库存扣减-订单的第一道关卡

> 向前爬行，肖申克先生，前方会有光。

> 这是一个系列的文章，我们先讲库存扣减，我们先搭建一个基本可用的骨架，然后再逐步完善。

前言

订单在创建的时候，需要扣除对应商品的库存，那么一个核心的问题就是我们不能超买，那么我们该如何实现不超买呢，那么我们可以借助数据库乐观锁的思想来实现库存扣减，首先我们有一张商品表，为了方面讨论我们的表里面就只有skuId和库存数量:

字段名	含义	字段类型
skuId	商品标识	长整型
num	库存数量	整型

spu是一个商品的宏观概念，比如小米 14 有不同的规格和颜色，每个不同的规格和颜色就是一个sku。现在我们流量比较小，实现不超卖的方式是通过数据库的乐观锁来实现的:

```
update product  set  num = num - 下单数量  where skuid = #{skuId} and num - 下单数量 > 0
```

之所以说这里是乐观锁的思想，是因为这条SQL也蕴含了乐观锁的思想比较并交换，但是具体到数据库实现的时候，如果多个事务执行上面的sql，那么其实是排队执行的，这意味着更大的流量奔涌而来的时候会扛不住。对此解决方案也有两种，为了避免多个事务操作一条记录，将热点商品热点库存拆成多行。我们本篇讲解的是另外一种方案，我们将目光盯上了Redis，那先更新Redis再更新数据库？ 这看起来更快一些，但是我们在《数据库缓存一致性学习笔记(一)》这篇文章已经讨论过了，如果先更新Redis后更新数据库，在下面的情况下数据库就会变成不对的:

```
updateRedis(key,data,time);
updateDB(data);
```

那更新失败了，回滚Redis？ 这不现实，首先Redis和MySQL是两个软件，Redis和MySQL的事务处理机制不同，Redis不支持回滚。这里我们简单回忆一下Redis中的事务。

说起Redis 中的事务

一般我们说起的事务，默认是数据库中的事务，数据库中的事务满足ACID四个特性:

- 原子性(**Atomicity**): 对于不可分割的操作，要么成功要么失败。
- 一致性(**Consistency**): 现实世界的一些约束到了软件世界也要予以保持，比如说人民币的最大币值等。如果数据库中的数据全部符合现实世界的约束，我们就说这些数据就是符合一致性的。
- 隔离性(**Isolation**): 对于现实世界的状态转换对应到某些数据库操作来说，不仅要保证这些操作以原子性的方式来执行完成，而且要保证其它的状态转换不会影响到本次状态转换，这个规则被称之为隔离性。

> 关于隔离性我的解读是事务在并发执行的时候，要有就像排队执行一样的效果

- 持久性(**Durability**): 当现实世界的一个状态完成后，这个转换的结果将永久保留，这个规则我们称之为持久性。当把现实世界的状态转换映射到数据库世界，持久性意味着转换对应的数据库操作所修改的数据都应该在磁盘上保留下来。

我们回忆一下Redis的事务:

> Redis Transactions allow the execution of a group of commands in a single step, they are centered around the commands MULTI, EXEC, DISCARD and WATCH. Redis Transactions make two important guarantees

Redis的事务允许在单个步骤中执行一组命令，它们主要围绕 MULTI、EXEC、DISCARD 和 WATCH 命令。

Redis 事务提供两个重要保证:

- 事务中的所有命令都是串行化的，并按顺序执行。在事务的执行过程中，其他客户端发送的请求永远不会被服务。这保证了命令作为单个独立操作执行。
- EXEC 命令触发事务中所有的命令执行，因此如果客户端在调用EXEC命令之前在事务的上下文中失去了与服务器的连接，则不会执行任何操作。相反，如果调用了EXEC命令，则会执行所有的操作。当Redis使用AOF(append-only file)持久化机制的时候，Redis 会确保使用单个 write(2) 系统调用将事务写入磁盘。但是如果 Redis 服务崩溃或或以某种强制方式(hard way)被系统管理员杀死, 则可能只有部分操作被注册 , Redis 将在重启时检测到这种情况,并将以错误退出。使用 redis-check-aof 工具可以修复仅附加文件,从而删除部分事务,以便服务器可以再次启动。

