浅谈12306核心模型设计思路和架构设计

优才网 2016-02-18

前言

春节期间，无意中看到一篇文章，文章中讲到12306的业务复杂度远远比淘宝天猫这种电商网站要复杂。后来自己想想，也确实如此。所以，很想挑战一下12306这个系统的核心领域模型的设计。一般的电商网站，购买都是基于商品的概念，每个商品有一定量的库存，用户的购买行为是针对商品的。当用户发起购买行为时，系统只需要生成订单并对用户要购买的商品减库存即可。但是，12306就不是那么简单了，具体复杂在哪里，我下面会进一步分析。

另外一个让我写这篇文章的原因，是我发现也许是否是因为目前12306的核心领域模型设计的不够好，导致用户购票时要处理的业务逻辑异常复杂，维护数据一致性的难度也几百倍的上升，同时面对高并发的订票也难以支持很高的TPS。我觉得，越是复杂的业务，就越要重视业务分析，重视领域模型的抽象和设计。如果不假思索，凭以往经验行事，则很可能会被以往的设计经验先入为主，陷入死胡同。我发现技术人员往往更注重技术层面的解决方案，比如一上来就分析如何集群、如何负载均衡、如何排队、如何分库分表、如何用锁，如何用缓存等技术问题，而忽略了最根本的业务层面的思考，如分析业务、领域建模。我认为越是复杂的业务系统，则越要设计一个健壮的领域模型。如果一个系统的架构我们设计错了，还有补救的余地，因为架构最终沉淀的只是代码，调整架构即可（一个系统的架构本身就是不断演进的）；而如果领域模型设计错了，那要补救的代价是非常大的，因为领域模型沉淀的是数据结构及其对应的大量数据，对任何一个大型系统，要改核心领域模型都是成本非常高的。

本文的重点不是在如何解决高并发的问题，而是希望从业务角度去分析，12306的理想模型应该是怎么样的。网上目前谈12306的文章貌似都是千篇一律的只谈技术，不谈业务分析和如何建模的。所以我想写一下自己的设计和大家交流学习。

需求简述

12306这个系统，核心要解决的问题是网上售票。涉及到2个角色使用该系统：用户、铁道部。用户的核心诉求是查询余票、购票；铁道部的核心诉求是售票。购票和售票其实是一个场景，对用户来说是购票，对铁道部来说是售票。因此，我们要设计一个在线的网站

系统，解决用户的查询余票、购票，以及铁道部的售票这3个核心诉求。看起来，这3个场景都是围绕火车票展开的。

查询余票：用户输入出发地、目的地、出发日三个条件，查询可能存在的车次，用户可以看到每个车次经过的站点名称，以及每种座位的余票数量。

购票：购票分为订票和付款两个阶段，本文重点分析订票的模型设计和实现思路。

其实还有很多其他的需求，比如给不同的车次设定销售座位数配额，以及不同的区段设置不同的限额。我觉得这个需求不是核心最重要的诉求，所以，本文针对这个需求不做具体讨论，也不是本文分析设计的重点。

需求分析

确实，12306也是一个电商系统，而且看起来商品就是票了。因为如果把一张票看成是一个商品，那购票就类似于购买商品，然后每张票都有库存，商品也有库存的概念。但是如果我们仔细想想，会发现12306要复杂很多，因为我们无法预先确定好所有的票，如果非要确定，那只能通过穷举法了。
我们以北京西到深圳北的G71车次高铁为例（这里只考虑南下的方向，不考虑深圳北到北京西的，那是另外一个车次，叫G72），它有17个站（北京西是01号站，深圳北是17号站），3种座位（商务、一等、二等）。表面看起来，这不就是3个商品吗？G71商务座、G71一等座、G71二等座。大部分轻易喷12306的技术人员（包括某些中等规模公司的专家、CTO）就是在这里栽第一个跟头的。

