链滴

领域驱动设计 DDD 之聚合

作者：valarchie
原文链接：https://ld246.com/article/1574070377089
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

为什么需要聚合?

当我们设计一个订单模块，用户下单时，我们需要确保用户的余额可供支付这笔订单，并且保存这个单。通俗的理解就是当下单的时候，必须生成订单表记录，并且检查用户余额是否足够支付，并修改户的余额表。再转换到我们领域驱动设计中，我们必须利用订单模型和账户模型联合来完成操作，并查保证业务规则（余额可供支付）。那此时有两点缺陷，在保证事务的应用服务中，这些领域知识（务规则）便从领域模型泄露应用服务层。代码中除了订单模型和账户模型在同一个应用服务中这点之，客户程序员很难知道在下单请求中订单模型和账户模型之间必须在同一个事务中进行修改。

聚合描述

每个聚合都有一个根和一个边界，边界内定义了聚合的内部有什么。根则是聚合所包含的一个特定的体。外部对象可以引用根，但不能引用聚合内部的其他对象，聚合内的对象之间可以相互引用，除了实体外，其他实体拥有本地标识。

举个例子:

在汽修厂的软件中会使用到汽车的模型，这里的汽车就是根实体，因为它具有全局唯一的标识：汽车别号。汽修厂想要跟踪每台汽车上四个轮胎的使用情况。轮胎在汽车里才是实体，它们拥有本地标识如汽车的四轮分为左前轮，右前轮，左后轮，右后轮。当轮胎报废了之后我们便不再关心这些轮胎的命周期了，我们也不会在系统中寻找某一个轮胎现在安在哪台车上。因此，汽车便是这个聚合的根实，轮胎就在这个聚合的边界之内。

再举个例子:

在订单模块中，下单操作必须生成订单主表和购买商品附表，订单主表即是一个根，通过这个根我们以找到当时这笔订单购买了哪些商品。但是我们并不会单独去查购买商品附表里的记录，脱离了订单表，这些记录就没有了意义。

定义

我们应该将实体和值对象分门别类的聚集到聚合当中，并定义聚合的边界。在每个聚合当中，选择一实体作为根。并通过根来控制边界内其他对象的所有访问。只允许外部对象保持对根的引用。对内部员的临时引用可以被传递出去，但仅在一次操作中有效。由于根控制访问，因此不能绕过它来修改内对象。这种设计有利于确保聚合中的对象满足所有固定规则，也可以确保在任何状态变化时聚合作为个整体满足固定规则。

不变性和一致性边界即是聚合的设计依据和精髓。

不变性和一致性边界

这里的不变性指的是业务规则，该规则应该始终保持一致。一致性边界的意思是单个事务的修改范围。原则上我们应该在一个事务里只修改一个聚合。

如果理解这个不变性和之前实体和值对象的不变性的区别呢?

之前的实体和值对象中的不变性，针对的是局部的规则，而聚合中的一致性针对的是，各个实体或者对象共同维持的规则

聚合的主要作用

1. 主要为了维护对象生命周期内的完整性。

关于聚合的生命周期，在初期的时候我们使用工厂Factory来创建聚合或者复杂对象，在生命周期的期末期我们使用资源库Repository来提供检索对象或者持久化对象。虽然工厂和资源库本身不属于领，但我们在使用聚合的过程当中，可以更容易的操作聚合。

2. 通过定义清晰的所属关系和边界，在这个边界中的模型元素在生命周期内必须维护一致性，通俗的就是业务规则。

聚合就是一组相关对象的集合，我们将他作为数据修改的单元。通俗的说，比如以往我们在一个事务需要修改三张表，那这三张表映射出的实体和值对象就可以组成一个聚合。

聚合大小（边界范围）

在具有复杂关联的模型中，要想保证对象更改的一致性是很困难的。不仅互不关联的对象需要遵守一固定规则。而且紧密关联的各组对象也要遵守一些固定规则。然而，过于谨慎的锁定机制又会导致多用户之间毫无意义地互相干扰，从而使系统不可用。

因为聚合涉及到了事务，因为如果事务边界太大的话，会导致经常发生冲突。所以我们应该合理的设事务边界。一方面为了对象组合上的方便而将聚合设计得很大，另一方面，我们设计的聚合又可能因过于贫瘠而丧失了保护真正不变条件的目的。所以聚合的边界不可过大也不可过小，但推荐聚合应该保证遵守固定规则的前提下尽可能的小。

