文章编号:1671-4598(2022)03-0234-05 DOI:10.16526/j.cnki.11—4762/tp.2022.03.039 中图分类号:TP311 文献标识码:A

基于高并发的分布式系统幂等设计

王 磊,孟利民
(浙江工业大学 信息工程学院,杭州 310023)

摘要:随着数字生活不断发展,分布式系统被广泛用来解决高并发等一系列问题;为了保证数据的准确性和一致性,分布式系统需要做幂等控制;通过分析对比各服务端幂等设计方法在高并发场景下的性能表现,提出一种改进的分布式锁设计方法;该方法主要解决由于集群服务主节点宕机可能造成分布式锁失效的问题,通过探讨RedLock算法方案及争论,提出对高并发请求二次拦截的理论,并将失效的锁通过消息队列服务进行通知,实现锁失效问题的排查以及服务监测治理;经实验测试表明,该方法实现了对失效锁的拦截,在高并发场景下有良好的性能表现,为分布式系统幂等设计提供了可行性方案。

关键词:分布式系统;高并发;数据一致性;幂等性;分布式锁

Idempotent Design of Distributed Systems Based on High Concurrency

WANG Lei, MENG Limin
(College of Information Engineering, Zhejiang University of Technology, Hangzhou 310023, China)

Abstract: With the development of digital life, the distributed systems are widely used to solve a series of problems such as high concurrency. In order to ensure the accuracy and consistency of data, the distributed systems need to design idempotent control. By analyzing and comparing the performance of each idempotent design method in high concurrency scenarios, an improved distribution lock design method is proposed. This method mainly solves the problem that the distributed lock may fail due to the downtime of the cluster service master node, by discussing the RedLock algorithm scheme and controversy, the secondary interception theory of the high concurrent requests is proposed, and the invalid lock is notified through the message queue service, the lock failure troubleshooting and service monitoring and management are realized. Experimental tests show that this method achieves the interception of invalid locks, and has good performance in high concurrency scenarios, and provides a feasible solution for the idempotent design of the distributed systems.

Keywords: distributed systems; high concurrency; data consistency; idempotence; distributed lock

0 引言

突如其来的新冠肺炎疫情一度让人们的线下生活按下“暂停键”,却也让数字生活[1]按下了“快捷键”。直播带货、外卖点餐、在线教育等行业的兴起,正在影响和改变人们的消费习惯和生活方式,蚂蚁集团CEO胡晓明在支付宝合作伙伴大会上表示,数字生活新服务将是下一个十年最大的互联网红利。数字生活依托于互联网和一系列数字科技技术应用,为了能够应对高并发场景并支撑多样化的服务,应用软件需要基于分布式思想[2]进行架构。基于分布式思想架构的系统(又称分布式系统)具有可靠性、高容错性和大吞吐量等特点,但也带来了新的问题。比如重复冗余操作会导致相同的请求分配到不同的服务实例,数据的插入与更新出现错乱,无法保证数据的一致性和准确性。特别是涉及金融支付等系统的开发时,如果不采取措施将会造成严重的后果,因而需要对分布式系统做幂等设计。

随着计算机技术的发展,幂等性设计方法也在不断演进。文献[3]探讨了基于唯一索引、悲观锁等方式保障系统的幂等性,采用数据库锁控制并发访问。文献[4]设计基于虚拟服务器的分布式PC共享平台时,采用了全局唯一ID机制,根据操作的内容生成一个全局ID,通过客户端与服务器端交互以控制页面的重复提交。文献[5]采用Redis分布式锁设计系统的幂等性,分布的服务实例通过竞争锁获取程序执行权限,利用缓存的高性能读写特性降低服务端损耗。

分布式系统在进行幂等设计时,应充分考量业务需求本身的高并发特性,以及幂等设计方法在高并发场景下的性能表现,同时需要关注服务端性能损耗问题。通过对比

收稿日期:2021-09-14; **修回日期**:2021-10-25。
基金项目:国家自然科学基金项目(61871349);浙江省自然科学基金项目(LQ19F010013)。
作者简介:王 磊(1987-),男,河南周口人,研究生,工程师,主要从事计算机软件开发与测试方向的研究。
孟利民(1963-),女,浙江杭州人,硕士生导师,教授,主要从事多媒体数字通信方向的研究。
引用格式:王 磊,孟利民.基于高并发的分布式系统幂等设计[J].计算机测量与控制,2022,30(3):234-238,243.

