锁、并发和事务

为了保证并发访问的正确性，Redis提供了两种方法

- [ ] 加锁是一种常用的方法，在读取数据前，客户端需要先获得锁，否则就无法进行操作。当一个客户端获得锁后，就会一直持有这把锁，直到客户端完成数据更新，才释放这把锁。

看上去好像是一种很好的方案，但是，其实这里会有两个问题：一个是，如果加锁操作多，会降低系统的并发访问性能；第二个是，Redis客户端要加锁时，需要用到分布式锁，而分布式锁实现复杂，需要用额外的存储系统来提供加解锁操作。

- [ ] 原子操作是指执行过程保持原子性的操作，而且原子操作执行时并不需要再加锁，实现了无锁操作。既能保证并发控制，还能减少对系统并发性能的影响。

一、并发访问中控制

并发访问控制，是指对多个客户端访问操作同一份数据的过程进行控制，以保证任何一个客户端发送的操作在Redis实例上执行时具有互斥性。

并发访问控制对应的操作主要是数据修改操作。当客户端需要修改数据时，基本流程分成两步：

- [ ] 客户端先把数据读取到本地，在本地进行修改；

- [ ] 客户端修改完数据后，再写回Redis。

这个流程叫做“读取-修改-写回”操作（Read-Modify-Write，简称为RMW操作）。当有多个客户端对同一份数据执行RMW操作的话，就需要让RMW操作涉及的代码以原子性方式执行。访问同一份数据的RMW操作代码，就叫做临界区代码。

为了保证数据并发修改的正确性，可以用锁把并行操作变成串行操作，串行操作就具有互斥性。一个客户端持有锁后，其他客户端只能等到锁释放，才能拿锁再进行修改。虽然加锁保证了互斥性，但是加锁也会导致系统并发性能降低。

二、Redis的两种原子操作方法

2.1 把多个操作在Redis中实现成一个操作，也就是单命令操作

Redis是使用单线程来串行处理客户端的请求操作命令的，所以，当Redis执行某个命令操作时，其他命令是无法执行的，这相当于命令操作是互斥执行的。

Redis提供了INCR/DECR命令，把这三个操作转变为一个原子操作了。INCR/DECR命令可以对数据进行增值/减值操作，而且它们本身就是单个命令操作，Redis在执行它们时，本身就具有互斥性。

2.2 把多个操作写到一个Lua脚本中，以原子性方式执行单个Lua脚本

Redis会把整个Lua脚本作为一个整体执行，在执行的过程中不会被其他命令打断，从而保证了Lua脚本中操作的原子性。如果有多个操作要执行，但是又无法用INCR/DECR这种命令操作来实现，就可以把这些要执行的操作编写到一个Lua脚本中。然后，使用Redis的EVAL命令来执行脚本。

Redis的Lua脚本可以包含多个操作，这些操作都会以原子性的方式执行，绕开了单命令操作的限制。不过，如果把很多操作都放在Lua脚本中原子执行，会导致Redis执行脚本的时间增加，同样也会降低Redis的并发性能。所以，在编写Lua脚本时，要避免把不需要做并发控制的操作写入脚本中。

三、使用Redis实现分布式锁

在分布式系统中，当有多个客户端需要获取锁时，需要分布式锁。此时，锁是保存在一个共享存储系统中的，可以被多个客户端共享访问和获取。

Redis本身可以被多个客户端共享访问，正好就是一个共享存储系统，可以用来保存分布式锁。而且Redis的读写性能高，可以应对高并发的锁操作场景

3.1 单机上的锁和分布式锁的联系与区别

对于在单机上运行的多线程程序来说，锁本身可以用一个变量表示。

• 变量值为0时，表示没有线程获取锁；

• 变量值为1时，表示已经有线程获取到锁了。

和单机上的锁类似，分布式锁同样可以用一个变量来实现。客户端加锁和释放锁的操作逻辑，也和单机上的加锁和释放锁操作逻辑一致：加锁时同样需要判断锁变量的值，根据锁变量值来判断能否加锁成功；释放锁时需要把锁变量值设置为0，表明客户端不再持有锁。

但是，和线程在单机上操作锁不同的是，在分布式场景下，锁变量需要由一个共享存储系统来维护，只有这样，多个客户端才可以通过访问共享存储系统来访问锁变量。相应的，加锁和释放锁的操作就变成了读取、判断和设置共享存储系统中的锁变量值。