从这两个重要保证我们可以看到Redis是如何保证持久性和隔离性的，保证隔离性Redis事务中的所有命令都是串行化的，并且按顺序执行，在事务执行过程中，其他客户端的请求永远不会服务，这就是在排队执行。那么持久性，上面提到了AOF持久化，也就是说如果 Redis 每执行一条写操作命令，就把该命令以追加的方式写入到一个文件里，然后重启 Redis 的时候，先去读取这个文件里的命令，并且执行它。

那原子性呢，Redis能否做到一组操作a、b、c、d，a和b执行成功，c执行失败，整体失败呢，Redis的回答是不能，不支持:

> Errors happening *after* EXEC instead are not handled in a special way: all the other commands will be executed even if some command fails during the transaction.
>
> 在EXEC命令被执行之后发生的错误不会有特殊处理: 即使在事务的执行过程中，有命令失败，事务中的其他命令也会继续执行。
>
> Redis does not support rollbacks of transactions since supporting rollbacks would have a significant impact on the simplicity and performance of Redis.
>
> Redis 不支持事务的回滚，支持回滚将会Redis的性能和简洁性。

> It's important to note that **even when a command fails, all the other commands in the queue are processed** – Redis will *not* stop the processing of commands.

> 要意识到，当命令执行失败，其他在队列里面的命令也会被执行，Redis将会停止处理这些命令。

将Redis执行的每一条写操作命令都把该命令写入到文件里面，这里面有两个问题值得我们关注:

1. 如果将Redis这个命令从内存写入到磁盘上，这会很慢，影响Redis 的响应速度
2. 这种操作产生的文件会很大。

对于第一个问题，Redis选择了使用系统调用选择将内容先刷新到系统缓冲区，但是UNIX中的系统调用只保证内容会被加入到缓冲区，并不保证一定会刷到磁盘上，所以还需要一个系统调用确保将数据一定刷到磁盘上，对于Redis有三个选项，一个是no，这表示Redis并不会进行主动刷盘，让操作系统进行刷盘，一个每秒刷一次，一个是每次调用完成之后就刷磁盘一次。

第一种当然性能最好，但是可靠性无法保证，第三种安全性最好，但是性能差。Redis选择了第二种吗？不完全是，这一种的缺点是上一次落盘之后的数据就没了，虽然会丢失数据，但是丢失的不多，最多丢失两秒的数据，在Redis的源码里面我们可以看到(见参考资料[2]), 如果当前正在写，可以看到，如果当前正在写，并且 `server.aof_flush_postponed_start == 0` 的话，就会跳过这一次；或者如果上一次跳过的时间 和现在相差不到2秒的话，也跳过，但是如果超过2秒，就要写入。这也是上面说最多会丢2秒数据的缘故。 除此之外，为了不影响Redis的读写操作，AOF持久化机制会开启一个线程去发起系统调用去刷磁盘。

对于第二个问题来说， Redis的答案是fork一个子进程去重写AOF文件。

所以还是应该先更新数据库

我们接着说回上面更新缓存，再更新数据库， 不采取先更新缓存后更新数据库的原因是我们一定要保证DB是对的，还有另外一种情况也会导致数据错乱:

时间	写请求A	写请求B	问题
T1	更新缓存为0		
T2		更新缓存为1	
T3		更数据库为1	
T4	更新数据库数据为0		此时缓存是对的，数据库的数据倒是不对了

因此先更新缓存再更新数据库这种策略我们应该避免采取，那如果我们选择在更新数据库之前删除缓存呢？ 其实也会有对应的问题:

