14 | 如何利用缓存+数据库构建高可靠的扣减方案?

在上两讲里分别介绍了使用数据库和纯缓存实现的扣减方案。在需求层面上，上述两者都能实现业务需求。但均存在一些缺陷：

- 数据库方案的性能较差；
- 纯缓存方案虽不会导致超卖，但因缓存不具备事务特性，极端情况下会存在缓存里的数据无法回滚，导致出现少卖的情况。且因“第 13 讲”是异步写库，也可能发生异步写库失败，导致多扣的数据再也无法找回的情况。

因此，本讲将向你介绍一种新的实现方案——使用数据库+缓存的方式规避上述存在的潜在问题。

顺序写的性能更好

本讲的方案是借助了“**顺序写要比随机更新性能好**”这个特性进行设计的。

在向磁盘进行数据操作时，向文件末尾不断追加写入的性能要远大于随机修改的性能。因为对于传统的机械硬盘来说，每一次的随机更新都需要机械键盘的磁头在硬盘的盘面上进行寻址，再去更新目标数据，这种方式十分消耗性能。而向文件末尾追加写入，每一次的写入只需要磁头一次寻址，将磁头定位到文件末尾即可，后续的顺序写入不断追加即可。

对于固态硬盘来说，虽然避免了磁头移动，但依然存在一定的寻址过程。此外，对文件内容的随机更新和数据库的表更新比较类似，都存在加锁带来的性能消耗。

数据库同样是插入要比更新的性能好。对于数据库的更新，为了保证对同一条数据并发更新的一致性，会在更新时增加锁，但加锁是十分消耗性能的。此外，对于没有索引的更新条件，要想找到需要更新的那条数据，需要遍历整张表，时间复杂度为 O(N)。而插入只在末尾进行追加，性能非常好。

