# **关于领域驱动（domain-driven desgin, ddd）开发 概述**

关于领域驱动开发，想写一个系列的文章。分享一下近两年在项目中运用ddd的经验，和自己的一些理解，教训。

第一篇文章先概述一下。

领域驱动设计 domain-driven design. 简称ddd。

在岛国这两年是一个比较火的名词，明明是一个有着多年历史的概念，不知道为什么过了那么久突然流行了起来。天朝的情况不是很了解，不过无所谓啦，思想啦理论啦不存在国界的！

什么是ddd?

这一个概念来自Eric Evans所著的<domain-driven desgin>一书。（P.S. 此书极为枯燥，而且理论繁多，阅读的障碍很大。）书中并没有很正规地用下定义来描述各种名词，所以个人认为导致了对领域驱动究竟是什么的的多种解释。

在这我说一下自己的理解（仅限于自己理解）

领域（以下都称domain）指需要解决的现实问题（业务），为了解决现实问题（domain）,我们会建立模型(以下称model)来帮助我们描述与理解domain，从而能够应对domain。

纯理论地说，domain是问题空间，model是解决空间（虽然一点都不好懂）。

比如，要开发一个考勤管理系统，考勤中的一些实际业务如打卡，休假申请就是domain。我们会建立一些model来应对这些domain。比如建立休假申请的model，然后赋予它一些状态，来描述休假申请这个业务的执行。这种domain与model的对应比较简单。如何管理考勤，已经有现有的经验与管理方式，从这个意义上说帮助我们理解的抽象的model已经存在，我们只是为了软件设计的方便对model在进行加工。

但如果一些复杂的问题，domain与model的关系就会不那么一目了然，比如要评价学生是一个现实课题（domain）。那用学生的考分进行评价就是一种model。

那domain驱动设计，顾名思义，由从domain出发，来设计，建模。

而ddd的大致流程便是

1 理解domain

2 建立应对domain的model （domain model）

3 根据model开发。

另外，ddd中，model一般特指domain model。即刚才说的应对domain的model。

为什么需要ddd?

这种看似理所当然的设计方式，为什么要像当作课题一样提出呢？

1 业务(domain)的复杂性，让理所当然的事情，做起来不那么简单

 没能完成需求的项目，必定是失败的。(当然项目的失败也有可能是其他原因，这不属于ddd讨论范畴。)当domain十分复杂时，我们必须有好的方法去理清，理解domain，并根据他来开发。

2 人们设计时当然是离不开model的。但model其实也是分好几种。比如domain model（领域模型）与data model（数据模型）。

 有时候会更倾向于对data model的思考，比如拿到需求就开始搞数据表设计，并不是说data model不好或者不重要，而是跳过了domain model，直接考虑data model，容易导致建立的model没能很好的应对实际的domain。这个以后会重点讲！

3 软件的开发，纵向分工过多，每一次分工都会建立新的model来教给下层的开发人员来实现，结果导致开发人员对domain的理解不正确，最终导致产品不能实现业务。

 从前（现在也可能）的软件开发行业，存在各种外包。有的软件公司只负责设计，然后把设计文档人给其他公司编码。这种现象在岛国的软件开发行业非常常见，以至于岛国的互联网行业。。。大概的流程会是需求分析，上流设计，详细设计，编码。

综上，ddd所提倡的是，不要建立过多的model，而是建立与domain最接近的domain model，然后根据domain model来进行开发。model必须尽量反应实际的domain。

ddd真的有效吗？

这个问题非常难。。。如果你去网上搜一下ddd的成功案例。结果会是。。。（应该没有）

也有种说法是ddd所追求的是对domain model的不断深化改造，所以在应用ddd开发的人们眼里，达到什么程度算是成功是不明确的。

从本人目前的工作经验上说，有项目运用了ddd，是否实现了ddd的精髓有待商榷，但开发者有了共同的设计思想和交流语言，使设计容易传达，说了更明显一点，就是代码容易读，容易修改。但有个人觉得这种效果其实需要非常多的前提，以后的文章会进一步讨论。

另外，有时候理论是用来解决某一些特定问题的。比如ddd提倡根据domain model来开发，防止对需求的认识偏差。但如果项目开发中，开发人员的需求的理解偏差不是一个很大的问题，那ddd在这个问题上就没有太多发挥余地了（个人认为）。当然除了这个ddd当然还有其他的作用，以后会介绍。

但底线是，我们可以通过学习ddd所提倡的思想和一些设计手法，帮助我们做一些更好的设计。

今后的内容

之后会写更多ddd的内容。顺序的话并不按照domain-drive design书中的章节（比如准备先讲战术，再讲战略）。而是类似于还原自己如何去接触ddd的过程，这样可能更容易理解。对于想学习应用程序架构，程序设计的人来说，可能是一种比较自然的理解/学习过程

目前能想到的大概有

ddd的战术: 分层设计

ddd的战术: entity, VO, aggregate, repository

ddd的战术: 如果没有domian model程序会是怎样的？

ddd的战术: domain event

让ddd发挥自己的优势，CQRS！

ddd的战略: bounded context

另外列一下推荐的参考书

implementing domain-driven desgin （比较通俗易懂）

domain-driven desgin distilled

domain-driven desgin （如果你想抱着试试的态度，千万别先读这本书，过早尝试会让你对ddd的兴趣丧失殆尽）

————————————————

版权声明：本文为CSDN博主「even\_he」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/abchywabc/java/article/details/79301163>