实际上，这个车次可以卖的票是非常多的。为了方便后面的讨论，我们先明确一下票是什么？

一张票的核心信息包括：出发时间、出发地、目的地、车次、座位号。持有票的人就拥有了一个凭证，该凭证表示持有它的人可以坐某个车次的某个座位号，从某地到某地。所以，一张票，对用户来说是一个凭证，对铁道部来说是一个承诺；那对系统来说是什么呢？不知道。这就是我们要分析业务，领域建模的原因，我们再继续思考吧。

明白了票的核心信息后，我们再看看G71这个车次的高铁，可以卖多少张票？
讨论前先说明一下，一辆火车的物理座位数（站票也可以看成是一种座位，因为站票也有数量配额）不等于可用的最大配合。所有的物理座位不可能都通过12306网站来销售，而是只会销售一部分，比如40%。其余的还是会通过线下的方式销售。不仅如此，可能有些

站点上车的人会比较多，有些比较少，所以我们还会给不同的区间配置不同的**限额**。比如D31北京南至上海共有765张，北京南有260张，杨柳青有80张，泰安有76张。如果杨柳青的80张票售完就会显示无票，就算其他站有票也会显示无票的。不管如何配置限制区段的配额和限额，我们总是针对车次进行配置，这点只是车次内部售票时的一些额外的判断条件（业务规则），不影响车次模型的核心地位。所以，为了本文讨论的清楚起见，我后续的讨论都不涉及配额和限额的问题，而是认为任何区段都可以享受火车最大的物理座位数。

为了讨论问题方便，我们减少一些站点来讨论。假设某个车次有A,B,C,D四个站点。那001这个人购买了A,B这个区间，系统会分配给001一个座位x；但是因为001坐到B站点后会下车，所以相当于x这个座位又空出来了，也就是说，从B站点开始，系统又可以认为x这个座位是可用的。所以，我们得出结论：同一个座位，其实可以同时出售AB,BC这两张票。通过这个简单的分析，我们知道，一列火车虽然只有有限的座位数，比如1000个座位。但可以卖出的票远远不止1000个。还是以A,B,C,D四个站点为例，假如火车总共有1000个座位，那AB可以卖1000张，BC也可以卖1000张，同样，CD也可以卖1000张。也就是说，理论上最多可以卖出3000张票。但是如果换一种卖法，所有人都是买ABCD的票，也就是说所有的票都是经过所有站点的，那就是最多只能卖出1000张票了。而实际的场景，一定是介于1000到3000之间。然后实际的G71这个车次，有17个站，那到底可以卖出多少个票，大家应该可以算了吧。理论上这17个站中的任意两个站点之间所形成的线段，都可以出售为一张票。我数学不好，算不太清楚，麻烦有数学好的人帮我算算，呵呵。

通过上面的分析，我们知道一张票的本质是某个车次的某一段区间（一条线段），这个区间包含了若干个站点。然后我们还发现，只要区间不重叠，那座位就不会发生竞争，可以被回收利用，也就是说，可以同时预先出售。
另外，经过更深入的分析，我们还发现区间有4种关系：1）不重叠；2）部分重叠；3）完全重叠；4）覆盖；不重叠的情况我们已经讨论过了，而覆盖也是重叠的一种。所以我们发现如果重叠，比如有两个区间发生重叠，那重叠部分的区间（可能夸一个或多个站点）是在争抢座位的。因为假设一列火车有100个座位，那每个原子区间（两个相邻站点的连线），最多允许重叠99次。

所以，经过上面的分析，我们知道了一个车次能够出售一张车票的**核心业务规则**是什么？就是：**这张车票所包含的每个原子区间的重叠次数加1都不能超过车次的总座位数，实际上重叠次数+1也可以理解为线段的厚度。**

