秒杀系统架构解密与防刷设计 - 高可用架构系列

吕毅 高可用架构 2015-07-21

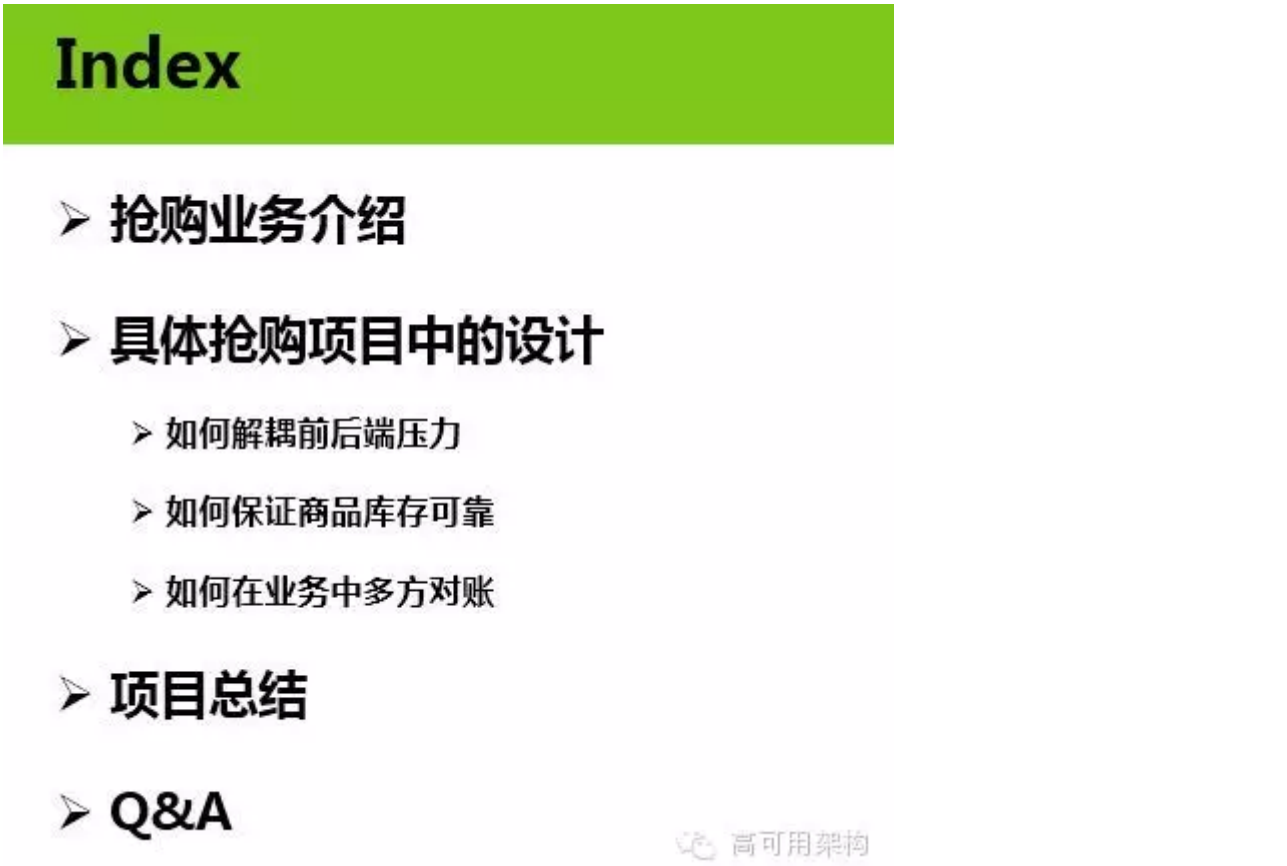
此文是根据吕毅在【QCON高可用架构群】中的分享内容整理而成，转发请注明出处。

> 吕毅，百度资深研发工程师，LAMP人。
> 2012年从新浪加入百度移动服务事业群。在百度期间，随着产品线发展和业务上QPS增长，架构设计方面略有所获，对移动端业务、优化有独特的理解和方法。

