美团点评技术团队

分布式系统互斥性与幂等性问题的分析与解决

蒋谞 ·2016-09-29 20:15

前言

随着互联网信息技术的飞速发展，数据量不断增大，业务逻辑也日趋复杂，对系统的高并发访问、海量数据处理的场景也越来越多。如何用较低成本实现系统的高可用、易伸缩、可扩展等目标就显得越发重要。为了解决这一系列问题，系统架构也在不断演进。传统的集中式系统已经逐渐无法满足要求，分布式系统被使用在更多的场景中。

分布式系统由独立的服务器通过网络松散耦合组成。在这个系统中每个服务器都是一台独立的主机，服务器之间通过内部网络连接。分布式系统有以下几个特点：

- 可扩展性：可通过横向水平扩展提高系统的性能和吞吐量。
- 高可靠性：高容错，即使系统中一台或几台故障，系统仍可提供服务。
- 高并发性：各机器并行独立处理和计算。
- 廉价高效：多台小型机而非单台高性能机。

然而，在分布式系统中，其环境的复杂度、网络的不确定性会造成诸如时钟不一致、“拜占庭将军问题”（Byzantine failure）等。存在于集中式系统中的机器宕机、消息丢失等问题也会在分布式环境中变得更加复杂。

基于分布式系统的这些特征，有两种问题逐渐成为了分布式环境中需要重点关注和解决的典型问题：

- 互斥性问题。
- 幂等性问题。

今天我们就针对这两个问题来进行分析。

互斥性问题

先看两个常见的例子：

例1：某服务记录关键数据X，当前值为100。A请求需要将X增加200；同时，B请求需要将X减100。

在理想的情况下，A先读取到X=100，然后X增加200，最后写入X=300。B请求接着从读取

X=300，减少100，最后写入X=200。
然而在真实情况下，如果不做任何处理，则可能会出现：A和B同时读取到X=100；A写入之前B读取到X；B比A先写入等等情况。

例2：某服务提供一组任务，A请求随机从任务组中获取一个任务；B请求随机从任务组中获取一个任务。
在理想的情况下，A从任务组中挑选一个任务，任务组删除该任务，B从剩下的的任务中再挑一个，任务组删除该任务。
同样的，在真实情况下，如果不做任何处理，可能会出现A和B挑中了同一个任务的情况。

以上的两个例子，都存在操作互斥性的问题。互斥性问题用通俗的话来讲，就是对共享资源的抢占问题。如果不同的请求对同一个或者同一组资源读取并修改时，无法保证按序执行，无法保证一个操作的原子性，那么就很有可能会出现预期外的情况。因此操作的互斥性问题，也可以理解为一个需要保证时序性、原子性的问题。

在传统的基于数据库的架构中，对于数据的抢占问题往往是通过数据库事务（ACID）来保证的。在分布式环境中，出于对性能以及一致性敏感度的要求，使得分布式锁成为了一种比较常见而高效的解决方案。

事实上，操作互斥性问题也并非分布式环境所独有，在传统的多线程、多进程情况下已经有了很好的解决方案。因此在研究分布式锁之前，我们先来分析下这两种情况的解决方案，以期能够对分布式锁的解决方案提供一些实现思路。

多线程环境解决方案及原理

解决方案

《Thinking in Java》书中写到：

> 基本上所有的并发模式在解决线程冲突问题的时候，都是采用序列化访问共享资源的方案。

在多线程环境中，线程之间因为公用一些存储空间，冲突问题时有发生。解决冲突问题最普遍的方式就是用互斥锁把该资源或对该资源的操作保护起来。

Java JDK中提供了两种互斥锁Lock和synchronized。不同的线程之间对同一资源进行抢占，该资源通常表现为某个类的普通成员变量。因此，利用ReentrantLock或者synchronized将共享的变量及其操作锁住，即可基本解决资源抢占的问题。

