07 | 分布式锁：关键重地，非请勿入

2019-10-07 聂鹏程

分布式技术原理与算法解析 进入课程 >

讲述：聂鹏程
时长 19:01 大小 17.42M

你好，我是聂鹏程。今天，我来继续带你打卡分布式核心技术。

我在第 3 篇文章中，与你一起学习了分布式互斥，领悟了其“有你没我，有我没你”的精髓，为你解释了同一临界资源同一时刻只能被一个程序访问的问题，并介绍了解决分布式互斥的算法。

不知道你有没有发现一个细节，在之前介绍的算法中，我主要讲了如何协调多个进程获取权限和根据权限有序访问共享资源，“获得访问权限的进程可以访问共享资源，其他进程必须等待拥有该权限的进程释放权限”。但是，我并没有介绍在访问共享资源时，这个权限是如何设置或产生的，以及设置或产生这个权限的工作原理是什么。

那么，在本讲，我就将带你一起打卡分布式锁，去学习分布式锁是如何解决这个问题的。

为什么要使用分布锁?

首先，我先带你认识一下什么是锁。

在单机多线程环境中，我们经常遇到多个线程访问同一个共享资源（这里需要注意的是：在很多地方，这种资源会称为临界资源，但在今天这篇文章中，我们统一称之为共享资源）的情况。为了维护数据的一致性，我们需要某种机制来保证只有满足某个条件的线程才能访问资源，不满足条件的线程只能等待，在下一轮竞争中重新满足条件时才能访问资源。

这个机制指的是，为了实现分布式互斥，在某个地方做个**标记**，这个标记每个线程都能看到，到标记不存在时可以设置该标记，当标记被设置后，其他线程只能等待拥有该标记的线程执行完成，并释放该标记后，才能去设置该标记和访问共享资源。这里的标记，就是我们常说的**锁**。

也就是说，**锁是实现多线程同时访问同一共享资源，保证同一时刻只有一个线程可访问共享资源所做的一种标记。**

与普通锁不同的是，**分布式锁**是指分布式环境下，系统部署在多个机器中，实现多进程分布式互斥的一种锁。为了保证多个进程能看到锁，锁被存在公共存储（比如 Redis、Memcache、数据库等三方存储中），以实现多个进程并发访问同一个临界资源，同一时刻只有一个进程可访问共享资源，确保数据的一致性。

那什么场景下需要使用分布式锁呢?

比如，现在某电商要售卖某大牌吹风机（以下简称“吹风机”），库存只有 2 个，但有 5 个来自不同地区的用户{A,B,C,D,E}几乎同时下单，那么这 2 个吹风机到底会花落谁家呢?

你可能会想，这还不简单，谁先提交订单请求，谁就购买成功呗。但实际业务中，为了高并发地接受大量用户订单请求，很少有电商网站真正实施这么简单的措施。

此外，对于订单的优先级，不同电商往往采取不同的策略，比如有些电商根据下单时间判断谁可以购买成功，而有些电商则是根据付款时间来判断。但，无论采用什么样的规则去判断谁能购买成功，都必须要保证吹风机售出时，数据库中更新的库存是正确的。为了便于理解，我在下面的讲述中，以下单时间作为购买成功的判断依据。

我们能想到的最简单方案就是，给吹风机的库存数加一个锁。当有一个用户提交订单后，后台服务器给库存数加一个锁，根据该用户的订单修改库存。而其他用户必须等到锁释放以后，才能重新获取库存数，继续购买。

在这里，吹风机的库存就是共享资源，不同的购买者对应着多个进程，后台服务器对共享资源加的锁就是告诉其他进程“**关键重地，非请勿入**”。

但问题就这样解决了吗？当然没这么简单。

想象一下，用户 A 想买 1 个吹风机，用户 B 想买 2 个吹风机。在理想状态下，用户 A 网速好先买走了 1 个，库存还剩下 1 个，此时应该提示用户 B 库存不足，用户 B 购买失败。但实际情况是，用户 A 和用户 B 同时获取到商品库存还剩 2 个，用户 A 买走 1 个，在用户 A 更新库存之前，用户 B 又买走了 2 个，此时用户 B 更新库存，商品还剩 0 个。这时，电商就头大了，总共 2 个吹风机，却卖出去了 3 个。

