15 | 数据库与缓存的扩展升级与扣减返还

在本讲里，将会对扣减中涉及的两个公共话题进行讨论，分别是**异步任务的设计和扣减中的返还的设计**。

在“第 14 讲”和“第 9 讲”里，均使用了异步任务（Worker）来做无状态存储到正式业务库的数据同步。但关于具体如何设计异步任务来保证高可用，以及如何设计任务的执行来保障执行的速度，避免产生任务积压，其实并没有过多介绍。

此外，在本模块前三讲介绍的扣减方案里，只涉及扣减的正向流程。对于扣减后可能发生的返还过程中涉及的技术点，我将在本讲进行逐一讲解。

实现无主架构的任务

对于无状态存储集群的数据同步任务，最简单的实现方式便是对于每一个分库启动一个自循环的 Worker，它的架构如下图 1 所示：

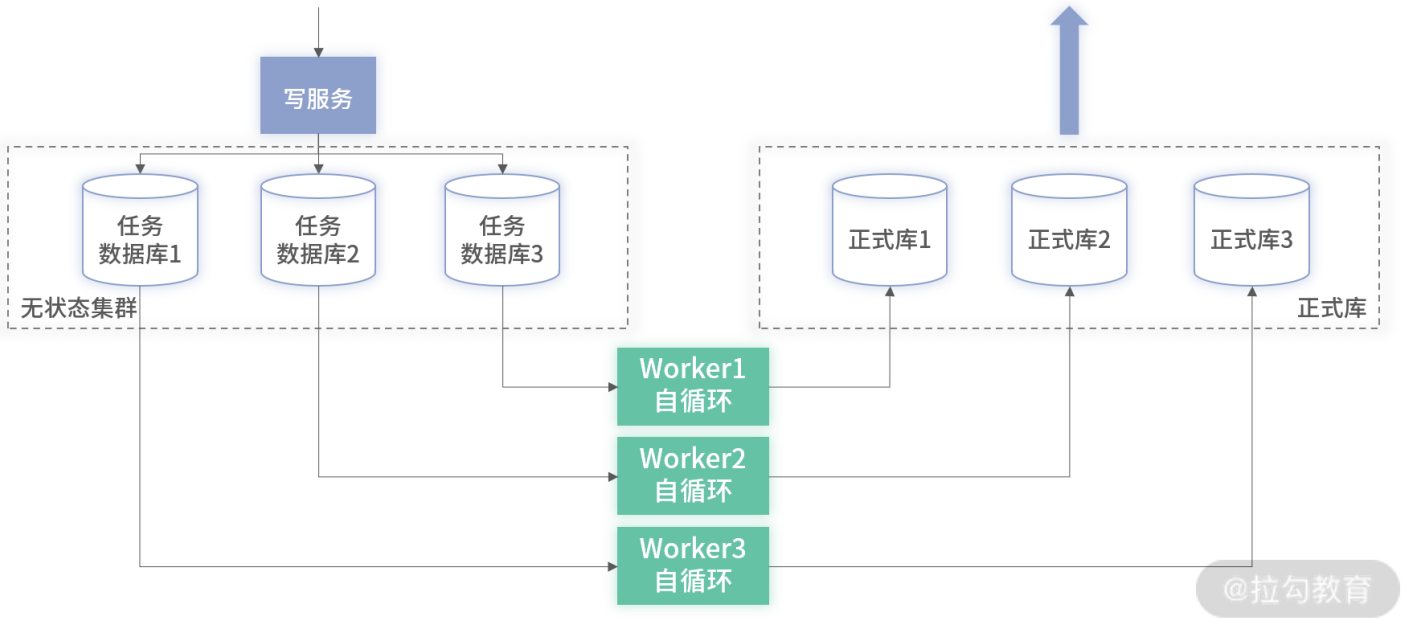

图 1：自循环的 Worker 架构

自循环的 Worker 在启动时，会开启一个不跳出的循环或者借助一些开源工具（如 Java 中比较出名的 Quartz）来保证任务不间断执行。在上述的循环内，会使用类似如下的 SQL 来批量获取未执行的任务或未同步的数据并执行同步，在任务执行成功后修改任务状态为“完成”。

```
select task_id,task_body,... from t_task where id>lastId and status='未执行' limit 一批数量 order by
```