时间	线程A(写请求)	线程B(读请求)	问题
T1	删除缓存成功		
T2		读取缓存失败，再从数据库读取数据100	
T3	更新数据库中数据的值为99		
T4		将读到的数据100写入缓存	缓存和数据库不一致

在读写并发的情况下，很轻松就出现了数据不一致，对于更新数据库之前删除缓存，有个很出名的策略叫延迟双删，既然在删除缓存之后会有时间窗口导致缓存和数据库不一致，那么我在执行写请求之后，等到差不多的时间再重新删除这个缓存值，那差不多是差多少，这个时间该怎么估测。

时间	线程 A（写请求）	线程 B（读请求）	线程 C（读请求）	潜在问题
T5	sleep(N)	缓存存在，读取到缓存旧值 100		其他线程可能在双删成功前读到脏数据
T6	删除缓存值			
T7			缓存缺失，从数据库读取数据的最新值（99）	

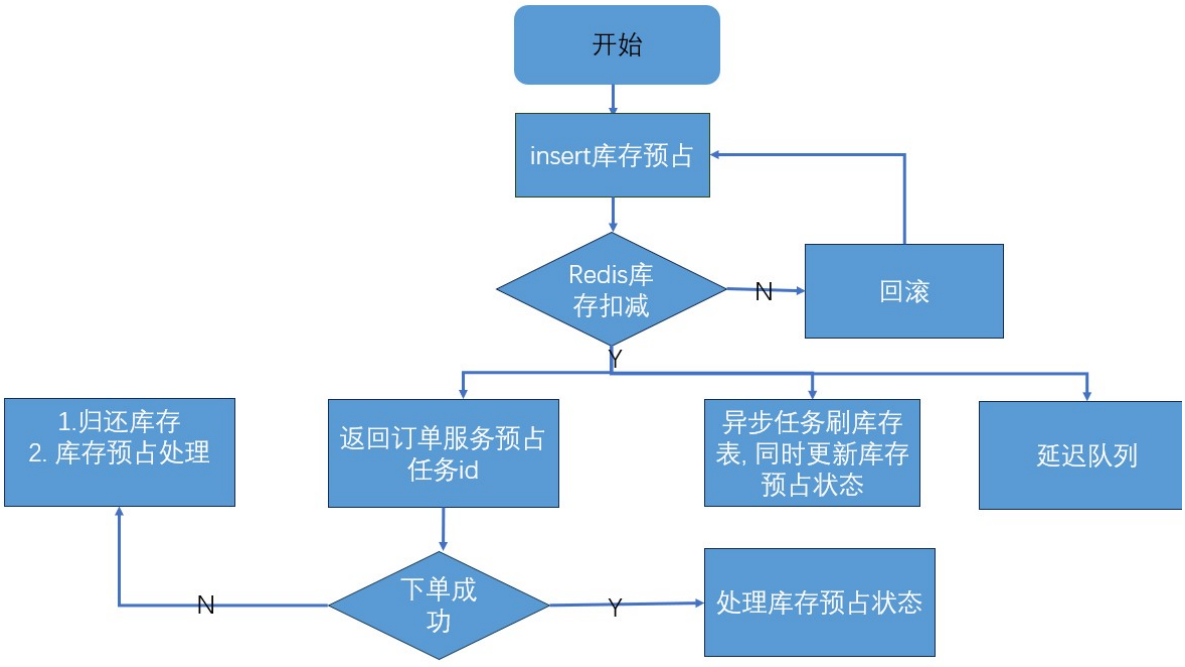
等待时间太短，线程 A 第二次删除缓存的时间依旧早于线程 B 把脏数据写回缓存的时间，那么相当于做了无用功。而 N 如果设置得太长，那么在触发双删之前，新请求看到的都是脏数据。

