# **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

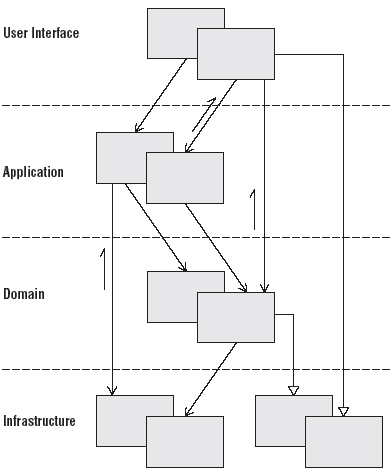
非严格的分层

 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

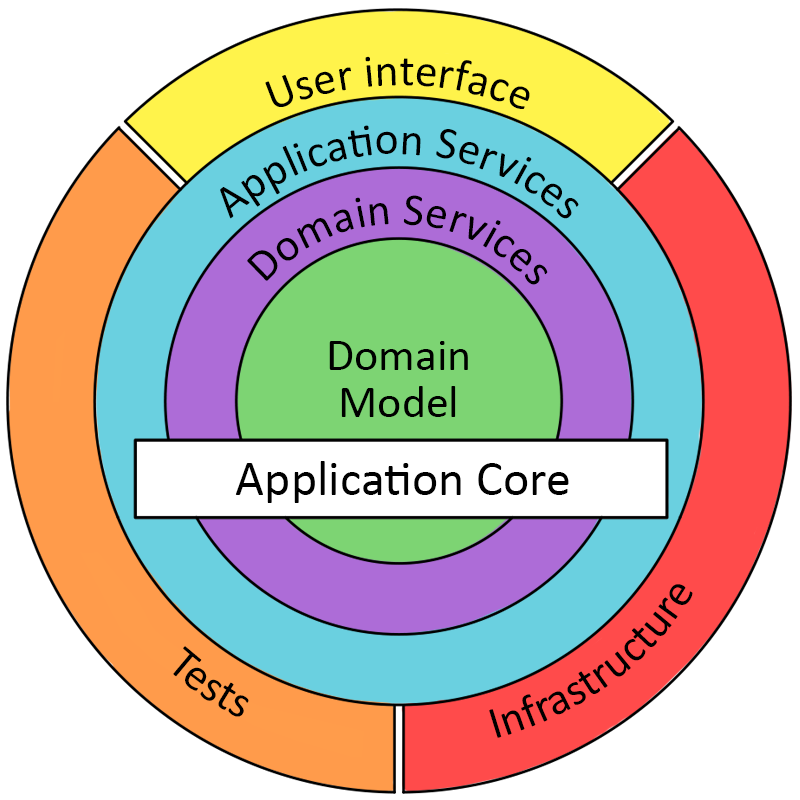
}

}

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



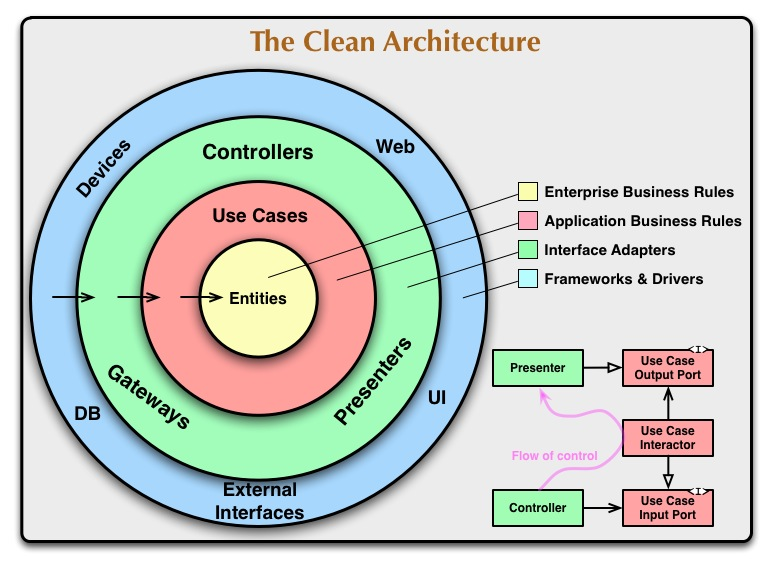
这个大家可以谷歌一下，因为《domain driven design》这本书出来的时候，可能还没有洋葱架构，书中自然没有探讨。但在《implementing domain driven design》中作过介绍。

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。

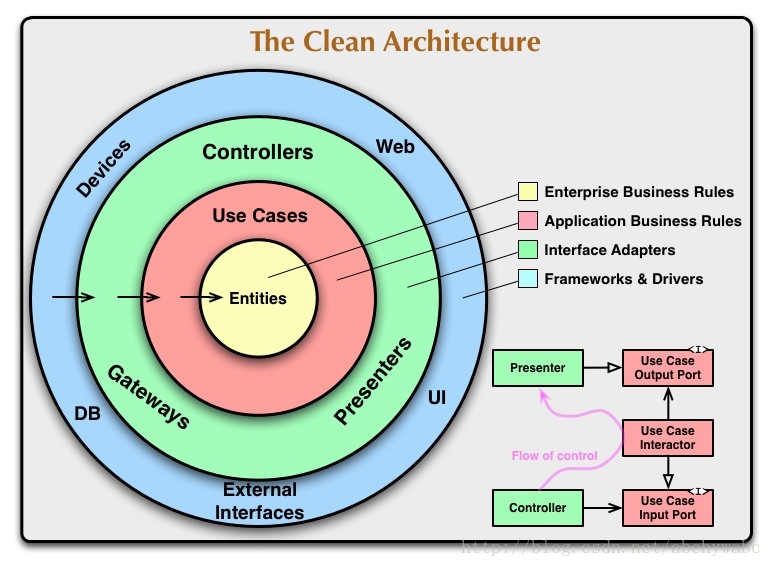
由于ddd提倡充血模型的缘故，我们在建模的时候要尽量避免制造domain service。尽量把业务逻辑放到其他的domain object中（比如entity, value object中）。

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

# **ddd的战术篇: Repository的实现**

上一篇文章讲到了Specification这个特殊的类，它是用来封装搜索的条件，也讲到了它的实现。然而Specification和Repository二者的实现是相关的，那这次就来讲一讲Repository的实现。

预备知识

复习一下Repository是什么

从字面意思上讲repository是类似于仓库的意思。那在ddd中，它便是存放domain object （entity, value object）的东西。

需要指出的是它与service类的区别。Repository的职责仅是数据永久化。它将数据永久化的部分抽象，使domain层能够专注对业务逻辑的实现。那理所当然repository是不能够包含业务逻辑的，所以我个人不推荐在repository定义像下面那样的方法。

findPersonByName(String name);

findAllPersonsByGroup(Group group);

1

2

在repository中应该吧查询条件更加抽象化，而起到这个作用的便是我们之前讲到的Specification。

实现的前提: 分层架构

还是略微提一下我们实现的repository是基于分层设计这个架构的。

按照分层设计分为以下三层:

application层

domain层 <– 定义repository

infrastructure层 <– 实现repository

实现repository

首先，这里仅谈论datasource是关系型数据库的实现。

repository和specifiction具体的实现其实是和db种类和选用的框架是有关系的，这里我们不能讲得面面俱到。

假设我们有一个表示商品的entity, Commodity。

public class Commodity {

private Long commodityId;

private String commodityName;

private String description;

/\*\*

\* 用于reconstruct的构造方法

\*/

private Commodity(Long commodityId, String commodityName, String description){

this.commodityId = commodityId;

this.commodityName = commodityName;

this.description = description;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

reconstruct

这里提一个概念叫reconstruct。

ddd中,domain object的实例化必须具有业务上的意义。很多情况下我们会用static的方法来表示其意义。比如

Commodity.createByName(commodityName);

1

这前我们还将过用Factory类来保证数据完整性。

但domain object还存在另一种实例化，那就是从db数据还原一个domain object。业务逻辑上没有因此增加一个domain object。只是将已经存在的domain object从repository里重新取出，故称作reconstruct。

我们为reconstruct定义了一个构造方法。请大家注意，这个构造方法仅用于还原domain object，在实现业务逻辑是我们原则上是不能调用这个构造方法的!

Commodity这个entity对应一个数据表commodities

id commodity\_name description

接口定义

我们定义一个IRepository接口，那应该开放哪些方法呢。我们写一下大致的构思。

public interface ICommodityRepository {

// 查询一个Commodity

Commodity findOne(ISpecification<Commodity> spec);

// 查询满足要求的所有商品

List<Commodity> find(ISpecification<Commodity> spec);

// 删除

void delete(Commodity commodity);

}

1

2

3

4

5

6

7

8

承接上面提到的内容，repository不应该包含业务逻辑，所以两个搜索的方法不包含任何业务的字眼，有关搜索条件全让Specification来定义。

然后具体的实例化的方法，会有两种方案。

add, update

void add(Commodity commodity);

void update(Commodity commodity);

1

2

save

void save(Commodity commodity);

1

这个会是比较难得选择题。从repository这个设计模式的思想上来说，比较理想的是让业务逻辑不要在意究竟是增加还是更新，只要关心把某个object存储就行。从这个角度来说save是更合适的。另外这个选择有时候取决于你选择的ORM。可以选择ORM更容易实现的方式。

实现

大致的思路是Repository, Specification隐藏了具体关于ORM对于的实现，所以RepositoryImpl, SpecificationImpl做的事情其实就是使用ORM的具体类来实现增删改查的功能。

比如jpa的话，基本就是把Specification转化为Predicate可以让EntityManager用来查询。

好了，之后基本就是关于ORM的事情了。

基于jpa的实现

jpa的实现，从代码量上面来说可能是最简洁的。不过与其说这是关于ddd的知识，更确切地说基本上完全是关于jpa的…

entity

我们必须对Commodity这个entity进行一些修改。加上与数据表映射用的信息。

@Entity

@Table(name = "commodities")

public class Commodity {

@Id

@Column(name = "ID", nullable = false)

private Long commodityId;

@Basic

@Column(name = "COMMODITY\_NAME", nullable = false)

private String commodityName;

@Basic

@Column(name = "DESCRIPTION", nullable = false)

private String description;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

repository

repository也就是几行代码

public interface ICommodityRepository extends Repository<Commodity, Long>, JpaSpecificationExecutor<Commodity> {

}

1

2

3

specification

之前的实现看起来实在是太简单了，更令人惊喜的是，jpa还很良心地提供了Specification的接口。

/\*\*

\* Specification in the sense of Domain Driven Design.

\*

\* @author Oliver Gierke

\* @author Thomas Darimont

\*/

public interface Specification<T> {

/\*\*

\* Creates a WHERE clause for a query of the referenced entity in form of a {@link Predicate} for the given

\* {@link Root} and {@link CriteriaQuery}.

\*

\* @param root

\* @param query

\* @return a {@link Predicate}, must not be {@literal null}.

\*/

Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder cb);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

连注释了都写domain driven design~

假设我们需要一个通过商品名来查询。那么我们可以定义一个。jpa里使用Predicate（命题？）来封装具体的查询条件的，所以我们要实现一个返回Predicate的toPredicate()方法。如下。

Specification

@AllArgsConstructor

public class CommoditySpecByName extends Specification<Commodity> {

@NonNull

private String commodityName;

@Override

public Predicate toPredicate(Root<Poll> poll, CriteriaBuilder cb) {

return cb.equal(root.get(Commodity\_.commodityName, commodityName);

}

}

1

2

3

4

5

6

7

8

9

10

11

然后具体使用时会是下面这样子

@Autowired

private ICommodityRepository commodityRepository;

pubic Commodity findCommodityByName(String commodityName){

Commodity commodity = commodityRepository.findOne(new CommoditySpecByName(commodityName);

return commodity;

}

1

2

3

4

5

6

7

关于jpa的实现

其实我们发现我们基本没有写什么具体实现，尤其是repository，只需定义一个接口，让它继承jpa的一些接口（好绕），jpa就会想变魔术一样，提供默认的实现。

指定repository开放的方法

JpaRepository开放了许多的方法，save(), findOne(), findAll(), count()等等。

jpa也只开放了save()这个方法，所以解决了一个本需要你烦恼的选择题。（除非你一定要选择insert/update）

但如果你觉得JpaRepository开放的方法太多了，很多根本不需要（比如delete(Entity entity)，count()），jpa还提供了其他的手段让你指定需要实现的方法）。本来想写一下具体的内容，大家还是在需要时去查文档吧，只要知道有这个选项就行了

domain object与data object的分离

大家可以看到，使用jpa的话，在entity的类里要写一些与db映射的注解。如果你有一些洁癖，不希望有这种依赖于db的信息，因为它实际上暴露具体的数据库实现。那自然而然你可能想要有一个专门与db映射的类，姑且称之为data object吧。

data object不是domain object，它没有业务的逻辑，只有关于db的一些信息。

那repository的实现就会变成

查询： 使用ORM查询得到data object, 将data object转化为domain object返回。

存储: domain object转化成data object, 使用ORM将它永久化。

而ddd关注的只是repository将domain object永久化的部分。它并不强求你去定义专门映射db的data model的。其实使用data model这个手段不仅增加了需要定义的类，还增加了类转换的逻辑，而你能够得到的回报知识讲业务与数据的更大程度的分离。我认为是性价比很低的一个做法。

题外话，那为什么还要提这个手段呢？

直接的原因是因为自己工作的地方是采用了这种方式。而做出这个选择的理由是我们没有使用jpa（holy sh#t!）

结果就是我们很悲惨地定义了一大堆domain object。但在repository里，我们要将domain object转换成ORM需要使用的data object。真的是无比啰嗦，仓了个天啊。