实现分布式锁的两个要求：

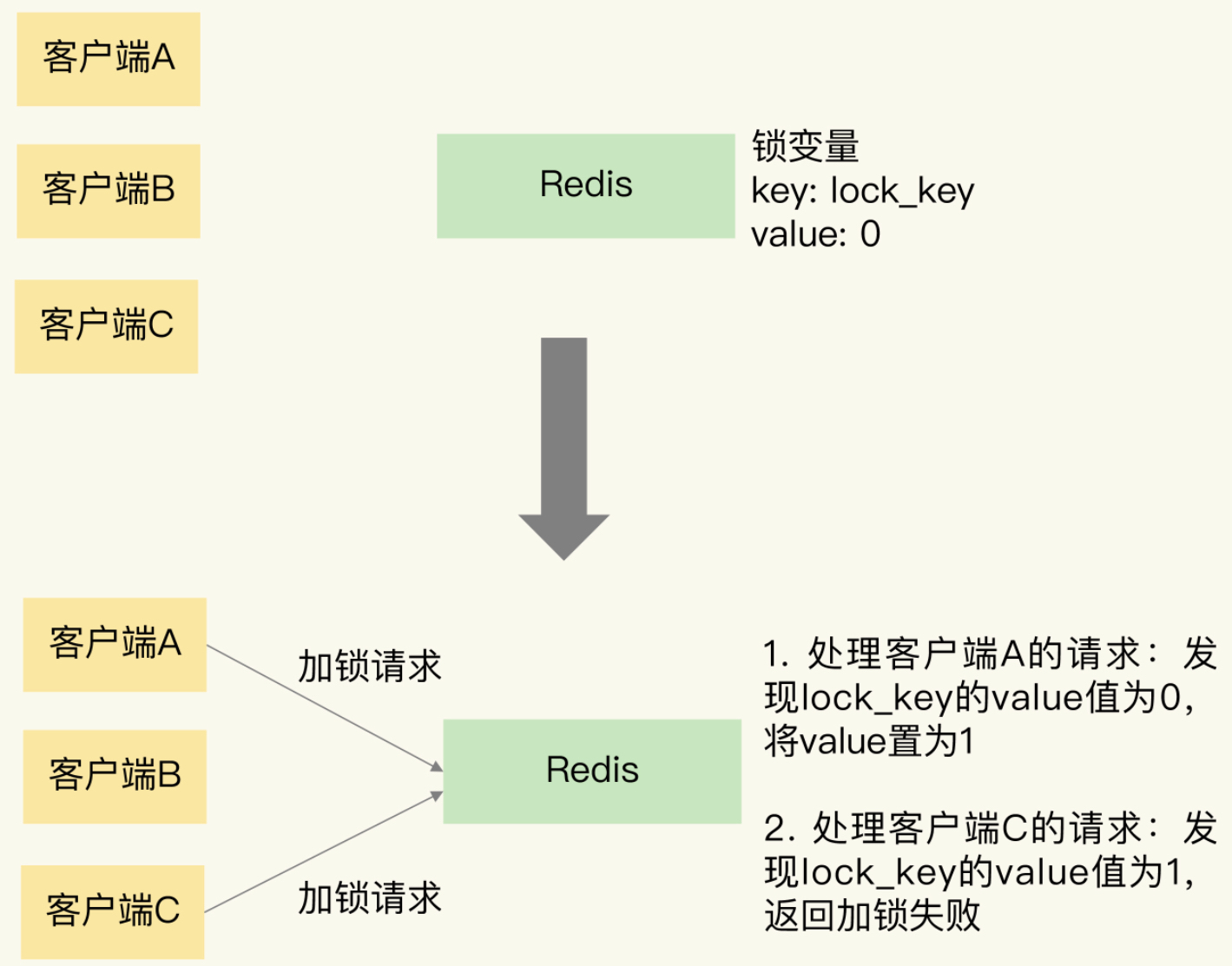
- [ ] 分布式锁的加锁和释放锁的过程，涉及多个操作。在实现分布式锁时，需要保证这些锁操作的原子性；