不难看出，如果只使用单机锁将会出现不可预知的后果。因此，在高并发场景下，为了保证临界资源同一时间只能被一个进程使用，从而确保数据的一致性，我们就需要引入分布式锁了。

此外，在大规模分布式系统中，单个机器的线程锁无法管控多个机器对同一资源的访问，这时使用分布式锁，就可以把整个集群当作一个应用一样去处理，实用性和扩展性更好。

分布式锁的三种实现方法及对比

接下来，我带你看看实现分布式锁的 3 种主流方法，即：

- 基于数据库实现分布式锁，这里的数据库指的是关系型数据库；
- 基于缓存实现分布式锁；
- 基于 ZooKeeper 实现分布式锁。

基于数据库实现分布式锁

要实现分布式锁，最简单的方式就是创建一张锁表，然后通过操作该表中的数据来实现。

当我们要锁住某个资源时，就在该表中增加一条记录，想要释放锁的时候就删除这条记录。数据库对共享资源做了唯一性约束，如果有多个请求被同时提交到数据库的话，数据库会保证只有一个操作可以成功，操作成功的那个线程就获得了访问共享资源的锁，可以进行操作。

基于数据库实现的分布式锁，是最容易理解的。但是，因为数据库需要落到硬盘上，频繁读取数据库会导致 IO 开销大，因此这种分布式锁**适用于并发量低，对性能要求低的场景**。对于双 11、双 12 等需求量激增的场景，数据库锁是无法满足其性能要求的。而在平日的购物中，我们可以在局部场景中使用数据库锁实现对资源的互斥访问。

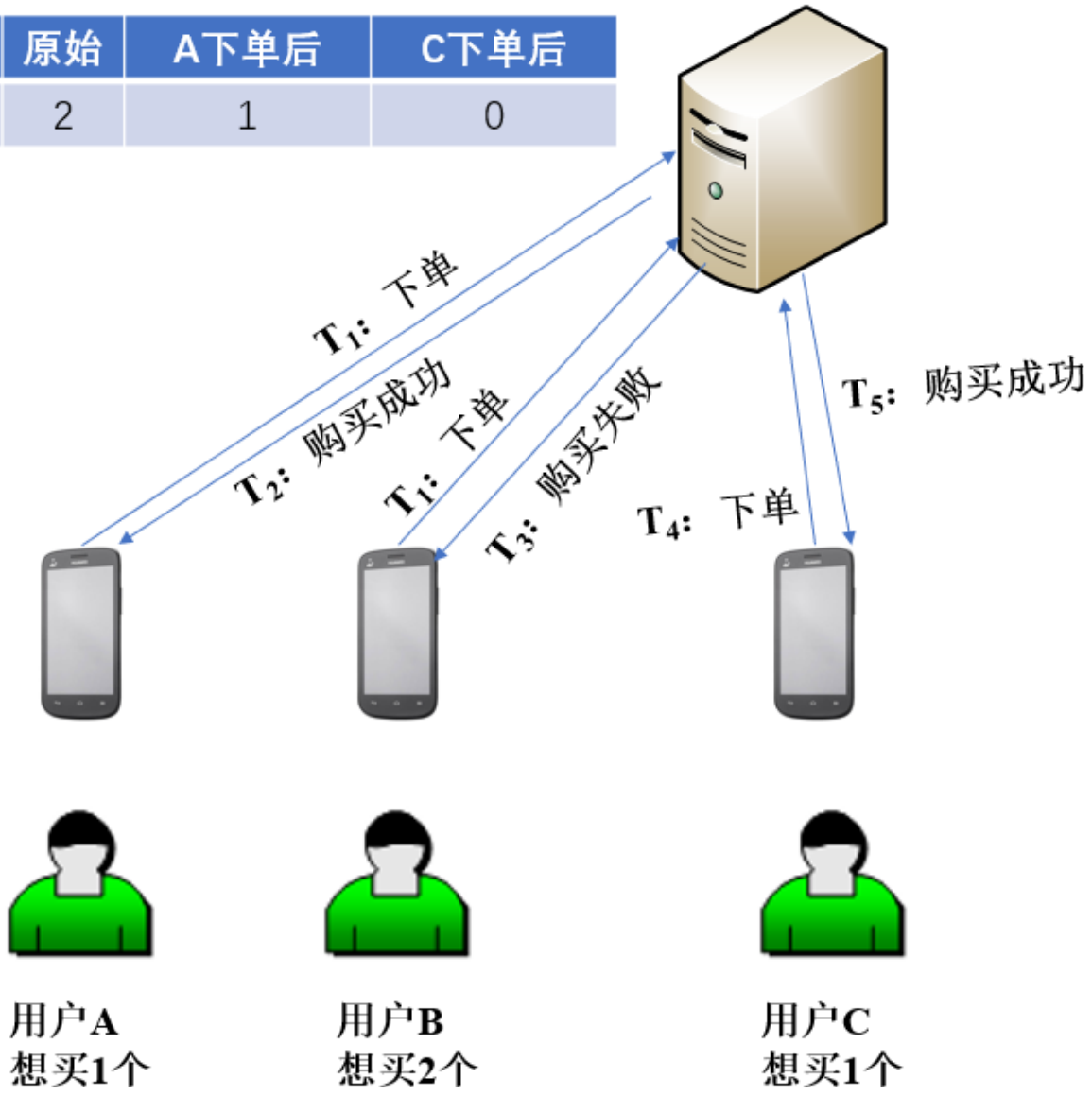
下面，我们还是以电商售卖吹风机的场景为例。吹风机库存是 2 个，有 5 个来自不同地区的用户{A,B,C,D,E}想要购买，其中用户 A 想买 1 个，用户 B 想买 2 个，用户 C 想买 1 个。