# **ddd的战术: 分层设计和应用架构**

应用程序的架构是个亘古不变的话题。

其中一个最典型的架构便是分层架构。分层是程序设计由来已久的一个概念。可以说肯定不是ddd所提出的，但ddd的角度，它对分层有着什么特别的做法呢？

注: 文章里的代码例子需要对ddd的domain object有了解。之后写文章专门讲解domain object。如果现在看起来太累，可以先草草读过，以后可以参照domain object的文章一期阅读。

分层

首先抛开ddd，简单说一下分层。

\*题外话，岛国人对分层的执着可能已经超越了分层的意义，他们根本就是为了分层而分层。然后各种层的对象转化几乎占了代码的大部分。。。

分层的思路就是把程序的流程进行纵向分割，然后每一层有每一层的职责，从而每一层干的事情就比较明确。

理论上理想的分层，对依赖关系处理地比较好的话，层是可以比较容易更换的，这就使重构什么的变得简单。

严格的分层

上层只准调用离自己最近的下一层

 比如从上到下有a, b, c三层。a只能调b,a不能调c

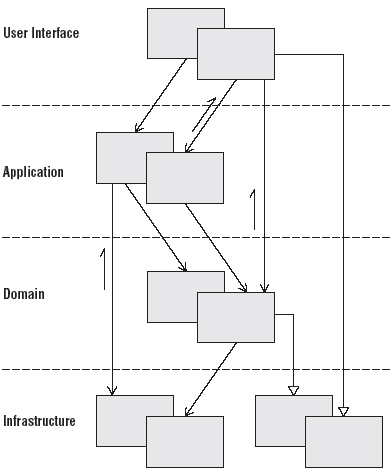
非严格的分层

 上次可以调用任何下层

 比如从上到下有a, b, c三层。a只能调b, c

ddd建议的一种分层

自上而下，依次分UI层，application层，domain层，infrastructure层



这是一种比较传统的分层架构

a. 最简单的domain层，就是指包含所有业务逻辑的层。职责：业务逻辑。

b. infrastructure层，最底层的实现层。是现在domain层定义的一些接口。职责：底层实现。最常见的当然是db的交互。domain层指关注业务，具体的db交互实现交给infrastructure层。

c. application层，迷惑性很大的层，很容易与domain层搞混。application层是提供应用程序（application）与domain对接的层，或者说他是一种程序的入口。但它不包含业务逻辑！！！ 但即使这么说，碰到具体问题时还是会混淆。

写个具体的例子。假如说写取款这个逻辑。

public class TransactionApplicationService{

@Autowired

private IAccountRepository accountRepository;

public void withdraw(AccountId accountId, Integer amount){

Account account = accountRepository.find(new AccountSpecificationById(accountId)); // 找到用户账户

account.withdraw(amount); // 取款

accountRepository.save(account); // 保存账户状态

}

}

private AccountId accountId;

private Integer balance;

public void withdraw(amount){

if(balance < amount){

throw new IllegalStateException("balance is not enough"); // 余额不够就不能取款

}

balance = balance - amount;

}

}

TransactionApplication属于application层。

Account，IAccountRepository这些类属于domain层。

infrastructure里会有IAccountRepository的具体实现，这里没有写

请注意取款的具体逻辑是在Account类里的。如余额够不够，和金额的具体计算（虽然很简单）。所以业务逻辑是在domain层。

TransactionApplication只是把具体的domain object找到，调用其开放的接口，不涉及具体逻辑。上边的例子找到用户的账户，告诉账户对象扣钱（具体怎么扣不管），保存账户的状态。

如果是下面的写法就值得商榷，有人认为account.getBalance() < amount的判断暴露了业务逻辑。也有人认为只要在account里有逻辑保证数据的正确性，这种写法也是可以接受的。我觉得在applicationService尽量减少判断逻辑，因为容易让人误会业务逻辑进入了application层。

public class TransactionApplicationService{

@Autowired

private IAccountRepository accountRepository;

public void withdraw(AccountId accountId, Integer amount){

Account account = accountRepository.find(new AccountSpecificationById(accountId)); // 找到用户账户