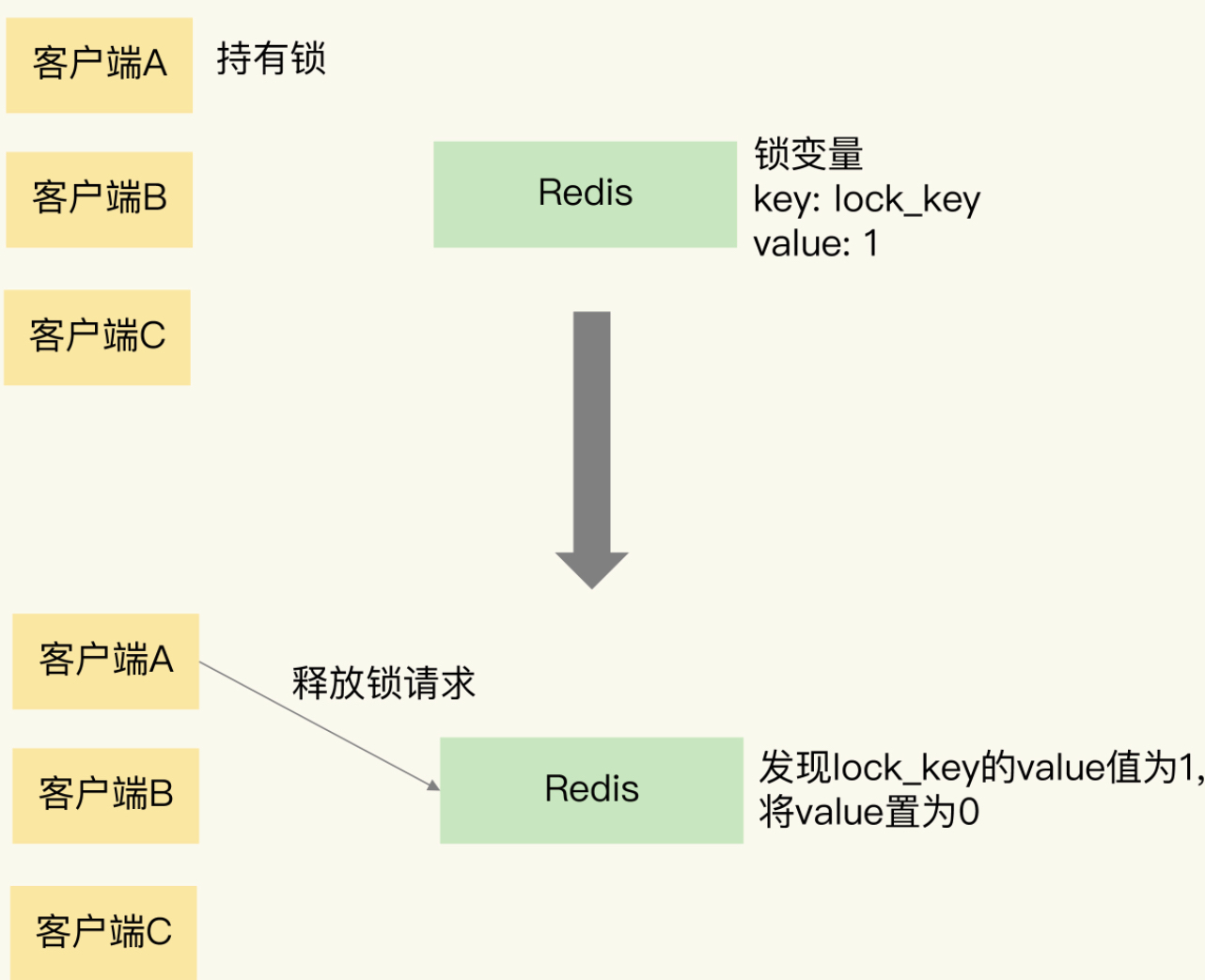
- [ ] 共享存储系统保存了锁变量，如果共享存储系统发生故障或宕机，那么客户端也就无法进行锁操作了。在实现分布式锁时，需要考虑保证共享存储系统的可靠性，进而保证锁的可靠性。

3.2 基于单个Redis节点实现分布式锁

作为分布式锁实现过程中的共享存储系统，Redis可以使用键值对来保存锁变量，再接收和处理不同客户端发送的加锁和释放锁的操作请求。

赋予锁变量一个变量名，把这个变量名作为键值对的键，而锁变量的值，则是键值对的值，这样一来，Redis就能保存锁变量了，客户端也就可以通过Redis的命令操作来实现锁操作。





因为加锁包含了三个操作（读取锁变量、判断锁变量值以及把锁变量值设置为1），而这三个操作在执行时需要保证原子性。

可以用SETNX和DEL命令组合来实现加锁和释放锁操作。

两个潜在的风险：

- [ ] 假如某个客户端在执行了SETNX命令、加锁之后，紧接着却在操作共享数据时发生了异常，结果一直没有执行最后的DEL命令释放锁。因此，锁就一直被这个客户端持有，其它客户端无法拿到锁，也无法访问共享数据和执行后续操作，这会给业务应用带来影响。解决方法是，给锁变量设置一个过期时间。

