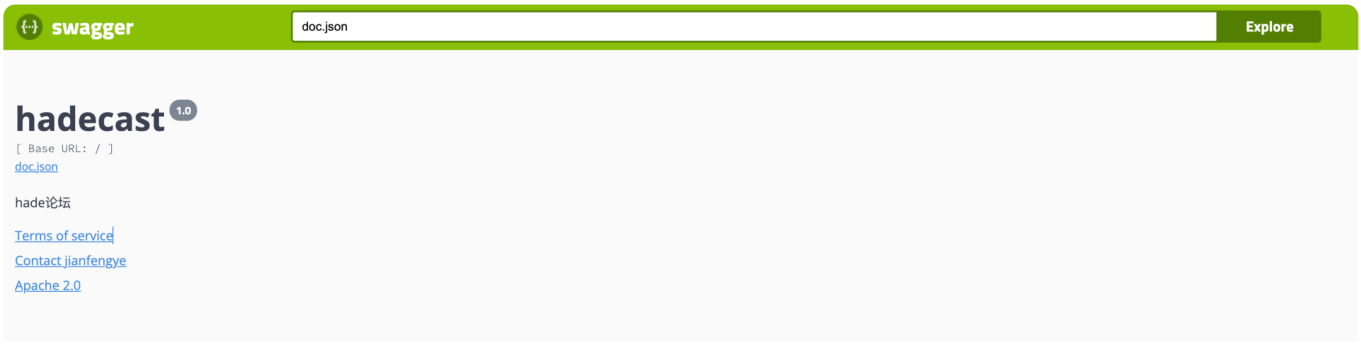
```
// Package http API.
// @title hadecast
// @version 1.0
// @description hade论坛
// @termsOfService https://github.com/swaggo/swag

// @contact.name jianfengye
// @contact.email jianfengye

// @license.name Apache 2.0
// @license.url http://www.apache.org/licenses/LICENSE-2.0.html

// @BasePath /
...
```

它最终对应的就是 swagger-UI 的头部：



接着，我们按照上节课的设计，在接口和返回对象的 app/module/user 目录中，接口不全放在一个文件里，而是一个接口用一个文件存放。那就要定义四个文件分别存放这四个接口：

app/http/module/user/api_register.go

app/http/module/user/api_verify.go

app/http/module/user/api_login.go

app/http/module/user/api_logout.go

然后定义好四个接口来实现具体的方法。关键是在方法的头部，要以swaggo的规范填写对这个接口的请求和返回值要求。这里我们就以参数稍微复杂一点的 api_register 为例，其他接口的实现大同小异，你可以参考 GitHub 上的代码。

复制代码

```
type registerParam struct {
   UserName string `json:"username" binding:"required"`
   Password string `json:"password" binding:"required,gte=6"`
   Email string `json:"email" binding:"required,gte=6"`
}

// Register 注册
// @Summary 用户注册
// @Description 用户注册接口
// @Accept  json
// @Produce  json
// @Tags user
// @Param registerParam body registerParam true "注册参数"
// @Success 200 {string} Message "注册成功"
// @Router /user/register [post]
func (api *UserApi) Register(c *gin.Context)  {
    // 验证参数
    userService := c.MustMake(provider.UserKey).(provider.Service)
    logger := c.MustMake(contract.LogKey).(contract.Log)
    param := &registerParam{}
    if err := c.ShouldBind(param); err != nil {
       c.ISetStatus(404).IText("参数错误"); return
    }
    ...
}
```

按照前面分析的，Register 接口是 POST 请求，请求参数放在 Body 体中，有三个字段 username、password、email。 所以我们定义一个 registerParam 结构来解析这些请求参数，并且使用 Gin 框架自带的 c.ShouldBind 方法，把请求 Body 体中的 JSON 结构解析为 registerParam 结构。