if(account.getBalance() < amount){

return；

}

account.withdraw(amount); // 取款

    accountRepository.save(account); // 保存账户状态

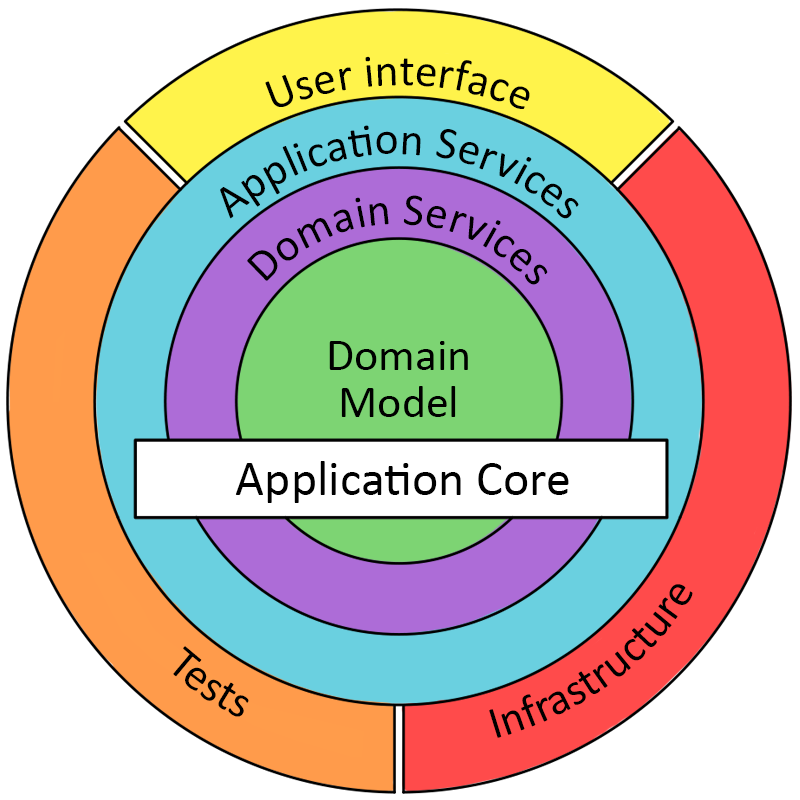
}

}

另外applciation层是有个连接UI层和domain层的作用。

比如我们用springMVC开发一个web应用。spring里的Controller只允许与application层（各种applicationService）交互。Controller直接访问domain对象（比如上面例子的Account, IAccountRepository）是不太好的。

洋葱架构



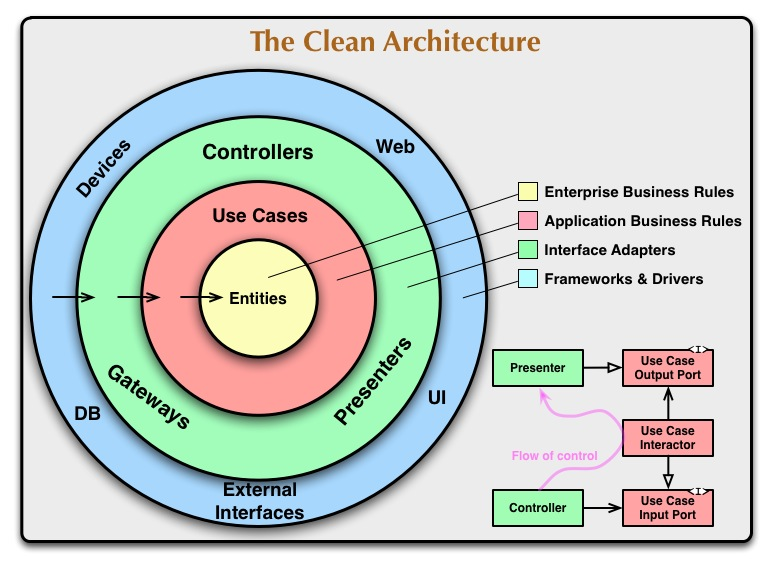
这个大家可以谷歌一下，因为《domain driven design》这本书出来的时候，可能还没有洋葱架构，书中自然没有探讨。但在《implementing domain driven design》中作过介绍。

大概的意思，像洋葱是一层包一层，最核心的是domain，外层application，再外层是UI呀data persistence等。

嗯？infrastructure去哪了？它被弄到了外边的层了。我觉得这可能也是中比较自然的结果。

回看一下上边的自上而下的分层。domain层会依赖于infrastructure层。但一个应用最核心的部分显然是domain层。domain层依赖某种实现是不合适的，所以（可能）给予控制反转的思想，在domain 层定义了接口，然后交由infrastructure来写。这样依赖关系就反转了，infrastructure依赖于domain定义的接口。（其实上边代码的例子，本人也是这么写的。）

clean architecture



这个中文不知道是什么，感觉是洋葱架构和六角形架构（六角形又是什么？）的混合加强版。和洋葱架构一样，也是一圈一圈的包起来，最核心的当然还是domain。感觉每一圈的名词和洋葱架构有些不同。比如use cases。

总结

ddd在分层上并没有严格规定要用哪种特定的架构。当然如果现在开始做项目的话，洋葱架构当然是很合适的。clean architecture与洋葱相比，感觉洋葱的一些名词更偏向ddd，容易理解。clean architecture它的思想应该不仅限于ddd。

其中很重要的一个思想便是domain层的重要性。domain是产品的核心，必须尽量减少它对外部的依赖。所以在写domain层时，我们尽量使用pojo类。

下一篇文章会介绍domain object，很重要哦！

————————————————

版权声明：本文为CSDN博主「even\_he」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/abchywabc/java/article/details/79301928>

# **ddd的战术篇: domain object之一**

首先ddd的战术这个讲法是不太好的。ddd书中说的是战术性建模（tactical modeling）。其意思是在战术层面的建模，那当然有战略层面的建模啦。以后会专门讲。

domain object是ddd区别于其他建模/设计方法的一个部分。他定义了概念帮助我们去建立model。如果没有接触过ddd，一开始会觉得有点莫名其妙，为什么要定这种规矩，这种规则？

Entity，value object, Aggregate

entity

entity是一类可识别的可追踪的对象。

说简单了，它必须有identifier，再简单点id（可识别，可被追踪）。

另外它是有可变对象，mutable。但即使状态变化了之后，entity还是原来的entity。（好绕口）

  现实一点的例子。一个人家“王帝”，他改名交了“王皇”，名字虽然变了，但还是同一人。当然现实世界里我们很难去寻找一个identifier(识别码)，如何定义一个不变的identifier会是个哲学问题，但编程时就简单多了，直接搞个id就就行。

value object

与entity对应的一个概念叫value object

它是一个值，是不可变的，immutable。没有identifier，也不需要被追踪！

  比如java中的字符串和value object的感念很相近。字符串生成之后就是不可变的。而且也没有什么id来识别它。

什么时候使用entity，什么时候使用value object

具体问题具体分析

比如我们需要对地址这个东西建模。如果我们关心的是地址的履历之类的信息，过去30年前这个地址可能叫霞飞路，现在可能叫淮海路，而且需求是我们必须知道霞飞路，淮海路指的是一个地址。那很可能我们需要的是entity。这个entity可能还要开发change()的方法来改变路名。