用户 A 和用户 B 几乎同时下单，但用户 A 的下单请求最先到达服务器。因此，该商家的产品数据库中增加了一条关于用户 A 的记录，用户 A 获得了锁，他的订单请求被处理，服务器修改吹风机库存数，减去 1 后还剩下 1 个。

当用户 A 的订单请求处理完成后，有关用户 A 的记录被删除，服务器开始处理用户 B 的订单请求。这时，库存只有 1 个了，无法满足用户 B 的订单需求，因此用户 B 购买失败。

从数据库中，删除用户 B 的记录，服务器开始处理用户 C 的订单请求，库存中 1 个吹风机满足用户 C 的订单需求。所以，数据库中增加了一条关于用户 C 的记录，用户 C 获得了锁，他的订单请求被处理，服务器修改吹风机数量，减去 1 后还剩下 0 个。

吹风机	原始	A下单后	C下单后
库存	2	1	0



可以看出，**基于数据库实现分布式锁比较简单，绝招在于创建一张锁表，为申请者在锁表里建立一条记录，记录建立成功则获得锁，消除记录则释放锁。**该方法依赖于数据库，主要有两个缺点：

单点故障问题。一旦数据库不可用，会导致整个系统崩溃。

死锁问题。数据库锁没有失效时间，未获得锁的进程只能一直等待已获得锁的进程主动释放锁。一旦已获得锁的进程挂掉或者解锁操作失败，会导致锁记录一直存在数据库中，其他进程无法获得锁。

基于缓存实现分布式锁

数据库的性能限制了业务的并发量，那么对于双 11、双 12 等需求量激增的场景是否有解决方法呢?

基于缓存实现分布式锁的方式，非常适合解决这种场景下的问题。**所谓基于缓存，也就是说把数据存放在计算机内存中，不需要写入磁盘，减少了 IO 读写。**接下来，我以 Redis 为例与你展开这部分内容。