借力顺序写的架构

有了上述的理论基础后，只要对上一讲的架构稍做变更，就可以得到兼具性能和高可靠的扣减架构了，整体架构如下图 1 所示：

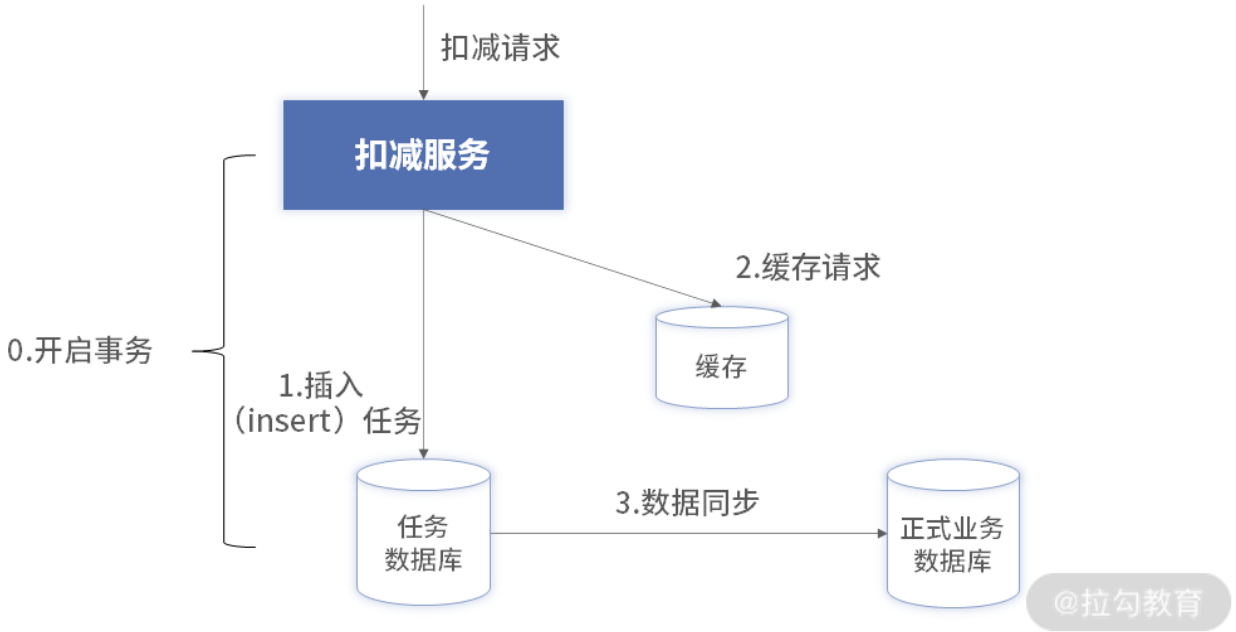


图 1：兼具性能和更加高可靠的扣减架构

上述的架构和纯缓存的架构区别在于，**写入数据库不是异步写入，而是在扣减的时候同步写入。**

这里你可能会有些疑问：同步的写入数据库是不是和“第 12 讲”讲述的内容类似？且数据库扣减的性能对于海量并发是扛不住的，这个方案是不是在倒退？

如果你仔细看架构图，会发现并非如此。同步写入数据库使用是 insert 操作，也就是顺序写，而不是 update 做数据库数量的扣减。因此，它的性能较好。

insert 的数据库称为**任务库**，它只存储每次扣减的原始数据，而不做真实扣减（即不进行 update）。它的表结构大致如下：

```
create table task{
  id bigint not null comment "任务顺序编号",
  task_id bigint not null
}
```

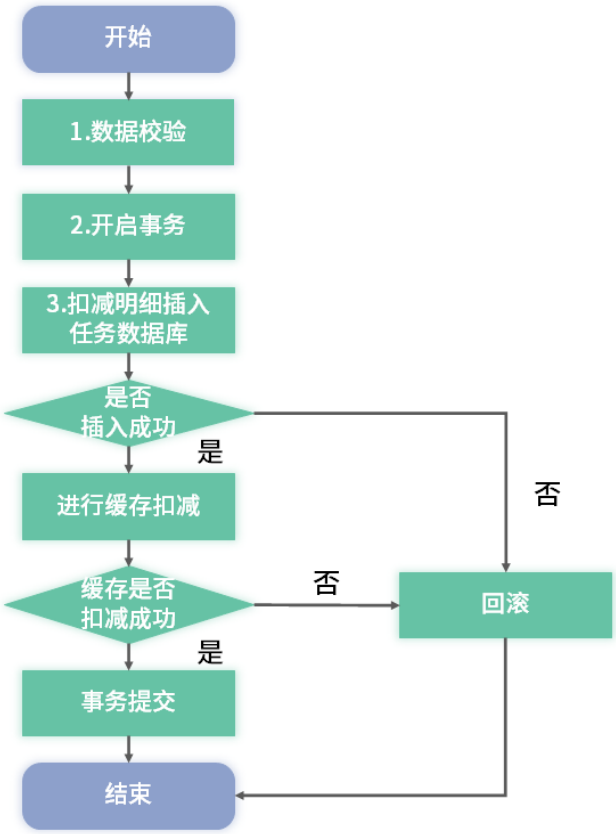
任务表里存储的内容格式可以为 JSON、XML 等结构化的数据。以 JSON 为例，数据内容大致可以如下：

```
{
  "扣减号":uuid,
  "skuid1":"数量",
  "skuid2":"数量",
  "xxxx":"xxxx"
}
```

在上述架构里，还有一个**正式业务库，这里面存储的才是真正的扣减明细和 SKU 的汇总数据。对于正式库里的数据，通过任务表的任务进行同步即可，此种方式保证了数据的最终一致性。

扣减流程

在引入了任务表之后，整体的扣减流程如下图 2 所示：



@拉勾教育

图 2：基于任务的扣减流程

上述的流程**和纯缓存的区别在于增加了事务开启与回滚的步骤，以及同步的数据库写入流程**，详细分析如下。

1. 首先是前置业务参数检验（包含基础参数、数量检验等），此步骤在本讲和前两讲的方案里都有。可以说，任何对外接口此功能都是不可或缺的，是完成业务验证性的必要一环。
2. 然后在图中编号 2 处，开始数据事务。
3. 当开始事务后，首先将此次序列化后的扣减明细写入到扣减数据库中的任务表里。
4. 假设数据库插入扣减明细失败，则事务回滚，任务表中无新增数据，数据一致，无任何影响。
5. 当数据库插入扣减明细成功后，便针对缓存进行扣减。和上一讲保持一致，使用 lua 等功能进行扣减即可。
6. 如果缓存扣减成功，也就是流程正常结束，提交数据库事务，给客户返回扣减成功。
7. 如果缓存扣减失败，有可能有两大类原因。
    1. 一类原因是此次**扣减的数量不够**，比如缓存里有 5 个数量，而实际此次扣减需要 10 个。当判断数量不够后，便调用缓存的归还并将数据库进行回滚即可。
    2. 第二类原因是缓**存出现故障，导致扣减失败**。缓存失败的可能性有很多，比如网络不通、调用缓存扣减超时、在扣减到一半时缓存突然宕机（故障 failover）了，以及在上述返回的过程中产生异常等。**针对上述请求，都有相应的异常抛出，根据异常进行数据库回滚即可，最终任务库里的数据都是准的。**