所以当你的项目也有类似问题，觉得不用jpa就实现不了ddd时，不要放弃，其实也是有办法的，但是如果运气不好，你是用的ORM和ddd很不对路，那开发成本会变得比较大。

基于mybatis的实现

ORM的选择有千万种，我们再试试另一个比较流行的框架mybatis。看看使用它时该如何实现Repository。

specification

定义一个specification接口，它的toQuery()方法返回sql语句。

public interface ICommoditySpecification {

String toQuery();

}

1

2

3

4

和之前例子一样，我们写一个用商品名来获取商品的specification。

@AllArgsConstructor

public class CommoditySpecByName implements ICommoditySpecification {

@NonNull

private String commodityName;

@Override

public String toQuery() {

return "SELECT \* FROM commodities WHERE commodityName = " + commodityName;

}

}

1

2

3

4

5

6

7

8

9

10

11

repository

写一个mybatis用来处理commodities这张表的类

public interface CommodityMapper {

@Results({

@Result(property = "commodityId", column = "id"),

@Result(property = "commodityName", column = "commodity\_name"),

@Result(property = "description", column = "description")

})

@Select("#{sql}")

Commodity findOne(String sql);

@Insert("INSERT into commodities(id, commodity\_name, description) VALUES(#{commodityId}, #{commodityName}, #{description})")

void add(Commodity commodity);

@Update("UPDATE commodities SET commodity\_name=#{commodityName}, description =#{description} WHERE commodity\_id =#{id}")

void update(Commodity commodity);

1

2

3

4

5

6

7

8

9

10

11

12

13

14

我们造了个万能的findOne()方法，它能接受任何查询，这是个潜在的安全性问题(估计在代买审核时，这个写法会被喷得体无完肤)。当然这里只想说明repository的实现，所以这个问题先不管。

repository

public CommodityRepositoryImpl implements CommodityRepository{

@Autowired

private CommodityMapper commodityMapper;

public Commodity findOne(ICommoditySpecification spec){

commodityMapper.findOne(spec.toQuery());

}

public void add(Commodity commodity) {

mapper.add(commodity);

}

public void update(Commodity commodity) {

mapper.update(commodity);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

关于mybatis的实现

仅仅是写了一部分的实现，大家就可以看出相比于jpa，repository的实现类还是需要一定代码量的。不过对喜欢写sql语句的人来说，mybatis也是个不错的ORM选择。而mybatis这个框架，努力一下还是能实现repository，specification模式的。

总结

这次通过jpa和mybatis的两个例子来说明了repository的实现方法。

相信大家已经找到了诀窍，无论是哪种ORM基本上都能实现Repository, Specification这个设计模式。当然成本可能千差万别。

到此为止我们基本上大致讲解了ddd的实现方法。之后还会在写一些文章来谈谈ddd战术方面的一些原则和建议。

————————————————

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/80037998

# **ddd的战术篇: aggregate的设计策略**

上一篇文章讲了repository的实现。结合前面的文章基本把ddd战术篇中的登场人物都介绍了一遍。依次是如下几个角色。

首先是位于domain层的domain objects

- aggregate

- entity

- value object

- domain service

- repository

- specification

- factory

另外有还有两种service。application service, infrastructure service.

当我们分析业务，实际建模时，最重要的当然是aggregate, entity与value object。

我们会将业务逻辑尽量写进entity与value object。

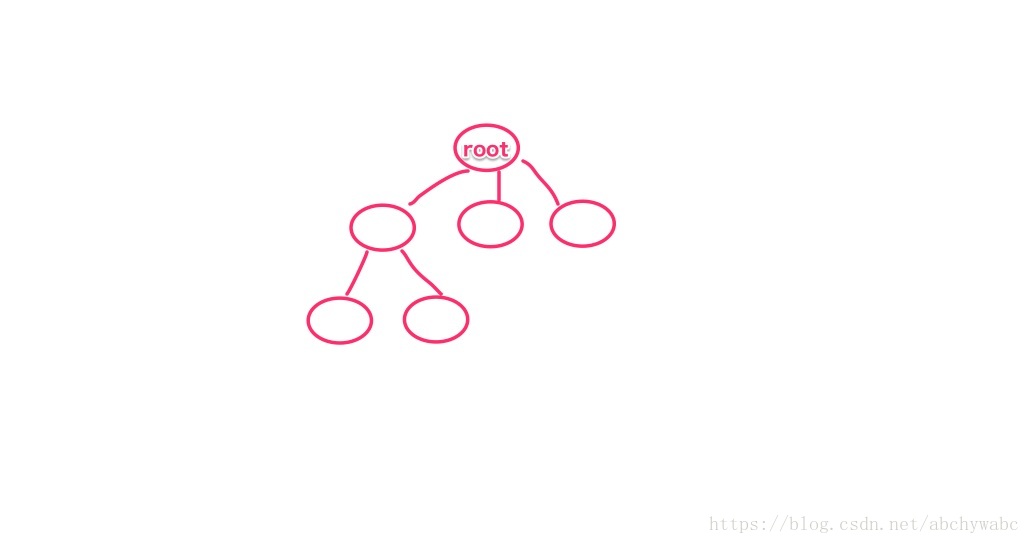
而aggregate还有确保数据完整性的责任。entity，value object也最终隶属于某个aggregate。（value object如果不需要永久化的话，那它可以独立于aggregate）

因此aggregate的设计是十分重要的，这篇文章就来谈谈aggregate的设计策略。

尽量设计小aggregate？

什么是小aggregate？

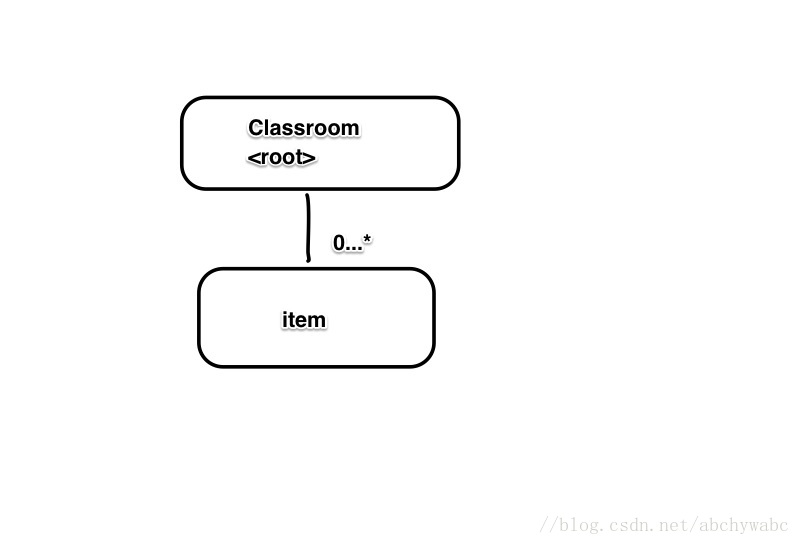
那我们先看看什么是大的吧～



如果所示，root entity直接引用了很多的entity。

我们用例子来说明一下。假设我们对处理一个教室物品管理的问题。教室里有椅子，桌子和黑板这些物品，为了解释数据完整性的问题，假设一个教室里不能有超过50样物品。另一个需求是假设这些物品需要维护，要记录各个物品是在何时维护的。

大概是这么一个关系



比较直观的方案当然是把教室当作一个aggregate，里边有它的物品。

首先是教室类，他是一个entity

public class Classroom {

private ClassroomId classroomId;

private String classRoomNo; // eg: 402

private List<Item> items;

public Item add(Item item) throws ItemLimitExceededException {

if(items.size() == 50) {

throw new ItemLimitExceededException();

}

item.add(item);

}

public Item findItem(ItemId itemId){

items.toStream().filter(item -> item.getItemId() == itemId).findFirst().getOrElse( null );

}

}

通过在add()方法中增加检验来保证数据完整性。

物品类（椅子，桌子，黑板什么的）

@Getter

public class Item {

private ItemId itemId;

private ItemType itemType;

private Date maintenanceDate;

public void maintain(){

maintenanceDate = DateUtils.now(); // 更新检测日

}

}

public enum ItemType {

Chair, Desk, Blackboard

}

需不需要具体的Desk，Chair，Blackboard类还是只要Item一个类就够了是可以讨论的问题，不过现在我们关注的不是这个，而是Classroom这个aggregate。当我们一这个方式实现时，我们获取Classroom的引用时，必然会把它所包含的物品一起获取。

想象一下ClassroomRepository的实现，获得某一个id的Classroom。不仅要读取classroom,我们不得不也把所有的物品items也一起读取。这个是个比较低效的做法的。

另外，按照这样的做法，如果你对某一个教室进行更新时，还会更新它所包含的物品。比如当你需要更改教室的名字，明明是和椅子，桌子无关的处理，但也必须去更新他们。这样的更新范围较大，当这个系统需要管理的东西变得十分庞大，并发处理很多时，容易导致锁问题的出现，造成更新的失败。（当然现实可能是这个系统永远不会变得那么大，难道这么一个管理系统会指望同时在线用户数亿？我承认这个例子是在是太不合适勒orz,不过意思能传达就好了…早知道该用论坛什么的做例子！）

此外，因为ddd的aggregate思想的一些限制，位于某一个aggregate内的entity，你必须通过它的root entity才能获取的。在这个例子中即你要获得一张Chair的引用，你必须先获得它所在Classroom的引用才行。Chair不是root aggregate，它没有repository，所以也不存在跳过Classroom的作弊方式。那这会有什么问题呢。假设我们需要维修了一张Chair。记录他的维护时间。

chair.maintain();

1

大概会是相面这个样子

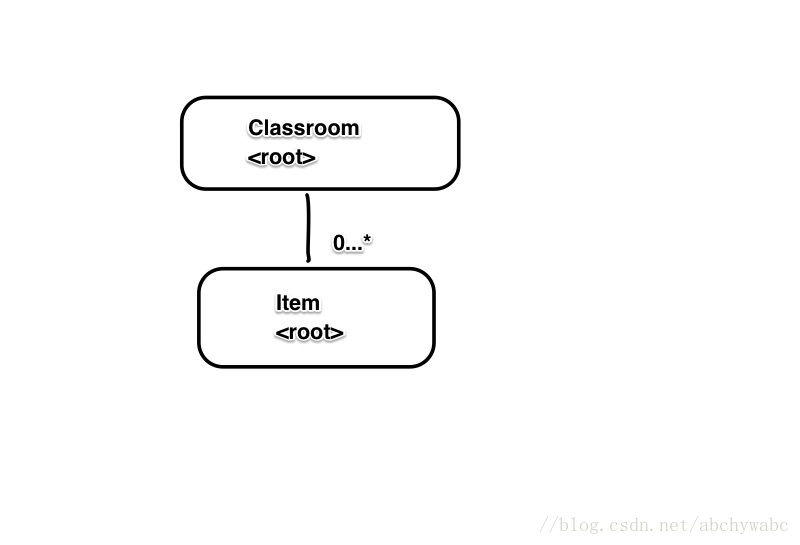
Classroom classroom = classroomRepository.findOne(new ClassroomSpecificationById(classroomId);

Item chair = classroom.findItem(chairId);

chair.maintain();

为了调用一张Item椅子的方法，我们必须把Classroom读取出，连带一群我们根本不在意的Desk，Chair。这样的做法超级低效，在高并发情况下，会有更新失败就更容易发生（这样用户体验会很差哦！如果你这么干，你就等着和产品经理或者设计师单挑吧！或者你是被群殴的那位！）

那如果不把所有object都扔进一个aggregate，怎么办。那就拆分成好几个aggregate。把物品Item也作为aggregate。



@Getter

public class Item {

private ItemId itemId;

private ItemType itemType;

private Date maintenanceDate;

public void maintain(){

maintenanceDate = DateUtils.now(); // 更新检测日

}

}

其实Item本身的实现没有什么变化啦～

接着是教室类

public class Classroom {

private ClassroomId classroomId;

private String classRoomNo; // eg: 402

private List<ItemId> itemIds;

public Item add(ItemId itemId) throws ItemLimitExceededException {

if(itemIds.size() == 50) {

throw new ItemLimitExceededException();

}

itemIds.add(itemId);

}

}

aggregate之间的引用，在ddd中是通过id引用的。例子中Classroom这个aggregate是持有的Item的root aggregate的id（好绕口）来表示Classroom与Item的包含关系的。

按照这样的设计，Item可以有自己的repository,你可以不通过Classroom对某个Item进行操作。

Item item = itemRepository.find(new ItemSpecification(itemId));

item.maintain();

当然你也必须放弃一个功能，便是从Classroom这个aggregate中直接获取某个Item引用的功能。findItem()这个方法就实现不了啦。

\* 我见过一些里类似于在entity 中放入repository的做法。比如像下面这样。

public class Classroom {

private ItemRespository itemRepository;

public Item findItem(ItemId itemId) {

return itemRepository.findOne(new ItemSpecificationById(itemId));

}

}

个人不是很推荐这种做法。因为entity依赖于外部的repository。另外repository是无状态的，和entity不太般配～

归纳一下

大aggregate的缺点。

1. 从aggregate的角度来说，每次获得一个aggregate的引用会读取过多的数据

2. 从aggregate的子entity的角度来说，所有的操作都必须经过root entity，如果对子entity的操作可能影响数据完整性，这种方式是可行的。反之，这样的方式便是很笨拙的。

3. 在高并发情况下，容易出现锁问题。

大aggregate的优点

1. 更好的维护数据完整性

大aggregate的用处

前面总结了大aggregate在维护数据完整性上是有优势的。但上面例子中的数据完整性即Classroom最多有50个物品，我们可以用小aggregate的方式来实现，所以我们倾向使用小aggregate。