所以我们的策略都是先更新数据库后对缓存做处理，对缓存做处理也有两种策略，一种是删除，一种是更新。写多读少用更新，读多写少用删除。那么我们如何维护数据库和缓存的一致性，一般我们采用的是binlog + mq的方式来保证一致性，数据库发生变化的时候触发MQ通知，做对应的更新。

最终的答案是库存预占

字段	含义	类型
id	唯一标识	长整型
skuId	商品标识	长整型
buy_num	购买数量	整型
expiration_time	过期时间	时间
user_id	预占人Id	字符串
pre_occupancy_status	预占状态	10待成单，20 成单成功 30 业务成单失败归还库存 40 过期归还库存 50 扣减库存失败
create_date	创建时间	时间
update_date	更新时间	时间
creator_name	创建人名称	字符串
is_delete	是否删除	整型

上面是一张库存预占表，下面是订单创建的前置流程的流程图:

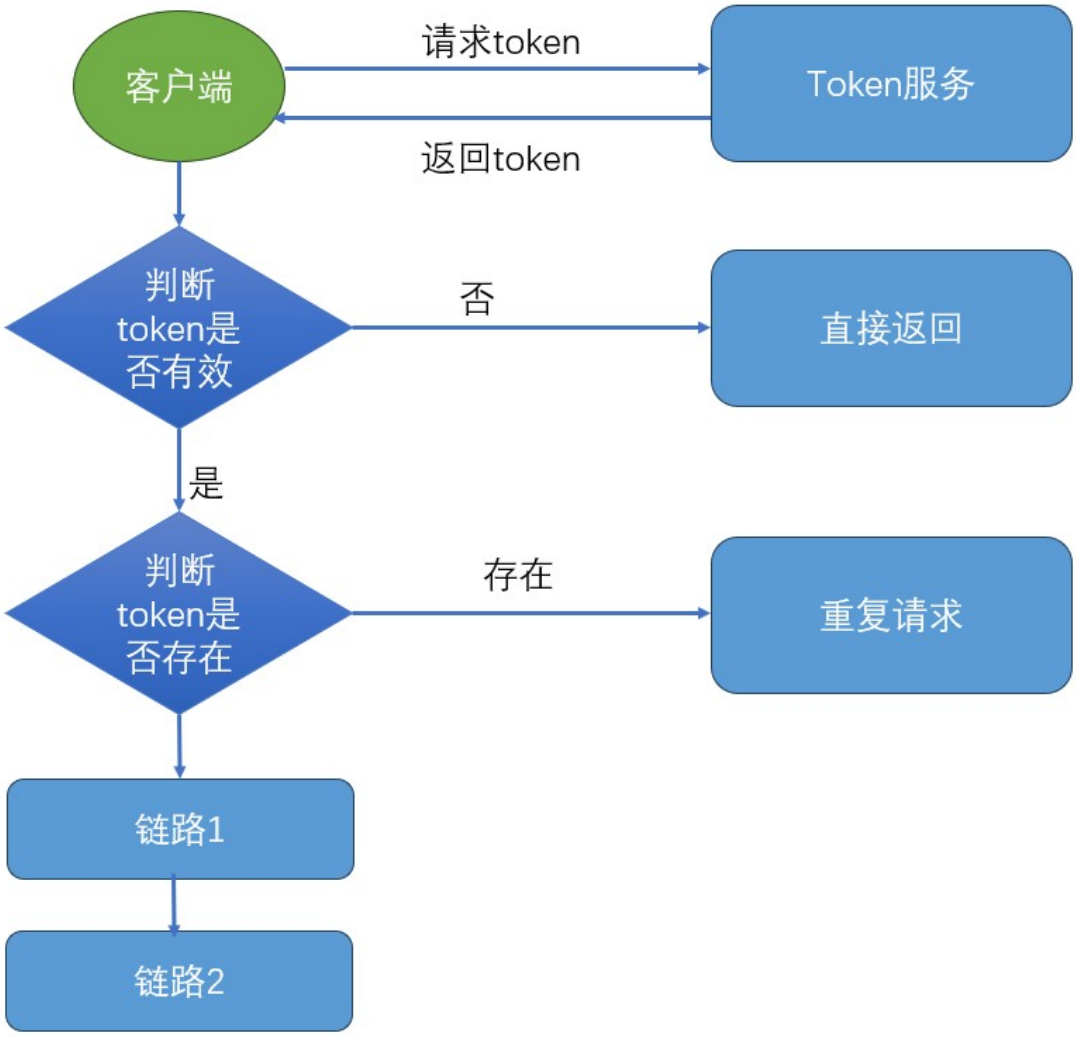


首先插入库存预占数据，表里面的字段如上面的表格所示，根据业务确定过期时间，首先插入库存预占，然后Redis(Lua脚本更新)扣减库存，事务状态下排队执行，所以可以信赖Redis，更新成功之后生成一个异步任务去刷库存表，同时根据业务过期时间一个延迟队列来归还库存。然后执行下单流程，下单失败归还库存，库存预占处理。下单成功处理库存预占状态。

这里的另外一个问题是防止用户重复提交，防止用户提交会有多层合作，每一层进行规避:

1. 页面提交接口之后，解决方案为: 提交接口再收到响应之前，按钮禁用或者处于loading状态。
2. 由于网络抖动导致接口超时，解决方案有两种一种是分布式锁，但在分布式系统中为了提升整体的吞吐量，扣减库存可能分布在多个节点上，这种情况下分布式锁的key会变得难以选择，比如库存服务选的是: stock_deduct(这是个字符串，只是用来区分业务而已) + userId + skuId + num，然后库存服务返回库存预占成功，此时调用订单服务生成订单超时。用户再点击一次仍然会超买。如果我们将库存扣减放到生成订单接口里面，这就违背了我们拆分服务的初衷，原本我们拆分服务的初衷就是提升系统整体的吞吐量。解决这个问题有两种思路，一种是允许在一段时间超卖，因为在上面的任务表里面我们会有一个延迟队列去归还库存。