完成上述步骤之后，便可以进行任务库里的数据处理了。任务库里存储的是纯文本的 JSON 数据，无法被直接使用。需要将其中的数据转储至实际的业务库里。业务库里会存储两类数据，一类是每次扣减的流水数据，它与任务表里的数据区别在于它是结构化，而不是 JSON 文本的大字段内容。另外一类是汇总数据，即每一个 SKU 当前总共有多少量，当前还剩余多少量（即从任务库同步时需要进行扣减的），表结构大致如下：

```
create table 流水表{
  id bigint not null,
  uuid bigint not null comment '扣减编号',
  sku_id bigint not null comment '商品编号',
  num int not null comment '当次扣减的数量'
}comment '扣减流水表'
```

商品的实时数据汇总表，结构如下：

```
create table 汇总表{
  id bitint not null,
  sku_id unsigned bigint not null comment '商品编号',
  total_num unsigned int not null comment '总数量',
  leaved_num unsigned int not null comment '当前剩余的商品数量'
}comment '记录表'
```

原理分析

本讲介绍的数据库+缓存的架构主要利用了**数据库顺序写入要比更新性能快**的这一特性。此外，在写入的基础之上，又利用了**数据库的事务特性来保证数据的最终一致性**。当异常出现后，通过事务进行回滚，来保证数据库里的数据不会丢失。

在整体的流程上，还是复用了上一讲纯缓存的架构流程。当新加入一个商品，或者对已有商品进行补货时，对应的新增商品数量都会通过 Binlog 同步至缓存里。在扣减时，依然以缓存中的数量为准。补货或新增商品的数据同步架构如下图 3 所示：

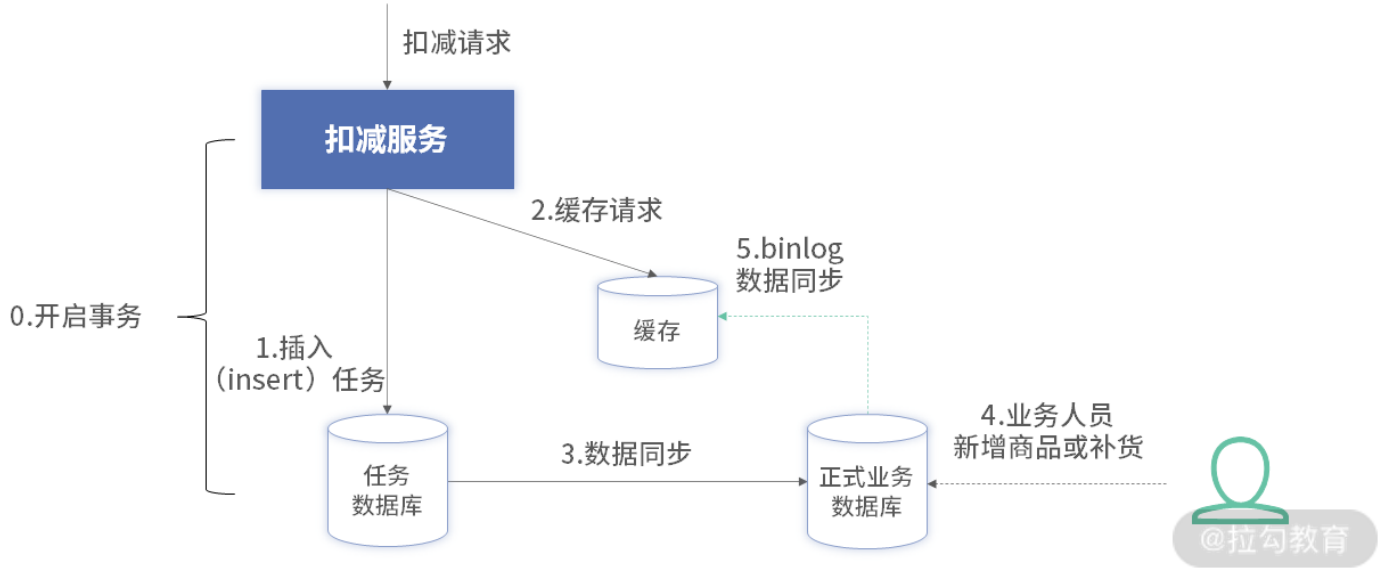


图 3：补货或新增商品的数据同步架构图

这里你可能会产生疑问：通过任务库同步至正式业务库里那份数据岂不是没用了？当然不是。**正式业务库异构的那份扣减明细和 SKU 当前实时剩余数量的数据，是最为准确的一份数据，我们以它作为数据对比的基准**。如果发现缓存中的数据不一致，就可以及时进行修复。对于数据校准，你可以参考“第 05 讲”里介绍的方案。