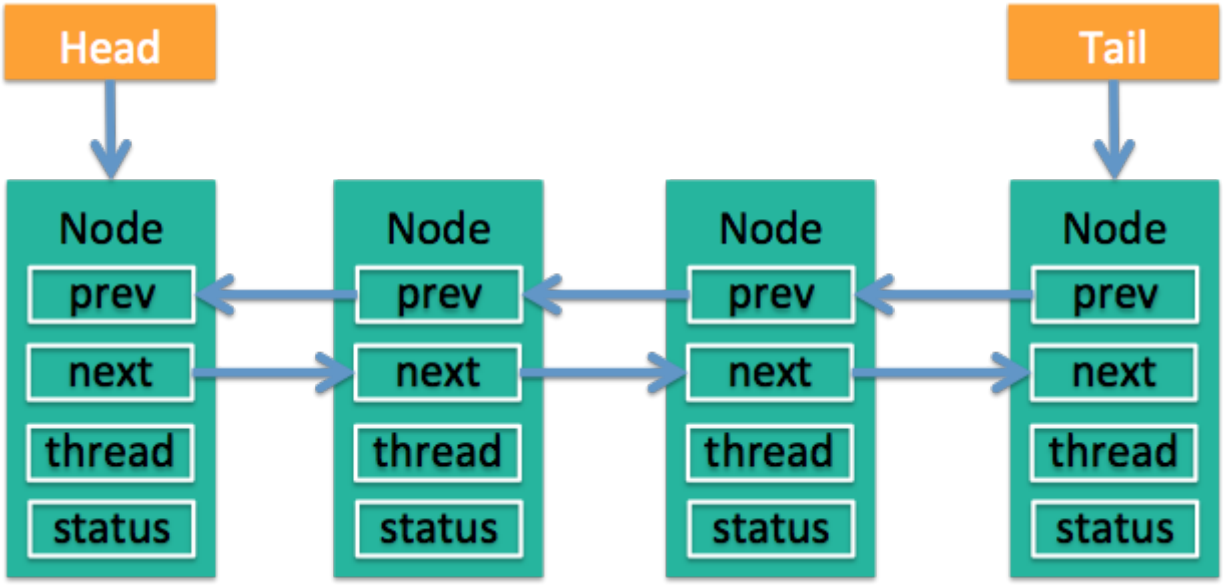
下面来简单聊一聊两者的实现原理。

原理

ReentrantLock

ReentrantLock主要利用CAS+CLH队列来实现。它支持公平锁和非公平锁，两者的实现类似。

- CAS：Compare and Swap，比较并交换。CAS有3个操作数：内存值V、预期值A、要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。该操作是一个原子操作，被广泛的应用在Java的底层实现中。在Java中，CAS主要是由sun.misc.Unsafe这个类通过JNI调用CPU底层指令实现。
- CLH队列：带头结点的双向非循环链表(如下图所示)：



ReentrantLock的基本实现可以概括为：先通过CAS尝试获取锁。如果此时已经有线程占据了锁，那就加入CLH队列并且被挂起。当锁被释放之后，排在CLH队列队首的线程会被唤醒，然后CAS再次尝试获取锁。在这个时候，如果：

- 非公平锁：如果同时还有另一个线程进来尝试获取，那么有可能会让这个线程抢先获取；
- 公平锁：如果同时还有另一个线程进来尝试获取，当它发现自己不是在队首的话，就会排到队尾，由队首的线程获取到锁。

下面分析下两个片段：

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

在尝试获取锁的时候，会先调用上面的方法。如果状态为0，则表明此时无人占有锁。此时尝试进行set，一旦成功，则成功占有锁。如果状态不为0，再判断是否是当前线程获取到锁。如果是的话，将状态+1，因为此时就是当前线程，所以不用CAS。这也就是可重入锁的实现原理。

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```

该方法是在尝试获取锁失败加入CHL队尾之后，如果发现前序节点是head，则CAS再尝试获取一次。否则，则会根据前序节点的状态判断是否需要阻塞。如果需要阻塞，则调用LockSupport的park方法阻塞该线程。

synchronized

在Java语言中存在两种内建的synchronized语法：synchronized语句、synchronized方法。

- synchronized语句：当源代码被编译成字节码的时候，会在同步块的入口位置和退出位置分别插入monitorenter和monitorexit字节码指令;
- synchronized方法：在Class文件的方法表中将该方法的access_flags字段中的synchronized标志位置1。这个在specification中没有明确说明。