如果我们的系统需求是不一样的，比如我们要记录教室中的物品位置，同时要保证相同位置上不能有两个物品（数据完整性！）。那么我们的模型就会发生变化。

@Getter

public class Item {

private ItemId itemId;

private ItemType itemType;

private Position position;

private maintenanceDate;

public void maintain(){

maintenanceDate = DateUtils.now(); // 更新检测日

}

}

@Value

public class Position {

private Integer x;

private Integer y;

}

当我们要向教室里加物品时，小aggregate的设计就满足不了需求了，于是我们可能不得不启用大aggregate的设计。

public class Classroom {

private ClassroomId classroomId;

private String classRoomNo; // eg: 402

private List<Item> items;

public Item add(Item item) throws ItemLimitExceededException, PositionOccupiedException {

if(items.size() == 50) {

throw new ItemLimitExceededException();

}

if(isOccupied(item.getPosition())) {

throw new PositionOccupiedException();

}

item.add(item);

}

private boolean isOccupied(Position position) {

return items.toStream().map(Item::getPosition)

.anyMatch(p -> p.equals(position));

}

public Item findItem(ItemId itemId){

items.toStream().filter(item -> item.getItemId() == itemId).findFirst().getOrElse( null );

}

}

总结

这次我们讲了一下aggregate的设计策略。讨论了大aggregate的优与劣。这部分的理论主要参考《实践ddd》，作者是提倡使用小aggregate的。个人认为，选择哪个策略归根到底会是两个问题。

1. 权衡数据完整性和抗并发的问题

2. aggregate的非root子entity是否拥有相当多的独立业务逻辑

根据上边的原则可以帮助我们决定aggregate的设计。

————————————————

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/80171967

# **ddd的战术篇: domain event(事件)**

承接上篇文章谈到的aggregate的设计策略。aggregate是用来保证数据（\*注: 之前的文章都是用数据完整性这个说法，其实是想表达data consistency这个意思，查了一下翻译，感觉还是数据一致性比较贴切）一致性的一个单位，但是如果设计大的aggregate的话，容易产生并发问题和资源浪费等问题。所以在考虑数据一致性的同时。我们要尽量设计小的aggregate。

另外自然会碰到这样的问题，aggregate的内部数据一致性由aggregate来保证。那aggregate之间会不会存在数据一致性呢？尤其在尽量设计小aggregate的前提下，这个问题好像更加容易发生。

跨aggregate的数据一致性

用例子来说明问题吧。

比如有一个类似博客的网站的功能。有用户User和文章post。

class User {

private UserId userId;

private String email;

private String username;

private AccountStatus accountStatus;

public void deactivate(){

accountStatus = AccountStatus.DEACTIVATED;

}

}

1

2

3

4

5

6

7

8

9

10

这里为了说明问题，我们就定义User的一个行为，注销deactive。

class Post {

private PostId postId;

private UserId authorId;

private String text;

private PostStatus postStatus;

private Long views;

public void view(UserId userId) {

if(userId == authorId){

throw new CannotAddViewToYourOwnPostException(postId);

}

this.likes = this.likes + 1;

}

public void delete(UserId userId){

if(userId != authorId){

throw new IllegalAccessException(postId, userId);

}

postStatus = PostStatus.DELETED;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

Post呢，简单起见也定义两更行为，阅览（记录阅览次数）和删除。如果你觉得物理删除是个邪恶的行为。那就定义一个状态来表示删除，例子中我们定义了一个枚举。

接下来要实现一个用例。当一个User账号删除后，与User关联的Post也必须删除。当User处在注销状态下，User的Post处在废删除状态是不合理的。这又是个数据一致性需要保障。

如何保障跨aggregate的数据一致性

先别想太多，就按普通的思路走，把对多个aggregate的操作写进一个事务不就行了？

class AuthorApplicationService {

@Transactional

public void deactivateAccount(Long userId){

User user = userRepository.findOne(new UserSpecificationById(userId);

user.deactivate();

userRepository.save(user);

Posts posts = postRepository.find(new PostSpecificationByUserId(user.getUserId()));

posts.forEach(post -> {

post.delete();

postRepository.save(post);

});

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

用户的注销和文章的删除处在同一个事务中，当用户备注小时她的文章肯定也会都被删除。

然而，这样的方式书中居然并不提倡。原书中有这样的阐述。

Any rule that spans aggregates will not be expected to be update at all times. Through event processing, batch processing, or other update mechanisms, other dependencies can be resolved within some specific time.

如果一个规则是跨aggregate的，那我们就不能指望在所有时刻都被遵守（直译很僵硬）。我们可以理解成对多个aggregates的操作，我们不能指望他们能同时进行。而实践ddd的书中也引用了这个规矩，并将其解释成”一个事务处理中应避免更新多个aggregate”。

也有说法表示因为aggregate需要保证数据的一致性，自身就代表了一个事务的边界。因此把对多个aggregate的处理放在一个事务中是很不自然的（这是一种直觉）。

那为什么要比面在一个事务中更新多个事务呢？

本人能想到的理由有两个

1. 可能会操作过多的数据，造成程序的低效。

尤其是你要更新好多个aggregate的时候。(这个问题在上面的例子中并没有很好地展示出。硬要说也就是万一有博主写了太多文章，一删删一万篇)

2. 同时操作过多的数据容易造成并发的问题。

假如说一个博主的文章超有人气，一天到晚有人看。导致对Post的更新很频繁，结果博主想要注销时，可能造成锁等待（悲观锁）或者提交失败（乐观锁）。当然这样的博主有必要注销吗？（好啦，我知道其实我在为谈问题而制造问题。）

这写话是不是好像在哪里听到过。感觉这好像就是让我们不要设计大aggregate的理由啊。

仔细想一想，如果我们允许在一个事务中对多个aggregate进行更新操作，我们是不是在某种意义上制造了一个更大的aggregate。结果就是我们明明设计了小的aggregate，结果反而没能得到我么预期的结果。

另外很多关于ddd的文章也有这种观点。ddd中通过aggregate来保证数据的一致性，所以aggregate自然而然产生了一个事务边界。出了aggregate的事务边界，自然无法很有效地保证数据一致性。如果当你发现自己必须把对多个aggregate的操作放进一个事务，很有可能你的建模是有问题的，应该重新审视自己的模型。

如此提倡小的aggregate设计，而跨aggregate的数据一致性却有又没有有效的手段来保证，那怎么办？搞一个大的aggregate？这不是打脸吗？

不过重新审视自己的模型，这话是没错啦～我们可以考虑是不是有更好的建模方法。按例子上来说，我们真的需要把统计阅览数的行为给Post来做吗？是不是可以把它分解出Post,个其他的模型来实现。这是一个关于更深入地理解业务，改进模型的思路。

另一个思路是aggregate的设计策略问题（也算是从另一个角度对业务进行理解）。即是前面的文章提到的，如果这是一种不能妥协的数据一致性，自然须将它放进一个边界，即一个aggregate。当然为此我们愿意付出代价即aggregate可能会变得巨大而导致性能问题。然而当我们希望优先性能及用户体验，数据一致性只须最终一致(eventual consistency)就行的话，我们有另一个选择。

domain event

从当数据的一致性只需要达到最终一致的话，ddd提倡利用event的设计模式。event也是解耦的一种常规操作。ddd也借鉴了这个模式来解决数据最终一致的问题。

当我们要对两个aggregate进行操作。两个处理会进行在独立的事务中。当第一个处理完成后，抛出一个event（事件），然后监听event的组件在接受到event后开始第二个操作。

比如在上面的例子里

User user = userRepository.findOne(new UserSpecificationById(userId);

user.deactivate();

userRepository.save(user);

1

2

3

当用户注销后，我们可以抛出一个UserDeactivatedEvent的事件，监听 UserDeactivatedEvent的组件会在接收到事件后进行文章删除的操作。

Posts posts = postRepository.find();

posts.forEach(post -> {

post.delete();

postRepository.save(post);

});

1

2

3

4

5

如此一来，我们便可以把处理分拆成两个事务，保证了性能和用户体验，同时也能达成数据的一致性。

是不是听起来很简单？但其实操作成面上并不简单。当A，B两个处理处在同一个事务中，A，B要么同时成功，要么同时失败。而通过event模式来做的话，A成功的话并不代表B一定会成功。那么B不成功时该怎么办？这会有多种策略

- 不断尝试进行B处理，直到B成功为止。

- B失败后抛出事件，对A处理进行回滚或者其他挽救处理

- 结合上边的两种方式

无疑为了实现最终一致性，我们引入了很多复杂度，需要考虑的状况会更多。

另外我们还需要一个事件订阅和分发的系统来帮我们实现这个功能。烦人的是如何实现事件订阅和分发的事情并不是ddd所关注的，ddd对实现没有任何的指向与意见。所以怎么解决就全得靠自己啦。。。

一篇文章码不了太多字，先介绍一下Spring提供的一个轻量级方案吧。

Spring Data 1.13后有下面一个方便实现aggregate的抽象类。

public class AbstractAggregateRoot {

/\*\*

\* All domain events currently captured by the aggregate.

\*/

@Getter(onMethod = @\_\_(@DomainEvents)) //

private transient final List<Object> domainEvents = new ArrayList<Object>();

/\*\*

\* Registers the given event object for publication on a call to a Spring Data repository's save method.

\*

\* @param event must not be {@literal null}.

\* @return

\*/

protected <T> T registerEvent(T event) {

Assert.notNull(event, "Domain event must not be null!");

this.domainEvents.add(event);

return event;

}

/\*\*

\* Clears all domain events currently held. Usually invoked by the infrastructure in place in Spring Data

\* repositories.

\*/

@AfterDomainEventPublication

public void clearDomainEvents() {

this.domainEvents.clear();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

其中有一个registerEvent()的方法，可供调用，它是用来创建时间的。注意当调用这个方法是，事件只是被创建，而没有被发布。

class User {

private UserId userId;

private String email;

private String username;

private AccountStatus accountStatus;

public void deactivate(){

accountStatus = AccountStatus.DEACTIVATED;

registerEvent(UserDeactivatedEvent(this.userId)); // 创建用户注销的事件

}

}

1

2

3

4

5

6

7

8

9

10

11

在用户注销的方法里，调用刚才提到的regsiterEvent()来创建事件。

而Spring data会在Repository.save()被调用的时候把实现发布出去。

class AuthorApplicationService {

@Transactional

public void deactivateAccount(Long userId){

User user = userRepository.findOne(new UserSpecificationById(userId);

user.deactivate();

userRepository.save(user); // UserDeactivatedEvent会被发布

}

}

1

2

3

4

5

6

7

8

然后订阅事件部分的实现会是下面这样。

@Service

public class PostService {

@Autowired

private IPostRepository postRepository;

@Async

@TransactionalEventListener

public void handleUserDeactivatedEvent(UserDeactivatedEvent event) {

Posts posts = postRepository.find(new PostSpecificationByUserId(event.getUserId()));

posts.forEach(post -> {

post.delete();

postRepository.save(post);

});

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

严格意义上说删除文章这个实现还是在一个事务中操作了多个aggregate的。因为只要作者有两篇以上的文章，删除时我们就等于在一个事务中操作了多个Post实例。所以还要把每个删除放进一个事务。

@Service

public class PostService {

@Autowired

private IPostRepository postRepository;

@Autowired

private PostTransactionService postTransactionService;

@Async

@TransactionalEventListener

public void handleUserDeactivatedEvent(UserDeactivatedEvent event) {

Posts posts = postRepository.find(new PostSpecificationByUserId(event.getUserId()));

posts.forEach(post -> {

postTransactionService.delete(post);

});

}

}

public class PostTransactionService {

@Autowired

private IPostRepository postRepository;

@Transactional

public void delete(Post post){

post.delete();

postRepository.save(post);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

总结一下spring data对domain event的实现如下

1. aggregate root来创建事件

2. repository.save()是发布事件

3. 用@TransactionalEventListener来声明对事件的处理

因为repository.save()是触发事件的地方，所以看起来有点像数据库的trigger功能。

关于最终一致性是否可行

从理想情况下而言，我们当然是希望数据在任何时刻都保持一致性。但很遗憾，很多时候因为很多的限制，我们必须做一些选择题来权衡。

最终数据的一致性意味着数据的状态会有延迟。那延迟会不会被接受。

其实数据的一致性是源自于业务的需求，而延迟能否被接受也不取决程序员。我们完全可以和domain expert（精通业务的人）来商讨，从业务角度来说多少的延迟是可以接受的。最近看到一篇关于ddd的文章说在计算机到来之前，人类用纸来管理业务的时代，延迟是家常便饭的事情，所以没必要视延迟为一种罪恶。。。