比如，当缓存扣减完成后，我们的应用客户端重启了，此时外部调用方的连接会断开，外部调用方判断此次调用失败。但因突然重启，当次完成的扣减在缓存里是没有完成返还的。但数据库采用的是事务，客户端重启时，事务就自动回滚了。此时，数据库的数据是正确的，但缓存的数据是少的。

在纯缓存的方案里，如果当时的异步刷库也失败了，则缓存数据一直都是少卖的。而数据库+缓存的方案，只会在一定时间出现少卖的情况，最终的数据一定是一致的。此方案会保证任务数据库和正式业务数据库里的数据准确性，出现故障后基于正式数据库进行异步对比修复即可。这便是两种方案的差异所在。

性能提升

进行方案升级后，我们便完成了一个更加可靠的扣减架构，且使用任务数据库的顺序插入也保证了一定的性能。但总的来说，即使是基于数据库的顺序插入，缓存操作的性能和数据库的顺序插入也不是一个量级，那么如何提升顺序插入任务数据库的性能和吞吐量呢？

这里我们回顾一下在“第 9 讲”（无状态存储）里介绍的内容和理念——**通过无状态的存储提升可用性**。同样的逻辑，**任务库主要提供两个作用，一个是事务支持，其次是随机的扣减流水任务的存取**。这两个功能均不依赖具体的路由规则，也是随机的、无状态的。因此，可以借鉴“第 9 讲”的架构对本讲的内容进行升级，升级后的架构如下图 4 所示：

