13 | 如何利用缓存实现万级并发扣减?

在上一讲的实现方案里我们讨论了采用纯数据库的扣减实现方案，如果以常规的机器或者 Docker 来进行评估，此方案较难实现单机过万的 TPS。之所以介绍，是想告诉你，**架构是面向业务功能、成本、实现难度、时间等因素的取舍，而不是绝对地追求高性能、高并发及高可用等非功能性指标**。

另外，在上一讲里介绍的扣减业务的技术实现需求点、数据库表结构信息等内容，其实是和技术无关的，它们属于通用的基本信息和标准定义。因此，今天我们讲解的方案将直接复用以上信息，不再赘述，有忘记的或者直接跳读此讲的同学可以翻到上一讲进行复习。

这一讲，我将由浅入深地介绍如何基于缓存来实现单机万级这一并发扣减目标。

纯缓存方案浅析

纯数据库的方案虽然避免了超卖与少卖的情况，但因采用了事务的方式保证一致性和原子性，所以在 SKU 数量较多时性能下降较明显。

> 注：事务本质上有四个特点 ACID：原子性（Atomicity）、一致性（Consistency ）、隔离性（Isolation）及 持久性（Durability）。

因为扣减有一个要求即当一个 SKU 购买的数量不够时，整个批量扣减就要回滚，因此，我们需要使用类似 for 循环的方式对每一个扣减 SQL 的返回值进行检查。另外一个原因是，当多个用户买一个 SKU 时，它的性能也并不乐观。因为当出现高并发扣减或者并发扣减同一个 SKU 时，事务的隔离性会导致加锁等待以及死锁情况出现。

现在看来，实现单机万级的并发扣减好像遥遥无期了。别急，还记得在上一讲里强调的，架构是在对问题清晰定义之后演化来的理念吗?

下面我们对问题再次梳理一遍，进而寻找可升级演化的方案。

首先，你要知道扣减只需要保证原子性即可，并不需要数据库提供的 ACID。在扣减库存时，重点是保证商品不超卖不少卖。而持久化这个功能，只有在数据库故障切换及恢复时才有需要，因为被中断的事务需要持久化的日志进行重演，也就是说持久化是主功能之外的后置功能、附加功能。

那么我们是否可以去掉持久化这一后置的、附加的功能？或者是否存在可替代方案？你先不着急回答这个问题，跟着我的思路咱们继续分析，自会寻得答案。

其次，在提升性能方面最简单、最快速的方案便是升级硬件。不管你使用的是哪一个厂商的数据库实现，提升或者替换部署数据库机器的硬件配置，都可以显著提升性能。虽然提升硬件可以解决问题，但与此同时也有另外一个问题——硬件的资金成本非常昂贵，动辄上百万、千万。

经过上面的分析，再请你思考第二个问题：**为什么当年阿里浩浩荡荡的发起了去 IOE 运动，转而采用性能相对较弱的 MySQL 及相对应的硬件呢**?

究其原因也是资金成本的考虑。此时，我们可以转换一个思路，既然提升或者替换机器配置可以提升性能，按此套路，是不是提升或者替换数据库存储也是一种方案？在不改变机器配置的情况下，把传统的 SQL 类数据库替换为性能更好的 NoSQL 类数据存储试试?

是不是有一个性能又好同时又能够满足扣减多个 SKU 具有原子性的 NoSQL 数据库呢？行文至此，答案显然是可以的。

Redis 作为最近几年非常流行的 NoSQL 数据库，它的原始版本或者改造版本基本上已经被国内所有互联网公司或者云厂商所采用。不管是微博爆点事件的流量应对，还是电商的大促流量处理，它的踪影无处不在，可见它在高性能上的能力是首屈一指。另外，因为 Redis 是开源软件且架构简单，部署在普通的 Docker 即可，成本非常低。

此外，Redis 采用了单线程的事件模型，保障了我们对于原子性的要求。对于单线程的事件模型，简单的比喻就是说当我们多个客户端给 Redis 同时发送命令后，Redis 会按接收到的顺序进行串行的执行，对于已经接收而未能执行的命令，只能排队等待。基于此特性，当我们的扣减请求在 Redis 执行时，也即是原子性的。此特性刚好符合我们对于扣减原子性的要求。