总结

ddd中由aggregate来保证数据一致性。当走出aggregate的边界后，我们能通过domain event来实现数据的最终一致性。

domain event具体如何实现又是一个能展开的话题，之后的文章还会进行论述。

————————————————

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/80289351

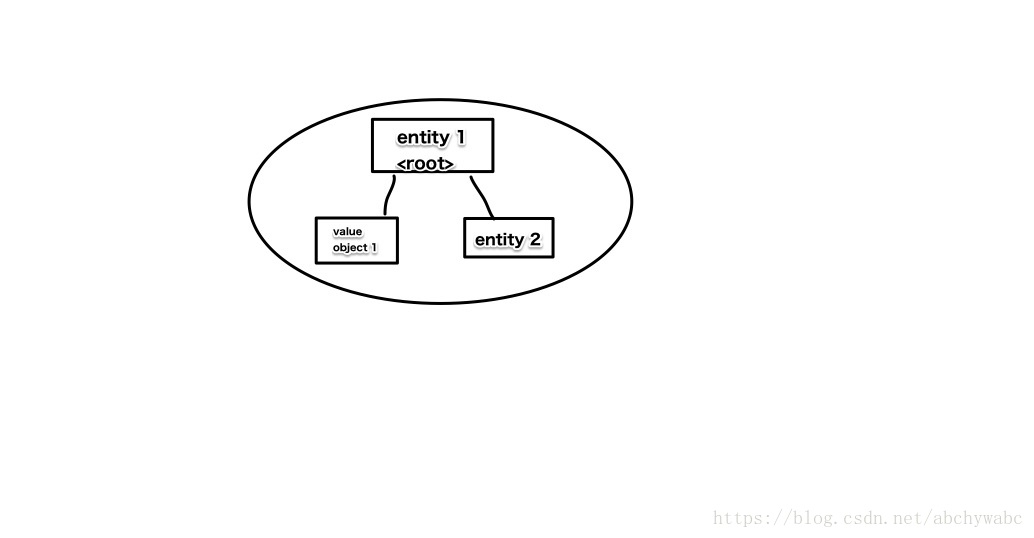
# **ddd的战术篇: CQRS**

之前的文章介绍了ddd在战术层面的要素，包括entity，value object，aggregate和一些设计模式如repository。在ddd中，repository几乎成为了等同于entity一样的基本要素。

关于aggregate与repository的回顾

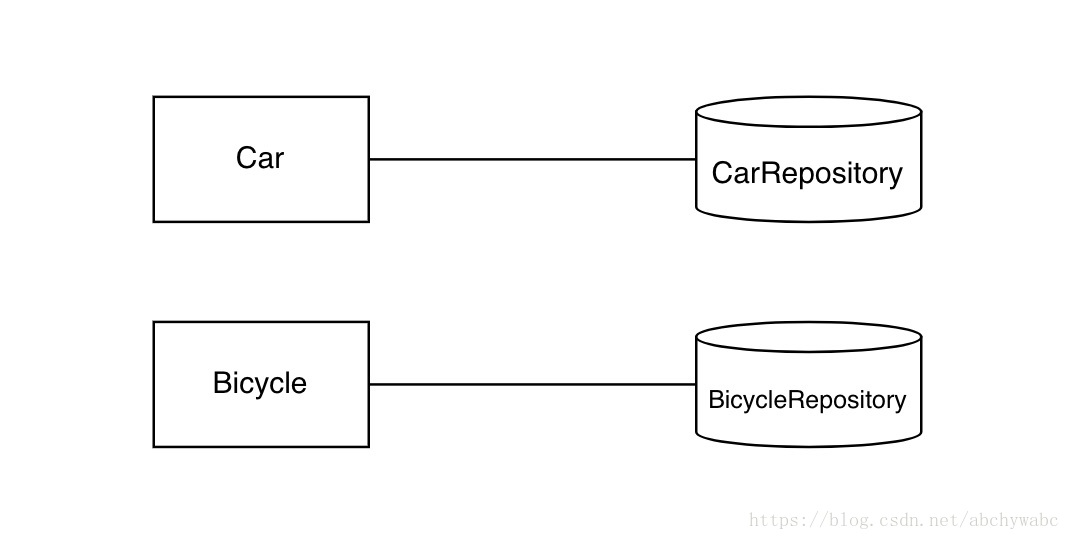
aggregate是entity和value object的聚合，它定义了一个边界，在此边界中，数据必须保证数据完整性。

aggregate有一个根root entity，从这个root entity可以获取aggregate内部的其他entity和value object



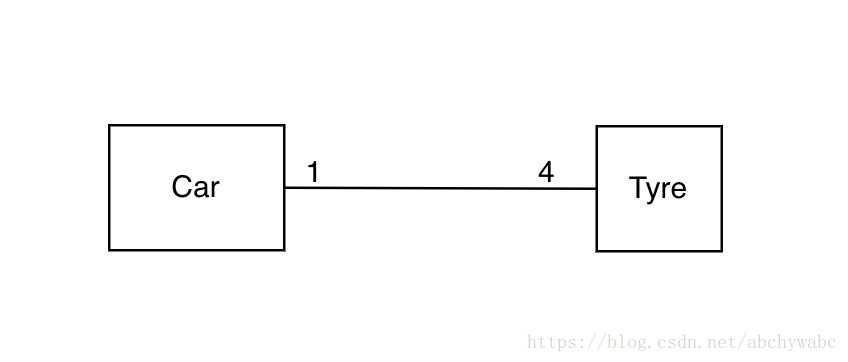
如图所示，entity 1是aggregate的root entity。

repository是存放和获取aggregate的地方。一种aggregate对应一种repository。

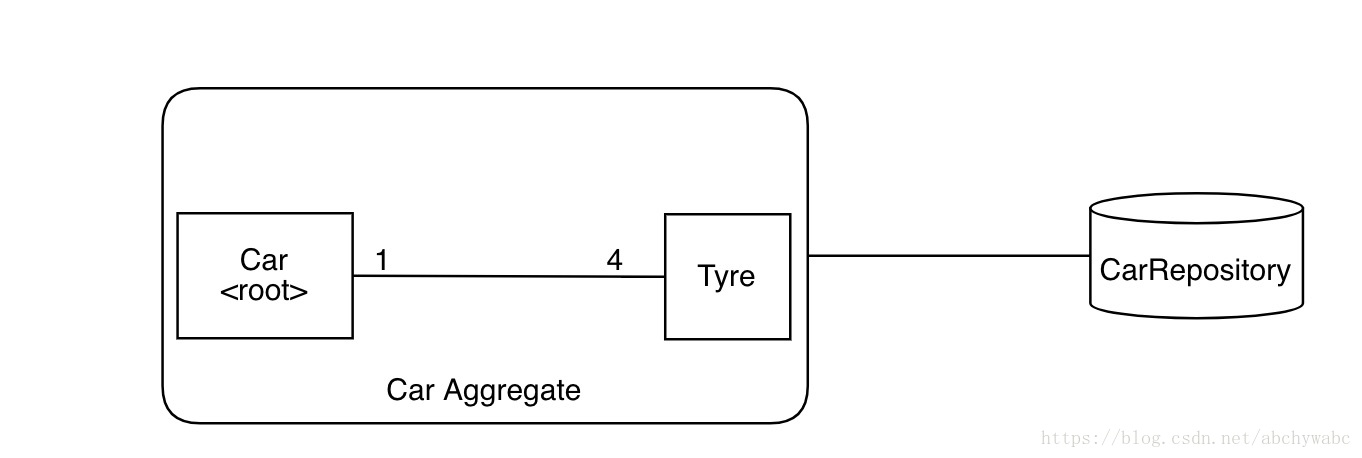


如图，Car与Bicycle都是aggregate，它们各自对应CarRepository与BicycleRepository。

然后Car有4个轮子，所以Car这个aggregate必须保证它有4个轮子的数据完整性。



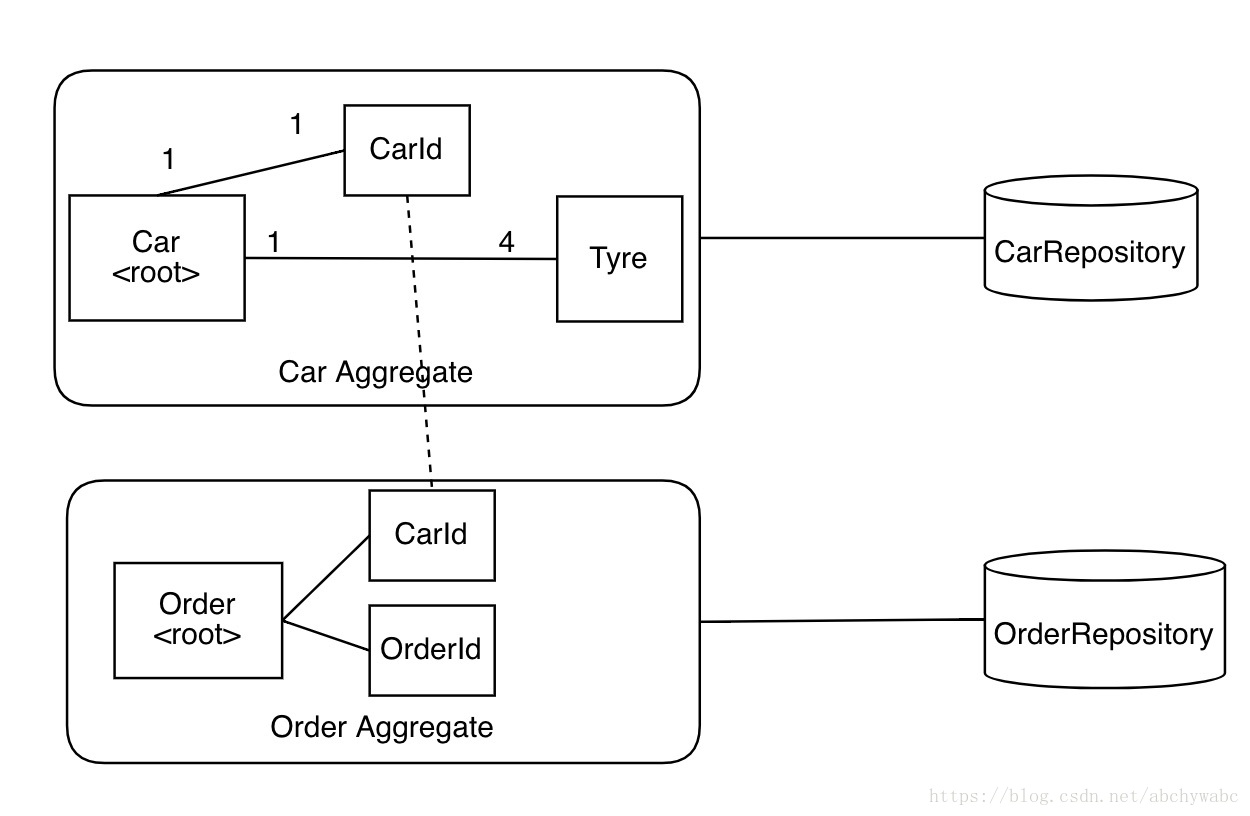
把这些都关联起来，我们可以话成这样的图



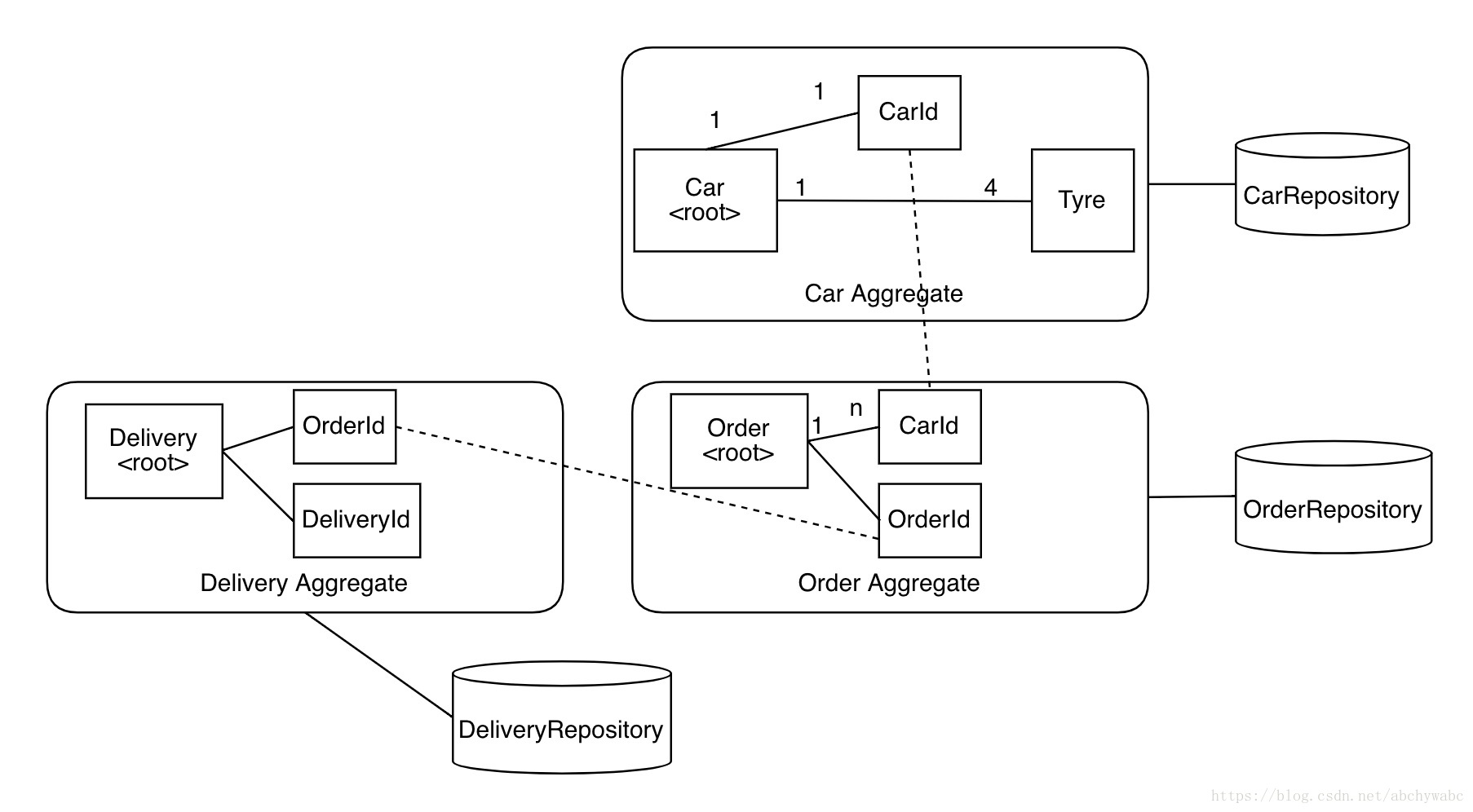
aggregate之间的引用

aggregate之间是通过entity的ID来引用的。Car这个entity，有一个CarId的value object作为Car的识别符。

现在我们想象要做一个购物平台，其中处理订单是我们的一个业务。那我们自然而然会想到需要一个表示订单的domain object。那我们就创建一个Order的entity。Order通过CarId来引用Car这个aggregate