模型设计

上面我分析了一下票的本质是什么。那接下来我们再来看看怎么设计模型，来快速实现购票的需求，重点是怎么设计商品聚合以及减库存的逻辑。

传统电商的思路

如果按照普通电商的思路，把票（站点区间）设计为商品（聚合根），然后为票设计库存数量。我个人觉得是很糟糕的。因为一方面这种聚合根非常多，另一方面，即便枚举出来了，一次购票也一定会影响非常多其他聚合根的库存数量（只要被部分或全部重叠的区间都受影响）。这样的一次订单处理的复杂度是难以评估的。而且这么多聚合根的更新要在一个事务里，这不是为难数据库吗？而且，这种设计必然带来大量的事务的并发冲突，很可能导致数据库死锁。总之，我认为这种是典型的由于领域模型的设计错误，导致并发冲突高、数据持久化落地困难。或者如果要解决并发问题，只能排队单线程处理，但是仍然解决不了要在一个事务里修改大量聚合根的尴尬局面。听说12306是采用了Pivotal Gemfire这种高大上的内存数据库，我对这个不太了解。我不可想象要是不使用内存数据库，他们要怎么实现车次内的票之间的数据强一致性（就是保证所有出售的票都是符合上面讨论的业务规则的）？

我的思路

通过上面的分析我们知道，其实任何一次购票都是针对某个车次的。我们看看一个车次包含了哪些信息？一个车次包括了：1）车次名称，如G71；2）座位数，实际座位数会分类型，比如商务座20个，一等座200个；二等座500个；我们这里为了简化问题，可以暂时忽略类型，我认为这个类型不影响核心的模型的设计决策。需要格外注意的是：这里的座位数不要理解为真实的物理座位数，很有可能比真实的座位数要少。因为我们不可能把一个车次的所有座位都在网上通过12306来出售，而是只出售一部分，具体出售多少，要由工作人员人工指定。3）经过的站点信息（包括站点的ID、站点名称等），注意：车次还会记录这些站点之间的顺序关系；4）出发时间；看过GRASP九大模式中的**信息专家模式**的同学应该知道，**将职责分配给拥有执行该职责所需信息的类**。我们这个场景，车次具有一次出票的所有信息，所以我们应该把出票的职责交给车次。另外学过DDD的同学应该知道，**聚合设计有一个原则，就是：聚合内强一致性，聚合之间最终一致性。**经过上面的分析，我们知道要产生一张票，其实要影响很多和这个票对应的直线相交的其他票的可用数量。因为所有的站点信息都在车次聚合内部，所以车次聚合内部自然可以维护所有的原子区间，以及每个原子区间的可用票数（相当于是库存数）。当一个原子区间的可用票数为0的时候，意味着火车针对这个区间的票已经卖完了。所以，我们完全可以让车次这个聚

合根来保证出票时对所有原子区间的可用票数的更新的强一致性。对于车次聚合根来说，这很简单，因为只是几次简单的内存操作而已，耗时可以忽略。一列火车假如有ABCD四个站点，那原子区间就是3个。对于G71，则是16个。

然后基于上面的聚合设计，**出票时扣减库存的逻辑是：**
根据订单信息，拿到出发地和目的地，然后获取这段区间里的所有的原子区间。然后尝试将每个原子区间的可用票数减1，如果所有的原子区间都够减，则购票成功；否则购票失败，提示用户该票已经卖完了。是不是很简单呢？知道了出票的逻辑，那退票的逻辑也就很简单了，就是把这个票的所有原子区间的可用票数加1就OK了。如果我们从线段的厚度的角度去考虑，那出票时，每个原子区间的厚度就是+1，退票时就是减一。就是相反的操作，但本质是一样的。

所以，通过这样的思路，我们将一次订票的处理控制在了一个聚合根里，用聚合根内的强一致性的特性保证了订票处理的强一致性，同时也保证了性能，免去了并发冲突的可能性。传统电商那种把票单做类似商品的核心聚合根的设计，我当时第一眼看到就觉得不妥。因为这违背了DDD强调的强一致性应该由聚合根来保证、聚合根之间的最终一致性通过Saga来保证的原则。