Redis 通常可以使用 setnx(key, value) 函数来实现分布式锁。key 和 value 就是基于缓存的分布式锁的两个属性，其中 key 表示锁 id，value = currentTime + timeOut，表示当前时间 + 超时时间。也就是说，某个进程获得 key 这把锁后，如果在 value 的时间内未释放锁，系统就会主动释放锁。

setnx 函数的返回值有 0 和 1：

返回 1，说明该服务器获得锁，setnx 将 key 对应的 value 设置为当前时间 + 锁的有效时间。

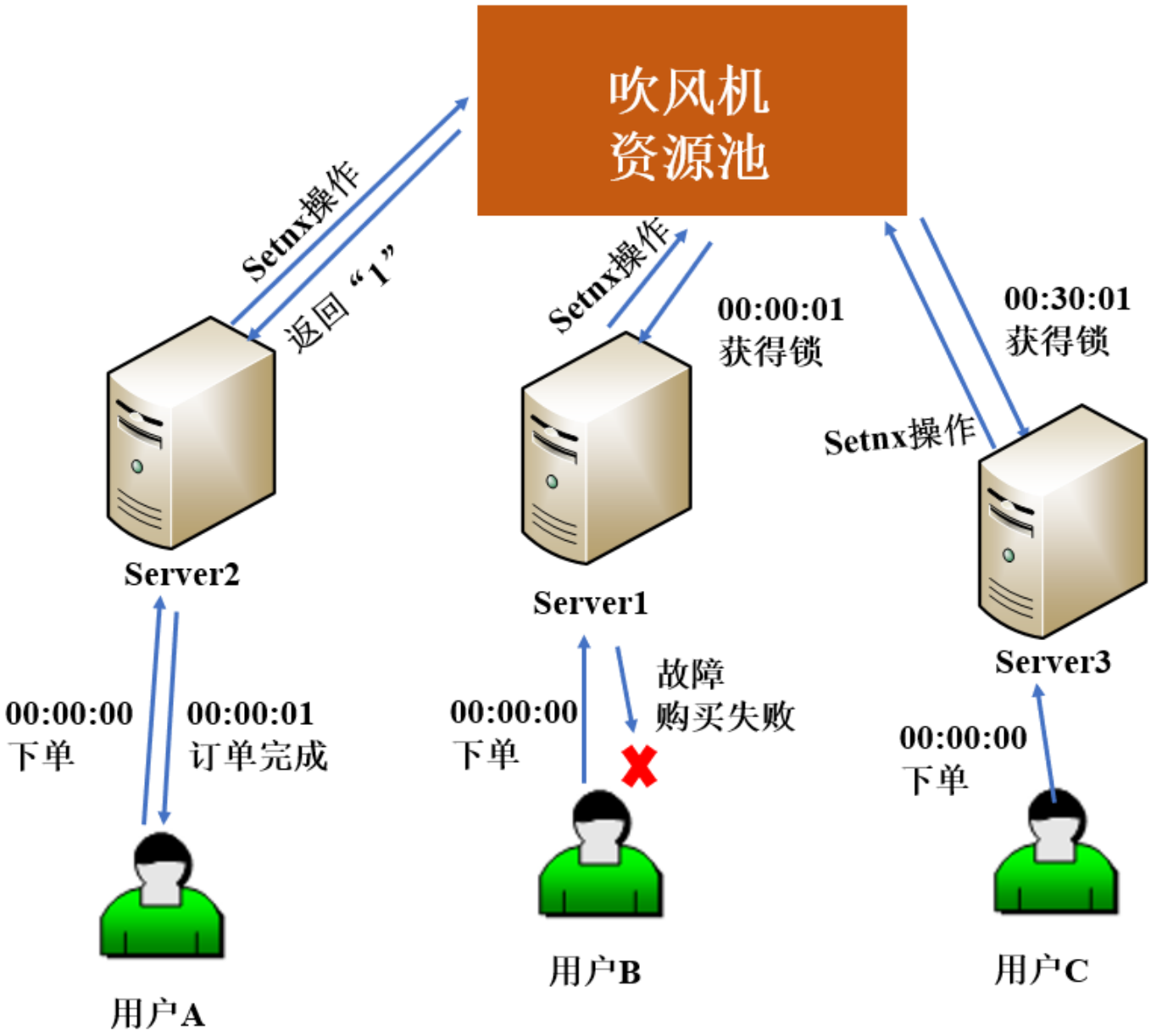
返回 0，说明其他服务器已经获得了锁，进程不能进入临界区。该服务器可以不断尝试 setnx 操作，以获得锁。