小聚合：其中只包含最小数量的属性或者值类型属性，这里的最小数量表示所需的最小属性集合即不也不少。怎么理解属性时所需的呢？即那些必须与其他属性保持一致的属性。比如三角形的三个角一要等于180度，你把这三个角放置到不同的聚合的话，那就不合适了。小聚合不仅有性能和可伸缩性的好处，也有助于事务的成功执行，即减少事务冲突。

举个例子：

比如我们设计一个订单模块。订单模块涉及两张表，一张是订单主表，一张是订单商品附表。我们不直接去访问订单商品附表，而是应该通过订单主表的id，间接的获取对应的订单商品列表。在领域模中则是，通过资源库获取订单根实体，再通过订单根实体去获取内部的订单列表。（订单列表可以一始就在资源库方法中封装好，也可以延迟加载）。这边我们可能会限制一个特殊规则，订单商品数量能超过订单时间的尾数。因此在每次创建订单或者修改订单的时候都不能破坏这个规则。这个规则是对订单主表和订单商品附表的一个联合规则。他们应该放置在同一个事务里。如果没有在一个事务中话，假设当前商品数量差一就会达到最大数量，此时张三和李四同时对这个订单新增了一个商品。此就破坏了这个规则。

聚合并不是简单理解为比如人类聚合由头、身体、四肢等实体或者值对象组成。它不一定映射到现实某一个事务的完整轮廓。比如我们的业务规则，只限定头和身子必须满足一定的比例，不关心四肢部。那么这时候的聚合只包含了头和身体。

聚合特征：

1. 根实体具有全局的标识，它最终负责检查固定规则。

2. 边界内的实体具有本地标识，这些标识只在聚合内部才是唯一的。

3. 聚合外部的对象不能引用根实体之外的聚合内部对象。根实体可以将内部实体的引用传递给它们，只能临时使用。或者传递一个值对象的副本出去，而不用关心它发生了什么变化。

4. 只有根实体才能直接通过数据库直接查询，其他对象必须通过遍历关联来发现。（意思是根实体可从资源库中的某个方法获取，但是聚合内的其他对象，资源库不提供直接的访问方法，而是在资源库生成聚合的时候，直接添加进聚合）

5. 根实体可以保持其他根实体的引用。

6. 删除操作，比如删除聚合边界内的所有对象。

7. 当对聚合边界内的任何对象做了修改时，整个聚合的所有固定规则都必须被满足。

设计原则

原则一：通过唯一标识去引用其他聚合

1. 引用聚合和被引用的聚合不可以在同一个事务中进行修改

2. 如果你在试图在单个事务中修改多个聚合，这往往意味着此时的一致性边界是错误的，发生这样的况通常是我们遗留了某些建模点，或者尚未发现通用语言中的某个概念。

3. 当试图修改多个聚合的话，我们也应该采用最终一致性而非原子一致性。

```
public class Order {
    private Product product;
}
```

应改为利用唯一标识去引用其他聚合。

```
public class Order {
    private ProductId productId;
}
```

原则二：利用应用层来处理聚合内的依赖关系，避免在聚合中使用资源库或者领域服。

如果实在需要特定的复杂依赖关系，可以在聚合的命令方法中使用领域服务和资源库。

原则三：在边界之外使用最终一致性。

如果单次用户请求，的确需要修改多个聚合实例的话，比如在一个聚合上执行命令方法时，如果还需在其他的聚合上执行额外的业务规则，那么则需要使用最终一致性。我们可利用消息中间件之类的机，完成最终一致性。

但以上这些原则不是完全不能打破的。当出现以下一些情况时，我们可以做出妥协。

1. 方便用户界面：用户界面可能允许用户一次性的给多个对象定义共有的属性，然后再进行批量处理。

2. 缺乏技术机制：最终一致性需要诸如消息，定时器，后台线程之类的技术，当我们的项目没有去使这些技术的时候。就只能在单个事务中去修改多个聚合实例。

3. 全局事务：考虑遗留技术和企业政策所带来的英雄。

4. 查询性能：有时候还是在一个聚合中维护其他聚合的直接引用，有助于资源库的查询性能。

我们不应该去找借口来打破聚合原则。长远的来看，遵循聚合原则对整个项目是有益的。有时候原子致性在技术、性能、资源上都不好实现的话。 我们可以弱化一致性，我们可以转而使用最终一致性。

关于DDD的理解各有不同，欢迎网友评论一起探讨。