这里的 ShouldBind 方法我们第一次遇到，稍微介绍一下。

ShouldBind 方法是 Gin 框架提供的一个很好用的参数绑定函数，**它不仅能将请求体直接绑定到一个数据结构中，还能根据标签，对每个字段进行参数验证**。比如我们希望 Password 是必须要填写的，且字段长度必须大于 6 个字符。那么在定义结构的时候，我可以用 tag 的形式标记这个规则，就像这样：

```
1 Password string `json:"password" binding:"required,gte=6"`
```

复制代码

当然绑定 tag 的写法是要按照 Gin 的绑定规则来填写的，更多的绑定规则你可以参考官网说明。

定义好了参数结构 registerParam，我们把注意力放到函数的 swagger 注释中。在 Register 的注释中使用了这些关键字：

Summary 接口简要说明

Description 接口详细说明

Accept 接口接受的请求格式

Produce 接口返回的 response 格式

Tag 接口的标签，便于归类

Param 接口参数

Success 接口成功返回的返回值

Router 接口路由说明

注释的详细写法你需要参考说明文档。这里还是看下最复杂的 Param 接口参数说明：

```
1 // @Param registerParam body registerParam true "注册参数"
```

复制代码

这句话告诉 swag：我们的接口参数名称为 registerParam，它是一个数据结构，存储在 Body 体中，参数是必须要填写的，参数说明为“注册参数”。

这样，register 接口的参数定义就完成了，根据 23 章讲过的使用 swagger 的方法，我们通过命令

`./bbs swagger gen` 来生成 swagger 文件：

```
~/Documents/workspace/gohade/bbs  geekbang/31 ●  ./bbs swagger gen
2021/11/22 09:55:35 Generate swagger docs....
2021/11/22 09:55:35 Generate general API Info, search dir:/Users/yejianfeng/Documents/workspace/gohade/bbs/app/http
2021/11/22 09:55:35 Generating user.loginParam
2021/11/22 09:55:35 Generating user.registerParam
2021/11/22 09:55:35 create docs.go at /Users/yejianfeng/Documents/workspace/gohade/bbs/app/http/swagger/docs.go
2021/11/22 09:55:35 create swagger.json at /Users/yejianfeng/Documents/workspace/gohade/bbs/app/http/swagger/swagger.json
2021/11/22 09:55:35 create swagger.yaml at /Users/yejianfeng/Documents/workspace/gohade/bbs/app/http/swagger/swagger.yaml
```

