多研究些架构，少谈些框架（3）

作者：bnvnvnv
原文链接：https://ld246.com/article/1531125207717
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)

2017-6-16 曹祖鹏

接上篇，我们采用了领域驱动的开发方式，使用了充血模型，享受了他的好处，但是也不得不面对他来的弊端。这个弊端在分布式的微服务架构下面又被放大。

事务一致性

事务一致性的问题在Monolithic下面不是大问题，在微服务下面却是很致命，我们回顾一下所谓的ACD原则

- Atomicity - 原子性，改变数据状态要么是一起完成，要么一起失败
- Consistency - 一致性，数据的状态是完整一致的
- Isolation - 隔离线，即使有并发事务，互相之间也不影响
- Durability - 持久性， 一旦事务提交，不可撤销

在单体服务和关系型数据库的时候，我们很容易通过数据库的特性去完成ACID。但是一旦你按照DD拆分聚合根-微服务架构，他们的数据库就已经分离开了，你就要独立面对分布式事务，要在自己的码里面满足ACID。

对于分布式事务，大家一般会想到以前的JTA标准，2PC两段式提交。我记得当年在Dubbo群里面，本每周都会有人询问Dubbo啥时候支撑分布式事务。实际上根据分布式系统中CAP原则，当P(分区容)发生的时候，强行追求C（一致性），会导致（A）可用性、吞吐量下降，此时我们一般用最终一致来保证我们系统的AP能力。当然不是说放弃C，而是在一般情况下CAP都能保证，在发生分区的情况，我们可以通过最终一致性来保证数据一致。

例：

在电商业务的下订单冻结库存场景。需要根据库存情况确定订单是否成交。

假设你已经采用了分布式系统，这里订单模块和库存模块是两个服务，分别拥有自己的存储（关系型据库），

在一个数据库的时候，一个事务就能搞定两张表的修改，但是微服务中，就没法这么做了。

在DDD理念中，一次事务只能改变一个聚合内部的状态，如果多个聚合之间需要状态一致，那么就要过最终一致性。订单和库存明显是分属于两个不同的限界上下文的聚合，这里需要实现最终一致性，需要使用事件驱动的架构。

事件驱动实现最终一致性

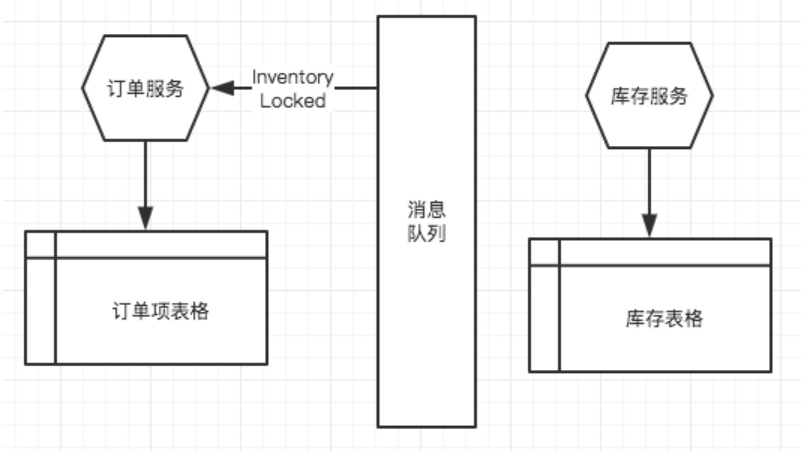
事件驱动架构在领域对象之间通过异步的消息来同步状态，有些消息也可以同时发布给多个服务，在息引起了一个服务的同步后可能会引起另外消息，事件会扩散开。严格意义上的事件驱动是没有同步用的。

例子：

在订单服务新增订单后，订单的状态是“已开启”，然后发布一个Order Created事件到消息队列上

库存服务在接收到Order Created 事件后，将库存表格中的某sku减掉可销售库存，增加订单占用库，然后再发送一个Inventory Locked事件给消息队列