但如果我们做的是一个送货软件。地址只是表示一个目的地而已，霞飞路和淮海路在我看来就是不同的，那就说明，你不必对地址本身的变化进行追踪（送货地址变了，对你很重要。但霞飞路改名成淮海路对你不重要。）。那value object就够了。

  你可能觉得何必这么费事搞个表示地址的value object, 搞个字符串不就行了？

   首先这不一定是一个关于ddd的问题。淡然很多情况下，我们可以的确用java的基础类来建模的。但从面向对象的封装角度来说，我们可以考虑创造专门的类。

   如果我们创建value object,它是可以拥有行为的。比如Address是一个value object。它可以拥有getCountry(), getProvince(), getCity()等方法。

Aggregate

Aggregate，集合。是有多个（也可以是一个）entity，value object组成的对象。

Aggregate可以看作一个树状结构的东西。根是一个entity。Aggregate的一个作用是保持domain object的关联性的正确。

Aggregate和Repository

Repository

Repository是用来存放一个aggregate的object。（是不是听起来像DAO？）

Repository的存在，让我们感觉我们存放在它里面的aggregate就好像on memory一样，不必去关心它具体是和rdb对接还是其他的形式。

请注意的是Repository对应的是aggregate而不是entity！虽然我们在实际偏码时不会去专门写aggregate的类，有关具体实现，之以后的文章会写。简单地讲根entity就可以代表aggregate。

domain object的做法与active record的区别

active record pattern也是很常见的一种模式。大概是建立于数据表的一一映射的类，然后有各自的DAO类。乍看起来和domain object的做法是一样的。

1. 映射关系

active record一般以一个类对应一个数据表为前提

repository则没有这个限制。当一个aggregate需要对应多个数据表时，那repository自然就对应多个数据表。repository隐藏了数据层的物理实现。

这里想多提一点。domain model的建立，称作理论设计比较合适。它尽量不关心物理的实现，只求对domain的正确反映。

而data model的设计，可以称作物理实现。是以数据永久化为目的。与ddd相关联，自然就是如何将domain object永久化。

2. domain object主张充血模式

active record常常会导致制造大量的于数据表对应的类，这些类一般不会有行为，而描述业务逻辑的职责就会到很多service类上。

domain object则希望类有自己的行为，而不是开放好多setter，让外部的类来控制自己的状态变化。

总结

讲了好多理论，大致介绍了domain object。

entity: stateful, mutable.

value object: immutable

aggregate: entity和value object的集合

repository: stateless，储存aggregate

之后的文章准备写一些例子来更好地说明这些概念。

————————————————

版权声明：本文为CSDN博主「even\_he」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/abchywabc/java/article/details/79304476>

# **ddd的战术篇: domain object之二**

前一篇文章介绍了domain object。这篇文章写一下具体的例子。

假设我们要实现一个账户管理的密码修改功能。

大致需求

账号通过邮箱地址来识别（identifier）

　- 密码不能为空

　- 密码必须加密（这个功能暂不实现）

修改密码

　- 修改密码时必须输入旧密码，旧密码输入正确后才可更改密码。

从aggregate开始

其实建模是一个过程，而且可能是程序设计里比较难有容易被忽略的过程。就像很多侦探片里的推理一样，推理很难，但往往侦探片只会告诉你答案，而不是告诉你如何推理的。很抱歉我这里也讲不了建模的过程，这回事另一个话题。但大概我们会想到如下的model。

Account aggregate

Account (entity, Account aggregate的root)

  因为密码可以更改可以理解为mutable，密码变化后，不代表账号就变成了另一个账号。所以把它定义成entity。

AccountId (value object)

  账号id生成后就不可变了。

Password (value object)

  密码设置后，密码本身是不可变的。

  更改账号密码应该理解为，用新密码换掉账号的旧密码。而非旧密码本身做了变化。

然后对应aggregate，会有AggregateRepository

具体的实现

Account

public class Account {

private AccountId accountId;

private EncryptedPassword encryptedPassword;

public void changePassword(String oldPassword, String newPassword){

if(encryptedPassword.verify(oldPassword)){

throw new IllegalArgumentException("old encryptedPassword is not correct");

}

encryptedPassword = new EncryptedPassword(newPassword);

}

public static Account createAccount(String email, String password){

Account account = new Account();

account.accountId= new Account(email);

account.encryptedPassword = new EncryptedPassword(password);

return account;

}

private Account(){

}

}

AccountId

@AllArgsConstructor

@Getter

public class AccountId {

@NonNull

private String email;

}

用来表示账号id的类。immutable。

EncyptedPassword

public class EncryptedPassword {

private String encryptedPassword;

public EncryptedPassword(@NonNull String password){

if(password.equals("")){

throw new IllegalArgumentException("password cannot be null");

}

// TODO password not encrypted.

this.encryptedPassword = password;

}

public boolean verify(String ps){

return encryptedPassword.equals(ps);

}

}

密码不能为空的验证构造方法中。这在ddd的设计中始终比较常用的方法，在构造方法中做验证，避免矛盾（inconsisitency）的发生。数据是否矛盾交给了类自己判断，因为只是类自己应该拥有的知识。当然，当验证逻辑变复杂后，可以把这部分逻辑放到外部专用的验证类。

