实现领域驱动设计心得

1. 分层
2. 领域
3. 领域服务
4. 聚合根
5. 实体
6. 值对象

<https://www.zhihu.com/question/25089273>

http://www.cnblogs.com/netfocus/archive/2012/02/12/2347938.html

http://blog.jobbole.com/99298/

分层

领域驱动设计分层

基础结构层，领域层，应用层和表现层。

表现层（UI层）：负责界面展示

应用层：负责业务流程。

领域层：负责领域逻辑。

基建层：负责提供基建。

分类的依据：越往上，预期变动越频繁；越往下，预期变动越少。

对程序员来说，表现层和基建层很容易分清，一个只管展示，一个只管提供持续储存，网络传输等基本设施，都没有业务等参与，容易混淆的是应用层和领域层，在这两层中存在的，就是应用模型和领域模型。

模型属于哪一层，有个粗略的判断方式，如果一个实体（entity）和针对实体的增删改查，就属于领域层；如果是一个场景，比如出现在UI菜单上的选项，就属于应用层。

前者是针对实体的操作，每一个实体都只有增删改查这样的操作，与之相反的是，要完整实现“购买商品”这个场景，也许需要检查库存，创建订单，创建交易等多个操作。

聚合

聚合根

具有全局唯一的标识，

聚合根，实体，值对象的区别

从标识的角度：

聚合根具有全局的唯一标识，而实体只有在聚合内部有唯一的本地标识，值对象没有唯一标识。

从是否只读的角度：

聚合根除了唯一标识外，其他所有状态信息都理论上可变，实体是可变的，值对象是只读的。

从生命周期的角度：

聚合根有独立的生命周期，实体的生命周期从属于其所属的聚合，实体完全由其所属的聚合根管理维护。

聚合根，实体，值对象对象之间如何建立关联

聚合根到聚合根：通过ID关联

聚合根到其内部的实体，直接对象引用。

聚合根到值对象，直接对象引用。

实体对其他对象的引用规则：1）能引用其所属聚合的聚合根，实体，值对象；

2）能引用外部聚合根，但推荐以ID的方式关联，另外也可以关联某个外部聚合内的实体，但必须是ID关联，否则就出现同一个实体的引用被两个聚合根持有，这是不允许的，一个实体的引用只能被其所属的聚合根持有。

聚合（以及聚合根）：聚合表示一组领域对象(包括实体和值对象)，用来表述一个完整的领域概念。而每个聚合都有一个根实体，这个根实体又叫做聚合根。举个简单的例子，一个电脑包含硬盘、CPU、内存条等，这一个组合就是一个聚合，而电脑就是这个组合的聚合根。博主觉得关于聚合的划分学问还是挺大的，需要在实践中慢慢积累。同一个实体，在不同的聚合中，它可能是聚合根，也可能不是，需要根据实际的业务决定。**聚合根是聚合所表述的领域概念的主体，外部对象需要访问聚合内的实体时，只能通过聚合根进行访问，而不能直接访问**。

如何识别聚合与聚合根

含义：一个界限上下文可能包含多个聚合，每个聚合都有一个根实体，叫做聚合根。一个界限上下文代表着某个独立的业务场景，这个业务场景操作的唯一对象总是该上下文边界的聚合根。

如何识别实体和值对象

实体是包含实体数据和行为的结合体。实体会被传递到聚合外部，但是因为实体在任何聚合外部都不能被修改，否则聚合的不变性无法确保，所以不能被修改，必须制度。

值对象和实体的区别：（1）实体拥有唯一标识，而值对象没有；（2）实体允许变化，而值对象不允许变化；（3）判断两个实体相等的方法是判断实体的标识相等，而判断两个值对象相等的标准是值对象内部所有属性值相等；

例子：

订单模型

order(订单)至少对应于一个orderLineItem(订单条)，对orderLineItem为主体，完全是在面对orderLineItem在做工作，比如向Order中增加明细，修改OrderLineItem的某个明细对应的商品的购买数量，从Order中移除某个明细，等等类似操作，我们从来不会从OrderLineItem为出发点去执行一些业务操作；另外，从生命周期去理解，那么OrderLineItem离开Order没有任何存在的意义，也就是说OrderLineItem的生命周期是从Order的。所以，我们可以很确信的回答，OrderLineItem是一个实体。