如果还有关于运输的需求。我们为此创建一个叫Delivery的domain object。（当然这个例子有点搞大了，很难想象一般消费者能像买纸巾一样买车子…）



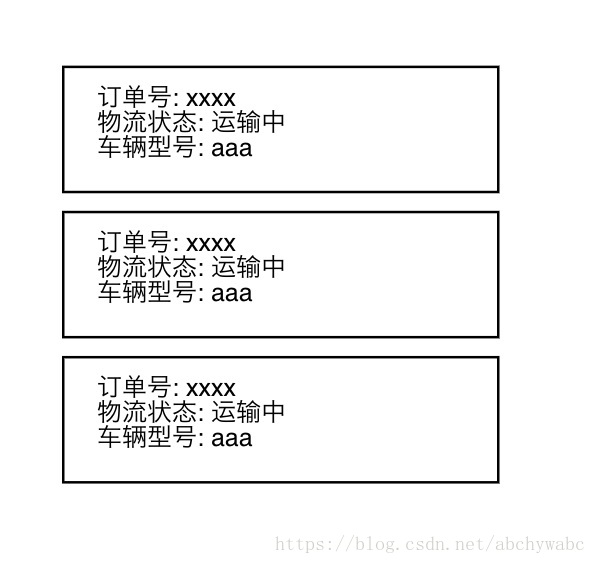
aggregate的缺陷

如果按照ddd提倡的建模方式，我们会比较自然地得出上面的模型。当然Aggregate的范围是可以有讨论余地的。比如把所有entity都扔进一个aggregate中。这方面的讨论可以参考下面的文章。

aggregate的设计策略

我们现在以上面的模型为前提来说说我们可能会遇到的问题。

如果我们需要实现一个订单列表的功能。



简而言之，我们需要获取Delivery,Order和Car的信息。

下面是代码

public List<OrderDTO> viewDeliveryList(UserId userId){

List<Delivery> deliveries=deliveryRepository.find(new DeliverySpecByUserId(userId));

List<OrderId> orderIds=deliveries.toStream().map(delivery->delivery.getOrderId()).collect(Collectors.toList());

List<Order> orders=orderRepository.find(new OrderSpecByIds(orderIds));

List<CarId> carIds=orders.toStream().flatMap(order->order.getCarIds()).collect(Collectors.toList());

List<Car> cars=carRepository.find(new CarSpecByIds(carIds));

return List<OrderDTO> orderDTOs=buildDTO(deliveries, orders, cars);

}

1

2

3

4

5

6

7

8

9

10

用于画面显示的DTO类，constructors, getter就省略了。

public class DeliveryDTO {

private Long deliveryId;

private String deliveryStatus;

private String deliveryAddress;

}

public class OrderDTO {

private Long orderId;

private List<CarDTO> cars;

private DeliveryDTO;

}

public class CarDTO {

private Long carId;

private Long carModelId;

private String imageUrl;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

大致的处理是从各个repository中获取entity(aggregate)。最后把取得的3种entity拼装成画面需要的DeliveryDTO。

那有什么问题吗？直观的感觉是对数据库进行了3次查询，而实际上如果使用关系型数据库的话，明明可以用一个集联查询来实现的东西，用了3次查询显得有些笨拙。

表面上看可能是我们选择ORM和模型设计的问题。比如如果是jpa的话，下面的模型设计是可以减少查询数量的。

public class Delivery {

private DeliveryId deliveryId;

}

public class Order {

private OrderId orderId;

private Delivery delivery;

private List<Car> cars;

}

public class Car{

private CarId carId;

private List<Tyre> tyres;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

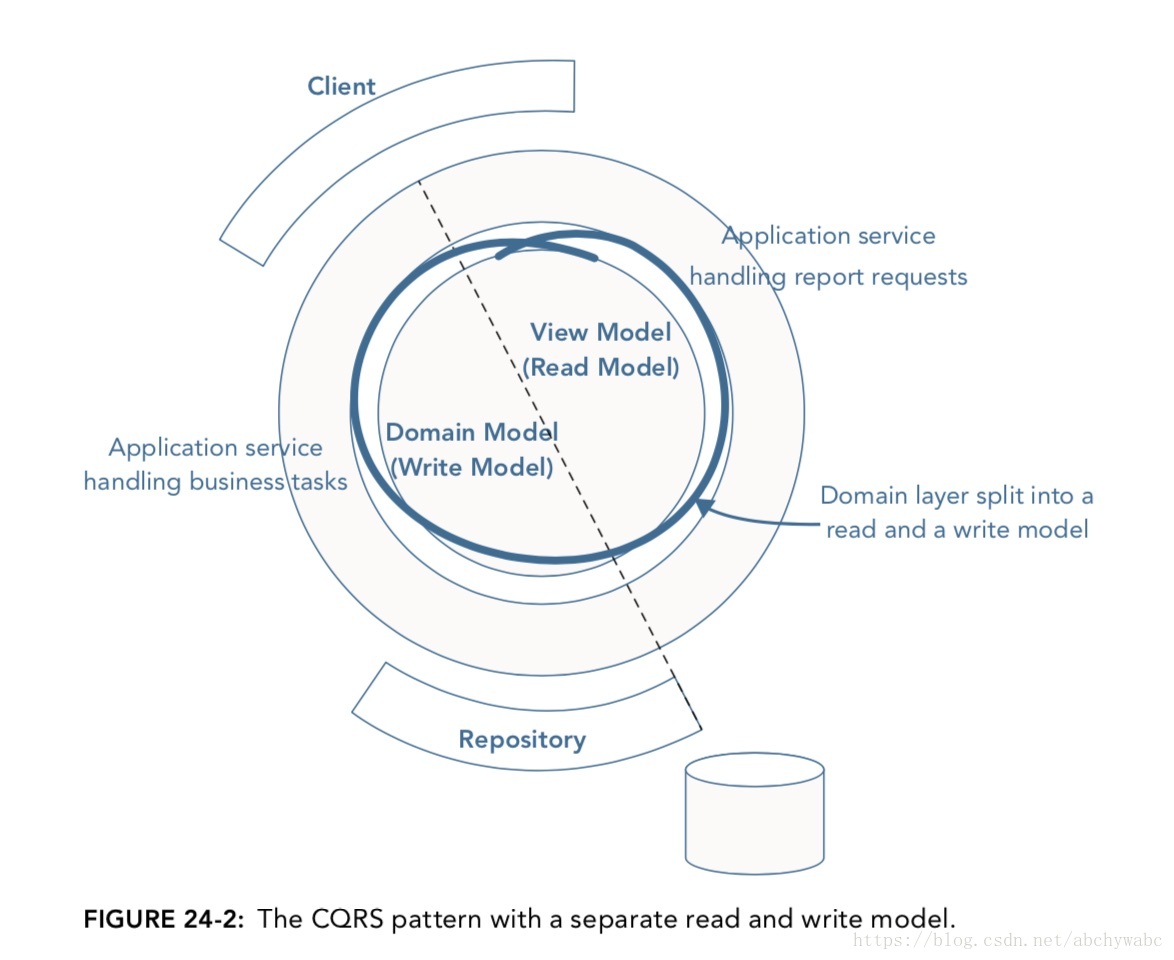
但结果是，这样的实现并不符合我们设计的domain object。而我们在entity中引用了entity(Order中直接引用了Delivery)，等于我们设计了一个很大的aggregate，包含了很多entity。当然这又与aggregate的设计策略有关，我们假设这不是我们想要的策略。

而更主要的原因是我们的模型并不契合我们的需求。

我们的需求是什么？就后端要实现的内容来说，我们需要返回画面需要的内容。

而我们的模型–domain model并不是以满足画面显示需求而设计的模型。domain model描述的是业务上的逻辑，所以模型会有很多的行为（类方法）。另外也多次强调过数据完整性这个概念，这也是domain model所关注的。从结论上说，我们倾向比较小的模型（小型aggregate）。

然而要实现画面显示功能时，domain model具体有什么行为，需要保证那些数据完整性显然不是我们关心的。而且模型的大小一般也不是考虑的因素，画面需要的信息我们都必须返回。



这个图引用了《patterns, principles, and practices of domain-driven design》一书的CQRS部分的插图。

书中的观点是，读与写本来就会有不同的需求，需要各自的模型。它将用于画面查询的模型称作view model，而view model与domain model是处理不同问题的，当我们使用不合适的模型来处理问题时，显然就会觉得比较变扭。

CQRS

既然模型不合适，我们就可以选择合适的模型。其实就订单列表这个功能来说，我们是不是已经有了需要的模型？对，DeliveryDTO！问题在于，我们是用domain object转成DTO的。自然我们可以想到，又没有办法跳过domain object这个步骤呢？这里我们就要说说CQRS这个思想。

CQRS，全称Command Query Responsibility Segregation。直译过来就是命令查询的职责分离。

Command指的是增加/更新数据的处理。

Query指的是查询数据的处理。它不会造成数据的变更。

我们将这两种处理用不同的模型来应对。

这又和ddd有什么关系呢？刚才说过ddd的domain model关注实际的业务行为以及业务上的数据完整性等问题。而这些问题在增加/更新数据时起着较大的作用。也就是说Command和ddd的契合度比较高。

而对复杂的画面提供数据这种功能，一般来说它对业务行为和数据完整性没有很多的要求，所以Query并比一定需要domain object。

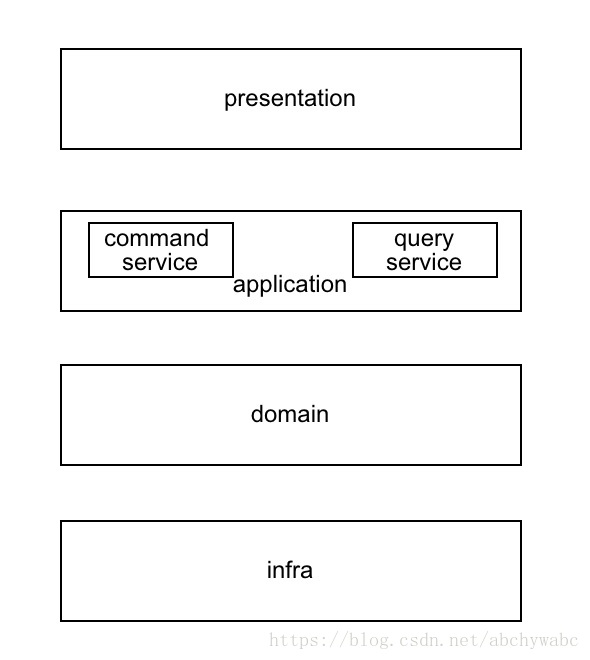
如何实现

首先，CQRS已经超过了ddd的范畴，它属于如何使用ddd的一种策略，或者说在Command处理时使用ddd，而在Query处理时则使用更合理的实现方式。

那具体实现层面，介绍一种做法。

之前说过ddd的一种分层方法是分成

presentation层，application层，domain层，infra层



在application层中，我们将本来的application service分成两种service，command service与query service。

Command处理

首先是command处理。command service会专门负责command处理。command处理包括创建与更新类型的处理。比如例子中下订单，取消订单的功能就会属于command service。

public OrderCommandService {

private OrderRepository orderRepository;

public orderId createOrder(UserId userId, Long carModelId){

Order order = Order.createOrder(userId, carModelId);

orderRepository.save(order);

return order.getOrderId();

}

public void cancelOrder(User userId, OrderId orderId){

Order order = orderRepository.findOrderById(new OrderSpecificationById(orderId));

if(order.getUserId() != userId) {

throw new IllegalAccessException();

}

order.cancel();

orderRepository.save(order);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

Query处理

query部分的处理采用的方法是不通过domain model，直接获取数据。概念是这样，实际上的实现是多种多样，一个重要的因素是用来与数据库交互的框架。个人感觉一些能直接写sql语句的框架是比较不错的选择。比如一个叫jooq的框架。

我们用jooq来写一个查询某个用户的订单列表的例子

@Component

public class OrderQueryService {

private final DSLContext jooq;

public List<OrderDTO> getOrderList(Long userId){

return jooq.select()

.from(ORDER)

.join(DELIVERY).on(DELIVERY.ORDER\_ID.eq(ORDER.ORDER\_ID))

.join(CAR).on(CAR.CAR\_ID.eq(ORDER.CAR\_ID))

.where(ORDER.ORDER\_ID.eq(userId))

.fetch()

.map(record ->

OrderDTO.builder()

.orderId(record.get(ORDER.ORDER\_ID))

.deliveryId(record.get(DELIVERY.ORDER\_ID))

.deliveryAddress(record.get(DELIVERY.ADDRESS))

.deliveryStatus(record.get(DELIVERY.STATUS))

.carId(record.get(CAR.CAR\_ID))

.carModelId(record.get(CAR.MODEL\_ID))

.carImageUrl(record.get(CAR.IMAGE\_URL))