还有一个很重要的概念我想说一下我的看法，就是座位和区间的关系。因为有些朋友和我讲，考虑座位号的问题，虽然都能减1，座位号也必须是同一个。我觉得座位是全局共享的，和区段无关（也许我的理解完全有误，请大家指正）。座位是一个物理概念，一个用户成功购买了一张票后，座位就会少一个，一张票唯一对应一个座位，但是一个座位有可能会对应多张票；而区间是一个逻辑上的概念，区间的作用有两个：1）表示票的出发地和目的地；2）记录票的可用数额。如果区间能连通（即该区间内的每个原子区间的可用数额都大于0），则表示允许拥有一个座位。所以，我觉得座位和票（区间）是两个维度的概念。

模型分析总结：我认为票不是核心聚合根，票只是一个计算的结果，一个凭证而已，票本身没有什么逻辑；12306真正的核心模型应该是车次，车次具有出票的职责，并以强一致性的方式维护一次出票（或退票）时所有原子区间的可用票数。

架构设计（非本文重点，没兴趣的朋友可以略过，呵呵）

我觉得12306这样的业务场景，非常适合使用CQRS架构；因为首先它是一个查多写少、但是写的业务逻辑非常复杂的系统。所以，非常适合做架构层面的读写分离，即采用CQRS架构。而且应该使用数据存储也分离的CQRS。这样CQ两端才可以完全不需要顾及

对方的问题，各自优化自己的问题即可。我们可以在C端使用DDD领域模型的思路，用良好设计的领域模型实现复杂的业务规则和业务逻辑。而Q端则使用分布式缓存方案，实现可伸缩的查询能力。

订票的实现思路

同时借助像ENode这样的框架，我们可以实现in-memory + Event Sourcing的架构。Event Sourcing技术，可以让领域模型的所有状态修改的持久化统一起来，本来要用ORM的方式保存聚合根最新状态的，现在只需要简单的通用的方式保存一个事件即可（一次订票只涉及一个车次聚合根的修改，修改只产生一个事件，只需要持久化一个事件（一个JSON串）即可，保证了高性能，无须依赖事务，而且通过ENode可以解决并发问题）。我们只要保存了聚合根每次变化的事件（事件的结构怎么设计，本文不做多的介绍了，大家可以思考下），就相当于保存了聚合根的最新状态。而正是由于Event Sourcing技术的引入，让我们的模型可以一直存活在内存中，即可以使用in-memory技术。不要小看in-memory技术，in-memory技术在某些方面对提高命令的处理性能非常有帮助。比如就以我们车次聚合根处理出票的逻辑，假设某个车次有大量的命令发送到分布式消息队列，然后有一台机器订阅了这个队列的消息，然后这台机器处理这个车次的订票命令时，由于这个车次聚合根一直在内存，所以就省去了每次要去数据库取出聚合根的步骤，相当于少了一次数据库IO。这样的好处是，因为一个车次能够真正出售的票是有限的，因为座位就那么几个，比如就1000个座位，估计一般正常情况也就出个2000个左右的座位吧（具体能出多少张票要取决于区间的相交程度，上面分析过）。也就是说，这个聚合根只会产生2000个事件，也就是说只会有2000个订票命令的处理是会产生事件，并持久化事件；而其余的大量命令，因为车次在内存计算后发现没有余票了，就不会做任何修改，也不会产生领域事件，这样就可以直接处理下一个订票命令了。这样就可以大大提高处理订票命令的性能。

另外一个问题我觉得还需要提一下，因为用户订票成功后，还需要付款。但用户有可能不去付款或者没有在规定的时间内完成付款。那这种情况下，系统会自动释放该用户之前订购的票。所以基于这样的需求，我们在业务上需要支持业务级别的2pc。即先预扣库存，也就是先占住这张票一定时间（比如15分钟），然后付款成功后再真实给你这张票，系统做真正的库存修改。通过这样的预扣处理，可以保证不会出现超卖的情况。这个思路其实和传统电商比如淘宝这样的系统类似，我就不多展开了，我之前写的Conference案例也是这样的思路，大家有兴趣的可以去看一下我之前录制的视频。