我还是以电商售卖吹风机的场景为例，和你说明基于缓存实现的分布式锁，假设现在库存数量是足够的。

用户 A 的请求因为网速快，最先到达 Server2，setnx 操作返回 1，并获取到购买吹风机的锁；用户 B 和用户 C 的请求，几乎同时到达了 Server1 和 Server3，但因为这时 Server2 获取到了吹风机数据的锁，所以只能加入等待队列。

Server2 获取到锁后，负责管理吹风机的服务器执行业务逻辑，只用了 1s 就完成了订单。订单请求完成后，删除锁的 key，从而释放锁。此时，排在第二顺位的 Server1 获得了锁，可以访问吹风机的数据资源。但不巧的是，Server1 在完成订单后发生了故障，无法主动释放锁。

于是，排在第三顺位的 Server3 只能等设定的有效时间（比如 30 分钟）到期，锁自动释放后，才能访问吹风机的数据资源，也就是说用户 C 只能到 00:30:01 以后才能继续抢购。



总结来说，**Redis 通过队列来维持进程访问共享资源的先后顺序。**Redis 锁主要基于 setnx 函数实现分布式锁，当进程通过 setnx<key,value> 函数返回 1 时，表示已经获得锁。排在后面的进程只能等待前面的进程主动释放锁，或者等到时间超时才能获得锁。

相对于基于数据库实现分布式锁的方案来说，**基于缓存实现的分布式锁的优势**表现在以下几个方面：

性能更好。数据被存放在内存，而不是磁盘，避免了频繁的 IO 操作。

很多缓存可以跨集群部署，避免了单点故障问题。

很多缓存服务都提供了可以用来实现分布式锁的方法，比如 Redis 的 setnx 方法等。

可以直接设置超时时间来控制锁的释放，因为这些缓存服务器一般支持自动删除过期数据。

这个方案的不足是，通过超时时间来控制锁的失效时间，并不是十分靠谱，因为一个进程执行时间可能比较长，或受系统进程做内存回收等影响，导致时间超时，从而不正确地释放了锁。

为了解决基于缓存实现的分布式锁的这些问题，我们再来看看基于 ZooKeeper 实现的分布式锁吧。

基于 ZooKeeper 实现分布式锁

ZooKeeper 基于树形数据存储结构实现分布式锁，来解决多个进程同时访问同一临界资源时，数据的一致性问题。ZooKeeper 的树形数据存储结构主要由 4 种节点构成：

