Redis分布式锁

我们知道`watch`，监测一个事务运行过程中有没有被其他的客户端或者其他的并发操作影响它监测的数据。其本质是乐观锁。那么，什么是乐观锁？是不是还有悲观锁？其实，我们在并发的概念中，对锁宏观的分类就是：乐观锁和悲观锁。

乐观锁与悲观锁

```
悲观锁（Pessimistic Lock），就是很悲观，每次去拿数据的时候,都认为别人会修改。所以每次在拿数据的时候,都会上锁。这样别人想拿数据就被挡住，直到悲观锁被释放。 —— 和synchronized是一样的。

乐观锁（Optimistic Lock），就是很乐观，每次去拿数据的时候,都认为别人不会修改。所以每次在拿数据的时候,不会上锁！但是如果想要更新数据，则会在更新前检查，在读取至更新这段时间，别人有没有修改过这个数据。如果修改过，则重新读取，再次尝试更新，循环上述步骤直到更新成功。 —— 和CAS的思想是一样的。

理解：乐观锁-夜不闭户（生活那么好了，晚上不会有人偷东西的）。悲观锁-晚上睡觉一定要把门锁上（不锁就有人来偷东西呢）。
乐观锁，虽然不锁门，但是每天早晨起床还是要看一看，东西有没有少（总是要检测一下有没有被修改，本质并没有上锁。这样的话，我们再去理解watch，其实没有上锁，只是监控，我们需要监控的键值的状态，如果被修改了，我们就不做什么事情，如果没有被修改，我们才拿过来这个键值做事情）。

悲观锁VS乐观锁
   1）悲观锁阻塞事务，乐观锁回滚重试。
   2）乐观锁适用于写比较少的情况下，即冲突很少发生时，可以省去锁的开销，加大了系统吞吐量。
   3）悲观锁适用于写比较多的情况下，因为如果乐观锁经常冲突，应用要不断进行重试，反倒降低性能。
```

乐观锁的实现方式CAS算法

```
Compare-and-Swap，即比较并替换，也有叫做Compare-and-Set的，比较并设置。
1、比较：读取到了一个值A，在将其更新为B之前，检查原值是否仍为A（未被其他线程改动）。
2、设置：如果是，将A更新为B，结束。 如果不是，则什么都不做。

允许多个线程同时读取（因为根本没有加锁操作），但是只有一个线程可以成功更新数据，并导致其他要更新数据的线程回滚重试。 也叫非阻塞同步（Non-block ing Synchronization）。

乐观锁策略也被称为无锁编程。换句话说，乐观锁其实不是“锁”，它仅仅是一个循环重试CAS的算法而已！

乐观锁的缺点——ABA 问题。如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 "ABA"问题。 —— 解决方案：AtomicReference 、 LongAdder。（详见：https://github.com/zhoudbw/source-analysis/tree/master/01-CAS%E5%8E%9F%E7%90%86%E5%AE%9E%E6%88%98）
```

分布式锁

分布式锁，本质上是悲观锁。是为了解决分布式领域的问题的。

在很多场景中，我们为了保证数据的最终一致性，需要很多的技术方案来支持，比如分布式事务、分布式锁等。
有时，我们需要保证一个方法在同一时间内只能被同一个线程执行。
在单机环境中，Java中其实提供了很多并发处理相关的API，但是这些API在分布式场景中就无能为力了。
也就是说单纯的Java Api并不能提供分布式锁的能力。

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。
  * 就相当于一个屋子里只有一个洗手间，大家都想使用的时候，使用的时候，势必要锁上，这样别人才进不来，我们才可以安心使用。
  * 分布式集群，实际上就是有多个分布式服务，对应就是有多个人，每个人有每个人自己的代码，怎么能控制这些代码对于同一个资源的使用时唯一的呢？那么，就是添加分布式锁，所以分布式锁的可靠性需满足以下四个条件：
        1. 互斥性。在任意时刻，只有一个客户端能持有锁。
            & 只有一个人能用洗手间。
        2. 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
            & 如果某个服务在持有锁的期间，发生了死锁，那么要能够解锁。也就是进去洗手间了，但是遇到一些不可抗因素，自己没法打开门出来了。这个洗手间的维护人员（保洁人员），就需要有钥匙能够把这个门给打开。
        3. 具有容错性。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。
            & 锁要有保障，不能用一会就坏了。
        4. 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

针对分布式锁的实现目前有多种方案。
分布式锁一般有三种实现方式：
1. 数据库乐观锁；
  * 当我要增加一条数据的时候，数据库其实已经帮我们有了一步校验的操作了。当前我们的主键出现重复，或者不可重的键出现了重复，在向数据库中插入就不成功。所以不论多少个服务去插入数据，只要这条数据的主键或者不可重复键已经存在了，那么都会插入失败。这其实就是一种乐观锁的方式，不管当前有没有别人在使用，线程自己就直接去使用，只不过用不了的时候，直接返回失败而已。
2. 基于Redis的分布式锁；
  * Setnx+Lua
3. 基于ZooKeeper的分布式锁。

Setnx+Lua

使用Redis实现分布式锁原理：
Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对Redis的连接并不存在竞争关系， 基于此，Redis中可以使用SETNX命令实现分布式锁。 SETNX——SET if Not eXist s（如果不存在，则设置） 若给定的 key 已经存在，则 SETNX 不做任何动作；如果需要解锁，使用 key 命令就能释放锁

lua脚本示例

```
* 在使用分布式锁的时候，往往要结合lua脚本来使用。
* 因为lua脚本可以保证多个连续子命令的原子性执行的。
* 也就是能够让，多个命令顺序执行，而不会被其他客户端服务所影响。
* 这是lua的强大之处。
* 但是我们只是学习lua自身的一个分布式锁。

- 获取锁（unique_value可以是UUID等）
SET resource_name unique_value NX PX 30000

- 释放锁（lua脚本中，一定要比较value，防止误解锁）
if redis.call("get", KEYS[1]) == ARGV[1] then
  return redis.call("del", KEYS[1])
else
  return 0
end
```

示例命令演示

```
127.0.0.1:6379> setnx k1 v1    由于现在的keys是empty，所以通过setnx可以成功设置。
(integer) 1
127.0.0.1:6379> get k1
"v1"
127.0.0.1:6379> expire k1 10000    当我们设置了一个key的时候，这个key是没有过期时间的。现在为k1设置过过期时间10000秒。
(integer) 1
127.0.0.1:6379> del k1     之后删除这个key
(integer) 1
这一系列的操作是非常顺畅的，这是因为我们现在在单一客户端下执行这些命令。
如果在setnx k1 v1 之后，我的客户端宕机了，那么这个k1没有过期时间，就会一直存在。
expire命令没法执行的过程，我们称之为死锁。
这个问题的根源是什么呢？因为setnx 和 expire是两条命令，而不是原子的，我们不能一起执行。从而导致中间会有被截断的时候。这样就会出现expire执行不了的情况。

* 解决方式，将setnx和expire组合成一条命令。
set key value [expiration EX seconds|PX milliseconds][NX|XX]

127.0.0.1:6379> set k2 v2 ex 9000 nx
OK  执行成功。
127.0.0.1:6379> set k2 v2 ex 9000 nx
(nil) nx判断该key值已经存在，设置失败。
127.0.0.1:6379> get k2
"v2"
127.0.0.1:6379> ttl k2
(integer) 8923

* 这就是分布式锁的一种使用方式。
```