研究，提出一种改进的分布式锁幂等设计方法。实验结果表明，该方法在高并发场景下能够保持数据的一致性和准确性，并具备良好的性能表现。

1　幂等相关概念

随着数字生活的深入发展，计算机系统越发面临高并发和业务多样化的考验。高并发是指系统运行过程中，遇到的一种“短时间内遇到大量操作请求”的情况。高并发时，系统需要处理所有请求，执行大量的业务逻辑处理，频繁地请求资源和操作数据库，服务器开销骤增。传统的单体架构系统由于业务模块耦合、单节点部署的特点，遭遇高并发场景时，单节点服务器性能下降，当请求数量超出服务器承载能力时会导致应用崩溃，服务宕机。

为了解决以上问题，分布式思想应运而生。它的发展经历了分布式架构、面向服务架构（SOA，service oriented architecture）、微服务架构等阶段。分布式架构将各业务模块拆分成子系统，并发访问量大的子系统可以进行多服务实例集群部署，请求经过 Nginx 反向代理分发，最终到达具体的服务实例完成业务处理，分布式架构原理框图如图 1 所示。随后出现的面向服务架构 SOA，在分布式架构的基础上集成了 ESB 企业服务总线[6]，实现路由转发、服务管理监控、统一安全管理等功能。微服务架构不再强调量级比较重的 ESB 企业服务总线，将业务系统彻底的组件化和服务化[7]，服务粒度比 SOA 架构更小，保证了服务的高可用、低耦合特性。

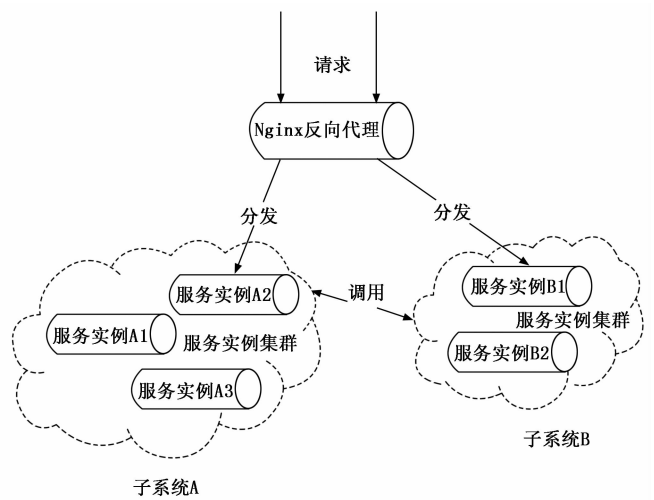

图 1　分布式架构原理框图

分布式系统为了应对高并发场景，保障数据的一致性和准确性，需要做幂等性设计。幂等概念来自数学，表示 N 次变换和 1 次变换的结果是相同的。移植到软件开发中主要指代，在 HTTP 协议中，除去请求超时、系统出现异常的情况下，系统的某些操作，对其调用一次和调用多次所产生的的效果是一样的。特别是在分布式系统中，业务繁忙的子系统需要多服务实例集群部署，请求的转发与控制情况比较复杂，幂等性设计显得尤为重要。

分布式系统中，需要进行幂等设计的场景众多，比较常见的有：

1）用户重复点击提交页面，请求被分配到多个服务实例进行处理，如果业务处理涉及数据的插入或更新，重复的操作将导致脏数据的产生。

2）分布式系统经常会设计重试机制[8]来提高请求的成功率，当请求被分配的服务实例出现异常或延时，系统触发重试，该请求被分配到其它服务实例，如果涉及数据插入或更新操作，可能导致脏数据的产生。