帖子模型

帖子和回复之间的关系比较弱，不像订单和订单项一样是一个内聚的关系；帖子和回复以及回复的回复之间就像是一个链，帖子是链中的第一个结点，回复的回复是第三个结点。帖子如果被删除了，就是这个链的头被移除了，但这个链的后面的结点还是在那里的。

实际上你没理由一定要把帖子和回复内聚在一起成为一个整体的，这样做会带来问题的，比如性能问题和并发问题，当我要发表回复时，每次都要取出帖子，然后往里面增加回复，然后保存整个帖子，在并发的情况下这是不行的。

如果帖子和回复在一个聚合内，聚合意味着“修改数据的一个最小的单元”，聚合内的所有对象都要看成是一个整体最小单元进行保存，这么要求是韵味聚合的意义是维护聚合内的不变性，数据一致性。仔细分析，你会发现帖子和回复之间没有数据一致性要求，所以你不需要设计在同一个聚合内。

从场景的角度，我们有发表帖子，发表回复，这两个不同的场景，发表帖子创建的是帖子，而发表回复创建的是回复，但是订单就不一样了，我们有创建订单，修改订单这两个场景，这两个场景都是围绕这订单这个聚合展开的。

识别顺序：**先找出哪些实体可能是聚合根，再逐个分析聚合根的边界，即该聚合根应该聚合哪些实体或者值对象；最后再划分界限上下文。**

聚合边界确定法则：根据不变性规则，不变性规则有两类：1）聚合边界必须具有哪些信息，如果没有这些信息就不能称为一个有效的聚合；2）聚合内的某些对象的状态必须满足某个业务规则。

领域服务和领域事件

领域服务的定义：当领域中某个重要过程或转换操作不属于实体或值对象的自然职责时，应该在模型中添加一个作为独立接口的操作，并将其申明为service。

领域服务层的作用：处理实体和实体之间的关系。DDD的设计原则之一是尽量丰满领域模型，主张充血的领域模型。处理实体和实体之间的关系一般来说领域模型，所以最好还是放在领域层。

我们举个例子：比如权限管理，如果需要一个功能，将指定的用户赋予制定的权限，比如接口这样定义：

void AssignPower(TB\_USERS oUser,TB\_ROLE oRole)

按照我们前面聚合的划分，这个接口是应该放在用户聚合还是角色聚合里面呢？很显然都不合适，这歌接口包含两个聚合的实体，所以像这种情况，可以考虑领域服务去解决。

领域事件

领域事件解耦代码。代码间解耦用事件，系统间解耦用MQ。

例子：订单系统和计费系统。大部分的业务流程都是由订单系统触发，然后计费系统做出相应的变更，最终，我们决定使用领域事件来讲订单系统和计费系统。通过orderEvent和BillEvent来将两个系统解耦开，然后将Event放到一个公共的Module中达到Module级别的解耦，效果完美。

总结使用领域事件解耦业务流程的应用场景：

1. 如果一个业务流程需要贯穿几个不同的受限上下文，那么可以通过以发布领域事件的方式来避免上游系统耦合下游系统，这种解耦方式的收益最大，因为其有利于后期系统的拆分。
2. 如果在同一个受限上下文中，也可以通过发布领域事件的方式来达到领域间解耦。

仓储repository

仓储作用之一是用于数据的持久化。从架构层面来说，仓储用于连接领域层和基础结构层，领域层通过仓储访问存储机制，而不用过于关心存储机制的具体细节。按照DDD的设计原则，仓储的作用对象的领域模型的聚合根，也就是说每个聚合都有一个独立的仓储。

使用仓储的意义：

1. 站在领域层更关心领域逻辑的层面，仓储作为领域层和基础结构层的连接组件，使得领域层不必过多的关注存储细节。在设计时，将仓储接口放在领域层，而将仓储的具体实现放在基础机构层，领域层通过接口访问数据存储，而不必过多的关注仓储存储的细节。
2. 站在架构的层面，仓储解耦了领域层和ORM之间的联系，这一点也就是很多人设计仓储模式的原因，比如我要更换ORM框架，我们只需要改变仓储的实现即可，对于领域层和仓储的接口基本不不需要做任何改变。