方案实现剖析

在确定了使用缓存来完成扣减和高性能后，为了帮助你理解，这里我们结合扣减服务的整体架构图来进一步分析：

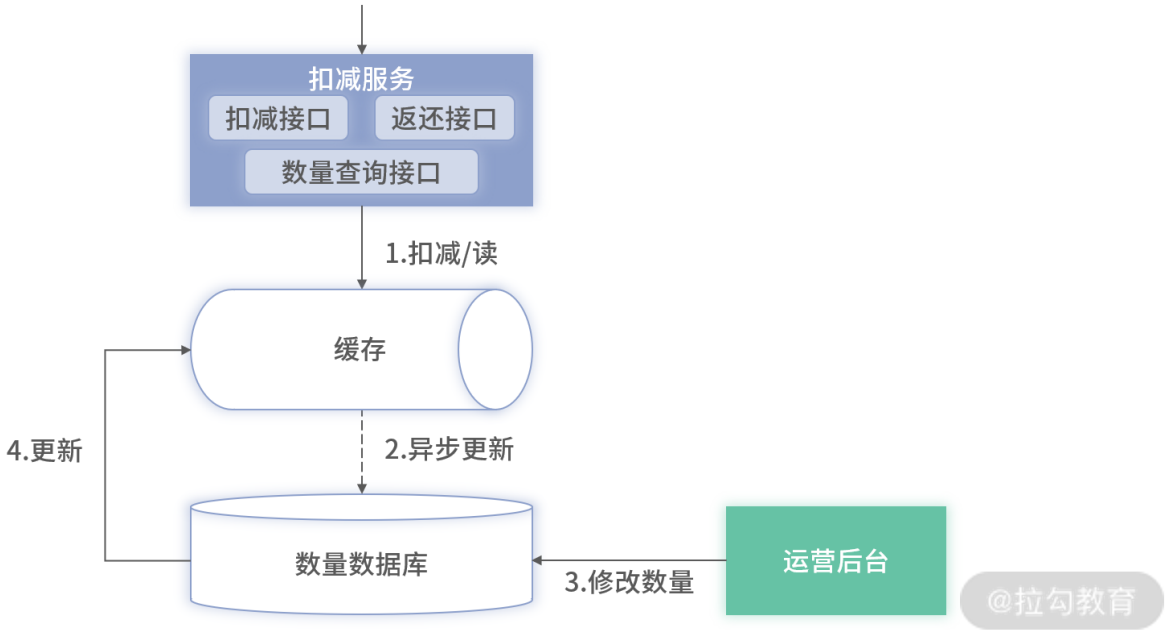

图 1：纯缓存架构

上图中的扣减服务和上一讲里的扣减服务一样，都提供了三个在线接口。但此时扣减服务依赖的是 Redis 缓存而不是数据库了。我们顺着上一讲的思路，继续以库存为场景讲解扣减服务的实现。

缓存中存储的信息和上一讲中的数据库表结构基本类似，包含当前商品和剩余的库存数量和当次的扣减流水，这里要注意两点。

- 首先，因为扣减全部依赖于缓存不依赖数据库，所有存储于 Redis 的数据均不设置过期并全量存储。
- 其次，Redis 是以 k-v 结构为主，伴随 hash、set 等结构，与 MySQL 以表 + 行为主的结构有一定的差异。Redis 中的库存数量结构大致如下：

key为：sku_stock_{sku}。前缀sku_stock是固定不变，所有以此为前缀的均表示是库存。{sku}是占位符，在实际存储时被具
value:库存数量。当前此key表示的sku剩余可购买的数量。

在实际应用中，上述 key 的 sku_stock_ 前缀一般会简写成 ss_ 或者可以起到和其他 key 区分的较短形式。当我们存储的 SKU 有百万、千万级别时，此方式可极大地降低存储空间，从而降低成本，毕竟内存是比较昂贵的。

对于 Redis 中存储的流水表采用 hash 结构，即 key + hashField + hashValue 的形式。结构大致如下：

key：sx_{sku}。前缀sx_是按上述缩短的形式设计的，只起到了区分的作用。{sku}为占位符
hashField：此次扣减流水编号。
hashValue：此次扣减的数量

在一次扣减时，会按 SKU 在 Redis 中先扣减完库存数量再记录流水信息。

我们在上一讲里已经介绍过，扣减接口支持一次扣减多个 SKU + 数量。查询 Redis 的命令文档时你会发现：

- 首先，Redis 对于 hash 结构不支持多个 key 的批量操作；
- 其次，Redis 对于不同数据结构间不支持批量操作，比如 KV 与 Hash 间。

如果对于多个 SKU 不支持批量操作，我们就需要按单个 SKU 发起 Redis 调用。在上文中提到过，Redis 不对命令间保证单线程执行。如果采用上述 Redis 的数据结构，一次扣减必须要发起多次对 Redis 的命令才可完成。这样，上文提到的利用 Redis 单线程来保证扣减的原子性此时则满足不了了。