   另一种思路是 我们需要的是一条链路的许可证，而不是单个资源的许可，解决这个问题的一个方案是token令牌机制，注意这个token不是登录token，属于资源许可类型的token，根据 userId + skuId + num + createDate + 随机数(或者你用UUID + 随机数生成也没关系，但是要注意防止假冒)，加密生成用来判断token的有效性，缓存一段时间(或者由最后一个节点删除token)，这个缓存时间取决于主链路的时间，没有推之四海皆准的规则。生成之后发放给客户端，客户端请求接口的时候携带此token。



，每经过一条链路就判断缓存是否存在，如果不存在就将其缓存在本地一段时间，经过链路的时候判断token是否存在，如果存在判定为重复请求。这种设计的前置要求意味着我们要尽可能减少主链路的请求时间。如果我们采用这种设计机制，由于网络是不稳定的，很有可能库存服务本地缓存过期，用户又重复点击了一次，这意味着我们前面两道防线被打穿，所以我们为了安全起见，可以在库存预占任务表

里面，再加入一个事件戳字段，使用userId+ buyNum + pre_occupancy_status +task_time_stamp。这个安全防御根据实际需要可设可不设。

总结一下

本篇我们总结了应对高并发下库存扣减的两种思路：热点商品分散和库存预占，库存预占的情况下，我们放了延时队列去做补偿防止超卖，核心思路就是库存预占 + Redis扣减 + 补偿。库存预占是为了方便排查问题，Redis扣减则是高性能的，延迟队列用于在后续的链路中没有正常形成订单将库存归还。

参考资料

[1] Understanding Redis Data Persistence: AOF vs. RDB https://medium.com/@gauravpatilsde/understanding-redis-data-persistence-aof-vs-rdb-58688dda2c32

[2] Redis源码阅读：AOF持久化 https://jiajunhuang.com/articles/2021_05_25-redis_source_code_5.md.html