持久节点。这是默认的节点类型，一直存在于 ZooKeeper 中。

持久顺序节点。也就是说，在创建节点时，ZooKeeper 根据节点创建的时间顺序对节点进行编号。

临时节点。与持久节点不同，当客户端与 ZooKeeper 断开连接后，该进程创建的临时节点就会被删除。

临时顺序节点，就是按时间顺序编号的临时节点。

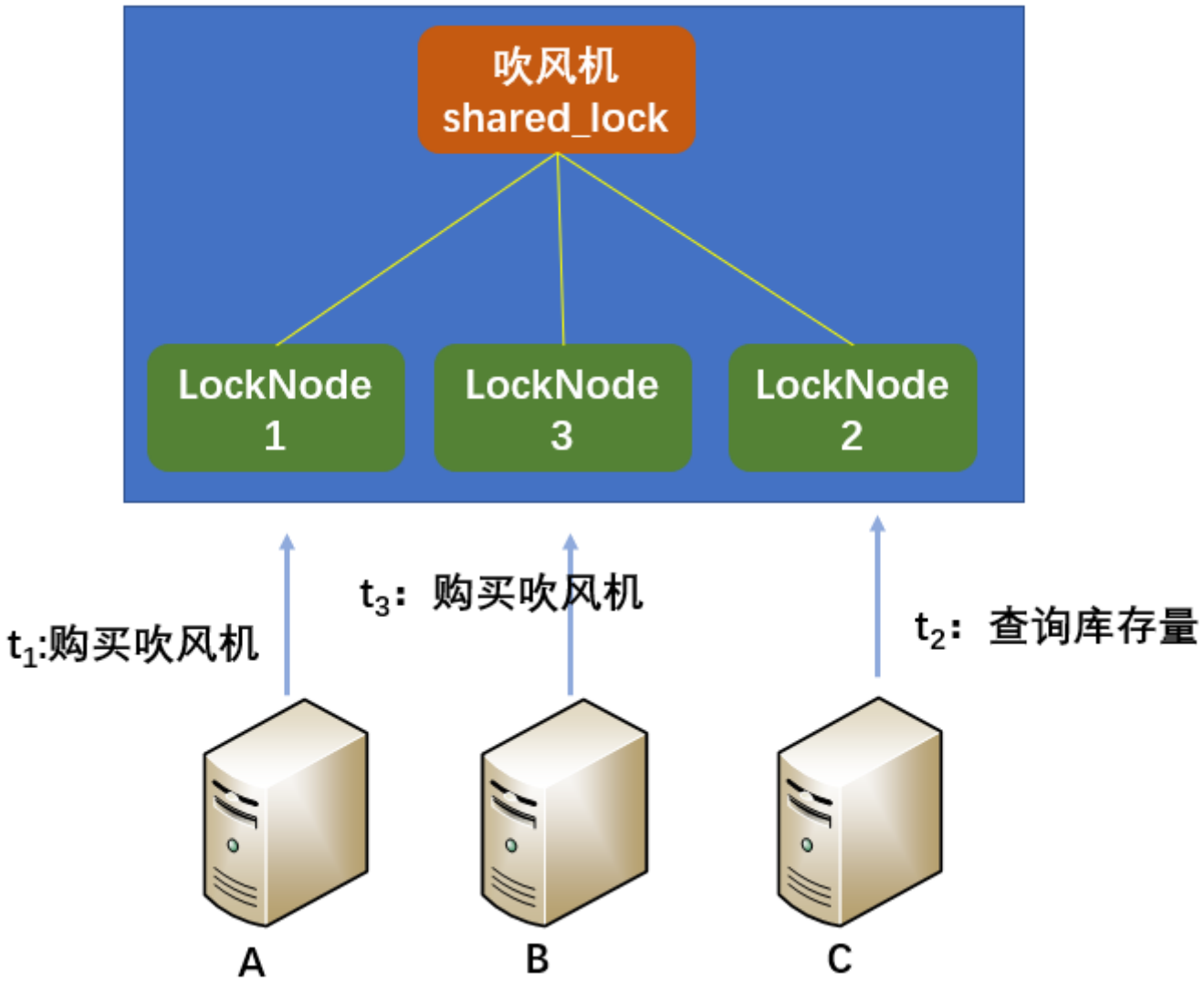
根据它们的特征，ZooKeeper 基于临时顺序节点实现了分布锁。

还是以电商售卖吹风机的场景为例。假设用户 A、B、C 同时在 11 月 11 日的零点整提交了购买吹风机的请求，ZooKeeper 会采用如下方法来实现分布式锁：

1. 在与该方法对应的持久节点 shared_lock 的目录下，为每个进程创建一个临时顺序节点。如下图所示，吹风机就是一个拥有 shared_lock 的目录，当有人买吹风机时，会为他创建一个临时顺序节点。
2. 每个进程获取 shared_lock 目录下的所有临时节点列表，注册子节点变更的 Watcher，并监听节点。
3. 每个节点确定自己的编号是否是 shared_lock 下所有子节点中最小的，若最小，则获得锁。例如，用户 A 的订单最先到服务器，因此创建了编号为 1 的临时顺序节点 LockNode1。该节点的编号是持久节点目录下最小的，因此获取到分布式锁，可以访问临界资源，从而可以购买吹风机。
4. 若本进程对应的临时节点编号不是最小的，则分为两种情况：

a. 本进程为读请求，如果比自己序号小的节点中有写请求，则等待；
b. 本进程为写请求，如果比自己序号小的节点中有读请求，则等待。

例如，用户 B 也想要买吹风机，但在他之前，用户 C 想看看吹风机的库存量。因此，用户 B 只能等用户 A 买完吹风机、用户 C 查询完库存量后，才能购买吹风机。