抢购业务介绍

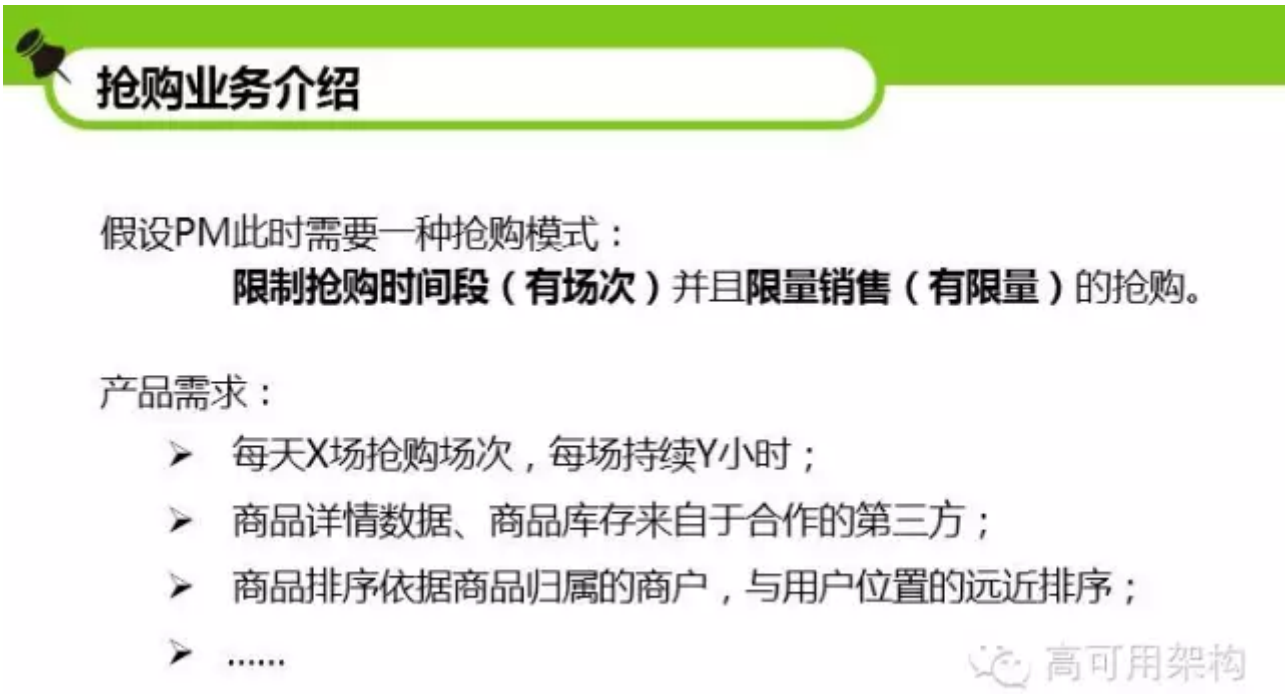
抢购、闪购，从国外风靡后，国内各类网站都开始做相似的业务，我们耳熟能详的唯品会、淘宝、京东都有这类业务。抢购，更多出现在电商网站。那么，今天和大家一起学习下抢购业务形态的业务架构设计。

我们常见的抢购业务分两种： 限时抢购、限量抢购，我简单分了下这些case，如下图：



想必小米的抢购运营的最火爆了，每发一款新品，都限量发售，每次搞的大家心里痒痒的。记得之前还因为抢购太火爆，站点打不开，崩溃了。那么问题来了：为什么抢购总是引发RD、OP恐慌？我理解是，爆品太火爆，瞬时请求太大，导致业务机器、存储机器都在抢购高峰时扛了太多压力。那么，我们今天以一个抢购业务场景为例，看看如何扛住压力，做好抢购业务！假设，这时候我们接到了产品层面的需