工厂

工厂不负责对象的持久化，工厂把持久化职责委托给仓储来完成，仓储不应该直接和应用层打交道，对于整个领域层来说，和应用层打交道的是service接口，而和持久化打交道的是repository接口。

举一个场景来说，根据订单号获取一个聚合复杂对象采购订单，对采购订单进行修改后再进行保存。这个时候和持久化层存在两次交互，第一次是数据的读取，第二次是修改后数据的存入。

对于数据读取，到领域层则是Factory需要实例化一个聚合对象并返回应用层。而Factory将该工作分解到聚合里美的每一个子实体，子实体通过Repository接口获取到ResultSet并进行OR转换后返回，Factory将拿到的所有实例化对象进行聚合返回一个完整的聚合对象实例。对于数据存储，仍然应该是Factory接管该操作，然后对数据进行分解后分别调用聚合中的每一个实体的仓储接口本身的保存方法，对数据进行持久化，在Factory层进行完整的事务控制并返回结果。

六边形架构

六边形每条不同的边代表了不同种类型的端口，端口要么处理输入，要么处理输出。该架构存在两个区域，分别是外部区域和内部区域，在外部区域中，不同的客户均可以提交输入，而内部的系统则用于获取持久化数据，并对程序输出进行存储（比如数据库），或者在中途将输出转发到另外的地方（比如消息）。

六边形架构也称为端口和适配器。对于每种外界类型，都有一个适配器与之相对应。外界通过应用层API与内部进行交互。

通常来说，我们不用自己实现端口，我们可以将端口想成HTTP，而将适配器想成Java的Servlet或JAX-RX的REST请求处理类。或者，我们可以为NServiceBus或RabbitMQ创建消息监听器，在这种情况下，端口是消息机制，而适配器则是消息监听器，因为消息监听器将负责从消息中提取数据，并将数据转化为应用层API（领域模型的客户）所需的参数。任何客户都可能向不同的端口发出请求，但是所有的适配器都将使用相同的API。

应用程序通过公共API接收客户请求。应用程序边界，即内部六边形，也是用例（或用户故事）边界。换句话说，我们应该根据应用程序的功能需求来创建用例，而不是客户数量或输出机制，当应用程序通过API接收到请求时，它将使用领域模型来处理请求，其中便包括对业务逻辑的执行。因此，应用层API通过应用服务的方式展现给外部。

六边形一般分为三层：

领域层（domain layer）：核心业务逻辑，一般不包含任何技术实现或引用。

端口层（Port Layer）：领域层之外，负责接收与用例相关的所有请求，这些请求负责在领域层中协调工作。端口层在端口内部作为领域层的边界，在端口外部则扮演了外部实体店角色。

适配器层：端口层之外，负责以某种格式接收输入，及产生输出。比如，对于HTTP用户请求，适配器会将其转换为对领域层的调用，并将领域层传回的响应进行封送，通过HTTP传回调用客户端，在适配器层不存在领域逻辑，它的唯一职责时在外部世界与领域层之间进行技术性的转换。适配器能够与端口的某个协议相关联并使用该端口，多个适配器可以使用同一个端口，在切换到某种新的用户界面时，可以让新界面同时使用相同的端口。

例子：

领域层想要获取某一个领域对象，来进行业务操作实现，然后告诉端口层说：“端口小弟，哥需要一个 XXX Domain Model，立马去叫人搞！”，端口小弟心想，老大发话了，得赶紧的啊，然后就在自己胸前，贴出了这样一段告示：能逮到 XXX Domain Model 的适配器杀手们，请速速到俺这里，必有重赏！告示一贴出，适配器杀手们蜂拥而至，然后根据自己的能力来进行判断，毕竟 XXX Domain Model 也不是那么容易擒服，也不是随便一个适配器杀手就能搞定的，这需要一定的能力，最后能做的适配器杀手进行揭此告示。

其实六边形架构，从内到外的一个过程的体现，反过来，从适配器到领域层也一样，这种方式一般是接收用户请求处理开始。

在六边形架构中，适配器是以组件性质的方式提供服务，他和领域层进行联系要通过端口层，这个端口层可以看作是服务的一种协议或规范，这就是SOA和REST的用武之地，来扮演适配器层和端口层的角色。