链滴

多研究些架构，少谈些框架（2）

作者：bnvnvnv
原文链接：https://ld246.com/article/1531125125530
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)

2017-6-12 曹祖鹏 https://github.com/JoeCao/JoeCao.github.io/issues/4

上篇我们聊了微服务的DDD之间的关系，很多人还是觉得很虚幻，DDD那么复杂的理论，聚合根、对象、事件溯源，到底我们该怎么入手呢？

实际上DDD和面向对象设计、设计模式等等理论有千丝万缕的联系，如果不熟悉OOA、OOD，DDD是使用不好的。不过学习这些OO理论的时候，大家往往感觉到无用武之地，因为大部分的Java程序开发生涯是从学习J2EE经典的分层理论开始的（Action、Service、Dao），在这种分层理论中，我基本没有啥机会使用那些所谓的“行为型”的设计模式，这里的核心原因，就是J2EE经典分层的开发式是“贫血模型”。

Martin Fowler在他的《企业应用架构模式》这本书中提出了两种开发方式“事务脚本”和“领域模”，这两种开发分别对应了“贫血模型”和“充血模型”。

事务脚本开发模式

- 事务脚本的核心是过程，可以认为大部分的业务处理都是一条条的SQL，事务脚本把单个SQL组织为一段业务逻辑，在逻辑执行的时候，使用事务来保证逻辑的ACID。最典型的就是存储过程。当然们在平时J2EE经典分层架构中，经常在Service层使用事务脚本。

使用这种开发方式，对象只用于在各层之间传输数据用，这里的对象就是“贫血模型”，只有数据字和Get/Set方法，没有逻辑在对象中。

我们以一个库存扣减的场景来举例：

- 业务场景

首先谈一下业务场景，一个下订单扣减库存（锁库存），这个很简单

先判断库存是否足够，然后扣减可销售库存，增加订单占用库存，然后再记录一个库存变动记录日志作为凭证）

- 贫血模型的设计

首先设计一个库存表 Stock，有如下字段

字段	说明
SpuId	Standard Product Unit （标准化产品单元）
SkuId	stock keeping unit(库存量单位)
StockNum	可销售库存
OrderStockNum	订单占用库存

设计一个Stock对象(Getter和Setter省略)

```
public class Stock {
    private String spuId;
    private String skuId;
    private int stockNum;
    private int orderStockNum;
}
```

● Service入口

设计一个StockService，在其中的lock方法中写逻辑

入参为(spuId, skuId, num)

实现伪代码

```
count = select stocknum from stock where spuId=xx and skuid=xx
if count>num {
    update stock set stocknum=stocknum-num, orderstocknum=orderstocknum+num  where
skuId=xx and spuId=xx
} else {
    //库存不足，扣减失败
}
insert stock_log set xx=xx, date= new Date()
```

● ok，打完收工，如果做的好一些，可以把update和select count合一，这样可以利用一条语句完成旋，解决并发问题（高手）。

小结一下：

有没有发现，在这个业务领域非常重要的核心逻辑 -- 下订单扣减库存中操作过程中，**Stock对象根本用出现**，全部是数据库操作SQL，所谓的业务逻辑就是由多条SQL构成。Stock只是CRUD的数据对象已，没逻辑可言。

● 马丁福勒定义的“贫血模型”是反模式，面对简单的小系统用事务脚本方式开发没问题，业务逻辑杂了，业务逻辑、各种状态散布在大量的函数中，维护扩展的成本一下子就上来，贫血模型没有实施服务的基础。