求，如下图：

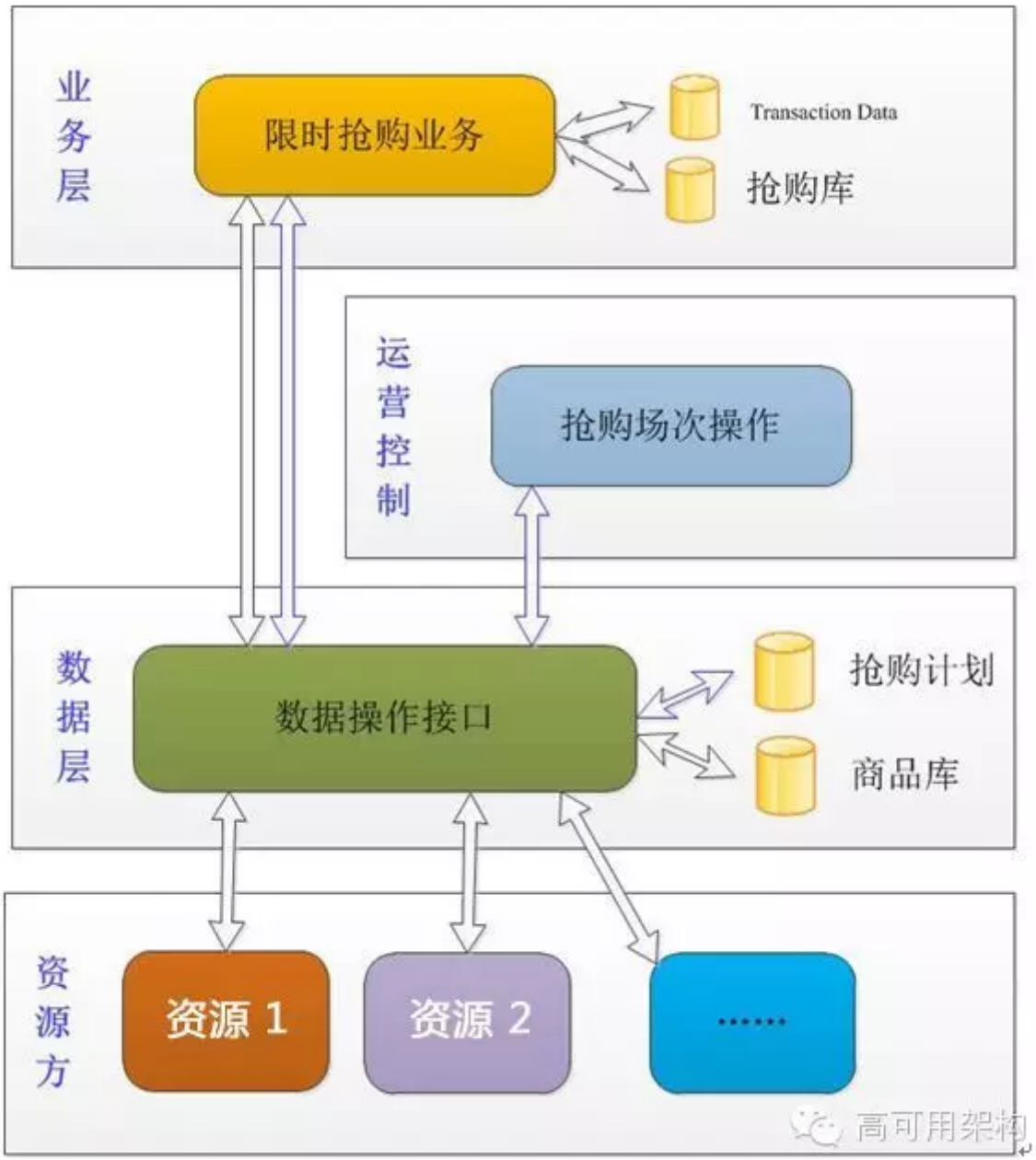


PM也挺呵呵的，又要有时段的要求、又要有限量的要求，大而全呐！
不过，对于咱们RD同学，也不是问题，我们一起来看看，如何设计业务架构，把需求满足的棒棒哒！
首先，我们冷静的看看需求。

> 需求说：商品数据来自资源方。（哦，我们没有商品数据。）
> 需求说：每天要有好几场抢购，每场抢购都有商品限制（哦，有点商场促销还限量甩的feel）
> 需求说：商品要基于用户位置排序（哦，移动端业务嘛，这种需求总是有的）
> 需求说：balabala.....

具体抢购项目中的设计

通过我们先行的抢购需求分析，我们画一个粗略的流程图，如下:



我们将自身简单划为两部分：业务层、数据层，并且旁路设计一个“运营控制”环节。
当然，数据源自第三方嘛，我们的数据层基于第三方资源数据构建。
这时，我们来看看这个草图里几个库和几个数据流，是怎样的。

首先,看看库。数据层的“商品库”，显而易见，用于存储第三方商品数据，通过第三方推、我们拉的方式来构建这个数据库信息。数据库层的“抢购计划”库，主要由旁路的“运营控制”环节产生的数据，由运营同学来维护抢购场次、商品数量。

