# **领域驱动最佳实践--用代码来告诉你来如何进行领域驱动设计**

做一个租户系统下的权限服务，接管用户的认证和授权，我们取名该服务为go-easy-login

     本文实质是[领域驱动设计之实战权限系统微服务](https://juejin.im/post/5cd3d1a8f265da034c7042c6" \t "https://juejin.im/post/_blank)的进一步总结和改进，学习领域驱动设计本身是循序渐进的过程，培养的是领域的概念和面向对象编程思想，而过去以及现在，包括未来，多数人只是披着面向对象的皮，干着面向过程，面向数据库的糙活，详情请看[为什么我们需要领域驱动设计](https://juejin.im/post/5ca16166e51d456708675612" \t "https://juejin.im/post/_blank)，如果你接触过领域驱动设计，但是苦于不知道如何动手，概念虽懂但不知如何实践，本篇将能为你打开实践领域驱动设计的大门，如果你未曾了解过领域驱动设计，这篇同样也是入门领域驱动设计的最好文章之一，带你感受领域驱动的非凡魅力。

#### 项目结构

     代码先行，先展示一下代码的目录结构以及相应的文件，大家可以先YY对应的作用，然后带着疑问去阅读。

login

base

encrypt.**go**

token.**go**

repoImpl.**go**

domain

service

loginService.**go**

loginService\_test.**go**

loginUser.**go**

loginUser\_test.**go**

mocks

EncryptHelper.**go**

LoginUserRepo.**go**复制代码

#### 如何脱离技术细节

     领域驱动设计更加强调业务逻辑以及相应对建立起的领域（模型），不应该出现任何 技术细节，即数据库，缓存等。面向对象是对于外在客观事物的一个模拟和反映，一 个User类应该具备eat,drink,play,happy等能力，一个User不可能具备连接数据库的能力，出现依赖任何技术细节是违反面向对象编程的。那么问题来了，道理我都懂，如何去做到，如果我们在一个项目中，****什么技术都不用到的话****，是不是就达到我们的目的了？（读者疑问：WTF,怎么可能？）

     新建一个项目，什么第三方包都不依赖，根据我们想做的功能，****做一个租户系统下的权限服务，接管用户的认证和授权****，我们新建一个LoginUserE来代表登陆用户，DoVerify执行认证过程，同时我们希望具备帐密登陆的时候，由调用系统决定加密方式，这就意味着LoginUserE这个领域需要可以根据EncryptWay来获取EncryptHelper，我们先新建一个loginUser.go

**package** domain

**type** LoginUserE **struct** {

Username **string**

IsLock **bool**

UniqueCode **string**

Mobile **string**

canLoginFunc **func**() **bool**

**EncryptWay**

}

**func** (user \*LoginUserE) **CanLogin**() **bool** {

**var** can **bool**

**if** user.canLoginFunc != nil {

can = user.canLoginFunc()

} **else** {

can = !user.IsLock

}

**return** can

}

**func** (user \*LoginUserE) **DoVerify**(sourceCode **string**, encryptedCode **string**) (**bool**, error) {

**if** !user.CanLogin() {

**return** false, errors.New("can not login")

}

match := user.EncryptHelper().Match(sourceCode, encryptedCode)

**return** match, nil

}复制代码

     这里的问题在于EncryptHelper()这个方法，我们知道加密方法，就拿MD5来说，必须需要依赖到其他包，而loginUser.go我们是不希望依赖到任何第三方包的，这似乎进入了一种矛盾。Alistair Cockburn 提出的六边形架构，****在于domain处于核心内部，其他的依赖通过接口进行交流****，再换句话说就是****domain层定义接口，基础设施层（技术层）实现接口****，我们定义EncryptHelper接口

**type** EncryptHelper **interface** {

Encrypt(password **string**) **string**

Decrypt(password **string**) **string**

Match(source, encryptedString **string**) **bool**

}复制代码

     然后在base基础设施层新建encrypt.go实现该类

**type** MD5Way **struct**{}

**func** (md5 MD5Way) **Match**(source, encryptedString **string**) **bool** {

**return** md5.Encrypt(source) == encryptedString

}

**func** (MD5Way) **Encrypt**(password **string**) **string** {

data := []**byte**(password)

md5Bytes := md5.Sum(data)