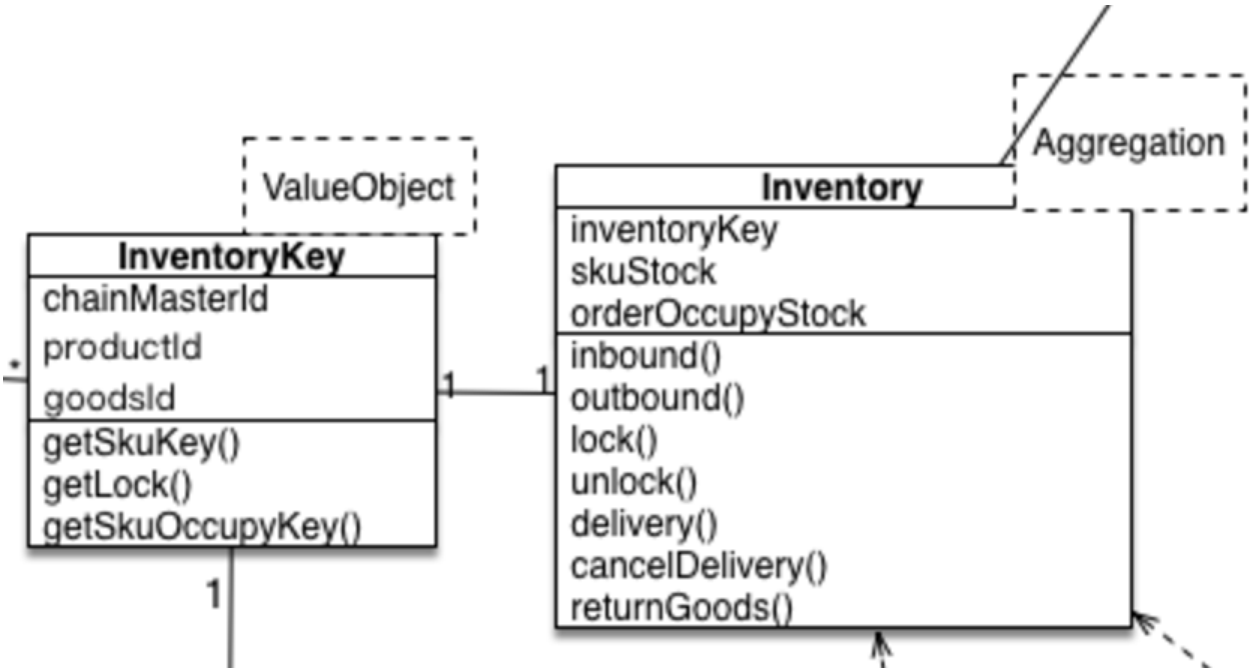
● 虽然我们用Java这样的面向对象语言来开发，但是其实和过程型语言是一样的，所以很多情况下大用数据库的存储过程来替代Java写逻辑反而效果会更好，（ps：用了Spring boot也不是微服务），