.build()

);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

其实我们就像写一条query语句一样

select \* from order o

inner join delivery d on o.order\_id = d.order\_id

inner join car c on order.car\_id = c.car\_id

where o.user\_id = ${userId}

1

2

3

4

注意，我们这里直接把query的结果转换成了画面需要的dto（用于画面显示的模型，我们也可称作view model）。

我们可以总结成一个原则，query service接受参数返回dto。query service中不允许存在domain object，如entity，value object, repository等。

注意点

query service与command service不要相互调用

query,与command他们服务于不同的目的，使用的模型也不同，所以他们应该没有交集。

例子中command service中引用domain object(entity, value object, repository, specification)，而query service使用了view model(例子中我们使用了dto)与query(jooq)。

请注意不是所有查询操作都要使用query service的，比如下面的写法是不正确的。

public OrderCommandService {

private OrderQueryService orderQueryService;

private OrderRepository orderRepository;

public void cancelOrder(User userId, OrderId orderId){

Order order = orderQueryService.findOrderById(orderId); // queryService返回了order这个entity !!!

if(order.getUserId() != userId) {

throw new IllegalAccessException();

}

order.cancel();

orderRepository.save(order);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

这个写法中query service返回了domain object。这是违反了query service的原则。当我们进行的command操作需要检索时，我们还是必须通过repository来对数据库进行检索。

可以将QueryService定义为接口

如果你觉对在queryService中出现了infra的实现不太满意的话，也可以使用控制反转，在application层定义queryService接口，在infra层进行实现。

使用queryService时，不要涉及业务逻辑

使用queryService有一个隐含的前提。之前提到过使用合适的model来处理合适的问题。因为查询这样的业务通常不涉及业务逻辑(domain)，所以我们自然不需要domain model。

然而即使画面显示这样的功能有时候也是会涉及业务逻辑的。比如说我们沿用上面购买汽车的例子。我们假设有一个订单确认的画面，画面中需要显示订单价格，其中价格的计算比较复杂，它包含促销打折等要素。

当然这样的实现方法会有很多种，我们说一个不太好的实现方式。那就是在query service中查询是否有促销打折，然后根据情况对订单的价格进行计算。价格计算显然是一种业务逻辑，他应该在domain object中实现。

具体的解决方法，个人觉得也有多种多样，最简单的就是虽然它是用于画面显示的功能，我们也不使用query service而把它写进一个application service中（不是command service）。当然这样会增加application层的复杂度，增加了一个service的种类。

另一种方法是我们在订单确认前就允许order的创建。用一个状态表示它未确认，已确认。order中的金额事先计算好。这样query service就可以查询到order表中的数据。这里可以根据实际的业务需求进行选择。

总结

CQRS的思想史把操作分成query, command两种操作，用各自合适的模型与框架来实现两种处理，query处理时我们可以选择mybatis, jooq这种抽象度较低，能直接写query的框架。

CQRS一方面让query的处理更加的高效，同时他会增加程序设计的复杂度。你必须判断什么样的查询需要使用query service。而当项目有一定规模时，查询可能和业务逻辑不能很轻易的分离，也会阻碍query service的使用。

所以你可以选择在处理效率遇到瓶颈时才引入query service。或者选择在功能设计时尽量避免查询功能涉及业务逻辑，但后者往往不完全取决于工程师的意见，具体的需求也是重要的因素，这不仅考验程序员的程序设计能力，还要要求工程师有沟通能力去与domian expert（具体对业务有了解，或者设计产品的人）交涉。

参考

《patterns, principles, and practices of domain-driven design》

————————————————

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/80879514

# **领域驱动设计(domain driven design)战略篇之一 战略 & Bounded Context**

之前的文章主要从战术层面的角度介绍了ddd。在岛国也被称为轻量级ddd。它提供了一些概念如aggregate, entity, domain event和一些设计模式如repository, specification来帮助我们建模和设计。各种战术还有能够扩展的地方，有机会还会再写下去。不过从这篇文章开始会写一写ddd战略方面的知识。

战略 vs 战术

究竟什么是战略与战术？他们有什么区别？

目测这两个词都来源于战争（自己的理解），战术是偏微观的策略，目的是取得某场战斗或者战役的胜利。诱敌深入，敌进我退之类的可能都属于战术吧。而战略是偏宏观的策略，目的是赢得一场战争，它所关注的不限于军事方面，可能资源的调配，甚至会牵涉到外交等。

扯远了～在领域驱动设计[domain driven design]中战术与战略的概念亦是如此。战术是微观层面的。在之前的文章中的例子中，基本都是为了解决某个十分具体问题而进行的模型设计。这些设计会反应具体的细节。比如一个如何去识别一个entity,一个entity会有什么样的行为。而这些设计会直接反映到我们的代码上。

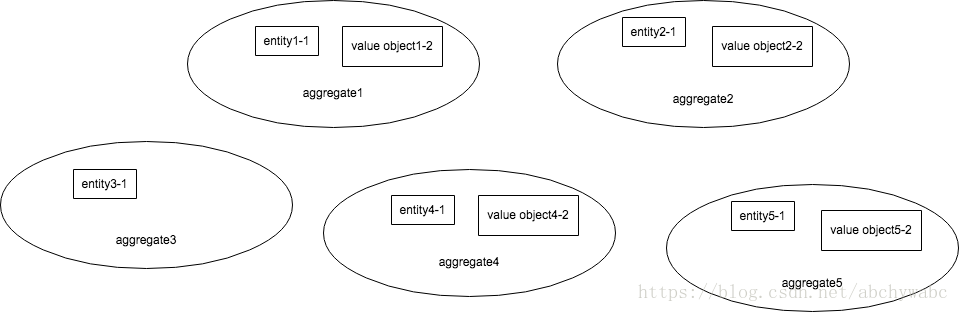
为什么需要战略

从很抽象的概念上来说，我们应该不难理解为什么我们需要战略。战略与战术他们需要解决的是不同层面的问题。当我们谈宏观问题时，自然是需要战略的。但话是这么说，什么从程序开发的角度来说，什么才是宏观问题？什么又是微观问题？这个好像很难用语言来具体地定义。

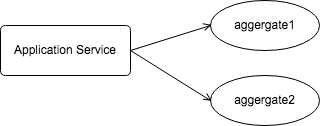
假设我们完全不考虑宏观问题。我们想象（想象…）一下只用ddd中介绍的战术知识来进行设计。如今互联网产品的行业可能比较现实的制造产品的方案会是，先做proto type，然后做mvp(minimum viable product 只包含核心功能的产品)，然后在mvp的技术上进行后续开发。

mvp

当我们在开发mvp的阶段时，产品的需求可能相对简单。我们分析需求，业务逻辑，然后设计能够描述这些业务的模型。可能10个到20个左右的aggregate就能解决问题。画个简图来表示我们定义的aggregate和其中的entity, value object。



这个时候你的application层的application service可能不需要与很多aggregate交互。



产品进化

随着产品经理对产品的认识更加深入，我们需要追加更多的开发，为了应对新的需求，我们要调整现有的模型，也需要增加新的模型来应对。某个时点，我们可以能已经到达了像图中的一个状况。我们已经无法再在一张图中精确描述aggregate中的元素。

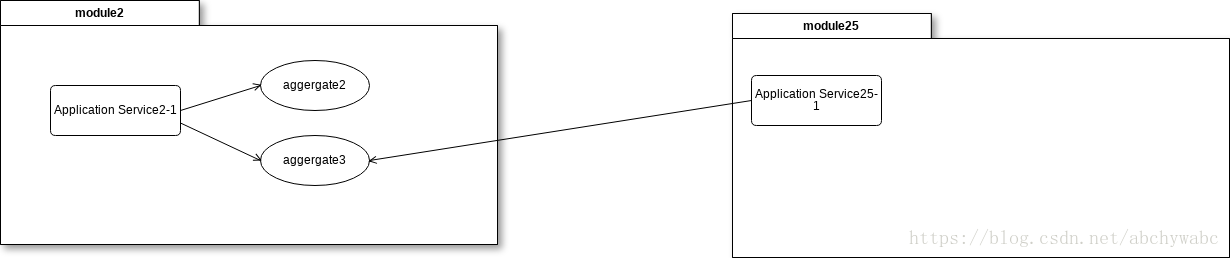


而此时，我们的application层的中的一些application service可能已经不得不与非常多的aggregate进行交互。



当application service包含了很多的aggregate操作时，我们可能不知不觉增加了各个模型间的耦合度，同时也冒着把业务逻辑写进application service的风险。理想情况下，我们希望所有的业务逻辑都在domain层，而application service是很薄的。

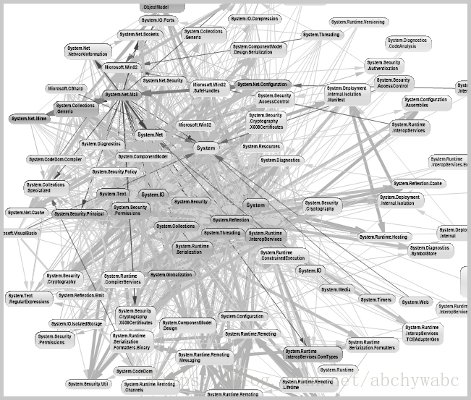
另外随着模型量的增加，我们对模型之间的关系也会慢慢模糊。再加上如果是在多人项目中，程序员之间交流会有极限。对于系统的认知里为下降，各种“神奇现象“会开始发生。如下图，模块2里的aggregate3，在设计之初可能设想只有模块2里的东西会对aggregate3进行操作，但项目大了，参与开发的人多了，之前完全没有想到的不知名模块25也对aggregate3进行了操作。对模块25的修改可能影响到了模块2。



注: 没有专门写文章说明module(模块)这个概念。如果你是java工程师，就把它当成package就行。

进化的究极体—-big ball of mud

当产品的复杂度不断增加，而我们有没有去控制控制这种复杂的话，我们的系统会成为ddd中称作big ball of mud（大泥球）的东西。



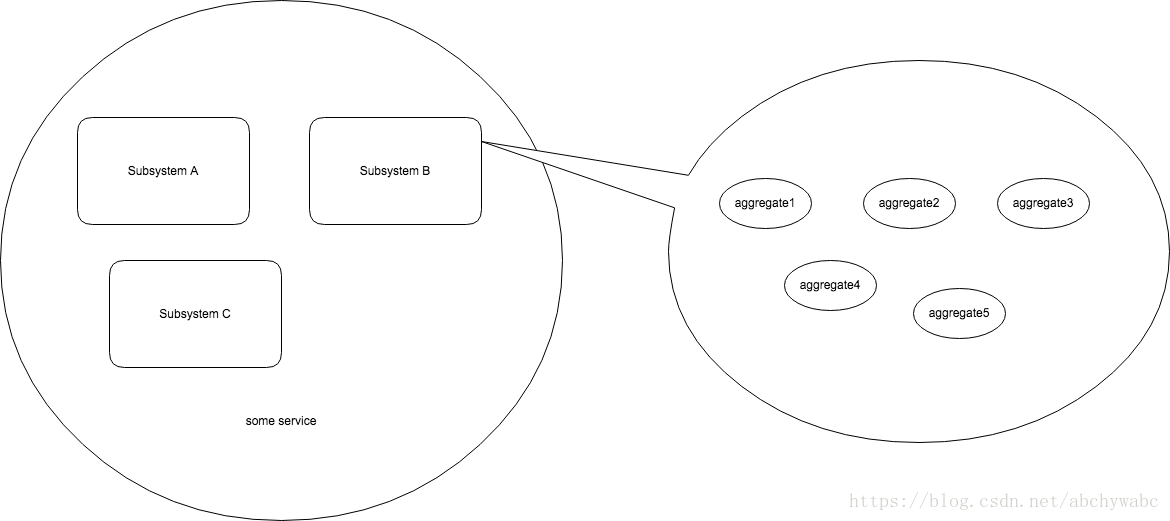
这个说法应该是比较形象的，系统的各种模块已经耦合在一起，很难直观地，整体地理解这个系统的构成。当我们对它无法整体地理解它，各种各样的误判就会发生，系统的各种行为变得难以捉摸。对于程序员来说，在这样的系统基础上开发会是个噩梦。

如何解决这种问题？

前面描述的方法哪里有问题？我们把注意力只集中在了小范围的设计上，我没有对全局进行把控。可能我们在各个模块的层面上，设计是合理的，当把视野放宽时，系统设计却变得很混乱。

很显然，除了战术层面的aggregate, module这样定义边界的概念，我们还需要一个抽象度更高，范围更广的概念来帮助我们把系统进行切分。其实这也符合一般的系统设计的思路，

在传说中的waterfall模式时代，我们会把需求定得十分明确，把尽可能地具体到每个细节（至少岛国就是这种状况。各种纸质设计文档，一审再审，一改再改，最后文档叠起来的高度会不亚于“上海中心“）。但即使在那种设计难以应对变化的年代，在程序设计时也不会一下子就设计到最细小的部分。还是会先分大的区块。比如一个服务，我们需要用多少个子系统来组成它，如何分割子系统，子系统之间如何交互。这写就是战略层面上要考虑的问题。而在子系统中我们会在进行更细节部分的设计。如下图所示，先将服务分成A, B, C三个子系统。其中的子系统B考虑它应该包含什么样的模块。



在[领域驱动设计中]提出了bounded context这个概念。直译过来时”边界确定的上下文”。这是一个帮助我们进行系统粗粒度切分的概念。

Bounded Context

个人认为Bounded Context强调的是边界这个概念。这个思想有一个前提，万能的模型是做不到的。我们建模是为了描述或解决现实的问题。而现实问题是复杂的，我们无法建立一个模型是面面俱到，一般我们只会截取自己所关注的一个切面，对它进行建模。