上述的流程虽然在功能上能够满足需求，但在高可用及性能上还是有一些不足：

1. 如果任务库中任务特别多，上述单 Worker 单库的方式不具备扩展性，随着任务不断变多，会出现任务积压的瓶颈且无法通过扩容解决；
2. 单库单 Worker 的方式存在单点问题，在执行过程中，当 Worker 发生故障导致宕机，如果没有监控等机制发现故障，Worker 得不到执行，任务就会一直积压。

对于上述两个问题，这里介绍一种可以提升任务执行速度，既具备扩展性、又能保障高可用的任务架构模式，如下图 2 所示：

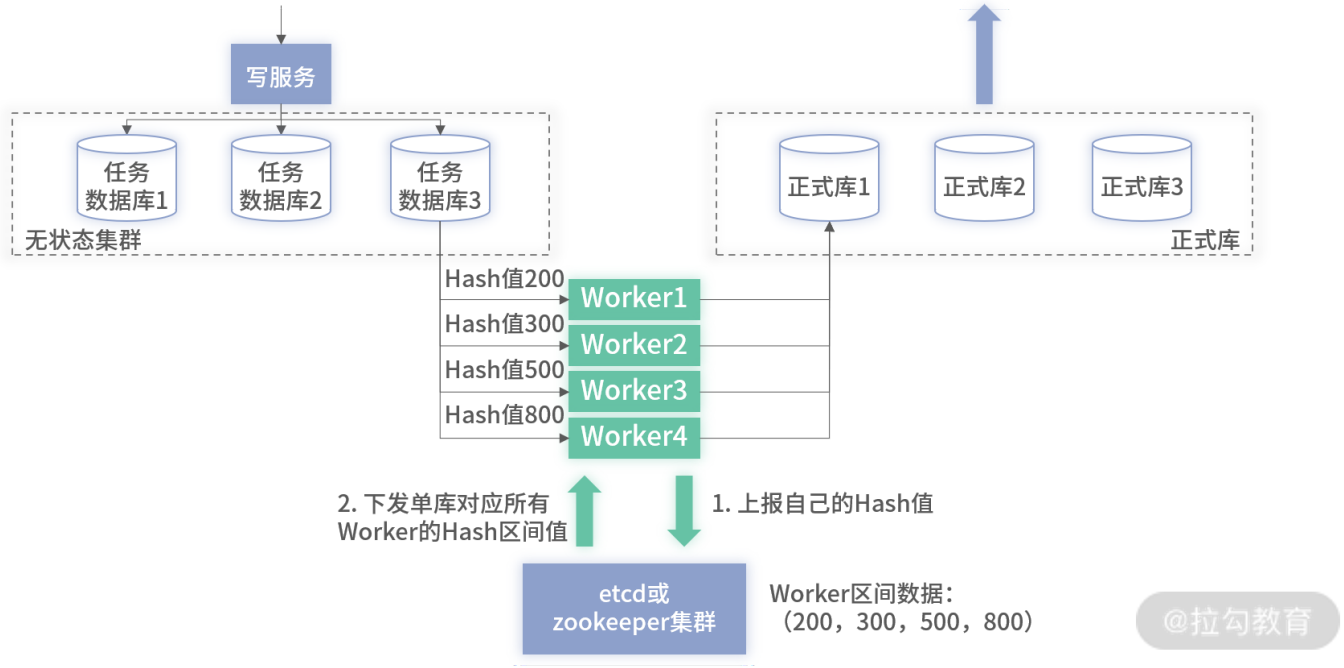

图 2：分布式无主架构图

在上述的整体架构里，每个分库对应的 Worker 的执行流程都类似，因此在讲解时，我只对一个分库的 Worker 进行分析，其余的可以以此类推。

1. 首先为了提升性能和高可用，单个分库的执行 Worker 配置的是多个并发进行执行。

2. 单个分库配置的多个任务在执行时使用自助协调，协调流程如下。

（1）每个 Worker 在启动时，会根据机器的 IP、随机数、当前时间戳等进行组合拼接计算一个唯一串，再在此基础上使用各种哈希工具计算一个**无符号整形哈希值**。