业务层的“抢购库”，其实是商品库的子集，由运营同学勾选商品并配好该商品放出多少用于抢购，发布到业务层面的抢购库中。

业务层的Transaction Data，一会我们讲到与第三方对账时候，我们再说它。

如何解耦前后端压力

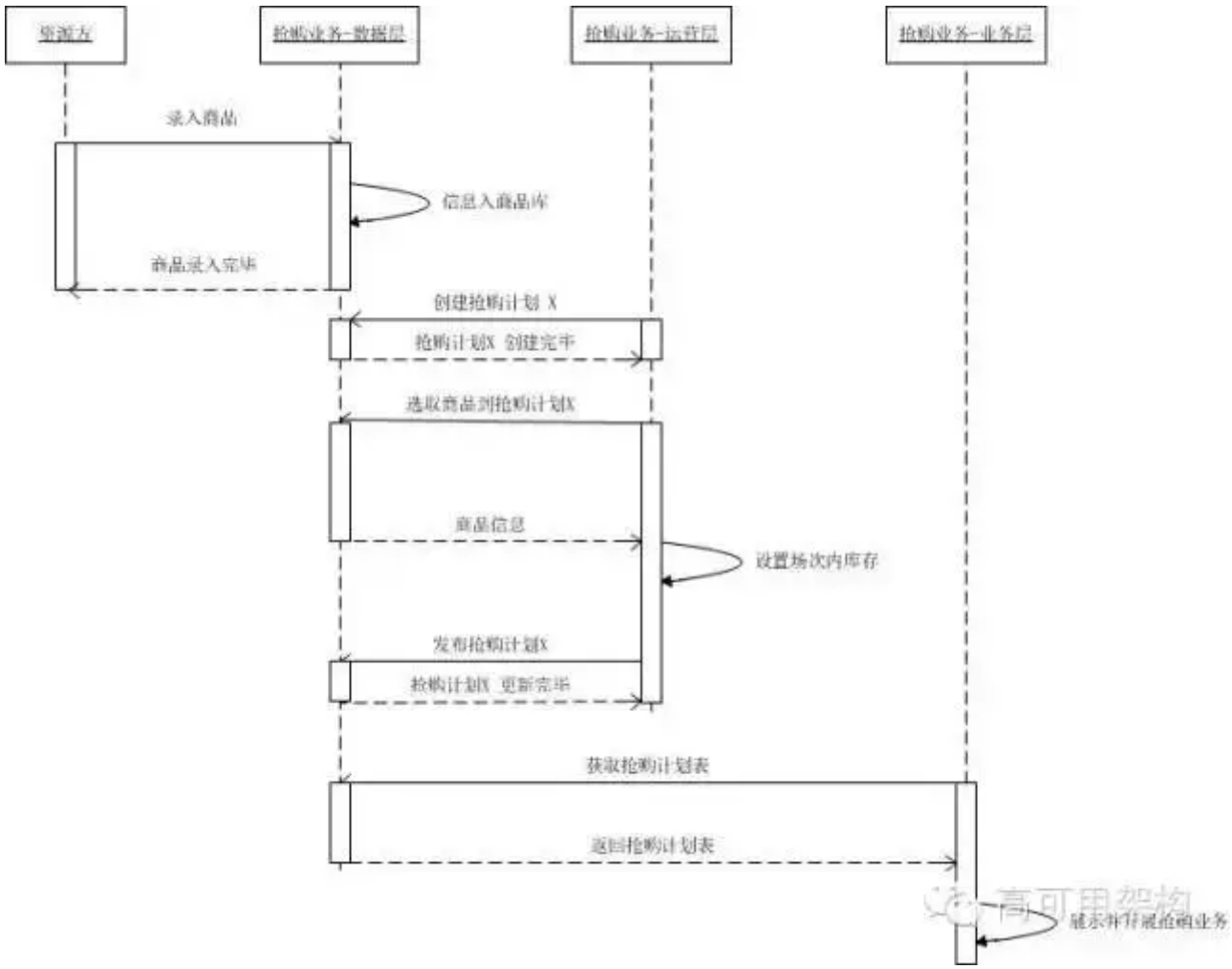
我们此时回顾下目录，目录中我们讲，如何隔离前后端压力呢？做法是：

1. 让我们业务的压力，不会传递到资源方，避免造成资源方接口压力同比增长。所以，我们自己建了商品库，此时，第三方笑了。
2. 业务层与数据层解耦，我们让抢购库位于业务层，让商品库位于数据层。因为我们可以想象到，抢购高峰来临时，查询“商品还有没有？”的请求是最多的，若“有没有”这种高频请求每次都去数据层，那我们其实就将业务、数据耦合在一起了，那么，就有了抢购库这个子库，在业务层抗压力。（这里可以明确的是，数据层的商品库为关系型存储，业务层的抢购库为nosql的）

有了业务层的nosql（我们就用redis吧）抗高频压力，数据层的商品库笑了。
这里就可以抛一个思想了：我们的架构设计中，需要分解压力，在互联网项目中，来自于用户的大流量不少见，这些流量最终都会落到一个地方，就看我们的设计如何分解这个压力了，如何避免它层层传递。抛个case，我们的水平分布业务机器，也是考虑通过水平扩展实例的方式，来分解大流量压力。
不扯概念的东西了，我们回归我们的抢购业务。
有了简单的分层设计，解决的大家都担心的压力问题，我们就看看抢购业务的时序是怎样的。
我们的时序图分两个视角来说明：