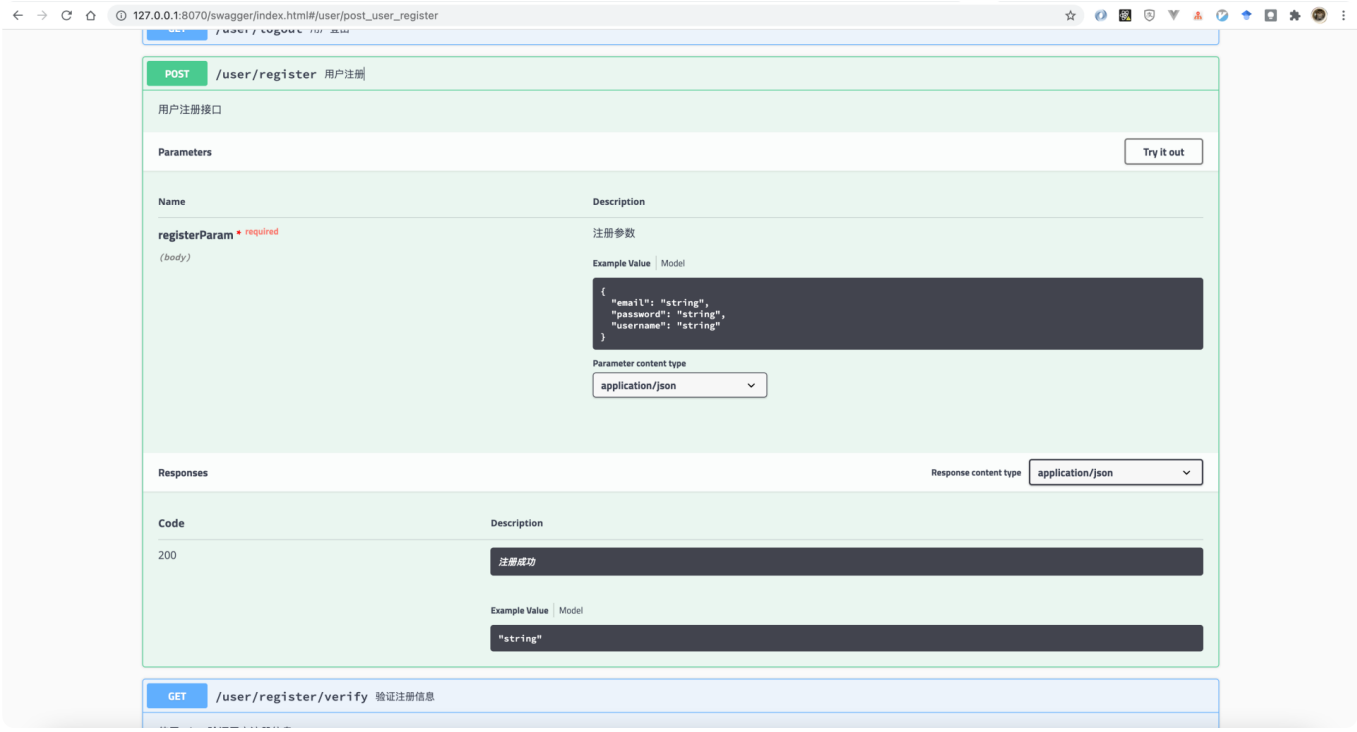
将生成的文件编译进入项目中，`./bbs build self`：

```
~/Documents/workspace/gohade/bbs  geekbang/31 ●  ./bbs build self
编译hade成功
```

然后你可以运行命令 `./bbs app start` 启动 App，或者选择调试模式 `./bbs dev backend` 来启动 swagger-UI，这里因为我们还是在调试状态，所以选择启动调试模式：

```
~/Documents/workspace/gohade/bbs  geekbang/31 ●  ./bbs dev backend
启动后端服务： http://127.0.0.1:8072
监控文件夹： /Users/yejianfeng/Documents/workspace/gohade/bbs/app
后端服务pid: 82695
代理服务启动: http://127.0.0.1:8070
[PID] 82695
app serve url: :8072
```

打开浏览器 http://127.0.0.1:8070/swagger/index.html，我们就能看到根据接口注释生成的 swagger-UI 界面了，并且后续可以通过这个界面直接调用接口请求来进行调试：



用户服务设计 - 模型

定义好接口，开始定义用户服务了。上节课介绍过了，根据一切皆服务的思想，我们已经创建好了用户模块服务，放在 app/provider/user 目录中。下面的关键是定义这个用户服务提供哪些服务。

首先，用户服务，要不要有一个用户结构来代表用户？是需要有的，我们在不同服务之间传递“用户”的实例，包含用户名、密码、邮箱等信息，后续也有可能扩展更多信息，所以**不论是为了服务传递的便捷，还是后续扩展的便捷，都有必要将用户信息封装为一个用户结构**。

所以，我们在 app/provider/user/contract.go 中定义一个 User 结构来表示一个用户，这个 user 结构中首先有 ID 字段，代表用户的唯一标识；其次有注册的用户名 UserName、密码 Password、邮箱 Email、创建时间 CreatedAt；并且由于用户注册和登录的时候会频繁使用 token，我们再定义一个字符串 Token 字段来存储注册 token 或者登录 token。