在Java虚拟机的specification中，有关于monitorenter和monitorexit字节码指令的详细描述：http://docs.oracle.com/Javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.monitorenter (http://docs.oracle.com/Javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.monitorenter)

sun.misc.Unsafe)。

monitorenter

> The objectref must be of type reference.
> Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes monitorenter attempts to gain ownership of the monitor associated with objectref, as follows:
>
> - If the entry count of the monitor associated with objectref is zero, the thread enters the monitor and sets its entry count to one. The thread is then the owner of the monitor.
> - If the thread already owns the monitor associated with objectref, it reenters the monitor, incrementing its entry count.
> - If another thread already owns the monitor associated with objectref, the thread blocks until the monitor's entry count is zero, then tries again to gain ownership.

每个对象都有一个锁，也就是监视器（monitor）。当monitor被占有时就表示它被锁定。线程执行monitorenter指令时尝试获取对象所对应的monitor的所有权，过程如下：

- 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者;
- 如果线程已经拥有了该monitor，只是重新进入，则进入monitor的进入数加1;
- 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

monitorexit

> The objectref must be of type reference.
> The thread that executes monitorexit must be the owner of the monitor associated with the instance referenced by objectref.
> The thread decrements the entry count of the monitor associated with objectref. If as a result the value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other threads that are blocking to enter the monitor are allowed to attempt to do so.

执行monitorexit的线程必须是相应的monitor的所有者。
指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个monitor的所有权。

在JDK1.6及其之前的版本中monitorenter和monitorexit字节码依赖于底层的操作系统的Mutex Lock来实现的，但是由于使用Mutex Lock需要将当前线程挂起并从用户态切换到内核态来执

Lock来实现的，但是由于使用Mutex Lock需要将当前线程挂起并从用户态切换到内核态来执行，这种切换的代价是非常昂贵的。然而在现实中的大部分情况下，同步方法是运行在单线程环境（无锁竞争环境）。如果每次都调用Mutex Lock将严重的影响程序的性能。因此在JDK 1.6之后的版本中对锁的实现做了大量的优化，这些优化在很大程度上减少或避免了Mutex Lock的使用。

多进程的解决方案

在多道程序系统中存在许多进程，它们共享各种资源，然而有很多资源一次只能供一个进程使用，这便是临界资源。多进程中的临界资源大致上可以分为两类，一类是物理上的真实资源，如打印机；一类是硬盘或内存中的共享数据，如共享内存等。而进程内互斥访问临界资源的代码被称为临界区。

针对临界资源的互斥访问，JVM层面的锁就已经失去效力了。在多进程的情况下，主要还是利用操作系统层面的进程间通信原理来解决临界资源的抢占问题。比较常见的一种方法便是使用信号量（Semaphores）。

信号量在POSIX标准下有两种，分别为有名信号量和无名信号量。无名信号量通常保存在共享内存中，而有名信号量是与一个特定的文件名称相关联。信号量是一个整数变量，有计数信号量和二值信号量两种。对信号量的操作，主要是P操作（wait）和V操作（signal）。

- P操作：先检查信号量的大小，若值大于零，则将信号量减1，同时进程获得共享资源的访问权限，继续执行；若小于或者等于零，则该进程被阻塞后，进入等待队列。
- V操作：该操作将信号量的值加1，如果有进程阻塞着等待该信号量，那么其中一个进程将被唤醒。

举个例子，设信号量为1，当一个进程A在进入临界区之前，先进行P操作。发现值大于零，那么就将信号量减为0，进入临界区执行。此时，若另一个进程B也要进去临界区，进行P操作，发现信号量等于0，则会被阻塞。当进程A退出临界区时，会进行V操作，将信号量的值加1，并唤醒阻塞的进程B。此时B就可以进入临界区了。