1. 商品的角度；
2. 用户的角度；

商品角度的时序图，从左到右：资源方、数据层、旁路-运营控制层、业务层。如下图：



录入商品 即商品从资源方发布到我们的数据层，形式可以是通过API、可以是通过文件传输、可以是我们去拉去。通过我们的代码逻辑，记录到我们数据层的“商品库DB”。

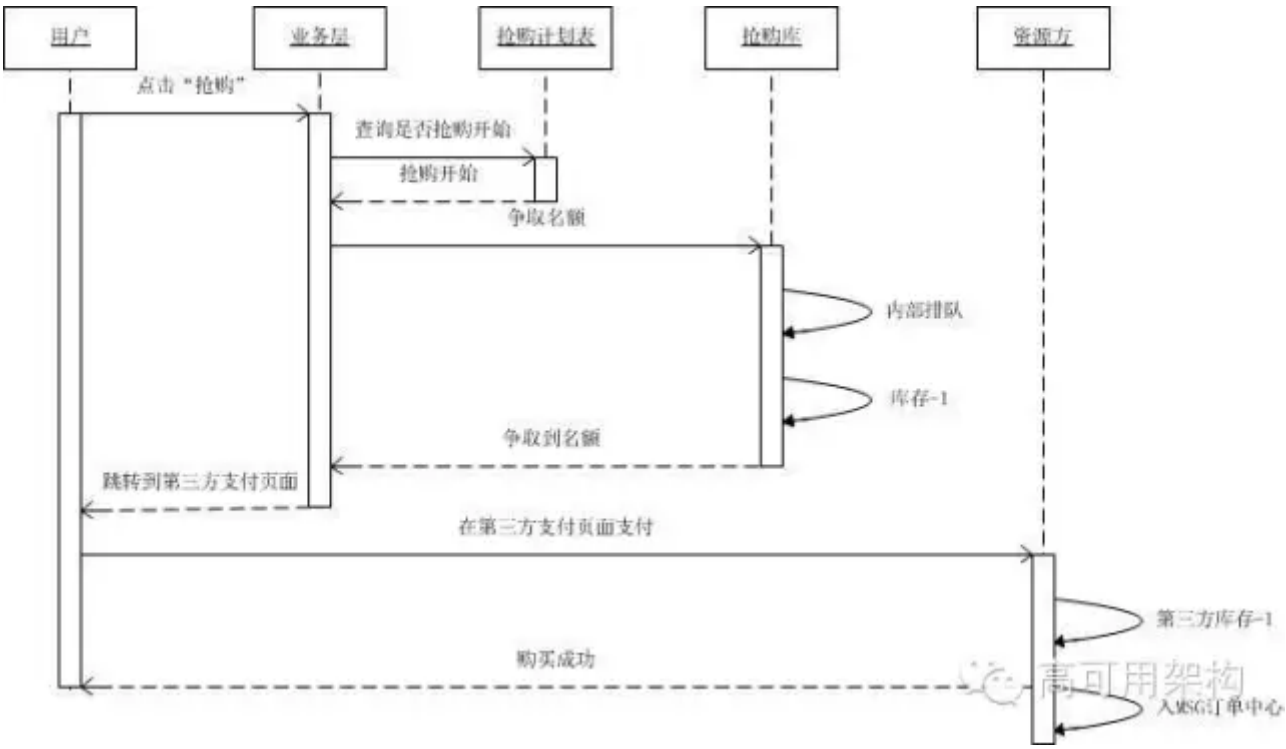
有了自建的商品库的数据，我们的运营同学就可以基于商品库设计每天的抢购场次（此事就有Web界面的事情，这里我们就不展开这块了），运营同学创建好一批抢购场次，记录在数据层的“抢购计划”这一关系型数据库中。

运营同学创建完抢购场次后，没完事，还得应产品需求，基于商品库，配置每场抢购场次中覆盖的商品，及商品的数量。这些抢购场次内的商品配置，会简单的记录在业务层的“抢购库”中。（抢购库记录的信息较为简单，例如商品库中ID为123123的商品有100件，业务层的抢购库中只存ID 123123商品运营配了在第X场抢购中有5件）