可以看到，使用 ZooKeeper 可以完美解决设计分布式锁时遇到的各种问题，比如单点故障、不可重入、死锁等问题。虽然 ZooKeeper 实现的分布式锁，几乎能涵盖所有分布式锁的特性，且易于实现，但需要频繁地添加和删除节点，所以性能不如基于缓存实现的分布式锁。

三种实现方式对比

我通过一张表格来对比一下这三种方式的特点，以方便你理解、记忆。

理解的容易程度	数据库 > 缓存 > ZooKeeper
实现的复杂性	ZooKeeper ≥缓存 > 数据库
性能	缓存 > ZooKeeper ≥数据库
可靠性	ZooKeeper > 缓存 > 数据库

总结来说，**ZooKeeper 分布式锁的可靠性最高，有封装好的框架，很容易实现分布式锁的功能，并且几乎解决了数据库锁和缓存式锁的不足，因此是实现分布式锁的首选方法。**

从上述分析可看出，为了确保分布式锁的可用性，我们在设计时应考虑到以下几点：

互斥性，即在分布式系统环境下，分布式锁应该能保证一个资源或一个方法在同一时间只能被一个机器的一个线程或进程操作。

具备锁失效机制，防止死锁。即使有一个进程在持有锁的期间因为崩溃而没有主动解锁，也能保证后续其他进程可以获得锁。

可重入性，即进程未释放锁时，可以多次访问临界资源。

有高可用的获取锁和释放锁的功能，且性能要好。

知识扩展：如何解决分布式锁的羊群效应问题？

在分布式锁问题中，会经常遇到羊群效应。所谓羊群效应，就是在整个分布式锁的竞争过程中，大量的“Watcher 通知”和“子节点列表的获取”操作重复运行，并且大多数节点的运行结果都是判断出自己当前并不是编号最小的节点，继续等待下一次通知，而不是执行业务逻辑。

这，就会对 ZooKeeper 服务器造成巨大的性能影响和网络冲击。更甚的是，如果同一时间多个节点对应的客户端完成事务或事务中断引起节点消失，ZooKeeper 服务器就会在短时间内向其他客户端发送大量的事件通知。

那如何解决这个问题呢？具体方法可以分为以下三步。

1. 在与该方法对应的持久节点的目录下，为每个进程创建一个临时顺序节点。
2. 每个进程获取所有临时节点列表，对比自己的编号是否最小，若最小，则获得锁。
3. 若本进程对应的临时节点编号不是最小的，则继续判断：

   若本进程为读请求，则向比自己序号小的最后一个写请求节点注册 watch 监听，当监听到该节点释放锁后，则获取锁；

   若本进程为写请求，则向比自己序号小的最后一个读请求节点注册 watch 监听，当监听到该节点释放锁后，获取锁。

总结

我以电商购物为例，首先带你剖析了什么是分布式锁，以及为什么需要分布式锁；然后，与你介绍了三种实现分布式锁的方法，包括基于数据库实现、基于缓存实现（以 Redis 为例），以及基于 ZooKeeper 实现。

分布式锁是解决多个进程同时访问临界资源的常用方法，在分布式系统中非常常见，比如开源的 ZooKeeper、Redis 中就有所涉及。通过今天这篇文章对分布式锁原理及方法的讲解，我相信你会发现分布式锁不再那么神秘、难懂，然后以此为基础对分布式锁进行更深入的学习和应用。

接下来，我把今天的内容通过下面的一张思维导图再全面总结下。

思考题

分布式锁与分布式互斥的关系是什么呢？

我是聂鹏程，感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎你把这篇文章分享给更多的朋友一起阅读。我们下期再会！

上一篇　06 | 分布式事务：All or nothing

精选留言 (2)

写留言

安排
2019-10-07

分布式互斥通过分布式锁来实现

展开

Geek_f6f02b
2019-10-07

分布式锁与分布式互斥的关系是什么呢?
我觉得分布式互斥是前提条件或者说实现目的，而分布式锁是手段或者说是实现方法。
还有文章中有2个地方不理解。一个是那个zookeeper实现分布式锁的，写请求为什么说是要等待比自己小的读请求结束，如果是比自己小的写请求就不用等待了吗？我理解是，自己为读请求就等待比自己小的写请求结束，写请求就等待所有比自己小的所有请求结束...

展开