针对上述问题，我们可以采用 Redis 的 lua 脚本来实现批量扣减的单线程诉求。

lua 是一个类似 JavaScript、Shell 等的解释性语言，它可以完成 Redis 已有命令不支持的功能。用户在编写完 lua 脚本之后，将此脚本上传至 Redis 服务端，服务端会返回一个标识码代表此脚本。在实际执行具体请求时，将数据和此标识码发送至 Redis 即可。Redis 会和执行普通命令一样，采用单线程执行此 lua 脚本和对应数据。

当用户调用扣减接口时，将扣减的 SKU 及对应数量 + 脚本标示传递至 Redis 即可，所有的扣减判断逻辑均在 Redis 中的 lua 脚本中执行，lua 脚本执行完成之后返还是否成功给客户端。

lua 脚本执行流程

当请求发送到 Redis 后，lua 脚本执行流程如下图 2 所示：

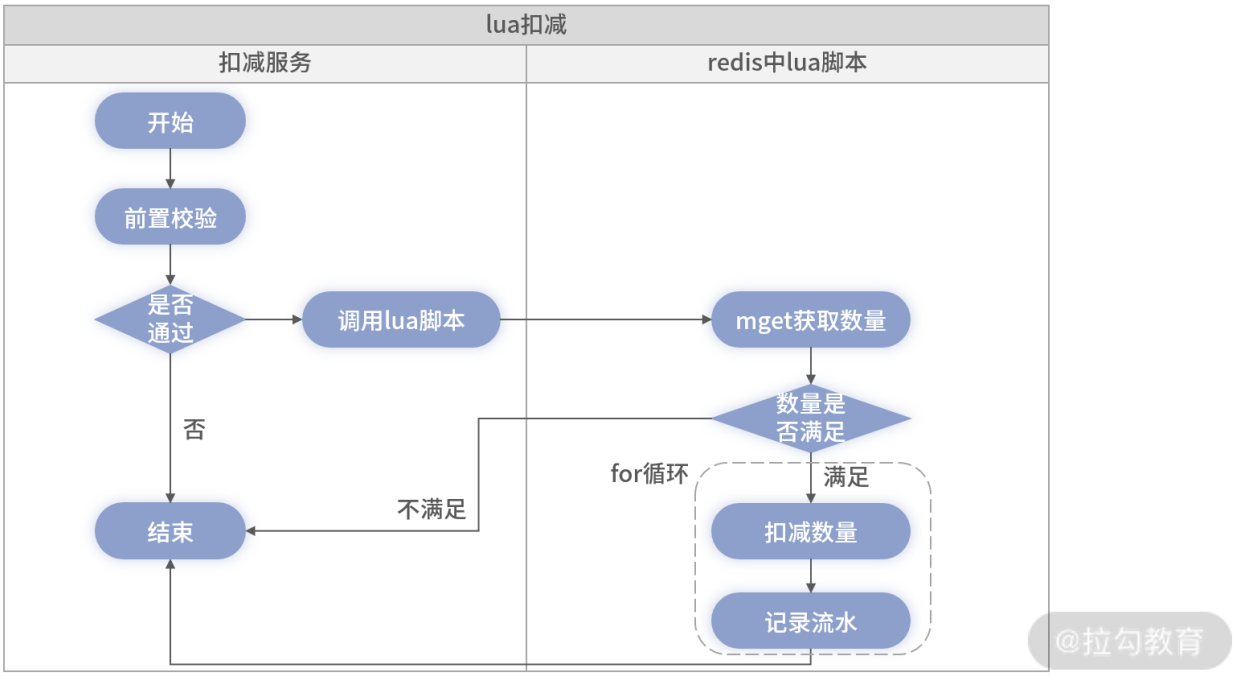

图 2：lua 脚本执行流程

Redis 中的 lua 脚本执行时，首先会使用 get 命令查询 uuid 是否已存在，如已存在则直接返回，并提示用户请求重复。当防重通过后，会按 SKU 批量获取对应的剩余库存状态并进行判断，如果其中一个 SKU 此次扣减的数量大于剩余数量，则直接给扣减服务返回错误并提示数量不足。通过 Redis 的单线程模型，确保当所有 SKU 的扣减数量在判断均满足后，在实际扣减时，数量不够的情况是不会出现的。同时，单线程保证判断数量的步骤和后续扣减步骤之间，没有其他任何线程出现并发的执行。