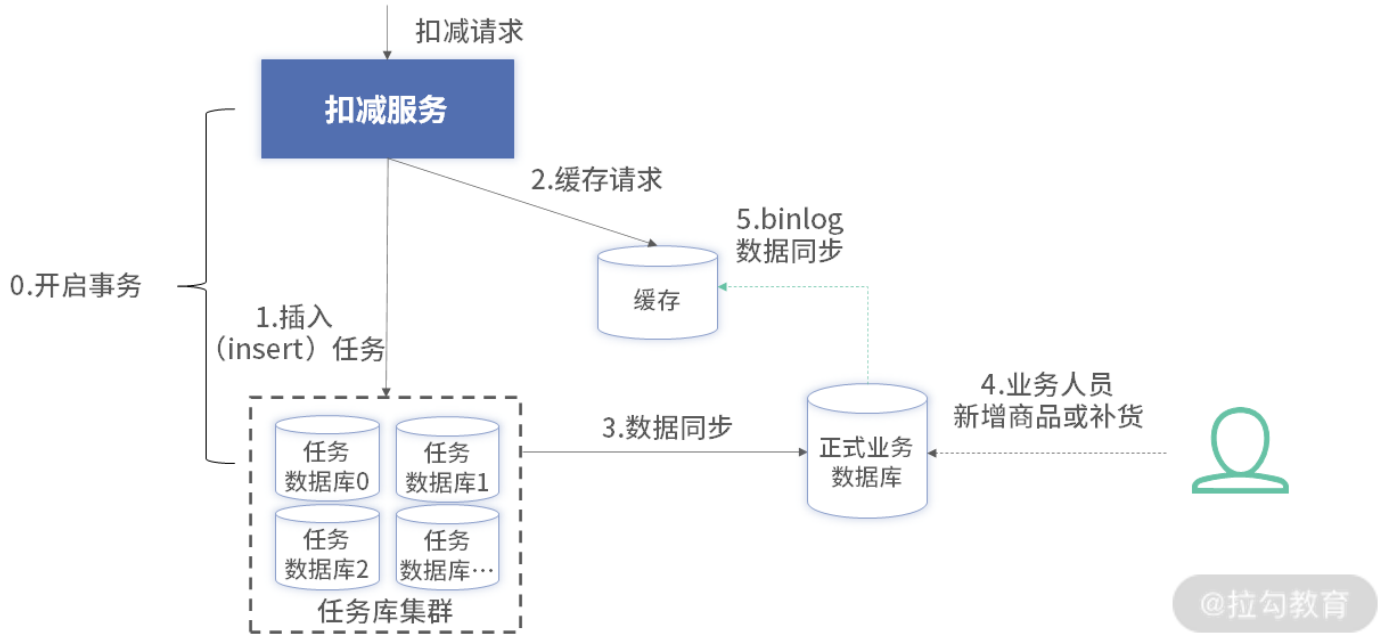


图 4：无状态存储的架构图

采用无状态存储后，任务库便可以进行水平扩展了，在性能和高可用上得到进一步的加强。具体的细节原理和落地步骤，你可以参考“第 9 讲”，这里不再赘述。

数据同步

任务库和业务正式库之间的数据同步和“第 9 讲”里介绍的无状态的存储基本类似，但整体实现上会更加简单。因为在业务上，扣**减前置依赖的均是缓存里的数据，业务正式库里的数据只用来做兜底使用**。因此最终只要使用 Worker 将数据从任务库同步至业务正式库即可，架构如下图 5 所示：

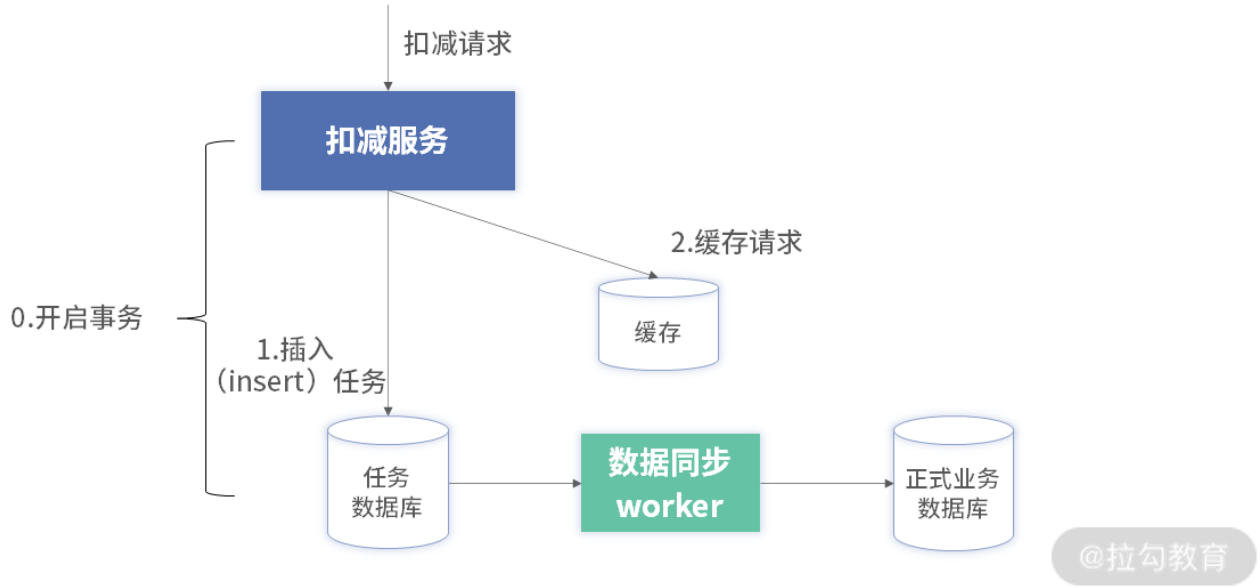


图 5：Worker 架构图

总结

在本讲里，我们介绍了通过**缓存和数据库结合**的方式，实现了一个更加可靠的扣减方案。相比纯缓存方案，即使使用了无状态的分库存储，它的性能也会有一定的损耗。但此方案的好处在于**数据更精准、更可靠**。对于类似额度扣减、实物库存扣减等场景，此方案均适用。而对于一些虚拟的次数限制，同时业务上能够容忍在一定概率下数据不准确的场景，也可以选择纯缓存的扣减方案。

此外，“**顺序追加写要比随机修改的性能好**”这个技巧，其实在很多场景里都有应用，是一个值得你深入学习和理解的技能。比如数据库的 Redo log、Undo log；Elasticsearch 里的 Translog 都是先将数据按非结构化的方式顺序写入日志文件里，再进行正常的变更。

当出现宕机后，采用日志进行数据恢复。

经过“12、13、14”这三讲的学习，我想你对“扣减”相关的内容已经掌握得如鱼得水了，欢迎你动一动手指，在留言区写一写思考、做一做总结。如果你对哪里有疑问，也可以在留言区提问，咱们一起讨论。