- [ ] 如果客户端A执行了SETNX命令加锁后，假设客户端B执行了DEL命令释放锁，此时，客户端A的锁就被误释放了。如果客户端C正好也在申请加锁，就可以成功获得锁，进而开始操作共享数据。这样一来，客户端A和C同时在对共享数据进行操作，数据就会被修改错误。解决方案是，区分来自不同客户端的锁操作。在加锁操作时，可以让每个客户端给锁变量设置一个唯一值，这里的唯一值就可以用来标识当前操作的客户端。在释放锁操作时，客户端需要判断，当前锁变量的值是否和自己的唯一标识相等，只有在相等的情况下，才能释放锁。

注：在释放锁操作中，使用Lua脚本。因为，释放锁操作的逻辑也包含了读取锁变量、判断值、删除锁变量的多个操作，而Redis在执行Lua脚本时，可以以原子性的方式执行，从而保证了锁释放操作的原子性。

3.3 基于多个Redis节点实现高可靠的分布式锁

为了避免Redis实例故障而导致的锁无法工作的问题，Redis的开发者Antirez提出了分布式锁算法Redlock。

Redlock算法的基本思路，是让客户端和多个独立的Redis实例依次请求加锁，如果客户端能够和半数以上的实例成功地完成加锁操作，那么就认为，客户端成功地获得分布式锁了，否则加锁失败。即使有单个Redis实例发生故障，因为锁变量在其它实例上也有保存，所以，客户端仍然可以正常地进行锁操作，锁变量并不会丢失。

Redlock算法的实现需要有N个独立的Redis实例。分成3步来完成加锁操作。

- [ ] 客户端获取当前时间。

- [ ] 客户端按顺序依次向N个Redis实例执行加锁操作。

这里的加锁操作和在单实例上执行的加锁操作一样，使用SET命令，带上NX，EX/PX选项，以及带上客户端的唯一标识。当然，如果某个Redis实例发生故障了，为了保证在这种情况下，Redlock算法能够继续运行，我们需要给加锁操作设置一个超时时间。

如果客户端在和一个Redis实例请求加锁时，一直到超时都没有成功，那么此时，客户端会和下一个Redis实例继续请求加锁。加锁操作的超时时间需要远远地小于锁的有效时间，一般也就是设置为几十毫秒。

- [ ] 一旦客户端完成了和所有Redis实例的加锁操作，客户端就要计算整个加锁过程的总耗时。

客户端只有在满足下面的这两个条件时，才能认为是加锁成功。

①客户端从超过半数（大于等于 N/2+1）的Redis实例上成功获取到了锁；

②客户端获取锁的总耗时没有超过锁的有效时间。

在满足了这两个条件后，需要重新计算这把锁的有效时间，计算的结果是锁的最初有效时间减去客户端为获取锁的总耗时。如果锁的有效时间已经来不及完成共享数据的操作了，可以释放锁，以免出现还没完成数据操作，锁就过期了的情况。

当然，如果客户端在和所有实例执行完加锁操作后，没能同时满足这两个条件，那么，客户端向所有Redis节点发起释放锁的操作。

在Redlock算法中，释放锁的操作和在单实例上释放锁的操作一样，只要执行释放锁的Lua脚本就可以了。这样一来，只要N个Redis实例中的半数以上实例能正常工作，就能保证分布式锁的正常工作了。

四、事务

事务是数据库的一个重要功能。所谓的事务，就是指对数据进行读写的一系列操作。事务在执行时，会提供专门的属性保证，包括原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability），也就是ACID属性。这些属性既包括了对事务执行结果的要求，也有对数据库在事务执行前后的数据状态变化的要求。

4.1 事务ACID属性的要求

- [ ] 原子性。一个事务中的多个操作必须都完成，或者都不完成。

- [ ] 一致性。指数据库中的数据在事务执行前后是一致的。

- [ ] 隔离性。它要求数据库在执行一个事务时，其它操作无法存取到正在执行事务访问的数据。

- [ ] 持久性。数据库执行事务后，数据的修改要被持久化保存下来。当数据库重启后，数据的值需要是被修改后的值。

4.2 Redis实现事务

Redis提供了MULTI、EXEC两个命令：

- [ ] 客户端要使用一个命令显式地表示一个事务的开启。在Redis中，这个命令就是MULTI。

- [ ] 客户端把事务中本身要执行的具体操作（例如增删改数据）发送给服务器端。这些操作就是Redis本身提供的数据读写命令，例如GET、SET等。不过，这些命令虽然被客户端发送到了服务器端，但Redis实例只是把这些命令暂存到一个命令队列中，并不会立即执行。