3）商品秒杀、抢票等业务场景中，涉及多个用户修改同一条数据记录。如果不做幂等设计，库存等需要计数的敏感字段更新将会产生错乱，影响整体业务的执行。

分布式系统幂等设计，从客户端角度出发，可以通过设置请求间隔时间防止页面的重复提交，但是间隔时间内无法保证服务端业务逻辑执行完毕，并且该方法不适用从服务端发起请求的场景，如系统接口对接。从服务端角度出发，可以通过区分业务操作类型从数据库端实现幂等设计，查询和删除是天然的幂等操作，而数据插入和更新一般是非幂等操作，执行一次和多次的效果往往不同，需要使用唯一索引、悲观锁等方法保障系统幂等性；除此之外可以通过借助第三方服务来维护分布式锁，无需区分业务操作的类型，分布的服务实例通过竞争分布式锁获得程序执行权限，进而保障业务执行的唯一性，常用的第三方服务依赖主要有 Zookeeper[9]和 Redis。

2　服务端幂等设计方法

基于服务端的分布式系统幂等设计经历了从数据库端到分布式锁发展的过程，由数据库端加锁控制事务的并发访问，到客户端与服务端交互的 token 机制，再到依赖第三方服务的分布式锁，幂等设计方法随着计算技术发展在不断演进。类比系统安全性设计，加入幂等设计后，系统需要牺牲部分性能来实现幂等，因而系统的性能损耗以及高并发场景下的性能表现，成为衡量分布式系统幂等设计方法的标准。

2.1　唯一索引

数据库索引[10]是数据库管理系统中基于排序的数据结构，以协助快速查询、更新数据库表中数据，作用类似书本的目录。唯一索引要求所在列的值必须唯一，如果是组合索引，则要求列值的组合必须唯一。唯一索引不仅能够加快数据的查询速度，而且保证了表中每一行数据的唯一性。

通过对数据库表设置唯一索引的方式，能够保障数据插入相关业务数据的准确性，因而常作为系统幂等设计的手段。唯一索引在高并发场景下，重复的数据插入将会被准确拦截，随着数据库表数据量的增长，系统响应时间会相应增加。当表中的数据增长到一定程度时，频繁的写入将导致磁盘 I/O 负载增加，需要做分表分库的操作，比较考验数据库性能开销。

需要注意的是，单一使用唯一索引时，如果涉及用户和系统交互，重复插入的数据被捕捉并以抛错的形式提示用户，可能导致用户更频繁的重试，高并发场景下会给系统带来比较大的压力。

2.2 悲观锁

数据库事务[11]是一个访问并可能操作各种数据项的数据库操作序列，悲观锁假定当前事务操作数据资源时，还会有其他事务同时操作该资源，为了避免当前事务被干扰，先将资源进行锁定。换而言之，当多个事务并发执行时，某个事务对数据加锁，其他事务只能等待该事务执行完毕，才能对当前数据进行修改。SELECT FOR UPDATE 是一个典型的悲观锁调用语句，常用于多个事务操作同一条记录，保证计数、余额扣减等字段值的准确性。如果程序执行出现异常，当前事务需要回滚，当前记录解除锁定，数据恢复至事务操作前的状态。

通过使用悲观锁，能够保障数据更新相关业务数据的准确性和一致性，满足了一定的幂等设计要求。但是悲观锁具有强烈的独占和排他特性，高度依赖数据库提供的锁机制，某个事务处理占用锁时，其它事务处于阻塞状态，因而加锁和释放锁的过程比较消耗资源，只适用于并发不高的场景。高并发场景下事务抢占资源容易造成死锁，进而导致应用系统崩溃，因而分布式系统幂等设计时，应慎重选择。

2.3 分布式锁

在分布式系统环境下，需要保证一个方法在同一时刻只能被一个服务实例的单个线程执行，来实现系统幂等。现有的做法是维护一把分布式锁[12]，存储在所有服务实例都能访问的地方，服务实例间通过高可用、高性能地获取锁和释放锁，完成并发访问控制，具体实现过程如图 2 所示。分布式锁的实现需要依赖第三方服务，常用的有 Zookeeper 和 Redis 等，这里主要分析基于缓存 Redis 实现分布式锁。