查询余票的实现思路

我觉得余票的查询的实现相对简单。虽然对于12306来说，查询的请求占了80%，提交订单的请求只占20%。但查询由于对数据没有修改，所以我们完全可以使用分布式缓存来实现。我们只需要精心设计好缓存的key即可；缓存key的多少要看成本，如果所有可能的查询都设计对应的key，那时间复杂度为1，查询性能自然高；但代价也大，因为key多了。如果想key少一点，那查询的复杂度自然要上去一点。所以缓存设计无非就是空间换时间的思路。然后，缓存的更新无非就是：自动失效、定时更新、主动通知3种。通过CQRS架构，由于CQ两端是事件驱动的，当C端有任何状态变化，都会产生对应的事件去通知Q端，所以我们几乎可以做到Q端的准实时更新。

同时由于CQ两端的完全解耦，Q端我们可以设计多种存储，如数据库和缓存（Redis等）；数据库用于线下维护关系型数据，缓存用户实时查询。数据库和缓存的更新速度相互不受影响，因为是并行的。对同一个事件，可以10台机器负责更新缓存，100台机器负责更新数据库。即便数据库的更新很慢，也不会影响缓存的更新进度。这就是CQRS架构的好处，CQ的架构完全不同，且我们随时可以重建一种新的Q端存储。不知道大家体会到了没有?

关于缓存key的设计，我觉得主要从查询余票时传递的信息来考虑。12306的关键查询是：出发地、目的地、出发日期三个信息。我觉得有两种key的设计思路：1）直接设计了该查询条件的key，然后快速拿到车次信息，直接返回；这种方式就是要求我们系统已经枚举了所有车次的所有可能出现的票（区间）的缓存key，相信你一定知道这样的key是非常多的。2）不是枚举所有区间，而是把每个车次的每个原子区间（相邻的两个站点所连成的直线）的可用票数作为key。这样，key就非常少了，因为车次假如有10000个，然后每个车次平均15个区间，那也就15W个key而已。当我们要查询时，只需要把用户输入的出发地和目的地之间的所有原子区间的可用票数都查出来，然后比较出最小可用票数的那个原子区间。则这个原子区间的可用票数就是用户输入的区间的可用票数了。当然，到这里我提到考虑出发日期。我认为出发日期是用来决定具体是哪个车次聚合根的。同一个车次，不同的日期，对应的聚合根实例是不同的，即便是同一天，也可能有多个车次聚合根，因为有些车次一天有几班的，比如上午9点发车的一班，下午3点发车的一般。所以，我们也只要把日期也作为缓存key的一部分即可。

总结

本文完全是凭自己对12306这个网站的核心业务的简单思考而得到的一些设计结果。如果真正的DDD领域建模，更多的是要和业务一线的工作人员、领域专家进行深入沟通，才能更深入的了解该领域内的业务知识，从而才能设计出更靠谱的领域模型和架构设计。我本人非常惭愧因为没有上12306买过火车票，家离的比较近，就算要买也是家人给我买:）所

以，本文所分享的内容难免是纸上谈兵。但我觉得12306这个系统的业务确实比传统的电商系统要复杂，且并发又这么高。所以，我觉得这个系统真的很值得大家重视模型的设计，而不只是只关注技术层面的实现。2016年，我有计划打算基于ENode实现一套12306的核心功能，比如余票查询、订票的功能。

来源：http://www.cnblogs.com/netfocus/p/5187241.html
作者：汤雪华

推荐阅读

前淘宝工程师谈12306：做它比做淘宝难
手机淘宝前端的图片相关工作流程梳理
淘宝搜索算法现状
淘宝技术专家谈大型网站架构

阅读原文