IAccountRepository

public interface IAccountRepository {

Account findById(String email);

void save(Account account);

}

最后写一下application层的application service（之后的文章会对application service做说明）

AccountApplicationService

public class AccountApplicationService {

@Autowired

private IAccountRepository accountRepository;

@Transactional

public void changePassword(String email, String oldPassword, String newPasssord){

Account account = accountRepository.findById(email);

account.changePassword(oldPassword, newPassword);

accountRepository.save(account);

}

}

其他说明

关于repository的实现

IAccountRepository的实现类没有写。这个会牵涉到具体的使用和中框架，orm啥的。

说一下比较令人烦恼的问题，是否该定义save()一个方法还是insert()，update()两个方法。

理想的来说save()是好的。因为domain就不用关系究竟是插入还是更新这种与db相关的操作。

然后当业务逻辑确实需要明确是登陆，还是更新时，insert()，update()会更方便。

这是个需要权衡的问题，感觉没有唯一解。

充血模型

可能大家也注意到，ddd的domain object都是采用充血模型的写法。以后也会写文章专门强调这一点。

value object真的有必要吗

AccountId真的有必要吗？直接一个String email吗？

EncryptedPassword真的有必要吗？

我承认，这些都存在讨论的空间。我也有可能是为了说明问题而使用了复杂的实现。

id类 vs String/Long

  如果id类都是Long或者String，那彼此之间是不能区别的。搞不清楚id究竟是哪个entity的id。特意定义id类，可以在一些方法要传递好多id时防止一些认为错误。比如getAnObject(Long objAId, Long objBId)。参数上objAId，objBId是不能搞错的。但如果数据类型都是Long，那编译器是核对不了的～

EncryptedPassword

  它有对密码进行核对的行为。这里提供了一种思路。当entity的逻辑过多，类变得过大时，我们可以将一部分逻辑分到value object中。淡然我们也可以在建模的时候就先考虑把一些逻辑放进value object中。这也是domain-driven design一书错推崇的（没记错的话～）。

总结

这次用简单的代码来说明了一下entity, value object等domain object的具体实现。

domain object中包含aggregate，但在实际代码中它只是像一个entity的身份出场。

代码的思路基本上是借鉴《implementing domain driven design》的做法。domain object中还有service, specification等概念没有介绍，可能很多地方还不能连贯起来，我会在之后的文章里再做说明。希望通过这个例子，大家可以以小窥大。下一篇我想写一下不使用ddd的做法，那代码会是怎样的，并做一下比较。

————————————————

版权声明：本文为CSDN博主「even\_he」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/abchywabc/java/article/details/79365266

# **ddd的战术篇: 如果没有domain object 世界会是怎样的？**

前一篇关于domain object的例子。（当然没写完整。）

如果不那么写，会是怎样的效果呢？我觉得我之后列出的代码可能是非常常见的一种写法。而且相比于ddd，大家可能更加熟悉。

贫血模型的世界

首先万年不变的实体（尸体）类

@Setter

@Getter

public class Account {

private String email;

private String password;

}

Account对应了可能叫accounts的一张表

然后我们自然而然回去想到CRUD的操作。那当然又是DAO

public interface IAccountDao {

void insert(Account account);

void update(Account account);

Account findById(String email);

}

按照这样的流程，那具体来使用这两个类的肯定就是Service了

public class AccountService {

@Autowired

private IAccountDao accountDao;

public void changePassword(String email, @NonNull String oldPassword, @NonNull String newPassword){

Account account = accountDao.findById(email);

if(oldPassword.equals("")){

throw new IllegalArgumentException("password cannot be empty");

}

if(!account.getPassword().equals(encryptPassword(oldPassword))){

throw new IllegalArgumentException("old password is wrong");

}

account.setPassword(encryptPassword(newPassword));

accountDao.update(account);

}

public void createNewAccount(String email, @NonNull String password){

Account account = new Account();

account.setPassword(password);

account.setEmail(email);

accountDao.insert(account);

}

private String encryptPassword(String password){

// TODO non implmented

return password;

}

}

贫血模式的缺点

感觉不是面向对象

这样的写法实体类就是一个数据的载体。业务的逻辑全心全意地交给了service类，实体类对业务一无所知。

感觉类和C的struct也没什么区别啊，就是可以查谁掉了setter方便点？

从宗教问题上讲，面向对象的一个目的就是开放有限接口，隐藏具体实现，具体数据。（当然，有人会说不是啦，面向对象的精髓应该是消息啥的，我们这里不讨论什么才是面向对象啦～）那至少实体类肯定是不符合这个要求的。

数据的完整性容易被破坏

实体类完全开放了setter，等于所有保持数据完整性的责任就交给了service。这是无法用语言特性来保障的，只能教育，加强代码的审核。否则，一不小心谁没通过AccountService，直接new Account()随便设了值就存，数据就可能不正常。

内聚度低

可能会觉得，和domain object的写法相比，类的数量少了啊，内聚度低？有没有搞错？

但是domain object的逻辑，其实就在Account这个entity里（好啦～其实还有很多逻辑在value object。我知道我多写了好多value object造成类数量的增多，自作孽。。。）

第内聚度的另一个后果当然是对外部依赖的增加，如果不太注意减低和控制外部依赖，很可能导致依赖关系的混乱，特别是没有明确分模块模块的情况下。于是A依赖于B，B又依赖于A这种双重依赖一不小心就产生了。或者是一个service类里又注入了十多个service，这些service之间的关系又是乱七八糟。