这种方式，其实和多线程环境下的加解锁非常类似。因此用信号量处理临界资源抢占，也可以简单地理解为对临界区进行加锁。

通过上面的一些了解，我们可以概括出解决互斥性问题，即资源抢占的基本方式为：

对共享资源的操作前后（进入退出临界区）加解锁，保证不同线程或进程可以互斥有序的操作资源。

加解锁方式，有显式的加解锁，如ReentrantLock或信号量；也有隐式的加解锁，如synchronized。那么在分布式环境中，为了保证不同JVM不同主机间不会出现资源抢占，那么同样只要对临界区加解锁就可以了。

然而在多线程和多进程中，锁已经有比较完善的实现，直接使用即可。但是在分布式环境下，就需要我们自己来实现分布式锁。

分布式环境下的解决方案——分布式锁

首先，我们来看看分布式锁的基本条件。

分布式锁条件

基本条件

再回顾下多线程和多进程环境下的锁，可以发现锁的实现有很多共通之处，它们都需要满足一些最基本的条件：

1. 需要有存储锁的空间，并且锁的空间是可以访问到的。
2. 锁需要被唯一标识。
3. 锁要有至少两种状态。

仔细分析这三个条件：

- 存储空间

锁是一个抽象的概念，锁的实现，需要依存于一个可以存储锁的空间。在多线程中是内存，在多进程中是内存或者磁盘。更重要的是，这个空间是可以被访问到的。多线程中，不同的线程都可以访问到堆中的成员变量；在多进程中，不同的进程可以访问到共享内存中的数据或者存储在磁盘中的文件。但是在分布式环境中，不同的主机很难访问对方的内存或磁盘。这就需要一个都能访问到的外部空间来作为存储空间。

最普遍的外部存储空间就是数据库了，事实上也确实有基于数据库做分布式锁（行锁、version乐观锁），如quartz集群架构中就有所使用。除此以外，还有各式缓存如Redis、Tair、Memcached、Mongodb，当然还有专门的分布式协调服务Zookeeper，甚至是另一台主机。只要可以存储数据、锁在其中可以被多主机访问到，那就可以作为分布式锁的存储空间。

- 唯一标识

不同的共享资源，必然需要用不同的锁进行保护，因此相应的锁必须有唯一的标识。在多线程环境中，锁可以是一个对象，那么对这个对象的引用便是这个唯一标识。多进程环境中，信号量在共享内存中也是由引用来作为唯一的标识。但是如果不在内存中，失去了对锁的引用，如何唯一标识它呢？上文提到的有名信号量，便是用硬盘中的文件名作为唯一标识。因此，在分布式环境中，只要给这个锁设定一个名称，并且保证这个名称是全局唯一的，那么就可以作为唯一标识。

- 至少两种状态

为了给临界区加锁和解锁，需要存储两种不同的状态。如ReentrantLock中的status，0表示没有线程竞争，大于0表示有线程竞争；信号量大于0表示可以进入临界区，小于等于0则表示需要被阻塞。因此只要在分布式环境中，锁的状态有两种或以上：如有锁、没锁；存在、不存在等等，均可以实现。

有了这三个条件，基本就可以实现一个简单的分布式锁了。下面以数据库为例，实现一个简单的分布式锁：
数据库表，字段为锁的ID（唯一标识），锁的状态（0表示没有被锁，1表示被锁）。
伪代码为：

```
lock = mysql.get(id);
while(lock.status == 1) {
    sleep(100);
}
mysql.update(lock.status = 1);
doSomething();
mysql.update(lock.status = 0);
```