复制代码

```
// User 代表一个用户，注意这里的用户信息字段在不同接口和参数可能为空
type User struct {
    ID int64  `gorm:"column:id;primaryKey"` // 代表用户id，只有注册成功之后才有这个
    UserName string `gorm:"column:username"`
    Password string `gorm:"column:password"`
    Email string  `gorm:"column:email"`
```

```
    CreatedAt time.Time `gorm:"column:created_at"`

    Token string `gorm:"-"` // token 可以用作注册token或者登录token
}
```

不知道你注意到了没，定义 User 结构的时候，我**在 tag 标签里也增加了 gorm 的标签**，意思是这个 User 对象，同时也可以作为 gorm 操作的对象使用的。

不光是这里有点小设计，其实我在 app/http/module/user 中还定义了一个 User 的相关对象：

复制代码

```
// UserDTO  表示输出到外部的用户信息
type UserDTO struct {
    ID int64
    UserName string
    CreatedAt time.Time
}
```

如果你没有太理解，因为这里涉及上一讲提到的分层“模型”的设计，我们来仔细思考一下。

首先不管分层逻辑是什么样，我们在实现一个业务模块的时候一般会有三种模型的需求。

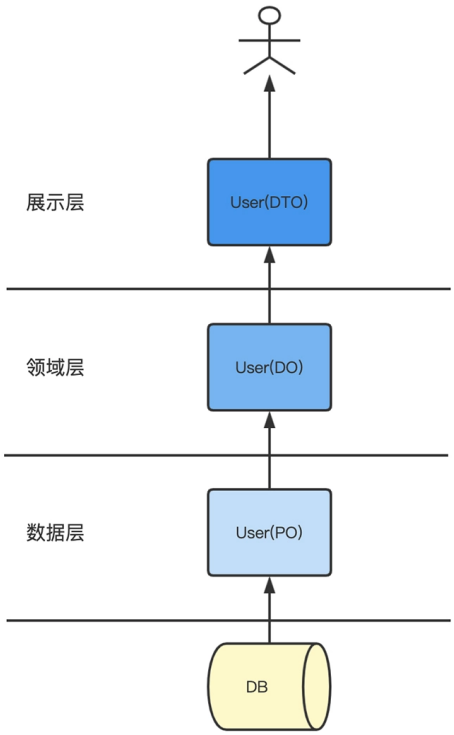
第一种模型是数据库模型，就是说这个这个模型对应数据库中的某个数据，这个也就是我们在第 25 章提到的 ORM 的概念，代码中的数据模型和数据表一一对应，这样操作这个模型就相当于能操作数据表。这种模型我们也称之为持久化对象 PO（Persistent Object），表示对象与数据库这种持久化层一一映射。

但是数据库模型和数据表关联太紧密了，一旦数据表修改，我们的数据库模型也需要对应修改，所以就会需要**第二种模型，领域模型，DO**。就是我们在业务中对某个事物的理解和抽象。

DO 模型不一定会依赖数据表的字段，而是依赖于我们对于要建立的业务的抽象。有了这个模型，我们在不同服务、不同模块之间的调用，都直接基于这个领域模型，就能保持各个模块对同一个事物的理解是一致的。

但是领域模型如果直接输出给用户，比如 Web 前端用户，有一些是需要进行加工的，比如一些涉及安全的字段需要隐藏、一些字段类型需要转换等。所以输出给前端的是**第三种模型，输出模型**，我们经常叫它 DTO，表示最终在网络上传输的数据对象。

这三种模型从底层到上层依次为 PO、DO、DTO。不管我们的业务是如何分层，基本上绕不开这三种模型的设计。



估计从这段分析中，你也能想到这三种模型之间是存在转换关系的，它们的转换也是一个比较繁琐的过程，比如 DO 转换为 DTO，记得吗我们还专门设计了一个映射层，就是 app/http/module/user/mapper.go 中定义的各种映射方法。