（2）所有的 Worker 会将自己的无符号整形哈希值上报到强一致的 etcd 或 ZooKeeper 存储集群里。

（3）**etcd 等集群具备通知功能（Watch）**。借助通知功能，所有的 Worker 都去订阅某一个分库下的其他 Worker 的哈希值，比如一个新的 Worker 启动了或者扩容新增了一个新的 Worker。

（4）每一个 Worker 都会获取到当前分库的所有其他 Worker 的哈希值。假设一个分库配置了四个 Worker，其中一个 Worker 会获取到自己及其他三个 Worker 的哈希值，假设为{200，300，500，800}。这四个 Worker 的 Hash 值便组成了一个环形区间，如下图 3 所示：

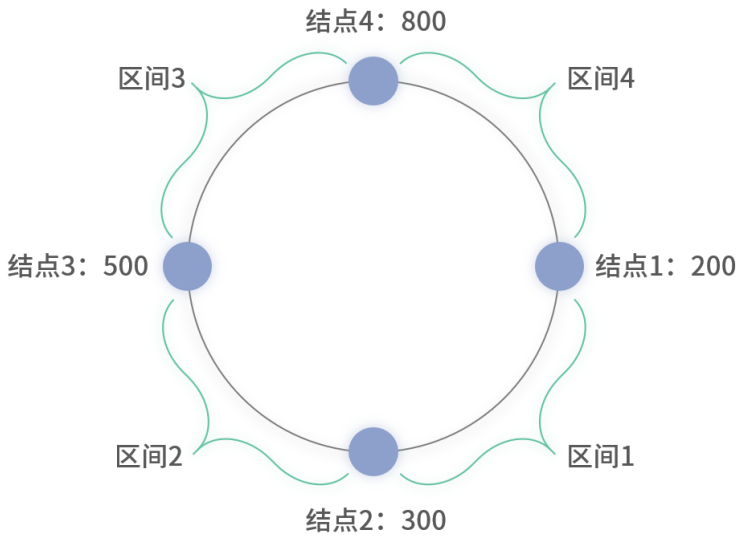

图 3：任务一致性 Hash 环

这个环形区间其实就类似一致性 Hash，每一个结点都代表一个 Worker，这个 Worker 负责任务编号在它区间范围内的任务的执行。

（5）有了上述的哈希值列表后，就可以做任务分配了。如果当前 Worker 的哈希值为 300，那么当前 Worker 就处理任务 ID 在区间[200,300)里的值。比如哈希值为 200 的 Worker 则执行区间为[800，无穷大)和[0，200)的任务（即任务编号大于等于 800 和处在[0,200)区间内的任务），其他以此类推。区间处在[200，300)的 Worker 获取任务的 SQL 大致如下：

```
select * from task where id>=200 and id<300 and status='待执行' order by id limit 100;
```

通过上述方式，无论是某一台 Worker 发生故障还是新扩容一台 Worker，通过 etcd 和 ZK 的通知机制，所有的其他 Worker 都可以立马感知，并更新自己所负责的任务区间。

比如上述介绍的案例里，四个 Worker 代表 300 的那一个发生故障，整个哈希值列表就从{200，300，500，800}变成了{200，500，800}，此时负责 500 的 Worker 就会执行[200，500)这个区间里的所有任务了，扩容 Worker 的流程和上述类似。

最后，**在 Worker 扩缩容的间隙里，可能存在临界的并发情况，即两个 Worker 可能获取到同一条任务**。对于此问题，可以从两点着手解决：

1. 首先，**任务执行需要保持幂等**，即任务可重复执行，这个可以从业务上着手实现；
2. 其次，**可以给任务增加状态**，如上述 SQL 里的 status 字段。当某一个 Worker 处理到该任务时，可以去修改该任务为处理中。其他 Worker 在获取任务时，显式指定状态，只处理为待执行的任务即可。