问题

以上的方式即可以实现一个粗糙的分布式锁，但是这样的实现，有没有什么问题呢？

- 问题1：锁状态判断原子性无法保证
  从读取锁的状态，到判断该状态是否为被锁，需要经历两步操作。如果不能保证这两步的原子性，就可能导致不止一个请求获取到了锁，这显然是不行的。因此，我们需要保证锁状态判断的原子性。
- 问题2：网络断开或主机宕机，锁状态无法清除
  假设在主机已经获取到锁的情况下，突然出现了网络断开或者主机宕机，如果不做任何处理该锁将仍然处于被锁定的状态。那么之后所有的请求都无法再成功抢占到这个锁。因此，我们需要在持有锁的主机宕机或者网络断开的时候，及时的释放掉这把锁。
- 问题3：无法保证释放的是自己上锁的那把锁
  在解决了问题2的情况下再设想一下，假设持有锁的主机A在临界区遇到网络抖动导致网络断开，分布式锁及时的释放掉了这把锁。之后，另一个主机B占有了这把锁，但是此时主机A网络恢复，退出临界区时解锁。由于都是同一把锁，所以A就会将B的锁解开。此时如果有第三个主机尝试抢占这把锁，也将会成功获得。因此，我们需要在解锁时，确定自己解的这个锁正是自己锁上的。

进阶条件

如果分布式锁的实现，还能再解决上面的三个问题，那么就可以算是一个相对完整的分布式锁了。然而，在实际的系统环境中，还会对分布式锁有更高级的要求。

1. 可重入：线程中的可重入，指的是外层函数获得锁之后，内层也可以获得锁，ReentrantLock和synchronized都是可重入锁；衍生到分布式环境中，一般仍然指的是线程的可重入，在绝大多数分布式环境中，都要求分布式锁是可重入的。
2. 惊群效应（Herd Effect）：在分布式锁中，惊群效应指的是，在有多个请求等待获取锁的时候，一旦占有锁的线程释放之后，如果所有等待的方都同时被唤醒，尝试抢占锁。但是这样的情况会造成比较大的开销，那么在实现分布式锁的时候，应该尽量避免惊群效应的产生。
3. 公平锁和非公平锁：不同的需求，可能需要不同的分布式锁。非公平锁普遍比公平锁开销小。但是业务需求如果必须要锁的竞争者按顺序获得锁，那么就需要实现公平锁。
4. 阻塞锁和自旋锁：针对不同的使用场景，阻塞锁和自旋锁的效率也会有所不同。阻塞锁会有上下文切换，如果并发量比较高且临界区的操作耗时比较短，那么造成的性能开销就比较大了。但是如果临界区操作耗时比较长，一直保持自旋，也会对CPU造成更大的负荷。

保留以上所有问题和条件，我们接下来看一些比较典型的实现方案。

典型实现

ZooKeeper的实现

ZooKeeper（以下简称“ZK”）中有一种节点叫做顺序节点，假如我们在/lock/目录下创建3个节点，ZK集群会按照发起创建的顺序来创建节点，节点分别为/lock/0000000001、/lock/0000000002、/lock/0000000003。

ZK中还有一种名为临时节点的节点，临时节点由某个客户端创建，当客户端与ZK集群断开连接，则该节点自动被删除。EPHEMERAL_SEQUENTIAL为临时顺序节点。

根据ZK中节点是否存在，可以作为分布式锁的锁状态，以此来实现一个分布式锁，下面是分布式锁的基本逻辑：

1. 客户端调用create()方法创建名为“/dlm-locks/lockname/lock-”的临时顺序节点。
2. 客户端调用getChildren(“lockname”)方法来获取所有已经创建的子节点。
3. 客户端获取到所有子节点path之后，如果发现自己在步骤1中创建的节点是所有节点中序号最小的，那么就认为这个客户端获得了锁。
4. 如果创建的节点不是所有节点中需要最小的，那么则监视比自己创建节点的序列号小的最大的节点，进入等待。直到下次监视的子节点变更的时候，再进行子节点的获取，判断是否获取锁。

释放锁的过程相对比较简单，就是删除自己创建的那个子节点即可，不过也仍需要考虑删除节点失败等异常情况。

开源的基于ZK的Menagerie的源码就是一个典型的例子：

https://github.com/sfines/menagerie (https://github.com/sfines/menagerie) 。

Menagerie中的lock首先实现了可重入锁，利用ThreadLocal存储进入的次数，每次加锁次数加1，每次解锁次数减1。如果判断出是当前线程持有锁，就不用走获取锁的流程。

通过tryAcquireDistributed方法尝试获取锁，循环判断前序节点是否存在，如果存在则监视该节点并且返回获取失败。如果前序节点不存在，则再判断更前一个节点。如果判断出自己是第一个节点，则返回获取成功。