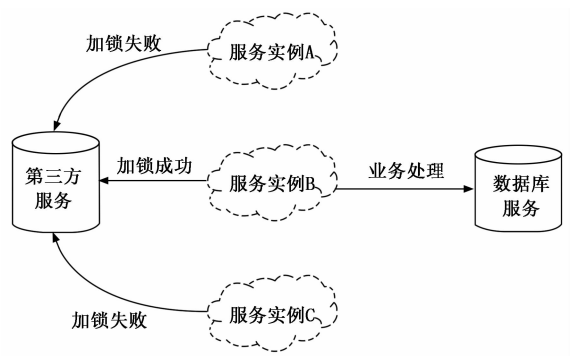

图 2 分布式锁实现过程

Redis 分布式锁主要利用了 Redis 缓存高性能读写的特性[13]，服务实例利用 setnx key value 命令进行加锁操作，如果 Redis 服务中该 key 值不存在，则设置 value 申请加锁成功，如果已存在该 key 值，表示已有服务实例持有该锁，从而加锁失败。当持有锁的服务实例方法执行完毕后，通过 del key 命令删除键值释放锁，其它服务实例可以重新竞争加锁，获取程序执行权限。为了保证操作的原子性，加锁和解锁需要使用 lua 脚本[14]执行。使用 Redis 分布式锁时，应设置合理的过期时间避免死锁问题，同时要保证分布式锁可重复可递归调用。

Redis 分布式锁相较于数据库层面的幂等设计有一定的优越性，其性能表现依赖于 Redis 服务的性能。高并发场景下，Redis 可以单机部署或者集群部署[15]，单机部署时，对服务器硬件配置要求较高，而且一旦单机服务宕机，虽然不进行数据处理但系统访问将报错，因而探讨 Redis 集群部署是必要的。Redis 集群是由一系列的主从节点（master-slave）群组成的分布式服务器群，具有复制、高可用和分片特性，服务实例访问 Redis 集群如图 3 所示。当主从节点中的主节点 master 宕机时，可以实现故障自动切换，把从节点 slave 升为主节点 master，解决了单机部署服务宕机问题。但是如果主节点 master 加锁成功，此时 master 出现异常宕机，由于主从节点切换是异步过程，加锁指令并未同步到从节点 slave 上，从节点 slave 被升为 master，该锁在新的主节点 master 上丢失了，进而出现短暂的锁失效问题，从而导致数据的插入或更新出现错乱，系统幂等无法被保障。

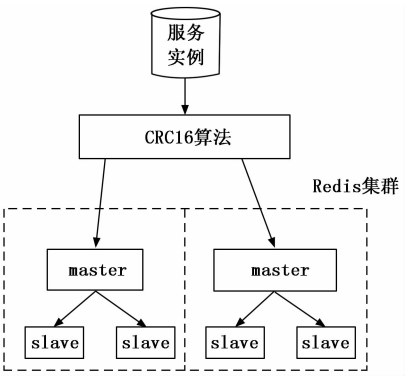

图 3 服务实例访问 Redis 集群

为了解决上述问题，Redis 作者提出了 RedLock 算法[16]方案，该方案实现需要部署 N 个独立的 Redis 实例，实例间没有主从关系，官方推荐实例数量 $N \geqslant 5$，方案模型如下：

1）服务实例先获取当前时间戳 T_1，并依次向 N 个 Redis 实例发起加锁请求，对每个加锁请求设置超时时间，如果某个实例由于锁被其它服务实例持有等原因导致加锁失败，就立即向下一个 Redis 实例申请加锁；

2）循环申请加锁完毕后，对（$N+1$）/2 进行向上取整运算得到结果 S，如果服务实例在大于等于 S 个 Redis 实例上加锁成功，再次获取当前时间戳 T_2，若 T_2-T_1 小于锁的过期时间，则认为该服务实例加锁成功，否则就认为加锁失败；