如何设计和实现扣减中的返还

下面将进入另外一个公共话题的讨论，如何设计和实现扣减中的返还。

什么是扣减的返还

扣减的返还指的是在扣减完成之后，业务上发生了一些逆向行为，导致原先已扣减的数据需要恢复以便供后续的扣减请求使用的场景。以在购买商品时的扣减库存举例，其中常见的逆向行为有：

1. 当客户下单之后，发现某个商品买错了（商品品类买错或是数量填错），客户便会取消订单，此时该订单对应的所有商品的库存数量都需要返还；
2. 其次，假设客户在收到订单后，发现其中某一个商品质量有问题或者商品的功能和预期有差异，便会发起订单售后流程，比如退、换货。此时该订单下被退货的商品，也需要单独进行库存返还。

返还实现原则

从上述的业务场景里可以看出，**相比扣减而言， 返还的并发量比较低**，因为下单完成后发生整单取消或者个别商品售后的情况概率较低。比如，对于热点商品或者爆品的抢购带来的扣减并发量是非常大的，但抢到爆品后再取消订单的概率是非常低的。此种场景里，扣减和返还的并发量的差距可能会达到上万倍。

因此，返还在实现上，可以参考商家对已有商品补货的实现，直接基于数据库进行落地。但返还自身也具备一些需要你注意的实现原则，可以总结为以下几点。

原则一：扣减完成才能返还

返还接口在设计时，必须要有扣减号这个字段。因为所有的返还都是依托于扣减的，如果某一个商品的返还没有带上当时的扣减号，后续你很难对当时的情况做出准确判断。

1. **当前商品是否能够返还。** 因为没有扣减号，无法找到当时的扣减明细，无法判断此商品当时是否做了扣减，没有做扣减的商品是无法进行返还的。
2. **当前返还的商品数量是否超过扣减值**。假设外部系统因为异常，传入了一个超过当时扣减值的数量，如果不通过扣减号获取当时的扣减明细，你无法判断此类异常。

原则二：一次扣减可以多次返还

假设你购买的一个订单里包含了 A、B 两件商品，且这两个商品你各买了 5 件，在产生购买订单时即对应一次扣减。后续使用过程中可能会对某件不满的商品发起售后退货申请。极端情况下，可能会发生四次退货的行为，如：第一次，先退 2 个 A；第二次，再退 3 个 B；最后一次退货，一起将剩余的 3 个 A 和 2 个 B 退回。

由上述案例可以看出，**一次扣减（即一个订单）在业务上可以对应多次返还。因此，在实现时需要考虑多次返还的场景。返还主要基于数据库实现**，下面介绍下支持多次返还的数据库表的设计。

```
create table t_return{
  id bigint not null comment '自增主健',
  occupy_uuid bigint not null comment '扣减的ID',
  return_uuid bigint not null comment '返还的唯一ID',
  unique idx_return_uuid (occupy_uuid,return_uuid) comment '返还标识唯一索引'
}comment '返还记录表';
create table t_return_detail{
  id bigint not null comment '自增主健',
  return_uuid bigint not null comment '返还标识',
  sku_id bigint not null comment '返还的商品ID',
  num bigint not null comment '返还的商品数量',
  unique idx_return_sku  (return_uuid,sku_id) comment '返还商品唯一标识'
}comment '返还商品记录表';
```

上述**返还记录表**实现了一次扣减多次返还的数据记录，**返还商品记录表**实现了一次返还里有多个商品的场景，也就是上述案例里的最后一次一起退了 A 和 B 两个商品的场景。