为了在别的线程占有锁的时候阻塞，代码中使用JUC的condition来完成。如果获取尝试锁失败，则进入等待且放弃localLock，等待前序节点唤醒。而localLock是一个本地的公平锁，使得condition可以公平的进行唤醒，配合循环判断前序节点，实现了一个公平锁。

这种实现方式非常类似于ReentrantLock的CHL队列，而且zk的临时节点可以直接避免网络断开或主机宕机，锁状态无法清除的问题，顺序节点可以避免惊群效应。这些特性都使得利用ZK实现分布式锁成为了最普遍的方案之一。

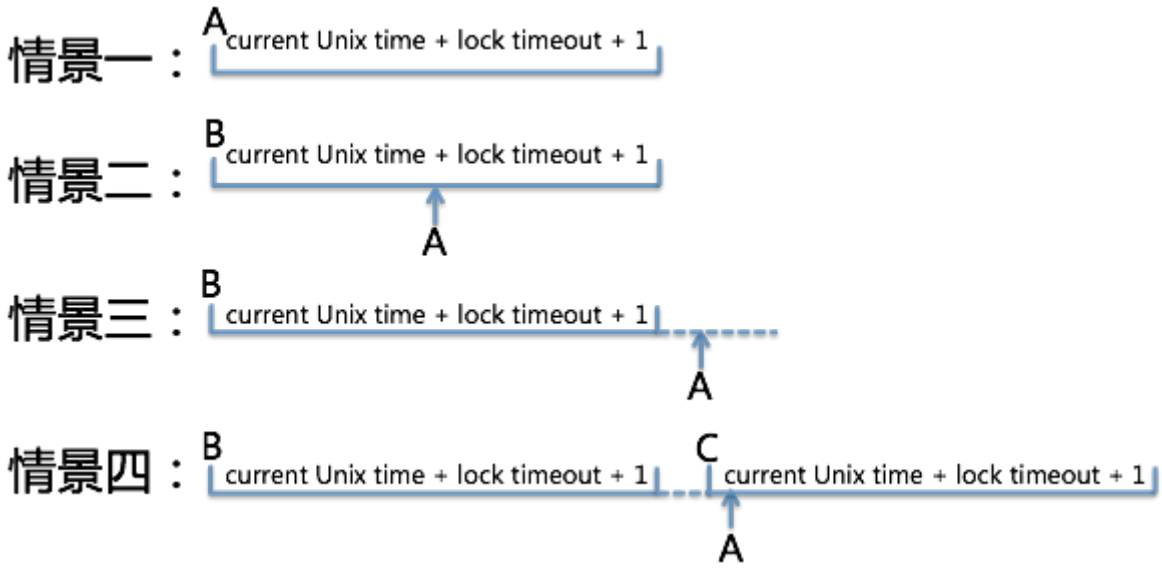
Redis的实现

Redis的分布式缓存特性使其成为了分布式锁的一种基础实现。通过Redis中是否存在某个锁ID，则可以判断是否上锁。为了保证判断锁是否存在的原子性，保证只有一个线程获取同一把锁，Redis有**SETNX**（即SET if Not eXists）和**GETSET**（先写新值，返回旧值，原子性操作，可以用于分辨是不是首次操作）操作。

为了防止主机宕机或网络断开之后的死锁，Redis没有ZK那种天然的实现方式，只能依赖设置超时时间来规避。

以下是一种比较普遍但不太完善的Redis分布式锁的实现步骤（与下图一一对应）：

1. 线程A发送SETNX lock.orderid 尝试获得锁，如果锁不存在，则set并获得锁。
2. 如果锁存在，则再判断锁的值（时间戳）是否大于当前时间，如果没有超时，则等待一下再重试。
3. 如果已经超时了，在用GETSET lock.{orderid} 来尝试获取锁，如果这时候拿到的时间戳仍旧超时，则说明已经获得锁了。
4. 如果在此之前，另一个线程C快一步执行了上面的操作，那么A拿到的时间戳是个未超时的值，这时A没有如期获得锁，需要再次等待或重试。