同一个概念，不同模型

比如在一个管理户籍时，我们只关注一个人的姓名，身份证号，住址。在写简历时，我们会关注一个人拥有的技能与工作经验。而在¥%@时，我们会关注一个人的性格等。在现实生活中同样的一个人的概念，它十分复杂，拥有很多的侧面，我们在建模时不会指望构建一个完美的模型，它能够复刻一个人。即使有这样的模型，它肯定也是极度复杂的（在本人目前的认知下），也很有可能超过了我们可以理解的范围。

因此，模型在某一个bounded context中，它是有限的，仅描述它所关注的部分。按上面的例子来说，在一个户籍管理系统中，Person可能就是下面的一个类

class Person (val identification: Identification,

var firstName: String,

var lastName: String,

var address: String) {

}

1

2

3

4

5

6

而在一个求职服务里，Person会是这样的

class Person (val id : Long,

val userName: String,

var emailAddress: String,

var resumeId: ResumeId){

}

class Resume(val id: ResumeId,

var education: String,

var professionalExperience: String

var skills: List<Skill>) {

}

1

2

3

4

5

6

7

8

9

10

11

尽管这两个Person在现实中是相关的，即使他们共同存在于某一个平台，在他们各自的bounded context中，他们是相对独立的，可以想像，这两个Person拥有的行为也会是不同的。

不同的语境，不同的概念

在英语中Account的这个词有不同的意思，一个表示银行的账户，一个表示账号。Account在银行业务，与网络业务的bounded context里会是完全不同的概念。

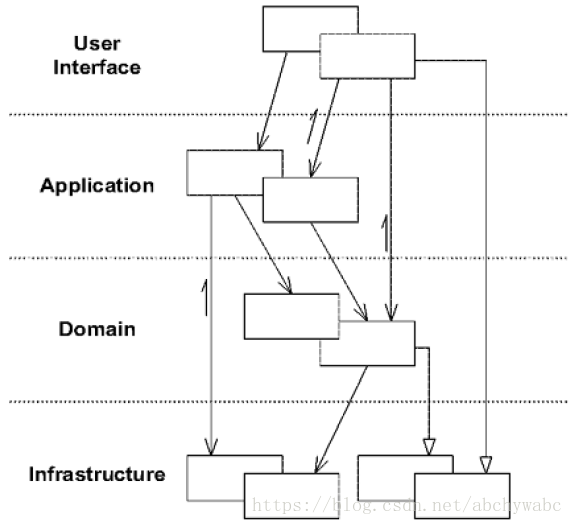
我们必须认识到这种思路和自然语言是不同的。在自然语言里，一个词可能是多义的，意义也可能是宽泛的。Bounded Context这个概念要求我们在Context中的模型必须是单义的，意义相对狭义的。当一个Bounded Context中的一个模型类开了Bounded Context,它的意义与行为会发生改变。

Bounded Context让我必须认清我们建模时所应该关注的地方。明确我们所关心的点后，我们能更有目的性地建模。同时把不应该在某一个Context关注的东西放到另一个Context。再次提醒！同一个概念，可以出现在在不同的Bounded Context，但是他们会以不同的模型来展现。

如何实现Bounded Context

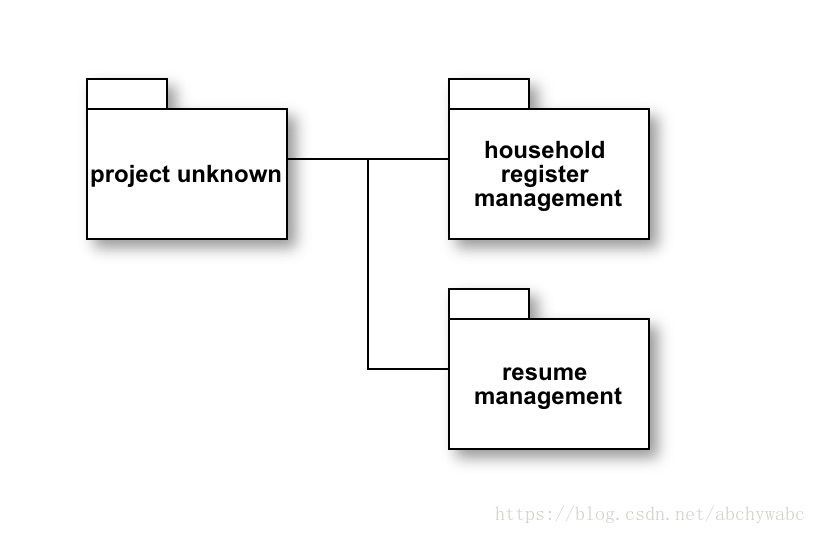
很遗憾的是，[领域驱动设计]一书中并没有给出操作层面实现Bounded Context的方法。这也导致了Bounded Context的实现成为了一个众说纷纭的话题。这里列举一些实现的方式。

首先Bounded Context并不是一个层，不是要在presentation, application, domain, infrastructure的4层架构中在加一个层。

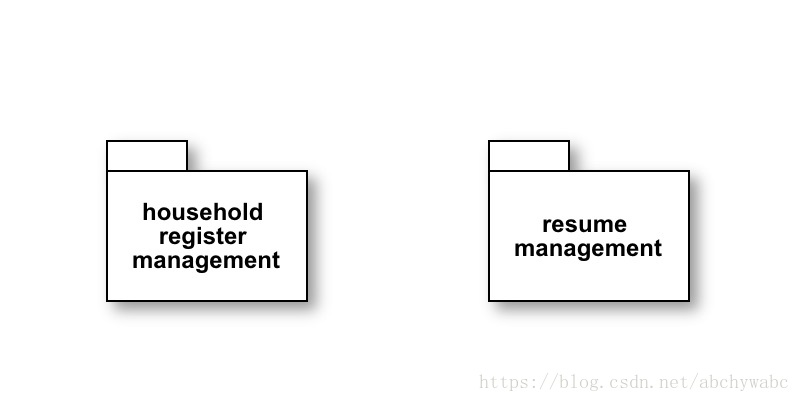


namespace级别的bounded context

如果是java的画，这个就是通过包(package)来构建bounded context。



project级别的bounded context



project又是一个很有歧义的词，如果你是java工程师，那project就是代码层面的一个叫project的单位。使用如果你使用intellij开发的话，它会被称为module。总之他们有各自的代码，如果不使用build工具，project A是无法引用project B的代码的。

micro-service级别的bounded context

这个可能不必赘述了把。用一个微服务来实现一个context。

比较

从边界的强度来说

namespace < project < micro-service

使用namespace来实现的话，如果编程语言不对namespace之间类的引用有限制的话，这种边界定义是很弱，很容易打破边界（有意无意）。对越界（package A是否引用package B的类）的检查可能需要人力，从长期来看成本很大。

project级别的话，基本不会有代码层面的越界。但是不同不同的project A, project B可能引用了同一个数据库，这可能会造成隐性的越界。比如A，B中的模块要对同一张数据表进行修改。但在A并不知道有B的存在。A所拥有的数据会在A不知情的情况被修改。

micro-service的边界是最牢固的。如果你不开发接口，基本上做不到越界。

边界初期投入上来说

namespace < project < micro-service

这个显而易见。如果你选择micro-service,意味着你选择了一个复杂的系统架构，你可能还需要一大堆配套的框架。而如果是namespace的话投入基本就是0。

如何实现bounded context是否是该尽早决定的

很显然如何实现bounded context没有一个完美的方案（可能本人没有找到）。无论选择那个选项，都必须做出牺牲，这就成了令人烦恼的权衡问题。

个人觉得，如要考虑的有如下几个要素

1. 对边界的认识是清晰明确

2. 团队技术实力

3. 项目规模

在这几个要素都十分明确的情况下，我们会有足够的信息作出选择。

但如果在边界还不是很清晰，对domain的理解也不成熟，就直接用micro service进行划分是比较危险的。因为一刀割下去割得太彻底，之后发现割错了也没办法缝合了。你可能觉得那只要把需求理解清楚后，把边界明确定义不就行了。很可惜现实情况是需求也是在变的，我们很在初期就把握所有的需求。

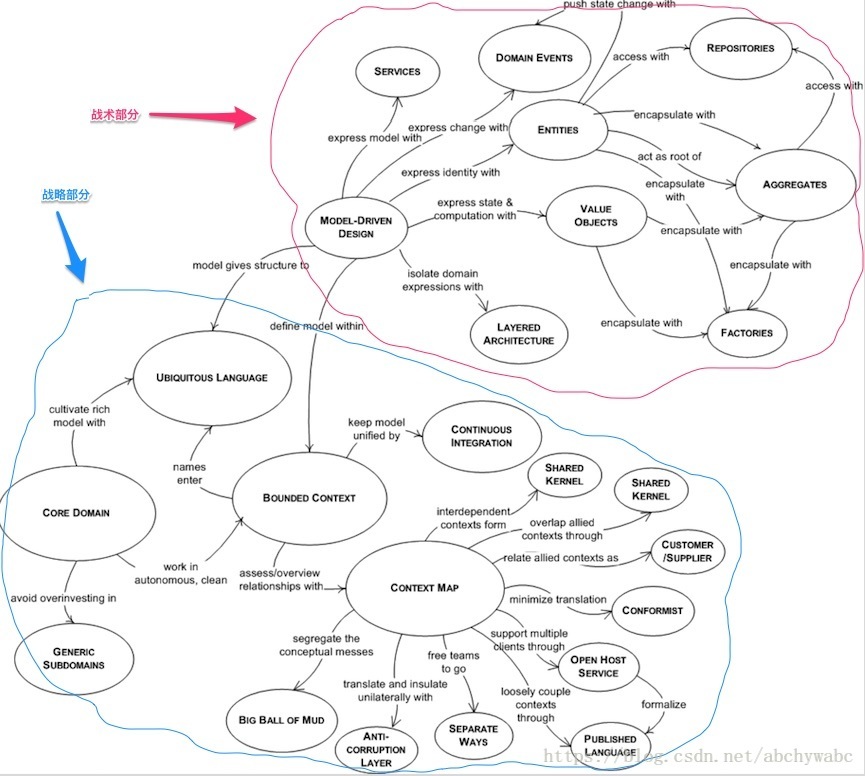
所以如果在各个要素都还不明确的情况下，比较实际的做法可能是，一开始选择边界强度低的实现方法，随着对domain理解的深入，明确边界后再过度到强度高的实现（micro service）。不过这种方法有一个隐性前提，边界分割方法的更改可以低成本地实现。如果monolithic的服务能够轻易地变换成微服务，那就不用逼我们在早期冒险做决定。这需要借助其他的一些手段。比如由尽量高的自动测试覆盖率等。

总结

这次讲了ddd的战术与战略的区别，战术帮我们优化局部，战略帮我们把握全局。

介绍了bounded context的这个概念以及实现的方式。那究竟如何能把一个系统分解成bounded context呢？之后的战略篇文章会进行说明。

网上发现了一张不错的图，总结了在ddd中，战略和战术都有那些概念。战略部分的概念之后会进行讲解。



参考资料

BoundedContext

————————————————

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/81410560

# **领域驱动设计(domain driven design)战略篇之二 Bounded Context**

之前的一篇文章谈了战略ddd的重要性与Bounded Context这个概念，最近在油管上看到一个2017年关于domain driven design的演讲。如下



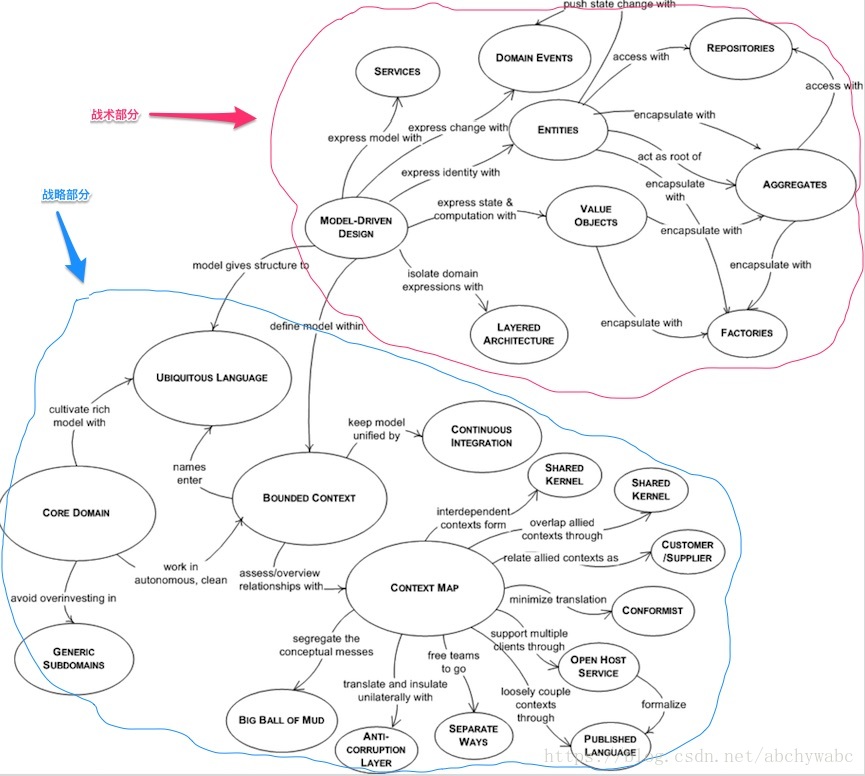
感觉与自己现在讲的主题十分相关，正好在这里展开说一下。

他认为Bounded Context可能是ddd中最重要的概念。而悲剧地是ddd的社群可能更倾向于讨论战术方面的各种设计模式，他顺便也吐槽了一下ddd这本书的章节编排。