原则三：返还的总数量要小于等于原始扣减的数量

看到原则三，你可能觉得这内容不需要单独讲解，因为从业务上来看，这是一个必要条件。但在真正实现时，很容易出现处理遗漏。依然以“原则二”里的案例来讲解，在并发返还时，即同一时刻有两个返还请求，一个请求返还 2 个 A，另一个请求返还 4 个 A。如果技术上没有并发的时序控制，在处理两个请求时，有可能都判断为可返还并实际进行返还，最终就会出现返还 6 个 A（实际当时只扣减了 5 个）的超返还的场景。具体如下图 4 所示：

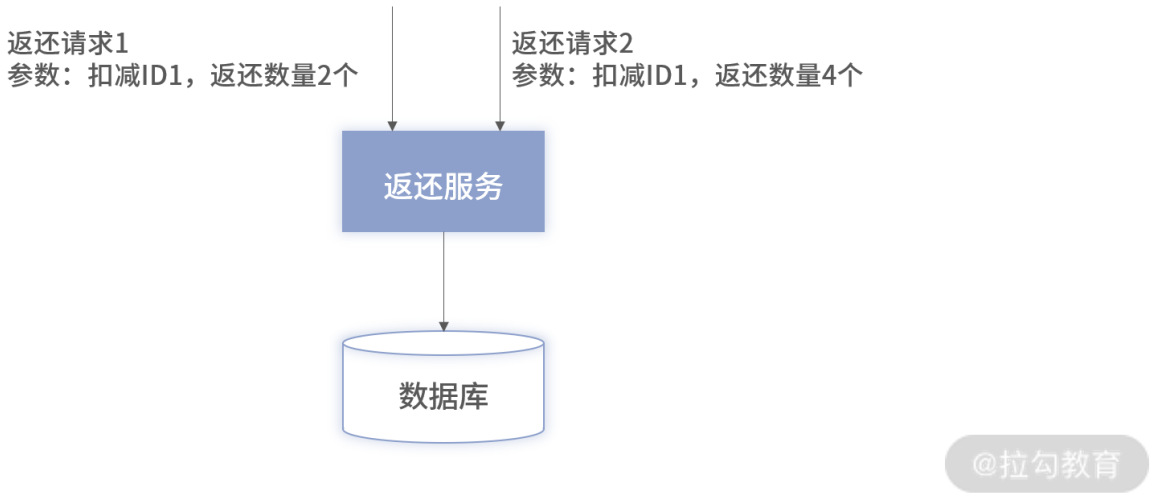

图 4：超返还的场景

对于上述潜在的风险，可以在返还前，对返还所属的扣减 ID 进行加锁来保证串行化操作，规避超卖的风险，架构如下图 5 所示：

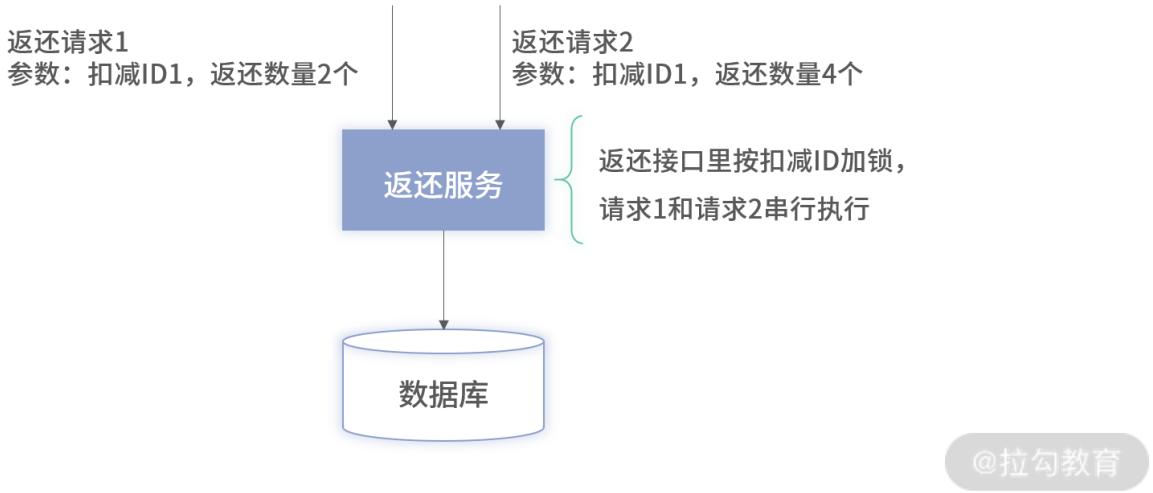

图 5：加锁串行的架构

在扣减 ID 上加锁，会导致该扣减 ID 下的所有返还都串行执行，有一定的性能损耗。但从业务上看，同一个扣减 ID 并发产生返还的场景极低且返还的调用次数也相对较少，从“架构是技术与业务场景的取舍”这个角度来看，暂不需要花费太大的人力去构建一个更加复杂的加锁架构。

原则四：返还要保证幂等