此时，数据层的 商品库有了资源方数据、数据层的 抢购计划库中有运营配的抢购计划，业务层的抢购库中每场抢购活动中商品的情况。

那么，业务层此时就可以基于时间，来展示运营配的抢购场次了。业务层，如何展示，这块就是拼装数据、前端效果了，这里也不展开了。

假设此事某场场次的抢购活动已经开始，我们再看看用户角度的时序图：

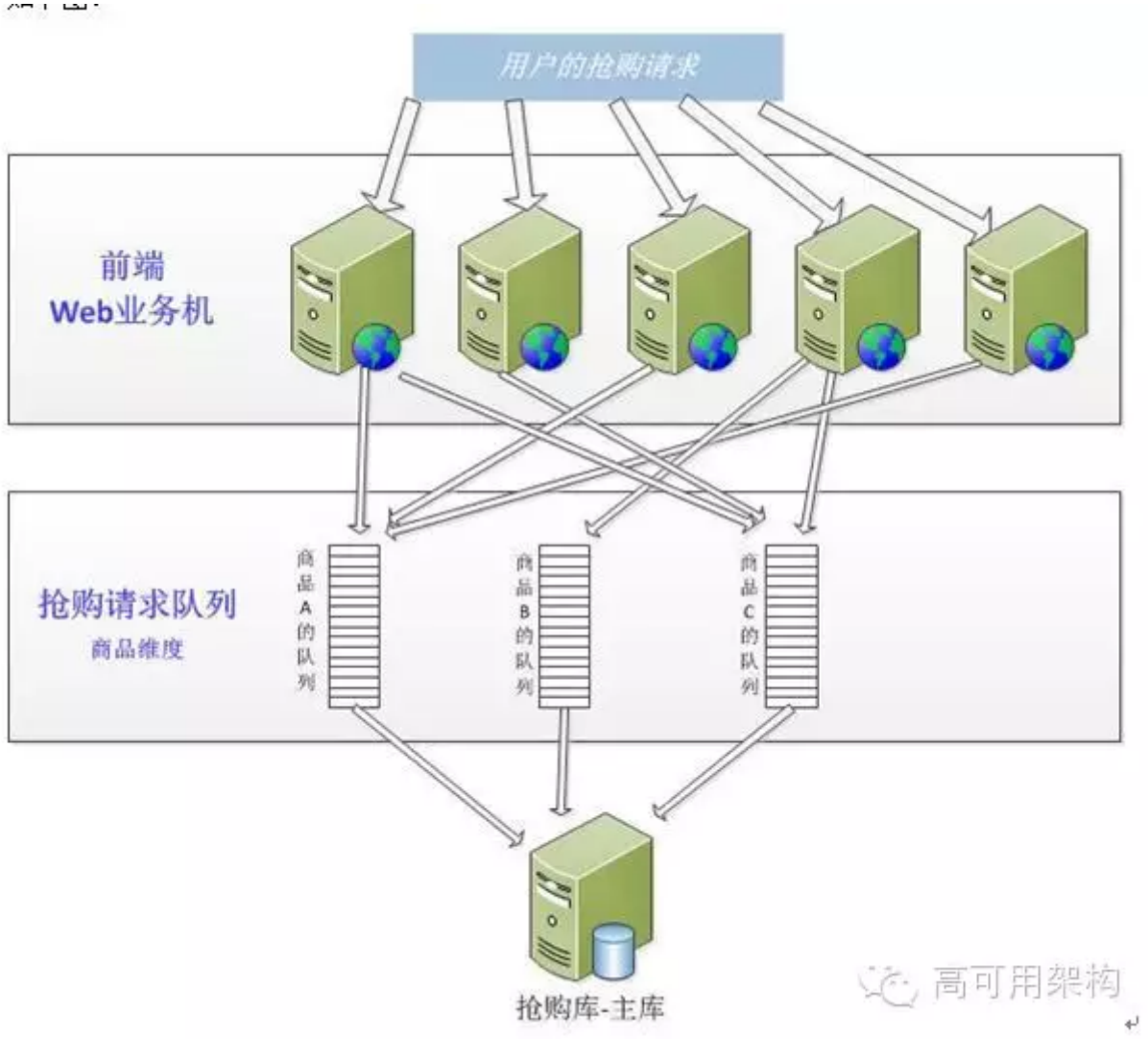


用户点击某个商品的抢购按钮，业务层代码首先去看看抢购计划库此时是否开始（此步可缓存、也可cache在前端页面或Client，若有cache的话，此步可忽略）。
若抢购在进行中，此时业务代码需要查询商品在本次抢购中的库存还有否（高频请求，即图中“争取名额”阶段）。
“争抢名额”这块，一会我们细讲，先把时序图说完。
若用户抢到了名额，就允许用户跳转到第三方的支付页面产生消费。（此时第三方笑了），产生消费后，第三方自己的库存-1，并且可以实时、异步、完事对账的方式通知我们。

如何保证商品库的库存可靠

此时，我们回顾下目录，“如何保证商品库的库存可靠”。
我们其实是将商品库的子库前置在业务层抗压力。那么，如何保证大家的库存情况稳定，不会因为抢购业务，导致库存波动影响用户体验。这里就需要提一个业务RD需要关注的问题，需要做好取舍。要么，我们保证大家看到的库存规律一致，要么，我们保证单个用户看到的库存规律一致。若保证大家看到的库存减少的规律一致，且同一时刻库存大家看到的库存都一样。这就对系统有数据强一致性要求，需要很大成本，还只能逐渐逼近此要求要求的效果。而我们若选择后者，仅保证单个用户看到的库存减少规律一致，虽放弃了数据强一致，但以更少的时间尽可能实现了最好的效果。所以，我们用到了用户来排队，若抢到名额了，在抢购库中的库存—（减减），这样单用户操作期间，能看到规律的减少，不会出现此事看剩10个，一会看还有11个的情况。这时我们说如何内部排队，如何来控制“查询商品在本次抢