写自动测试难度高

因为业务逻辑存在于accountService中，所以如果要写测试，测试对象会是accountService。但它有一些成员变量需要注入啊。是的，你可以使用mock啊。

但是，如果按照domain object来写，写自动测试，你不觉得更轻松愉快吗？

贫血模型的使用

也许是我的错觉，我觉得目前来说贫血模式是web应用开发的主流（纯感觉，没有任何可靠根据）。

那贫血模式好在哪里呢？

1 它以最简单的方式，将数据和行为进行了分离（即使它很不面向对象）。

2 相比于充血模型要思考如何合理地建模，充血模型因为业务逻辑都在stateless的servcie类中，所以写起来会比较简单。

个人认为，对于比较简单的系统，或者业务基本能用CRUD来描述的系统，使用贫血模型是个可以考虑的选择。

从经验上来说，当系统变得复杂，贫血模型的系统会更快变得很难维护。（随着系统变得复杂充血模型当然也会遇到困难，但它的逻辑更加分散，所以变得难以维护的速率可能低一点）。

另外，如果使用贫血模型时，做好模块的横向划分，会使它更能对抗复杂的系统。

其实话说回来，domain model，并不是我们不去思考它，它就不存在的。使用贫血模型容易使我们意识不到的domain model变得没有规则，没有规律。就像我们不去考虑类设计，设计出来的类自然就可能混乱。所以如果我们建立好domain model，即使不用贫血模型去实现它，系统的耐复杂度也会变得更好。

————————————————

版权声明：本文为CSDN博主「even\_he」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/abchywabc/java/article/details/79305626

# **ddd的战术篇: application service, domain service, infrastructure service**

之前的一篇文章谈到了贫血模型，而ddd是提倡充血模型的，即尽量把逻辑写在domain object中，而不是写一大堆的service类，对数据类进行操作。那么为什么ddd里会有service类呢？这篇文章会对service进行说明。

ddd中的service

首先这个在这篇文章里讨论的service基于一个前提，就是采用view, application, domain, infrastructure的分层架构。ddd提倡的架构有很多种，大家可以看前面的文章。《domain driven design》原书也是基于分层架构来论述的。至于现在流行的洋葱架构该怎么办，有机会再讲吧～

ddd中有三种service。分别是application service, domain service, infrastructure service。

application service

首先从简单的开始讲。application service是应用程序的某个功能的入口（end point）。如果你使用的是分层架构，那它是位于presentation和domain之间。

我们想像我们要写一个api用来注册账号。

url: api/accounts

method: post

如果我们使用springMVC来实现这个api的话，那Controller会调用AccountApplicationService。

@RestController

@RequestMapping("api/accounts")

public class AccountController{

@Autowired

private AccountApplicationService accountApplicationService;

@RequestMapping(value ="", method = RequestMthod.POST)

public void register(@RequestBody AccountDTO accountDTO){

accountApplicationService.register(accountDTO);

}

}

上面的代码，controller讲画面传过来的dto传给了AccountApplicationService。

AccountApplicationService再来调用各种domain object。比如Account (entity), AccountRepository。

public class AccountApplicationService {

@Autowired

private IAccountRepository accountRepository;

public void register(AccountDTO accountDTO){

Account account = Account.createAccount(accountDTO.getEmail(), accountDTO.getPassword());

accountRepository.save(account);

}

}

infrastructure service

infrastructure service实现不依赖于业务(domain)的功能。简单的例子来讲，比如打印日志（log），发送邮件（如果你的应用软件不是处理邮件问题的话）

infrastructure service位于最底层的infrastructure层。

domain service

实现domain的service类。三种service中，唯一可以写业务逻辑的地方。

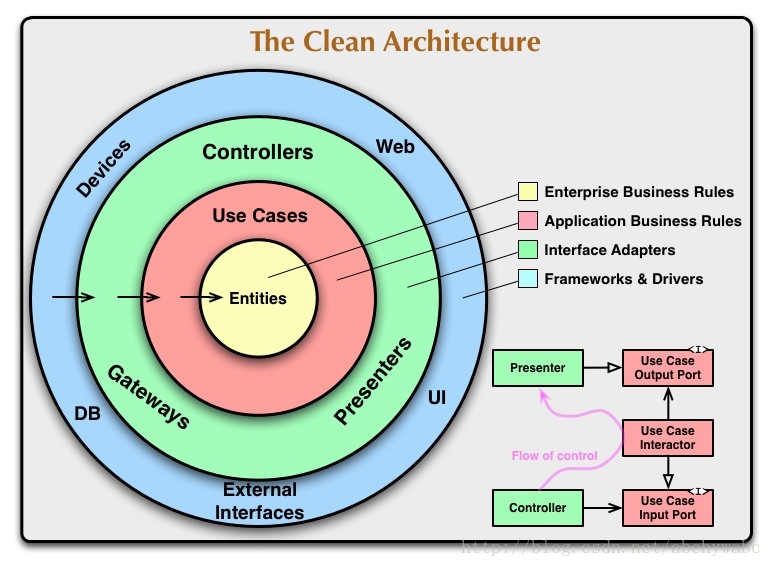
由于ddd提倡充血模型的缘故，我们在建模的时候要尽量避免制造domain service。尽量把业务逻辑放到其他的domain object中（比如entity, value object中）。