订单服务接收到Inventory Locked事件，将订单的状态改为“已确认”



有人问，如果库存不足，锁定不成功怎么办？ 简单，库存服务发送一个Lock Fail事件， 订单服务接后，把订单置为“已取消”。

好消息，我们可以不用锁！事件驱动有个很大的优势就是取消了并发，所有请求都是排队进来，这对们实施充血模型有很大帮助，我们可以不需要自己来管理内存中的锁了。取消锁，队列处理效率很高事件驱动可以用在高并发场景下，比如抢购。

是的，用户体验有改变，用了这个事件驱动，用户的体验有可能会有改变，比如原来同步架构的时候有库存，就马上告诉你条件不满足无法下单，不会生成订单；但是改了事件机制，订单是立即生成的很可能过了一会系统通知你订单被取消掉。 就像抢购“小米手机”一样，几十万人在排队，排了很久诉你没货了，明天再来吧。如果希望用户立即得到结果，可以在前端想办法，在BFF（Backend For Frontend）使用CountDownLatch这样的锁把后端的异步转成前端同步，当然这样BFF消耗比较大。

没办法，产品经理不接受，产品经理说用户的体验必须是没有库存就不会生成订单，这个方案会不断生成取消的订单，他不能接受，怎么办？那就在订单列表查询的时候，略过这些cancel状态的订单吧也许需要一个额外的视图来做。我并不是一个理想主义者，解决当前的问题是我首先要考虑的，我们计微服务的目的是本想是解决业务并发量。而现在面临的却是用户体验的问题，所以架构设计也是需妥协的:(但是至少分析完了，我知道我妥协在什么地方，为什么妥协，未来还有可能改变。

多个领域多表Join查询

● 我个人认为聚合根这样的模式对修改状态是特别合适，但是对搜索数据的确是不方便，比如筛选出批符合条件的订单这样的需求，本身聚合根对象不能承担批量的查询任务，因为这不是他的职责。那必须依赖“领域服务（Domain Service）”这种设施。

当一个方法不便放在实体或者值对象上，使用领域服务便是最佳的解决方法，请确保领域服务是无状的。

● 我们的查询任务往往很复杂，比如查询商品列表，要求按照上个月的销售额进行排序； 要按照商品退货率排序等等。但是在微服务和DDD之后，我们的存储模型已经被拆离开，上述的查询都是要涉及单、用户、商品多个领域的数据。如何搞？ 此时我们要引入一个视图的概念。比如下面的，查询用户下订单的操作，直接调用两个服务自己在内存中join效率无疑是很低的，再加上一些filter条件、分页没法做了。于是我们将事件广播出去，由一个单独的视图服务来接收这些事件，并形成一个物化视图materialized view），这些数据已经join过，处理过，放在一个单独的查询库中，等待查询，这是一典型的以空间换时间的处理方式。

● 经过分析，除了简单的根据主键Find或者没有太多关联的List查询，我们大部分的查询任务可以放单独的查询库中，这个查询库可以是关系数据库的ReadOnly库，也可以是NoSQL的数据库，实际上们在项目中使用了ElasticSearch作为专门的查询视图，效果很不错

限界上下文（Bounded Context）和数据耦合

除了多领域join的问题，我们在业务中还会经常碰到一些场景，比如电商中的商品信息是基础信息，于单独的BC，而其他BC，不管是营销服务、价格服务、购物车服务、订单服务都是需要引用这个商信息的。但是需要的商品信息只是全部的一小部分而已，营销服务需要商品的id和名称、上下架状态订单服务需要商品id、名称、目录、价格等等。这比起商品中心定义一个商品（商品id、名称、规格规格值、详情等等）只是一个很小的子集。这说明不同的限界上下文的同样的术语，但是所指的概念一样。 这样的问题映射到我们的实现中，每次在订单、营销模块中直接查询商品模块，肯定是不合适因为