判断数量满足之后，lua 脚本后续就可以按 SKU 进行循环的扣减数量并记录流水。

当 Redis 扣减成功后，扣减接口会异步的将此次扣减内容保存至数据库。异步保存数据库的目的是防止出现极端情况—— Redis 宕机后数据未持久化到磁盘，此时我们可以使用数据库恢复或者校准数据。

最后，在纯缓存的架构图（图 2）中还有一个运营后台，它直接连接了数据库，是运营和商家修改库存的入口。当商品补齐了新的货物时，商家在运营后台将此 SKU 库存数量加回。同时，运营后台的实现需要将此数量同步的增加至 Redis，因为当前方案的所有实际扣减都在 Redis 中。

至此，采用纯缓存扣减的基本方案已经介绍结束了。因为实际的压测和很多因素相关，比如机器配置、压测的参数等，此处就不给出具体数字。但目前这个方案已经可以满足支撑单机万级的扣减了。下面我们再来看一看如何应对异常情况。

异常情况分析

因为 Redis 不支持 ACID 特性，导致在使用 Redis 进行扣减时相比纯数据库方案有较多异常场景需要处理，此处我挑选几个重要的给你讲解。

- **第一个场景是 Redis 突然宕机的场景**。

如果 Redis 宕机时，请求在 Redis 中只进行了前置的防重和数量验证，此时则没有任何影响，直接返回给客户扣减失败即可。

但如果此时 Redis 中的 lua 脚本执行到了扣减逻辑并做了实际的扣减，则会出现数据丢失的情况。因为 Redis 没有事务的保证，宕机时已经扣减的数量不会回滚。宕机导致扣减服务给客户返回扣减失败，但实际上 Redis 已经扣减了部分数据并刷新了磁盘，当此 Redis 故障处理完成再次启动后或者 failover 之后，部分库存数量已经丢失了。

为了解决此问题，可以使用数据库中的数据进行校准。常见方式是开发对账程序，通过对比 Redis 与数据库中的数据是否一致，并结合扣减服务的日志。当发现数据不一致同时日志记录扣减失败时，可以将数据库比 Redis 多的库存数据在 Redis 中进行加回。

- **第二个场景是扣减 Redis 完成并成功返回给客户后，异步刷新数据库失败了的情况。**

此时，Redis 中的数据库是准的，但数据库中的库存数据是多的。在结合扣减服务的日志确定是 Redis 扣减成功但异步记录数据失败后，可以将数据库比 Redis 多的库存数据在数据库中进行扣减。

升级纯缓存实现方案

上述的纯缓存方案在使用了 Redis 进行扣减实现后，基本上完成了扣减的高性能和高并发，满足了我们最初的需求。那整体方案上还有哪些可以优化的空间呢?

在“**第 12 讲**”里我们介绍过，扣减服务不仅包含扣减接口还包含数量查询接口。查询接口的量级相比写接口至少是十倍以上，即使是使用了缓存进行抗量，但读写都请求了同一个 Redis，将会导致扣减请求被读影响。

其次，运营在后台进行操作增加或者修改库存时，是在修改完数据库之后在代码中异步修改刷新 Redis。因为数据库和 Redis 不支持分布式事务，为了保证在修改时它们数据的一致性，在实际开发中，需要增加很多手段保证数据一致性，成本较高。

对于上述两个问题，我们可以做两方面的改造。

第一个是和“**第 12 讲**”里 MySQL 的优化方案思路一样，增加一个 Redis 从结点，在扣减服务里根据请求类型路由到不同的 Redis 节点。使用主从分离的好处是，不用太多的数据同步开发，直接使用 Redis 主从同步方案，成本低开发量小。

第二个是运营后台修改数据库数量后同步至 Redis 的逻辑使用 binlog 进行处理，关于如何接入和使用 binlog，你可以参见“**第 12 讲**”的内容。

当商家修改了数据库中的数量之后，MySQL 等数据库的 binlog 会自动发出，在数据转换模块接受 binlog 并转换格式插入 Redis 即可。因为 binlog 消费是采用 ack 机制，如果在转换和插入 Redis 时出错，ack 不确定即可。下一次数据转换代码运行时，会继续上一次未消费的 binlog 继续执行。最终，binlog 的机制不需要太多逻辑处理即可达到最终一致性。相比采用不借助 binlog 的方式，此方案成本和复杂度均较低。