3）服务实例加锁成功后，执行业务逻辑处理，加锁失败，则向全部 Redis 实例发起释放锁请求。

RedLock 算法方案目前存在争论，质疑者认为 RedLock 通过循环 Redis 实例申请加锁，开销大效率低；同时 Redis 节点会因为机器时钟修改或跳跃导致锁到期，造成分布式服务实例间持有锁冲突，最终的结果是数据严重错误、永久性不一致或丢失，因而认为 RedLock 无法解决 Redis 集群主从节点切换导致的锁时效问题。

3　改进的分布式锁设计

针对 RedLock 存在的争论问题，提出一种改进的分布式锁设计方法，具体的设计过程是：部署 Redis 集群环境，通过分布式锁的方式对高并发请求实施第一道拦截；针对极端情况下 Redis 集群可能出现的主从节点切换导致分布式锁失效问题，通过判断业务操作类型施加唯一索引或者数据锁，实现第二道拦截；最后将失效的分布式锁通过消息队列异步发送通知消息，实现 Redis 集群服务的监测和治理。

上述改进的分布式锁设计方法，实施的第一道拦截采用 Redisson 分布式锁。Redisson[17] 是 Java 技术栈封装的用于操作 Redis 的工具，基于 Netty 框架进行事件驱动。相较于 Jedis、Lettuce 等客户端工具，Redisson 实现了分布式和可扩展的数据结构，促使使用者对 Redis 的关注分离，提供了很多分布式相关操作服务，如分布式锁、分布式集合等。Redisson 分布式锁的工作过程是：分布的服务实例通过 lock 或 tryLock 方法进行加锁操作，底层通过 exists 指令判断锁标识是否存在，若锁标识不存在，则使用 hset 指令进行加锁，再通过 pexpire 指令设置锁过期时间；若锁标识存在，则根据业务需求选择不停尝试加锁或者停止申请加锁。业务逻辑执行完毕后，使用 unlock 方法时释放锁，底层通过 del 指令删除锁标识。

Redisson 加锁和释放锁操作基于 lua 脚本实现，以确保底层 exists、hset、pexpire 一系列指令不受服务实例宕机的影响，能够执行完毕，保证操作的原子性。另外 Redisson 还提供了 watch dog 自动延期机制，后台线程每隔 10 s 检查一次，若服务实例仍持有锁标识，将不断延长锁的过期时间，防止业务逻辑未执行完毕自动释放锁的情况，保障系统的幂等性。

改进的分布式锁设计方法，针对 Redis 集群可能出现的主节点 master 宕机问题，在数据库层面进行第二道拦截。根据业务数据操作类型进行判断，如果是数据插入操作，则施加唯一索引限制数据重复插入的问题；如果是数据更新操作，则施加数据库锁，保证数据更新的正确性。由于悲观锁采用的是阻塞模式，不适用于高并发场景下数据更新操作，方法选用一种乐观锁[18]的方式进行实现。

乐观锁是相对于悲观锁而言的，它假设数据一般情况下不会产生冲突，只有在事务提交时才会对数据冲突与否进行检测。乐观锁沿用了 CAS 的思想，通过数据库表增加“版本号”Version 字段，检测事务冲突，其工作过程如图 4 所示：事务 1 读取并记录时版本号为 1，执行更新时 Version 自动加 1 并更新为版本号 2；事务 2 顺序执行，将读取的版本号 2 更新为版本号 3，此时两个事务提交不产生冲突。如果事务 1 和事务 2 同时读取的记录版本号为 1，事务 1 执行更新时 Version 自动加 1 并更新为版本号 2，事务 2 同样准备将版本号更新为 2，但此时已查询不到版本号为 1 的当前记录，发生冲突。乐观锁的优势在于不对数据进行行锁和表锁处理，减小了数据库的压力开销，对改进的分布锁设计高并发场景的性能表现是一个提升。