- 商品中心需要适配每个服务需要的数据，提供不同的接口
- 并发量必然很大
- 服务之间的耦合严重，一旦宕机、升级影响的范围很大。

特别是最后一条，严重限制了我们获得微服务提供的优势“松耦合、每个服务自己可以频繁升级不影其他模块”。这就需要我们通过事件驱动方法，适当冗余一些数据到不同的BC去，把这种耦合拆解开这种耦合有时候是通过Value Object嵌入到实体中的方式，在生成实体的时候就冗余，比如订单在生的时候就冗余了商品的信息；有时候是通过额外的Value Object列表方式，营销中心冗余一部分相关商品列表数据，并随时关注监听商品的上下级状态，同步替换掉本限界上下文的商品列表。

下图一个下单场景分析，在电商系统中，我们可以认为会员和商品是所有业务的基础数据，他们的变应该是通过广播的方式发布到各个领域，每个领域保留自己需要的信息。

保证最终一致性

最终一致性成功依赖很多条件

- 依赖消息传递的可靠性，可能A系统变更了状态，消息发到B系统的时候丢失了，导致AB的状态不致
- 依赖服务的可靠性，如果A系统变更了自己的状态，但是还没来得及发送消息就挂了。也会导致状不一致

我记得JavaEE规范中的JMS中有针对这两种问题的处理要求，一个是JMS通过各种确认消息（Client Aknowledge等）来保证消息的投递可靠性，另外是JMS的消息投递操作可以加入到数据库的事务中-没有发送消息，会引起数据库的回滚（没有查资料，不是很准确的描述，请专家指正）。不过现在符JMS规范的MQ没几个，特别是保一致性需要降低性能，现在标榜高吞吐量的MQ都把问题抛给了我自己的应用解决。所以这里介绍几个常见的方法，来提升最终一致性的效果。

使用本地事务

还是以上面的订单扣取信用的例子

- 订单服务开启本地事务，首先新增订单；
- 然后将Order Created事件插入一张专门Event表，事务提交；
- 有一个单独的定时任务线程，定期扫描Event表，扫出来需要发送的就丢到MQ，同时把Event设置“已发送”。

方案的优势是使用了本地数据库的事务，如果Event没有插入成功，那么订单也不会被创建；线程扫后把event置为已发送，也确保了消息不会被漏发（我们的目标是宁可重发，也不要漏发，因为Event理会被设计为幂等）。

缺点是需要单独处理Event发布在业务逻辑中，繁琐容易忘记；Event发送有些滞后；定时扫描性能消大，而且会产生数据库高水位隐患；

我们稍作改进，使用数据库特有的MySQL Binlog跟踪（阿里的Canal）或者Oracle的GoldenGate技可以获得数据库的Event表的变更通知，这样就可以避免通过定时任务来扫描了

不过用了这些数据库日志的工具，会和具体的数据库实现（甚至是特定的版本）绑定，决策的时候请重。

使用Event Sourcing 事件溯源

事件溯源对我们来说是一个特别的思路，他并不持久化Entity对象，而是只把初始状态和每次变更的Evnt记录下来，并在内存中根据Event还原Entity对象的最新状态，具体实现很类似数据库的Redolog的现，只是他把这种机制放到了应用层来。

虽然事件溯源有很多宣称的优势，引入这种技术要特别小心，首先他不一定适合大部分的业务场景，旦变更很多的情况下，效率的确是个大问题；另外一些查询的问题也是困扰。

我们仅仅在个别的业务上探索性的使用Event Souring和AxonFramework，由于实现起来比较复杂具体的情况还需要等到实践一段时间后再来总结，也许需要额外的一篇文章来详细描述

以上是对事件驱动在微服务架构中一些我的理解，文章部分借鉴了Chris Richardson的Blog，https://www.nginx.com/blog/event-driven-data-management-microservices/ 在此我向他表示致谢 。