领域模型的开发模式

● 领域模型是将数据和行为封装在一起，并与现实世界的业务对象相映射。各类具备明确的职责划分使得逻辑分散到合适对象中。这样的对象就是“充血模型” 。

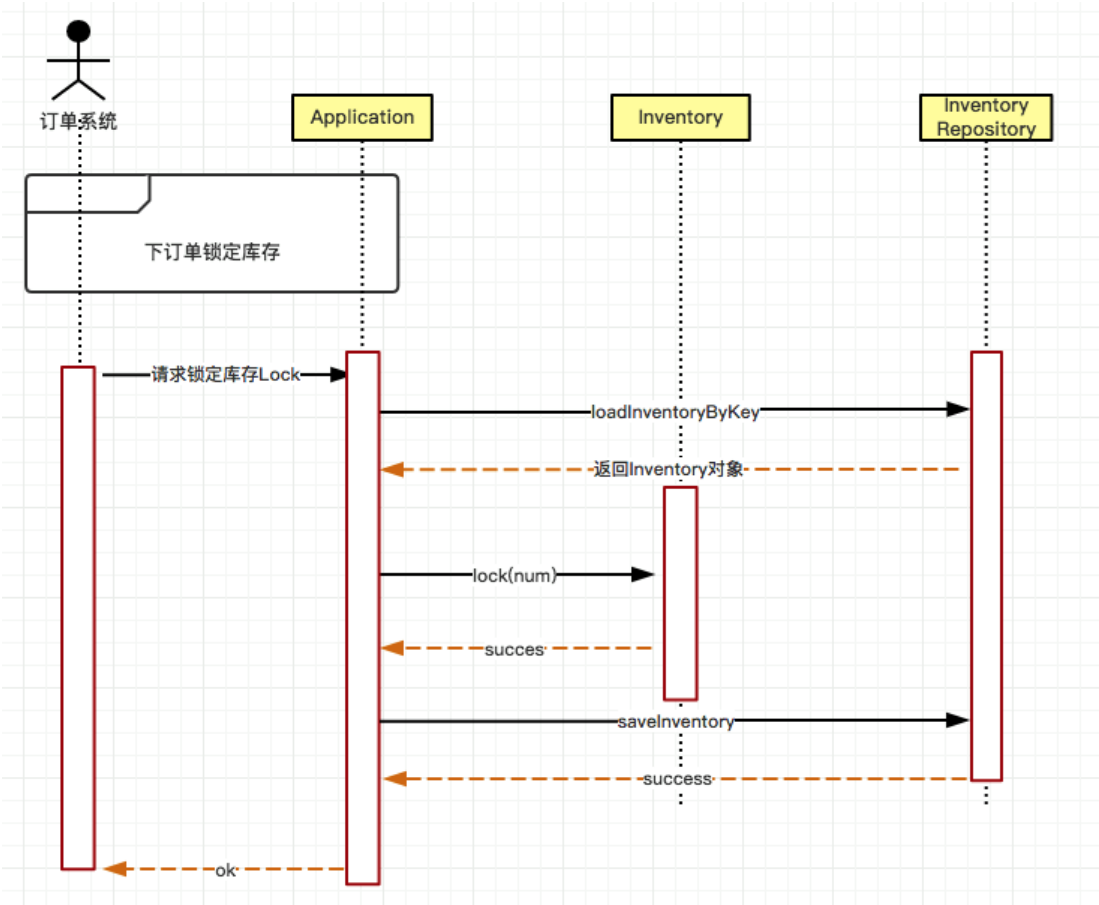
● 在具体实践中，我们需要明确一个概念，就是领域模型是有状态的，他代表一个实际存在的事物。

是接着上面的例子，我们设计Stock对象需要代表一种商品的实际库存，并在这个对象上面加上业务辑的方法



这样做下单锁库存业务逻辑的时候，每次必须先从Repository根据主键load还原Inventory这个对象然后执行对应的lock(num)方法改变这个Inventory对象的状态（属性也是状态的一种），然后再通过epository的save方法把这个对象持久化到存储去。

完成上述一系列操作的是Application,Application对外提供了这种集成操作的接口



原文链接：多研究些架构，少谈些框架（2）

领域模型开发方法最重要的是把扣减造成的状态变化的细节放到了Inventory对象执行，这就是对业逻辑的封装。

Application对象的lock方法可以和事务脚本方法的StockService的lock来做个对比，StockService完全掌握所有细节，一旦有了变化（比如库存为0也可以扣减），Service方法要跟着变；而Application这种方式不需要变化，只要在Inventory对象内部计算就可以了。代码放到了合适的地方，计算在合层次，一切都很合理。这种设计可以充分利用各种OOD、OOP的理论把业务逻辑实现的很漂亮。

- 充血模型的缺点

从上面的例子，在Repository的load 到执行业务方法，再到save回去，这是需要耗费一定时间的，是这个过程中如果多个线程同时请求对Inventory库存的锁定，那就会导致状态的不一致，麻烦的是对库存的并发不仅难处理而且很常见。

贫血模型完全依靠数据库对并发的支撑，实现可以简化很多，但充血模型就得自己实现了，不管是在存中通过锁对象，还是使用Redis的远程锁机制，都比贫血模型复杂而且可靠性下降，这是充血模型来的挑战。更好的办法是可以通过事件驱动的架构来取消并发。

领域模型和微服务的关系

上面讲了领域模型的实现，但是他和微服务是什么关系呢？在实践中，这个Inventory是一个限界上文的聚合根，我们可以认为一个限界上下文是一个微服务进程。

不过问题又来了，一个库存的Inventory一定和商品信息是有关联的，仅仅靠Inventory中的冗余那点品ID是不够的，商品的上下架状态等等都是业务逻辑需要的，那不是又把商品Sku这样的重型对象引了这个微服务？两个重型的对象在一个服务中？这样的微服务拆不开啊，还是必须依靠商品库？！

请参考下一篇，通过事件驱动架构来完成领域间的松耦合。