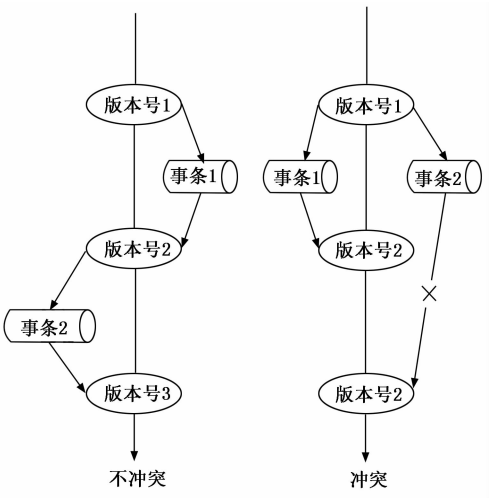

图 4　版本号实现乐观锁过程

改进的分布式锁设计通过消息队列的方式将数据库拦截的失效锁，以消息的形式通知给开发维护人员，方便进行锁失效问题的排查，如果是 Redis 集群服务主节点宕机的原因，可以快速地重启服务节点。方法选用 RabbitMq 消息服务[19]实现消息的生产和消费，通知消息以异步的形式进行处理，防止出现同步阻塞影响主要业务逻辑的处理。

改进的分布式锁设计整体工作过程如图 5 所示。

1）高并发请求经过 Nginx 负载均衡服务被分配到具体的服务实例执行。

2）服务实例收到转发的请求，程序接口根据请求内容中的字段或组合字段生成锁标识，Redisson 通过 lock 或 tryLock 方法调用 Redis 集群服务进行加锁操作。根据返回结果判断服务实例是否竞争到锁，如果加锁成功将进入业务逻辑处理环节，加锁失败可以结束当前线程或者等待其它服务实例释放锁后重新竞争加锁。

3）业务逻辑处理阶段，根据数据操作类型进行判断，若为数据插入操作则执行唯一索引逻辑，若为数据更新操作则执行乐观锁逻辑，完成对失效锁的拦截，拦截成功后结束当前线程，对当前请求返回错误提示。