Service的比较

application service vs domain service

我想infrastructure是没有什么值得讨论的。而application service与domain service会是让人纠结的一个话题。

最近看到了clean architecture的设计后，感觉可能稍微容易解释一些。(对不起，说好用只讨论分层架构的，却又提起了clean architecture，因为好难解释…)



entities的部分对应的便是domain部分。

而application service对应的是use cases 用例。

相比于application service这个有点不知所云的名次，用例是不是更容易理解一些了（也许不是。。。）？

用例

用uml图来画的话，就是下面这样的东西。

用例具体教科书般的定义，大家可以自行谷歌一下。大致就是描述一个系统大致的功能而并不描述具体内部的结构或者设计。

那在clean architecture中，domain object的操作必须通过「用例」这个入口。

那在操作层面上如何处理application service和domain service呢？

首先application service既然是入口，在一个模块中，它必定是存在的。与之相反，domain service则不一定需要。

因此，再做类设计时可以先假定domain service不存在。直接写application service，在application service中对其他domain object进行操作。

理想情况下，application service存在的代码基本上就是它调用其他domain object的方法，具体的业务逻辑都会在domain object的方法中。所以当application service中出现if/else之类的语句，或者application service的一个方法变得很长时，我们就该警惕是不是把业务逻辑写到了application service中。这个时候我们该考虑是否需要重构，比如把逻辑放进domain object，或者增加一个domain service。

按上面一个注册账号的例子，这个时候我们并不需要domain service。

但如果我们首先要确认email是否被注册，那这时候代码就会变成下面那样

public class AccountApplicationService {

@Autowired

private IAccountRepository accountRepository;

public void register(AccountDTO accountDTO){

Account account = accountRepository.find(new AccountSpecificationByEmail(accountDTO.getEmail()));

if (account != null) {

throw new EmailAlreadyRegisteredException();

}

account = Account.createAccount(accountDTO.getEmail(), accountDTO.getPassword());

accountRepository.save(account);

}

}

尽管方法不是很长，但判断账号能否被注册的逻辑（业务逻辑）写在了application service中。我们必须考虑将这个逻辑移到其他地方。

当然理想情况下是把它放进Account中，但查询账号是否存在的逻辑使用到AccountRepository，这个很难放进Account中，所以AccountService自然会是一个选择。

public class AccountService {

@Autowired

private IAccountRepository accountRepository;

public void register(String email, String password){

Account account = accountRepository.find(new AccountSpecificationByEmail(email));

if (account != null) {

throw new EmailAlreadyRegisteredException();

}

account = Account.createAccount(email, password);

accountRepository.save(account);

}

}

因为不想让domain层的东西依赖于application层的form，dto类，所以方法的参数没有用dto。

那application service会变成

public class AccountApplicationService {

@Autowired

private AccountService accountService;

public void register(AccountDTO accountDTO){

accountService.register(accountDTO.getEmail(), accountDTO.getPassword());

}

}

另外，有了AccountService这个类，并不是所有关于Account的逻辑都必须放进那里。

比如AccountApplicationService里的changePassword()，没有必要放进AccountService中。

public class AccountApplicationService {

@Autowired

private IAccountRepository accountRepository;

@Autowired

private AccountService accountService;

public void register(AccountDTO accountDTO){

accountService.register(accountDTO.getEmail(), accountDTO.getPassword());

}

public void changePassword(String email, String oldPassword, String newPasssord) {

Account account = accountRepository.findById(email);

account.changePassword(oldPassword, newPasssord);

accountRepository.save(account);

}

}

总结

ddd中把service类分成三种。

application service, domain service, infrastructure service。

domain service中可以写业务逻辑，但同时理想情况下我们尽量不实用domain service。

另外，我们要注意不要讲业务逻辑写到application service中

————————————————

版权声明：本文为CSDN博主「even\_he」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/abchywabc/article/details/79362975

# **ddd的战术篇: Factory和Specification**

之前的文章中讲到了entity, value object, repository等domain object。这次终于能将一些相对比较轻松的话题了

Factory

这个设计模式中应该有一个叫工厂模式，ddd可能也是借鉴了它。

ddd比较注重数据的完整性。

有关数据完整性，百度了一下，结果

存储在数据库中的所有数据值均正确的状态

复习一下，ddd中有aggregate（集合）这个概念，集合中的entity, value有一定的必须保持恒定不变的状态。而ddd中的数据完整性指的就是这中概念。

比如有一个aggregate叫Person。其中有两条腿Leg(ValueObject)

public class Person {

List<Leg> legs;

}

1

2

3

那数据完整性观点来讲，无论我们调用什么方法，绝对不能出现下面这种情况

legs.size() != 2

1

另外ddd提倡充血模式。所以比如说我们创建一个entity的类之后，不会再调用一大堆setter来初始化。entity被创建了，它的状态时必须是符合业务逻辑要求，而不是需要进一步加工的。听起来很绕口。来说个实际例子吧。

比如我们要做一个购物网站，需要一个商品的类。Commodity。下面的贫血模型是ddd所反对的。

@Setter

@Getter

public class Commodity{

private CommodityId id;

private Category category;

private String commodityName;

private String description;

private Date dateCreated;

private Double price;

...

}

所以一般情况，自然我们必须在构造方法中对类进行初始化（原本很自然而然的做法，构造方法当然是来构造类的，但因为贫血模型的流行，用含参数的构造方法构造完整的类反而变成了非主流。）

public class Commodity{

private CommodityId id;

private Category category;

private String commodityName;

private String description;

private Date dateCreated;

private Double price;

public Commodity(CommodityId id, Category category, String commodityName, String description, Date dateCreated, double price){

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

如刚才所说，创建一个类，他必须符合业务逻辑的要求，那检查是否符合要求则自然成了构造方法所要做的事情。最简单的如null检查

public Commodity(CommodityId id, Category category, String commodityName, String description, Date dateCreated, double price){

if(id == null){

throw new IllegalArgumentException("id cannot be null.")