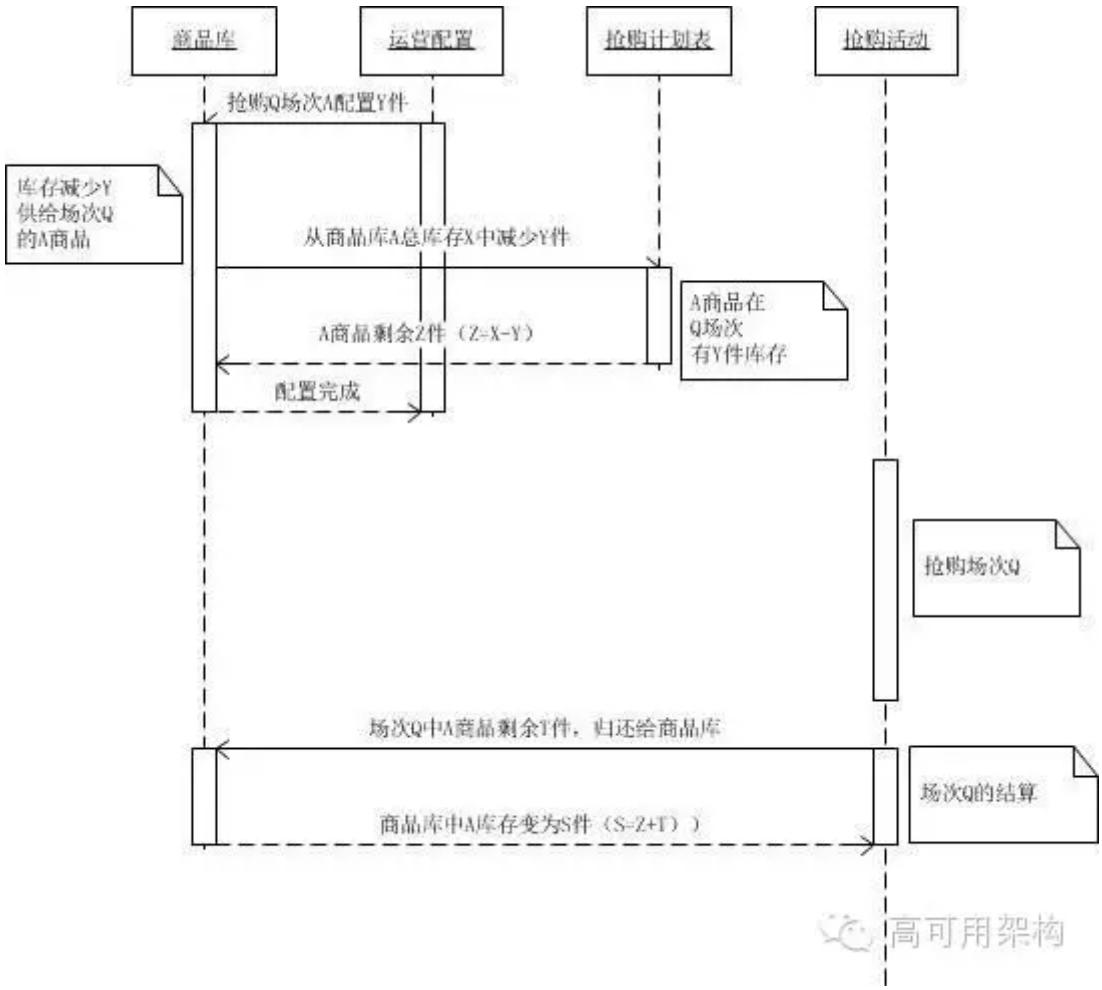
购中的库存还有否（高频请求）”这个高频请求。



我们构建商品维度的cache，上图中虽然说是“队列”，我们可以用redis的list来真正实现个队列，也可以通过 /—来实现。

假设商品A，运营配了20件，此事来了N多用户的请求，业务代码都会来查询cache_prefix_a_id这个队列的长度，若队列长度≤0，则有权去—（减减）抢购库的商品库存。若队列长度在20件内，则通过业务代码内的等待来等待队头的位置，然后获得抢购权限。若队列长度太长，则可以直接返回，认为商品已被抢空。

这时插入一个运营配库的时序，便于大家理解。该时序图有详细的说明和标注，就不展开了，如下图：



此时，我们可以想象，若上游用户的请求压力是N，这个N会压在业务层的抢购库，俗话说“责任止于此”