最后，设计的返还接口需要支持幂等性。比如外部系统调用返还接口超时后，因为外部系统不知道是否调用成功，就会再一次重试。如果返还接口不满足幂等性要求，且上次超时实际是执行成功的，则会导致同一个返还号产生两次数据的返还。处理这个问题最简单的做法是：**在返还接口增加返还编号（上述表结构中的 return_uuid）字段并由外部系统传入，通过数据库唯一索引来防重，进而实现幂等性**，大致的架构如下图 6 所示：

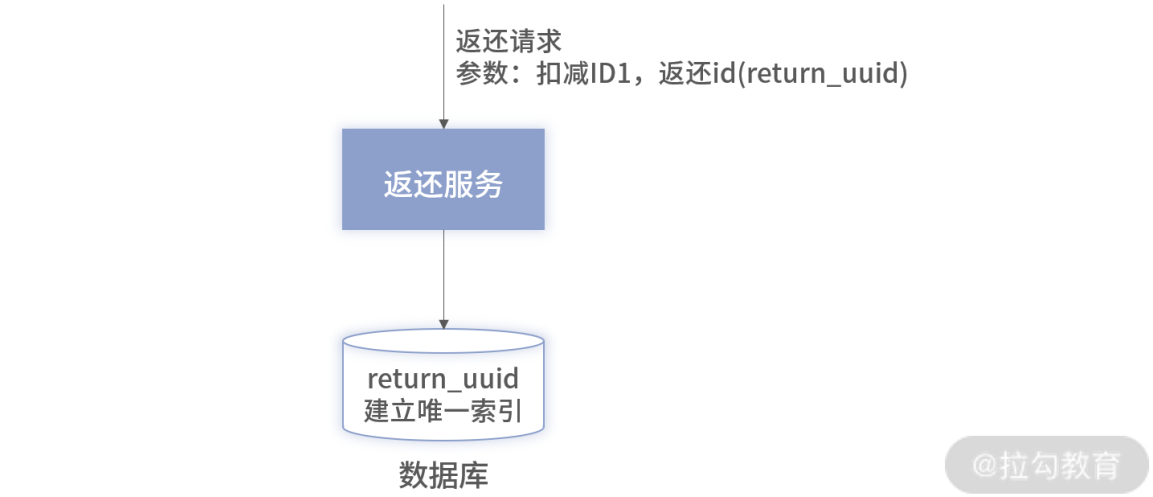

图 6：幂等的返还架构图

总结

在本讲里，讲解了几种扣减方案里都会涉及的**任务执行**和**扣减返还**这两个公共话题，不管你的业务场景采用了哪种扣减方案，你都可以参考上述的返还和任务执行方案。

最后，我再给你留两道思考题，一道题是需要你动手操作的，另一道题则是需要你深入思考的。

动手题：上述提供的分布式 Worker 扩容两台机器后，etcd 或 ZK 里的哈希列表值，以及后续任务执行的区间是如何变化的，你可以试着梳理下。

思考题，取消订单后，除了要返还商品的库存数量，还需要做哪些内容的返还呢？

这一讲就到这里，感谢你学习本次课程，接下来我们将学习 16 |秒杀场景：热点扣减如何保证命中的存储分片不挂？再见。