- [ ] 客户端向服务器端发送提交事务的命令，让数据库实际执行第二步中发送的具体操作。Redis提供的EXEC命令就是执行事务提交的。当服务器端收到EXEC命令后，才会实际执行命令队列中的所有命令。

4.3 Redis的事务机制保证的属性

4.3.1 原子性

- [ ] 在执行EXEC命令前，客户端发送的操作命令本身就有错误（比如语法错误，使用了不存在的命令），在命令入队时就被Redis实例判断出来了。

对于这种情况，在命令入队时，Redis就会报错并且记录下这个错误。此时，还能继续提交命令操作。等到执行了EXEC命令之后，Redis就会拒绝执行所有提交的命令操作，返回事务失败的结果。这样一来，事务中的所有命令都不会再被执行了，保证了原子性。

- [ ] 事务操作入队时，命令和操作的数据类型不匹配，但Redis实例没有检查出错误。

在执行完EXEC命令以后，Redis实际执行这些事务操作时，就会报错。不过，需要注意的是，虽然Redis会对错误命令报错，但还是会把正确的命令执行完。在这种情况下，事务的原子性就无法得到保证了。

Redis中并没有提供回滚机制。虽然Redis提供了DISCARD命令，但是，这个命令只能用来主动放弃事务执行，把暂存的命令队列清空，起不到回滚的效果。

- [ ] 在执行事务的EXEC命令时，Redis实例发生了故障，导致事务执行失败。

在这种情况下，如果Redis开启了AOF日志，那么，只会有部分的事务操作被记录到AOF日志中。需要使用redis-check-aof工具检查AOF日志文件，这个工具可以把未完成的事务操作从AOF文件中去除。这样一来，使用AOF恢复实例后，事务操作不会再被执行，从而保证了原子性。

当然，如果AOF日志并没有开启，那么实例重启后，数据也都没法恢复了，此时，也就谈不上原子性了。

小结

命令入队时就报错，会放弃事务执行，保证原子性；

命令入队时没报错，实际执行时报错，不保证原子性；

EXEC命令执行时实例故障，如果开启了AOF日志，可以保证原子性。

4.3.2 一致性

事务的一致性保证会受到错误命令、实例故障的影响。所以，按照命令出错和实例故障的发生时机，分成三种情况来看。

- [ ] 命令入队时就报错

在这种情况下，事务本身就会被放弃执行，所以可以保证数据库的一致性。

- [ ] 命令入队时没报错，实际执行时报错

在这种情况下，有错误的命令不会被执行，正确的命令可以正常执行，也不会改变数据库的一致性。

- [ ] EXEC命令执行时实例发生故障

如果我们没有开启RDB或AOF，那么，实例故障重启后，数据都没有了，数据库是一致的。

如果使用了RDB快照，因为RDB快照不会在事务执行时执行，所以，事务命令操作的结果不会被保存到RDB快照中，使用RDB快照进行恢复时，数据库里的数据也是一致的。

如果使用了AOF日志，而事务操作还没有被记录到AOF日志时，实例就发生了故障，那么，使用AOF日志恢复的数据库数据是一致的。如果只有部分操作被记录到了AOF日志，我们可以使用redis-check-aof清除事务中已经完成的操作，数据库恢复后也是一致的。

4.3.3 隔离性

事务的隔离性保证，会受到和事务一起执行的并发操作的影响。而事务执行又可以分成命令入队（EXEC命令执行前）和命令实际执行（EXEC命令执行后）两个阶段：

- [ ] 并发操作在EXEC命令前执行，此时，隔离性的保证要使用WATCH机制来实现，否则隔离性无法保证；

WATCH机制的作用是，在事务执行前，监控一个或多个键的值变化情况，当事务调用EXEC命令执行时，WATCH机制会先检查监控的键是否被其它客户端修改了。如果修改了，就放弃事务执行，避免事务的隔离性被破坏。然后，客户端可以再次执行事务，此时，如果没有并发修改事务数据的操作了，事务就能正常执行，隔离性也得到了保证。

- [ ] 并发操作在EXEC命令后执行，此时，隔离性可以保证。

4.3.4 持久性

- [ ] 如果Redis没有使用RDB或AOF，那么事务的持久化属性肯定得不到保证。

- [ ] 如果Redis使用了RDB模式，那么，在一个事务执行后，而下一次的RDB快照还未执行前，如果发生了实例宕机，这种情况下，事务修改的数据也是不能保证持久化的。

- [ ] 如果Redis采用了AOF模式，因为AOF模式的三种配置选项no、everysec和always都会存在数据丢失的情况，所以，事务的持久性属性也还是得不到保证。