那有没有节省代码量，快速开发的方法呢？有的，就是将这三种模型合并。

至于怎么合并，不同的业务有会有不同的选择了。有的人会选择将领域模型和输出模型合并，就是直接将领域模型输出给前端业务；也有的人选择将领域模型和数据库模型合并，在领域模型中增加数据库的操作字段；当然有更极致的设计将这三种模型统一合并，使用一个模型，不仅操作数据库，也操作前端返回值。

不同的合并策略各有好处和劣势。不过好处基本一样，能有效降低某些层的代码量，提升开发效率。**而缺点也差不多，就是合并之后，一旦合并的两层有不同的需求，修改起来可能就会非常复杂了。**比如将领域模型和输出模型合并后，如果不断有需求要修改输出模

型，原先的 name 字段要修改为 username，或者原先输出的电话号码要将某些位置替换为 * 号，又需要将其分离。

所以三层不同“模型”的优化，实际上需要你对业务有足够的理解，才能精准判断出大致哪几层合并后在“未来”不会有重构的需求。

比如这里我实际上创建了两个模型，UserDTO 和 User，也就是选择将数据库模型和领域模型合并。因为我觉得我们这个网站的数据库模型比较简单，并且基本上和业务领域中的用户逻辑是一致的，只有一两个字段会稍微有些不同。所以我有把握，后续对“用户”这个业务逻辑的需求，都不会影响数据库和业务领域的一致，就做了这个选择。

但是提醒一下，如果在你的业务中某个逻辑比较复杂，比如“用户”这个逻辑基本上要由几个数据表才能得出一个完整的用户逻辑，那么这种选择就是错误的了。

另外坚持将 UserDTO 单独分开，这也是我的经验之谈。**越接近前端，需求变化越频繁**，修改 / 增加 / 删除某个前端展示字段的需求在实际工作中比比皆是。所以这个输出层模型我一般习惯单独写出来。

好讲到这里，相信你就可以理解在“用户服务”中定义模型 User 会同时带着 gorm 标签的含义了，就是代表，我会使用这个模型来操作数据库的。

后端开发剩下的两个步骤，我们下一节课再继续讲解。这节课所有的代码都上传到 geekbang/31分支了。你可以对比查看。

小结

分析用户模块的注册和登录两个部分后，我们开始开发后端了，但并不是一上手就开始接口逻辑编写，也不是一开始就考虑我应该用数据库还是缓存实现用户存储。而是依照四个步骤：

1. 接口 swagger 化
2. 定义用户服务协议
3. 开发模块接口
4. 实现用户服务协议

hade 框架的整体都是不断在强调“协议优于实现”，接口是后端和前端定义的协议，用户服务是后端模块与模块之间定义的协议。对于一个业务，最重要的是定义好、说明清楚这些协议，然后才是实现好这些协议。希望 hade 框架带给你的不仅仅是工具和功能上的便利，更多是开发思维和流程的转变。

思考题

看完关于“模型”的讨论，你一定会有很多观点想要交流，比如你的实际项目中，业务架构师是怎么分层的，在每一层是否有定义对应的模型，是否有合并不同的模型？具体业务中的模型设计方式你是否认可？如果你来重新设计一个模型设计，会怎么设计？

欢迎在留言区分享你的思考。感谢你的收听，如果觉得今天的内容对你有所帮助，也欢迎分享给你身边的朋友，邀请他一起学习。

上一篇 30 | 设计先于实战：需求设计和框架搭建

下一篇 32 | 通用模块（下）：用户模块开发

精选留言 (1)

写留言

宙斯

2021-12-02

看项目情况，老项目或迭代项目并没有做过多考虑，model层就直接和数据库和业务服务层打交道，写业务时会做过多的传输数据的转化。
不复杂的新项目会做基础层，存储层，服务层，聚合层，领域层，访问层，其中通常存储层，服务层，领域层会给出传输对象，不过这也不绝对视情况而定。

展开