如何和第三方多方对账

那么，我们回顾目录“如何和第三方多方对账”？
这里就要提到“Transaction Data”这个库了。
Transaction ID为用户维度的Session记录，用户从进入抢购业务开始，产生一个Transaction ID，该Transaction ID生命周期截止到用户跳转去第三方支付为止。期间在生活服务中产生的浏览、抢购行为均会挂靠到该Transaction ID之下，并会在跳转去第三方支付页时携带该Transaction ID凭证。最主要的是需要记录下：用户获得商品名额后，跳转去第三方时，这一行为。
考虑到Transaction ID为抢购业务中，用户操作行为的关键字段，值需要保证唯一。故此处可以采用发号器之类的能力。
我们构建的Transaction Data记录，就可以按照DailyRun的方式，与第三方对账，来fix两方数据库库存不一致等问题。
为什么会产生我们和资源方的库存不一致，可能是因为用户在第三方消费后，第三方callback我们时候失败造成，也可能是因为用户跳去第三方后并没有真正支付，

但我们的商品库、抢购库的库存都已经减少造成的。原因可能有很多，对账机制是必要的。

项目总结

最后，我们回顾回顾设计，压力问题在业务层解决了，库存不一致问题我们通过对账机制解决了，产品的需求我们也通过旁路可配解决了，嗯，可以喝杯茶，发起评审，评审通过后开始写代码了。 :)

感谢大家。分享中的数据强一致那块，以及如何做取舍，都是很有意思的点，都可以展开聊很久，这里没展开，大家可以事后查查资料。

Q & A

Q1:防刷是怎么做的？一般抢购都有很大优惠。如果有人恶意刷，那正常的用户就失去了购买的机会。比如，抢购的商品数为1000，有人恶意刷了900，那只有 100 被正常用户抢到。等恶意抢到的 900 经过后面的支付环节验证后，可能已经过了抢购时间了。就算恶意抢到的 900 都支付成功，那对正常用户也是不公平的。

> 在这个业务场景中，我们做的是商品展示、商品的购买权的发放，真正产生消费是在第三方。那么，用户刷的问题，需要我们和第三方支付页面一起来控制。在用户通过排队机制，获得了购买名额后，跳转去第三方时候，我们按照和第三方约定的加密方式传递加密信息，第三方按照约定的解密方式解密成功后才允许用户支付，加密解密的过程中可以带具有生命周期的内容。这样，用户在高频请求支付页面获取商品时候，实际只有：1）加密对；2）第一次，才可能获得。不过，第三方都是为了销售出商品，所以这类合作的成功几率不大。恶意刷，的确会在我们的业务层面展示商品没量了。导致想买的用户没了机会，但可以保证第三方不受损。这种刷的情况，若想在我们业务层规避，我想这就是一个通用的防SPAM的问题了。这块自己真懂得不多。

Q2:要想准确的放刷，判断的维度就多，逻辑就复杂；与之矛盾的，抢购要求的是响应迅速。

> 对的，抢购业务因为请求压力大、热门商品抢购并发高，切忌增加过多逻辑，切忌过多后端依赖，越简单效果越好。我们在设计系统时候，很多事不是咱们一个系统能cover的，多少需要一些前置模块、能力的构建ready后，我们的系统才能run的不错。建议构建帐号体系、用户消费记录这两部分。

Q3:对账只是和第三方去对比商品的库存量吗，金额是否去对比？

> 对账，其实是对比的消费数据。避免出现我们统计今日产生了X件商品共价值Y的消费，第三方给出的是消费了N件共M价值的消费。避免金额不一致，造成结算、分成等问题的出现。我想你问题中的库存量的diff问题，还得靠第三方定期的通过我们数据层的接口来update他们提供的商品。其实在我们的商品库中，商品不一定只允许第三方提供，也可以允许第三方通过接口减少商品嘛，比如和一个卖水果的第三方合作，第三方上周发布说有100件，但这周线下热销，只剩20件了，我们也应该允许第三方来update到一个低值。但这样，我们的系统中就会复杂挺多。

Q4：防刷，避免第三方的推广效果达不到问题。

> 对的，用户ID维度、IP维度，都是有效办法。看具体场景。有帐号体系的业务，用用户ID维度效果最好，借助存储记录下每个用户的购买记录，来控制就好。市面上的电商网站，基本是抢购业务都需要登录，并且限制每件商品单人购买数量，其实就是通过存储记录用户的消费，并且再次产生消费前查询并增加代码逻辑来控制。

Q5：每次抢购活动的时候用一套新的验证码？

> 验证码这个东东，属于图灵测试嘛，只要测试方法好，并且尽可能保证每次产生的验证信息从未出现过且无规律，就是好的验证码啦。

感谢刘世杰的记录与整理，国忠的校对与发布，其他多位编辑组志愿者对本文亦有贡献。读者可以通过搜索“ArchNotes”或长按下面图片，关注“高可用架构”公众号，查看更多架构方面内容，获取通往架构师之路的宝贵经验。转载请注明来自“高可用架构（ArchNotes）”公众号。

Read more