4）对于数据库拦截的失效锁，通过 RabbitMq 消息生产者将相关信息放入消息队列，等待 RabbitMq 消息消费者

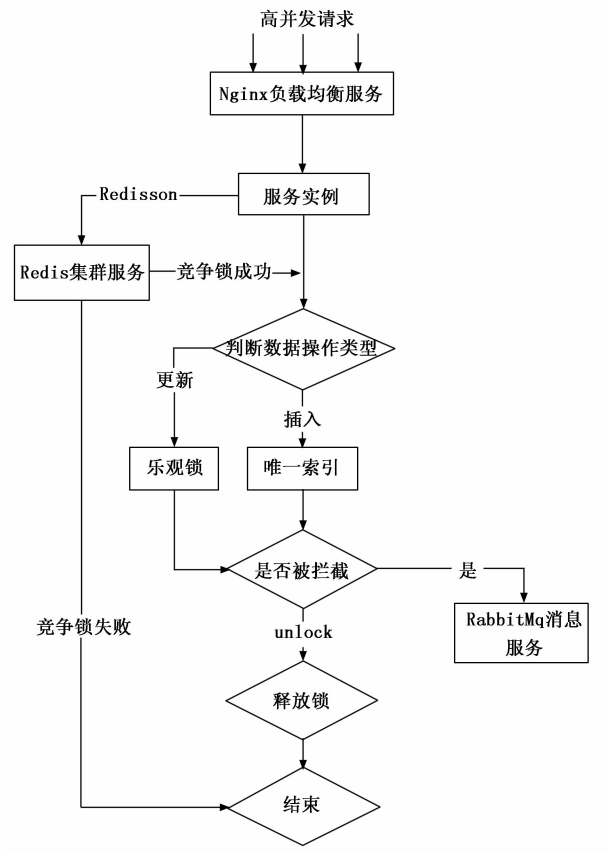

图 5 改进的分布式锁工作过程

进行异步处理。

5）对于持有 Redisson 分布式锁的服务实例，程序执行完毕后，通过调用 unlock 方法将。

Redis 集群服务中的锁标识删除，保证后续服务实例能够继续竞争使用该锁标识。

高并发场景下，网络请求首先经过 Redis 集群进行加锁，基于缓存高性能读写特性完成操作，保障了服务端性能损耗主要在访问 Redis 集群服务上，只有极端情况下主从切换出现短暂的锁失效问题时，才会触发数据库层面的拦截，避免单一使用数据库幂等设计导致重复试错带来的数据库死锁等风险。同时本设计还包含了 Redis 集群服务的监测和治理，方便开发人员能够快速的了解和掌握 Redis 服务的健康状况，有助于解决节点宕机问题和系统优化。

4 实验结果与分析

为了验证分布式系统幂等设计方法在高并发场景下的性能表现和性能损耗问题，通过实验模拟和还原高并发场景进行测试。实验需要部署 Redis 集群服务、RabbitMq 服务、多个服务实例、JMeter 测试工具[20]以及千万级别数据量的数据库表。测试方法为：通过 JMeter 设置 1 000 个并发线程数，分别测试单独使用悲观锁、乐观锁、唯一索引、Redis 分布式锁 4 种幂等设计方法的性能表现，然后测试改进的 Redisson 分布式锁在 Redis 集群主动停掉一个主节点的情况下的性能问题。悲观锁、乐观锁设置为秒杀 50 个商品库存的场景，Redis 分布式锁和改进的 Redisson 分布式锁在本实验中只针对数据插入的场景，并且测试插入的数据每隔一条设置重复数据模拟高并发请求。

实验结果如表 1 所示，其中成功次数和拦截次数反映了幂等设计方法保障数据一致性和准确性的能力，平均响应时间和吞吐量反映了高并发场景下系统性能开销和损耗。通过实验结果对比发现：乐观锁相较于悲观锁响应时间短，系统吞吐量也有提升，有良好的拦截事务冲突能力；Redis 分布式锁相比于数据库层面的幂等设计有更好的性能表现；通过对比 Redis 分布式锁和改进的 Redisson 分布式锁发现，即使在主动宕机一个 Redis 集群主节点时，改进的 Redisson 分布式锁仍能保证数据拦截的准确性，并且其平均响应时间和吞吐量指标和 Redis 分布式锁相当，同时 RabbitMq 消费者收到一条失效的锁信息，表明数据库层面的二次拦截生效。

表 1 幂等设计各方法性能参数

幂等方法	操作类型	并发数	成功次数	拦截次数	平均响应时间 ms	吞吐量/s
悲观锁	更新	1 000	50	950	1 135	302.8
乐观锁	更新	1 000	50	950	988	316.8
唯一索引	插入	1 000	500	500	1 050	295.5
Redis 分布式锁	插入	1 000	500	500	794	323.2
改进的 Redisson 分布式锁	插入	1 000	500	500	768	323.7

5 结束语

随着分布式架构思想的广泛应用，如何保证系统数据的一致性和准确性愈发受到关注。通过分析服务端幂等设计方法的原理、应用场景以及性能表现，提出一种改进的 Redisson 分布式锁设计方法，来保证分布式系统数据的一致性和准确性。该方法对 Redis 分布式锁进行了升级，针对 RedLock 存在争论的基础之上，采用二次拦截的方式，解决 Redis 集群主从节点切换造成的锁失效问题。并且通过消息队列服务实现通知，方便 Redis 集群服务的监测和治理。最后通过实验验证了改进的 Redisson 分布式锁设计的可行性。

参考文献：

[1] 陈桂龙. 数字改变生活 [J]. 中国建设信息化，2021 (13)：41－43.

[2] 叶子安. 基于分布式的高性能 Web 站点的设计与实现 [D]. 广州：华南理工大学，2018.

[3] 曾泽堂. 一种数据同步系统的设计与实现 [D]. 南京：南京大学，2019.

[4] 方徐伟. 基于虚拟服务器的分布式 PC 共享平台设计及实现 [D]. 成都：电子科技大学，2019.

（下转第 243 页）