我也很赞同他的看法。具体说一下这两点。

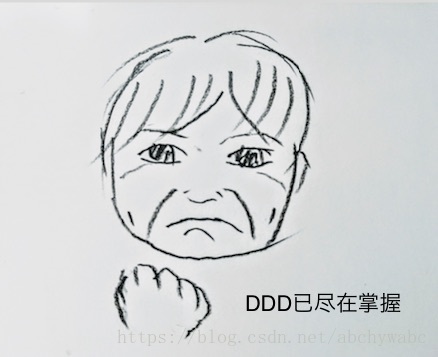
1. ddd的战略部分容易被忽略

还是摆出这张图

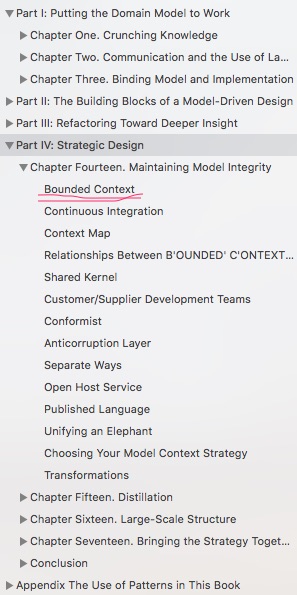


学习ddd的人在开始可能更热衷于战术层面设计模式。如上图里repository, specification, aggregate那部分概念，以及在代码层面如何实现它们。

本人自己也是如此，在读《domain driven design》时基本把战略部分给跳过了。而且战略部分也是书的最后几章。读完了战术篇的部分，就已经觉得



也由于此书对读者的极其不友好，读到最后几章时基本气力已经耗尽。(如下图，全书共17章，Bounded Context居然到了14章才登场。)



然而,在实际的项目中，如果我们只是去讨论如何实现这些设计模式，那会是很可笑的。最重要的是如何实现项目的需求。虽说如此，本人在写博客时，还是先从ddd的战术开始写的。原因大概有这么几点

1.1 这反映了自己对ddd的认知历程。本人一开始也深陷ddd战术的话题，而对战略层面的东西基本选择跳过。所以认为很多人学习时也会遇到同样的问题，如果一开始就谈战略，可能基本都会被大脑过滤掉。

1.2 由于1.1的缘故，战略方面的知识也是自己薄弱的部分。

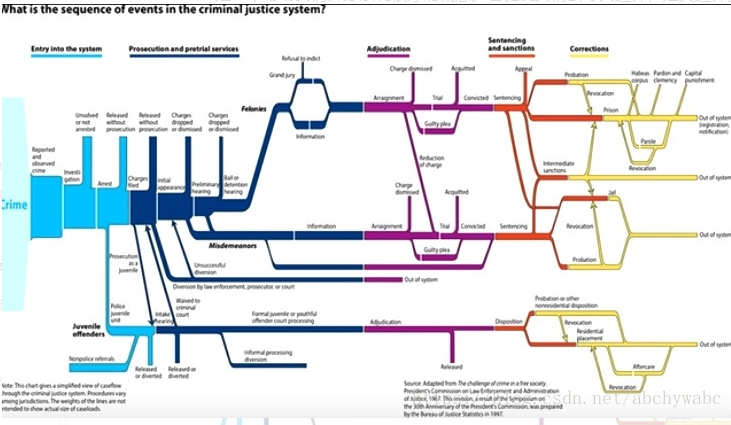
1.3 ddd战术的设计模式比较具体，容易在实际项目上使用。而想战略层面的Bounded Context的话，很抽象，这就造成这个说明很困难。（尤其在自己的理解不是很透彻，很多情况都是人云亦云的情况下）

从结论上说，自己还是做了妥协，把比较重要的部分挪到了后边来讲。

2. Bounded Context是很重要！

Bonded Context是一个帮助我们拆分系统很好的概念。它促使我们去寻找边界，定义边界。这里先说说Jimmy Bogard在视频中的例子。Jimmy Bogard他曾经为美国德州政府建造一个管理青少年犯罪事物的系统。

他最后整理出的一张表示这个系统的图。

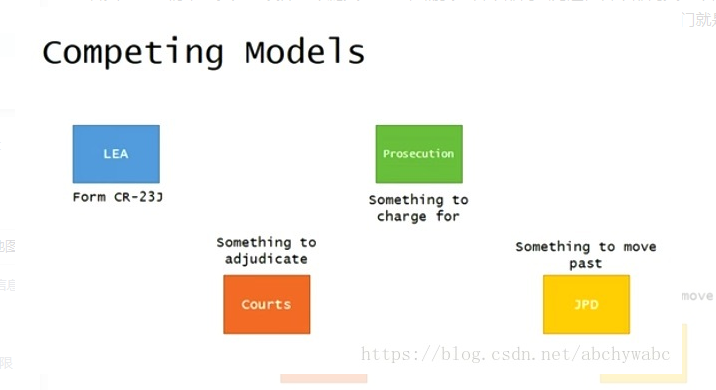


当然这只是想表达这个系统的复杂度。

他还整理了使用这个管理系统的用户，大概有七十多种。



接着头痛的事情来了。为了设计一个庞大的系统，需要与各个部门去沟通，各个部门又有不同的概念和用语，而让各个部门的人对某一个概念拥有共同的认识也基本是不可能的。比如对青少年的解释，各个部门就是各有各的说法。



执法部门(图中的LEA)根本就不管什么青少年。他们觉得这只是某张表格。。。法院(图中的Courts)觉得青少年就是判决对象。检察机关(图中的Prosecution)认为青少年就是起诉的对象。面对这种境地，他们尝试了创建非常抽象的概念，如人Person，但实际上在实际的业务中，没有使用person这个概念。这就悲剧地表示建立的模型没有对应的业务概念。这当然会有悖于model driven design这种思想。

此外各个部门关注的事情也不同。比如执法机关只关注”抓人”。法院则关注“审判“，检察机关则直观“案子“（可能指的是起诉）。



如此多的人物登场，而所关心的业务也各不相同，其实是一种征兆。尤其出现如“青少年“这种貌似是同一个概念却有着不同解释及业务的情况时，我们就应该意识到我们在不同的Context里讨论某个概念，划分界线，使用不同的模型可能是一个更好的选择。

当时Jimmy Bogard在承包这个项目时也觉得，他在做一个系统想要满足所有人，所有需求，这实在是难度太高，根本不切实际。不过他还是义无反顾地奋勇向前了。原因是甲方只想做一个系统。而当时也没有SOA, microservice什么的，他可能也没有拆分系统的好的手段。

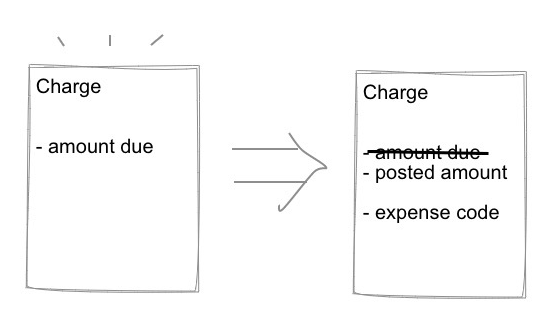
最后项目时上线了。但这份经历基本算是用来反省的。这个项目最终也成了big ball of mud,没有人愿意触碰的系统。据说被开发人员称作一拖翔。

不关注Bounded Context的潜在危险

从上边的例子，应该不难想象，当系统过于庞大时不拆分Bounded Context的痛苦。再结合一下《domain driven design》一书说说无视Bounded Context的危险。

对模型的错误认识

在《domain driven design》一书中就有一个例子。说是某公司要开发一个customer-invoicing(发票)模块，需要一个叫Charge（收费）的对象。expense code(支出代码)与posted amount(记账金额)两个属性。开发团队A发现另一个开发团队B的代码已经存在了Charge这个对象。不过这个Charge没有expense code和posted amount属性。于是团队A对加上了expense code，又发现Charge有一个叫amount due（未支付金额）的属性，他们以为和posted amount应该是一个意思，于是就把这个amount due改成了posted amount。



团队A之后又为Charge加上了几个方法，在不影响既有功能的情况下作出了customer-invoicing模块。同时因为Charge对象是借用既有的东西，Charge对象里包含了和customer-invoicing完全没有关系的各种逻辑。当然这个模块发布后各种报错。很明显本来就不该用同一个Charge。(当然这个故事听上去是有点扯~)

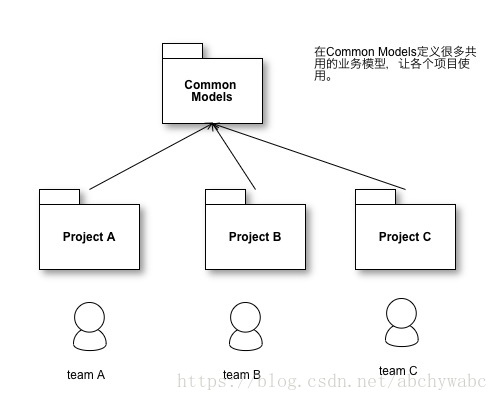
如果再用Bounded Context划界的情况下，开发团队A可能就会意识到，在他们的这个Bounded Context里还不存在Charge这个对象。而开发团队B使用的Charge是在他们的Bounded Context中才有意义的模型，团队B设计Charge这个模型的目的和用途可能是不一样的，所以不能拿来就用。

从自己的开发经验上讲也是如此。对没有划界线的东西自然会倾向于重用。这很容易导致我们用现有的模型(model)套用到新的问题(domain)，而忽略了是否应该使用完全不同的模型。最终这也会导致我们的系统变得过于复杂，难以维护。更可怕的是我们可能制造出与需求完全不同的东西。

维护统一模型的沟通与理解的成本

如果一个系统由多个团队同时开发，而没有明确的界线。团队A修改的东西要注意不影响团队B，如果有影响还要与团队B去协调。然而当项目规模到达人脑难以完全把握的程度，即使有团队沟通后也不一定能百分百地解决问题。

按自己的工作经验来说讲，团队按照不同的业务应对分了组，理论上在业务层面有各自的界线，但开发时并没有在意这种界线，反而去一些建立一些横跨多个服务的共用模型(如下图的common models)，让这个模型应对不同服务的需求。



结果界线其实就是名存实亡。结果就是明明团队A要开发一个功能，但这个功能会影响到团队B的东西，接着就要想团队B去确认影响，如果团队B的人记忆力不好还得全部自己查。

当然这样的设计有一定的历史原因。如项目本身只有一个database。project A, project B, project C的数据其实是共用的。然后用orm生成了所有table的java entity类。自然习惯性地都把他们放到Common Models中，最终也变成了big ball of mud。

ddd对Bounded Context的定义

Bounded Context是一个抽象的概念，而且在各种设计思想中被借鉴。说了一大堆它的重要性，最后希望能给这个抽象概念有更好的解释。

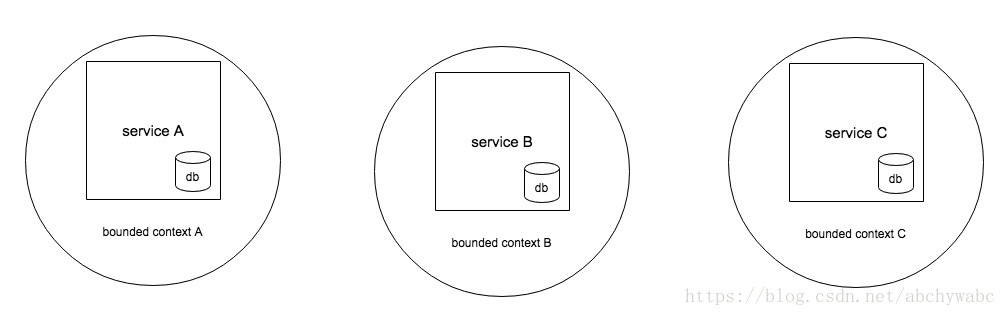
很遗憾在《domain driven design》一书中虽然出现了Bounded Context,但感觉没给出很明确的定义。而关于Context则有这样的说明。

The context of a model used in an example in this book is that particular example section and any later discussion of it. The model context is whatever set of conditions must apply in order to be able to say that the terms in a model have a specific meaning.

Context是能够明确一个模型意义所需要的所有条件集。Bonded则突出了“界线“这个意思。Bounded Context是有界线的Context,在这个Context中的模型，意义是明确的，没有歧义的。

从自己的项目经验上来讲，自己还是比较实用主义。在系统划分时就是使用了Bounded Context这个用语，而没有十分在意它的实体是什么东西（可能就没有实体）。不过至少Bounded Context这个概念给出了一个启示。不要去设计一个能够应对所有情况的模型。把模型要解决的东西限定在某一个范围。而这个范围就是一个Bounded Context。

另外这几年微服务也十分流行，在《Building Microservices》一书中，作者也谈到Bounded Context。个人也觉得microservice其实于Bounded Context有着映射关系。



总结

这次通过实际的例子又讲了讲Bonded Context这个概念。帮助我们更有目的性地去建立明确的模型。那当我为某个项目拆分出各个Bounded Context后，接下来又能做什么呢？下一篇文章讲讲Context Map

————————————————

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/81900794

# 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>

<https://blog.csdn.net/abchywabc/article/details/80037998>

<https://blog.csdn.net/abchywabc/article/details/80171967>

<https://blog.csdn.net/abchywabc/article/details/81410560>

<https://blog.csdn.net/abchywabc/article/details/81900794>