}

if(category == null){

throw new IllegalArgumentException("category cannot be null.")

}

if(commodityName == null){

throw new IllegalArgumentException("commodityName cannot be null.")

}

if(description == null){

throw new IllegalArgumentException("description cannot be null.")

}

this.category = category;

this.commodityName = commodityName;

this.description = description;

...

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

再比如字数限定，还有更复杂的检查，比如保质期比可以比生产日期早等等。

if(dateCreated > dateExpired){

throw new IllegalArgumentException("dateCreated cannot be greater dateExpired");

}

1

2

3

当这些逻辑变得复杂，构造方法就会变得很大，那类也当然会变得很大。自然而然我们就能想到把这部分逻辑分到专门的类里，这就是Factory。

public CommodityFactory{

public Commodity create(Commodity(CommodityId id, Category category, String commodityName, String description, Date dateCreated, double price)){

if(id == null){

throw new IllegalArgumentException("id cannot be null.")

}

if(category == null){

throw new IllegalArgumentException("category cannot be null.")

}

if(commodityName == null){

throw new IllegalArgumentException("commodityName cannot be null.")

}

if(description == null){

throw new IllegalArgumentException("description cannot be null.")

}

return new Commodity(id, category, commodityName, description, dateCreated, price);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

这里是一个职责分离的思想，把类的实例构造逻辑从entity类移到了Factory里。

当然这是有些争议的地方，这些验证处理不是可以在entity类构造前，通过一些Validation的工具先处理掉。

个人觉得domain object是保证数据完整性的最后防线，所以我比较倾向把这些逻辑放在domain object，然后抛出异常，然后在前台，或者application层根据异常来做处理。

承接一下开头提到的aggregate的数据完整性的话。在ddd的使用factory时，会使用factory来构建aggregate（根entity）。aggregate是一个要求数据完整和不变性（invariant)的单位，对aggregate的验证自然也会写在Factory类中。

比如，一个商品需要有标签，但标签不能超过三个（开头开腿说过人只能有两条腿～）。那Commodity这个entity/repository里会有CommodityTag这个表示标签的value object。

Commodity(CommodityId id, Category category, String commodityName, String description, Date dateCreated, double price, List<CommodityTag> tags){

...

if(tags.size == 0){

throw new IllegalArgumentException("tags cannot be empty.")

}

if(tags.size > 3){

throw new IllegalArgumentException("tags cannot be over 3.")

}

...

}

1

2

3

4

5

6

7

8

9

10

11

Specification

之前说到过，domain object是通过Repository来永久化的，一种Aggregate对应一个Repository。

Repository一般提供findById()的方法，把aggregate给返回。但如果碰到一些比较复杂的查询逻辑，findById()就不够用了，但我们肯定又不想每次有新的查询逻辑就创建一个新的方法，于是就用Specification pattern来解决这个问题。

在ICommodityRepository中定义方法

interface ICommodityRepository{

Commodity find(ICommoditySpecification spec);

}

1

2

3

定义Specification接口，这里toQuery的返回值是和实际的实现相关的，假设用hibernate的话，就返回Criteria。

interface ICommoditySpecification{

Criterion toQuery();

}

1

2

3

4

然后可以根据不同的查询逻辑建立不同的Specification类

通过id来查询的Specification

@AllArgsConstructor

public class CommoditySpecificationById implements ICommoditySpecification{

private CommodityId id;

public Criterion toQuery(){

return Restrictions.eq("commodityId", id);

}

}

1

2

3

4

5

6

7

8

9

通过产品名称来查询的Specification

@AllArgsConstructor

public class CommoditySpecificationByName implements ICommoditySpecification{

private CommodityName commodityName;

public Criterion toQuery(){

return Restrictions.eq("commodity\_name", commodityName);

}

}

1

2

3

4

5

6

7

8

9

大概是这个思路。

另外Specification把查询的条件也抽象化了，理论上可以解除domain和具体db实现的依赖关系。

比如我们不用hibernate了，而使用更底层的sql语句，那可以改写成下面那样。

interface ICommoditySpecification{

String toQuery();

}

1

2

3

@AllArgsConstructor

public class CommoditySpecificationByName implements ICommoditySpecification{

private CommodityName commodityName;

public String toQuery(){

return "select \* from commodity where commodity\_name = " +

}

}

1

2

3

4

5

6

7

8

9

假设我们连db都变掉了，变成nosql，那理论上我们只需修改Specification和Repository的实现，而不会影响到domain的逻辑。

细心的朋友可能注意到，我迟迟没有讲关于Repository的实现。没有Repository的实现，那Specification的实现是没有意义的，他们两个是相互依赖的。

那么下一篇文章终于还是要讲一讲Repository的实现方法了。突然觉得压力好大！

————————————————

版权声明：本文为CSDN博主「even\_he」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/abchywabc/java/article/details/79778220

# 参考

<https://blog.csdn.net/abchywabc/article/details/79301163>

<https://blog.csdn.net/abchywabc/article/details/79301928>

<https://blog.csdn.net/abchywabc/article/details/79304476>

<https://blog.csdn.net/abchywabc/article/details/79365266>

<https://blog.csdn.net/abchywabc/article/details/79305626>

<https://blog.csdn.net/abchywabc/article/details/79362975>

<https://blog.csdn.net/abchywabc/article/details/79778220>