优化后的整体方案如下图 3 所示：

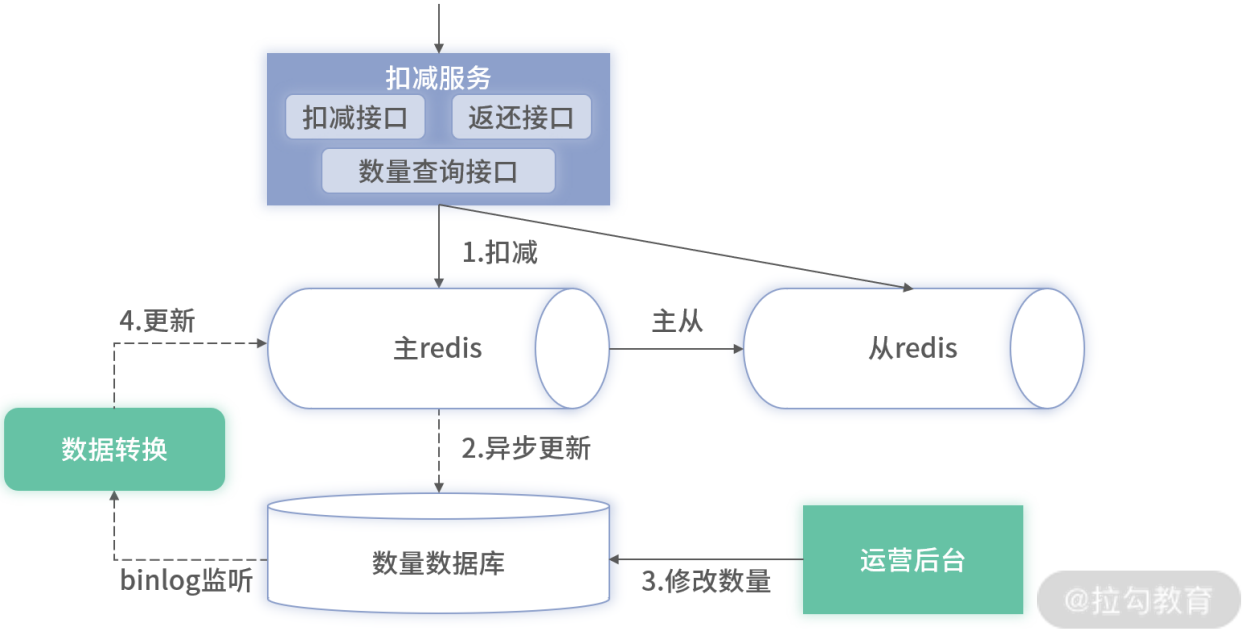

图 3：纯缓存升级版架构

纯缓存方案适用性分析

相比于纯数据库扣减方案，纯缓存方案也存在一定的优缺点和适用性。

纯缓存方案的主要优点是性能提升明显。使用缓存的扣减方案在保证了扣减的原子性和一致性等功能性要求之外，相比纯数据库的扣减方案至少提升十倍以上。

除了优点之外，纯缓存的方案同样存在一些缺点。Redis 及其他一些缓存实现，为了高性能，并没有实现数据库的 ACID 特性。**导致在极端情况下可能会出现丢数据，进而产生少卖**。另外，**为了保证不出现少卖，纯缓存的方案需要做很多的对账、异常处理等的设计，系统复杂度会大幅增加**。

对于纯缓存的扣减的优缺点有了一定了解后，可以发现纯缓存在抗并发流量时，效果非常显著。因此，它较适合应用于高并发、大流量的互联网场景。但在极端情况下，可能会出现一些数据的丢失。因此，它优先适合对于数据精度不是特别苛刻的场景，比如用户购买限制等。

但如果上述的异常场景都有降级方案应对，保证最终一致性。它也是可以应用在库存扣减、积分扣减等等场景。在我所经历的和了解的实践中，是有很多公司将此方案应用在非常精度的场景里的。

总结

在上一讲中的纯数据库方案无法完全满足量级要求时，本讲介绍了纯缓存的扣减方案。着重讲解了为什么纯缓存可以满足扣减的功能需求，对于分析的过程希望你能够理解并应用，而不是关注最终提出的方案。**作为一名优秀的开发人员，你要知道架构图是一个最终态，是静止的，它并不能 100% 直接应用到你所面对的场景，而分析思路却是可以复制和模仿的**。

其次，本讲也分析了纯缓存方案存在的一些异常场景。在实战中，正常流程是简单的，而异常流程的思考与处理十分的复杂与烦琐，同时也最能体现技术性，请你务必注意与加强。

最后，我再留给你一道思考题，如果此处的 Redis 是一个集群，而不是一个单独实例，方案应该如何演化？你可以把你的答案、思路或者课后总结写在留言区，我们一起交流。