**return** **string**(md5Bytes[:])

}

**func** (MD5Way) **Decrypt**(password **string**) **string** {

panic("not support")

}复制代码

      问题还没能够解决，****base层的具体实现类，如何让domain层中不直接依赖的同时，又能使用呢？最好的方法实际上是依赖注入****，但是引入依赖注入又陷入另一种悖论--不依赖任何技术细节，依赖注入也可以归纳为技术的一种，下文再继续探讨这点，且看我如何不用依赖注入实现。在loginUser.go我们新建一个全局变量var EncryptMap = make(map[EncryptWay]EncryptHelper)

**var** EncryptMap = make(**map**[EncryptWay]EncryptHelper)

**func** (encryptWay EncryptWay) **EncryptHelper**() **EncryptHelper** {

**if** helper, ok := EncryptMap[encryptWay]; ok {

**return** helper

} **else** {

panic("can not find helper")

}

}

**func** **AddEncryptHelper**(encryptWay EncryptWay, helper EncryptHelper) {

EncryptMap[encryptWay] = helper

}复制代码

     核心层之外的类，通过AddEncryptHelper注册相应的EncryptHelper,这种设计初看尚可，但是一旦项目中具备更多个领域，再采用这种方法则会导致代码的维护成本的提高，****依赖注入实则是屏蔽构建具体实现类的过程****，要不要在domain层引入，因人而异，因项目而异。若有更好的方法，欢迎在评论区中提出。

#### 领域服务

     若你是初涉或者从未涉及过领域驱动设计，你的思维会比较固定，如[为什么我们需要领域驱动设计](https://juejin.im/post/5ca16166e51d456708675612" \t "https://juejin.im/post/_blank),长期以来你习惯以数据表为核心进行分析设计，想着某个功能我们应该如何建表。我敢打包票，在上面讲解的功能过程中，你一开始就在思考这个表怎么设计的，我从表中取哪个字段再怎样怎样，这是绝对的****被数据库，被技术绑架了的思维****但是，问题也在于，我们最终总是需要解决数据库这个问题，这个问题也就是在领域服务中解决。 领域服务解决的另一个问题是****组装逻辑****，举个例子，LoginUserE.DoVerify虽然不依赖任何第三方包或者同级的其他类，但是他的入參被我们依赖隔离，这个入參可能就依赖其他domain，因此，我们需要领域服务去组装这一层。 我们来讲一下登陆是如何实现的，首先我们定义LoginCmd为Login的入參,

//implemention will show right behind this**func** (service \*LoginService) **Login**(loginCmd common.LoginCmd) (**string**, error)

**type** **LoginCmd** **struct** {

Username **string**

TenantId **string**

EffectiveSeconds **int**

Mobile **string**

SourceCode **string**

LoginWay **string**

EncryptWay **string**

}复制代码

     这里就到了设计数据库的地方，我们需要去查找判断这个用户是否存在，那么问题来了，我们不能直接依赖数据库技术，但是我们又需要，这可咋整？类似的当然是****定义接口****

**type** LoginUserRepo **interface** {

GetOne(username, tenantId **string**) \*domain.LoginUserDO

}复制代码

但是这里又回到了上文讨论依赖注入的地方了，这里我为了简单起见，仍然没有用到依赖注入，但是我个人是建议使用的

**var** loginService \*LoginService**type** LoginService **struct** {

LoginUserRepo

}**func** **NewLoginService**(repo LoginUserRepo,) \***LoginService** {//do not argue to use double check lock,it's a example and does not hurt anyway

**if** loginService == nil {

**return** &LoginService{

LoginUserRepo: repo,

}

} **else** {

**return** loginService

}

}复制代码

     这样我们就在初始化LoginService的时候将repoImpl传送进行，达到了依赖隔离的目的。****领域服务不需要知道任何仓储手段，甚者无需知道底层用的是什么数据库，我只关心取和拿，我只要结果，定义接口的实质目的也在于此。****

      回到GetOne(username, tenantId string) \*domain.LoginUserDO这个方法，这里还暴露了一个点在于，DataObject类是定义在domain层中，而不是在service，更不是在base中，我以前纠结的一点是，既然domain层不依赖数据库技术，是不是也应该不关心DataObject，DataObject是不是放在base层下更加合适？

     现在之所以把DataObject放在domain层，原因在于

1.domain核心层不直接依赖其他层，如果DataObject放在base层势必违背这点；

2.domain层作为接口定义者，有权根据他自身的需求定义他想要的存储内容，其他层只需要服从并且实现。复制代码

     同时，我们不希望代码中充斥着大量的convert，从cmd转到DO，从DO转到E，所以我们提炼出了dto.go这个文件，用于存放concert代码。最终的代码形式如下。

**func** (service \*LoginService) **Login**(loginCmd common.LoginCmd) (**string**, error) {

userDO := service.GetOne(loginCmd.Username, loginCmd.TenantId)

userE := common.ToLoginUserE(\*userDO)

userE.EncryptWay = domain.EncryptWay(loginCmd.EncryptWay)

//login way contains PASSWORD and SMS ,encryptCode（）is to get which one to be verify ,so userE will not to care about which way is exactly by logining

encryptCode := service.encryptCode(loginCmd.LoginWay, userDO)

**if** \_, err := userE.DoVerify(loginCmd.SourceCode, encryptCode); err != nil {

**return** "", err

}

//todo add login event and callback

**return** service.token(userE.UniqueCode, loginCmd.EffectiveSeconds), nil

}

**func** (service \*LoginService) **encryptCode**(way **string**, userDO \*domain.LoginUserDO) **string** {

**switch** way {

**case** "PASSWORD":

**return** userDO.Password

**case** "SMS":

**return** service.FindSmsCode(userDO.Mobile)

**default**:

panic("unknown login way")

}

}复制代码

      新的风暴又出现了，service.token(userE.UniqueCode, loginCmd.EffectiveSeconds)这段逻辑是什么意思，上文中也没有出现，待我慢慢需讲解。正常登陆下我们校验成功之后需要授予token，但是token的生成技术细节,用JWT还是什么其他的，domain不应该关心，所以我们给loginService 加一个类型为函数的field

**type** LoginService **struct** {

LoginUserRepo

token **func**(uniqueCode **string**, effectiveSeconds **int**) **string**

}

**func** **NewLoginService**(repo LoginUserRepo, token **func**(uniqueCode **string**, effectiveSeconds **int**) **string**) \***LoginService** {

**if** loginService == nil {

**return** &LoginService{

LoginUserRepo: repo,

token: token,

}

} **else** {

**return** loginService

}

}复制代码

#### 最终效果

****拷贝不走样****，屏蔽技术细节，强调业务逻辑，最终目的是实现业务逻辑可重用，组织为一个可重用的自封闭的业务模型。最终我们很好的构建了这样的一个模型。

     这个业务模型无论置身于任何技术框架，任何Web框架，还是其他的场景，都不会受到破坏，无论选择任何数据库技术，也不会影响到这个模型。外在技术的细节这里就不跟着大家一起实现了，本篇文章重在构建模型，技术的选择就由自己去做决定，这也丝毫影响不了模型。

#### 测试驱动使领域驱动更加完美

     全篇下来的奥义在于隔离依赖，这些都是经验积累，有没有行之有效的规范得以遵守，答案是我也不知道，但是如果你遵守测试驱动的行为的话，这会迫使你去思考，什么该依赖，什么不该依赖，****因为所有的第三方依赖，都需要用Mock去代替****，这就是为什么目录中存在mocks这个文件。测试写得好，烦恼多不了。

#### 总结

     回头看这是我写的第三篇涉及领域驱动设计的文章，目的在于能够让更多的人更加容易理解并且实践领域驱动设计，写出优秀的代码，得出接任者的称赞，提高代码质量。

     路漫漫其修远兮，看官点个赞呗！

     作者：plz叫我红领巾

     出处：[领域驱动最佳实践--用代码来告诉你来如何进行领域驱动设计](https://juejin.im/post/5d7770faf265da03e05b3c63" \t "https://juejin.im/post/_blank)

    源码地址：[github.com/iamlufy/go-…](https://github.com/iamlufy/go-easy-login" \t "https://juejin.im/post/_blank) 代码会逐步完善，可以看看git提交记录～

    本博客欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。