该实现还有一个需要考虑的问题是全局时钟问题，由于生产环境主机时钟不能保证完全同步，对时间戳的判断也可能会产生误差。

以上是Redis的一种常见的实现方式，除此以外还可以用SETNX+EXPIRE来实现。Redisson是一个官方推荐的Redis客户端并且实现了很多分布式的功能。它的分布式锁就提供了一种更完善的解决方案，源码：https://github.com/mrniko/redisson (https://github.com/mrniko/redisson) 。

Tair的实现

Tair和Redis的实现类似，Tair客户端封装了一个expireLock的方法：通过锁状态和过期时间戳来共同判断锁是否存在，只有锁已经存在且没有过期的状态才判定为有锁状态。在有锁状态下，不能加锁，能通过大于或等于过期时间的时间戳进行解锁。

采用这样的方式，可以不用在Value中存储时间戳，并且保证了判断是否有锁的原子性。更值得注意的是，由于超时时间是由Tair判断，所以避免了不同主机时钟不一致的情况。

以上的几种分布式锁实现方式，都是比较常见且有些已经在生产环境中应用。随着应用环境越来越复杂，这些实现可能仍然会遇到一些挑战。

- **强依赖于外部组件**：分布式锁的实现都需要依赖于外部数据存储如ZK、Redis等等，因此一旦这些外部组件出现故障，那么分布式锁就不可用了。
- **无法完全满足需求**：不同分布式锁的实现，都有相应的特点，对于一些需求并不能很好的满足，如实现公平锁、给等待锁加超时时间等等。

基于以上问题，结合多种实现方式，我们开发了Cerberus（得名自希腊神话里守卫地狱的猛犬），致力于提供灵活可靠的分布式锁。

Cerberus分布式锁

Cerberus有以下几个特点。

特点一：一套接口多种引擎

Cerberus分布式锁使用了多种引擎实现方式（Tair、ZK、未来支持Redis），支持使用方自主选择所需的一种或多种引擎。这样可以结合引擎特点，选择符合实际业务需求和系统架构的方式。

Cerberus分布式锁将不同引擎的接口抽象为一套，屏蔽了不同引擎的实现细节。使得使用方可以专注于业务逻辑，也可以任意选择并切换引擎而不必更改任何的业务代码。

如果使用方选择了一种以上的引擎，那么以配置顺序来区分主副引擎。以下是使用主引擎的推荐：

功能需求	Tair	ZK
并发量高	✓	
响应时间敏感	✓	
临界区执行时间长		✓
公平锁		✓
非公平锁	✓	
读写锁		✓

特点二：使用灵活、学习成本低

下面是Cerberus的lock方法，这些方法和JUC的ReentrantLock的方式保持一致，使用非常灵活且不需要额外的学习时间。

- void lock();
  获取锁，如果锁被占用，将禁用当前线程，并且在获得锁之前，该线程将一直处于阻塞状态。
- boolean tryLock();
  仅在调用时锁为空闲状态才获取该锁。
  如果锁可用，则获取锁，并立即返回值 true。如果锁不可用，则此方法将立即返回值 false。
- boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
  如果锁在给定的等待时间内空闲，并且当前线程未被中断，则获取锁。
  如果在给定时间内锁可用，则获取锁，并立即返回值 true。如果在给定时间内锁一直不可用，则此方法将立即返回值false。
- void lockInterruptibly() throws InterruptedException;
  获取锁，如果锁被占用，则一直等待直到线程被中断或者获取到锁。
- void unlock();

void unlock();

释放当前持有的锁。

特点三：支持一键降级

Cerberus提供了实时切换引擎的接口:

- String switchEngine()
  转换分布式锁引擎，按配置的引擎的顺序循环转换。
  返回值：返回当前的engine名字，如："zk"。
- String switchEngine(String engineName)
  转换分布式锁引擎，切换为指定的引擎。
  参数：engineName - 引擎的名字，同配置bean的名字，"zk"/"tair"。
  返回值：返回当前的engine名字，如："zk"。

当使用方选择了两种引擎，平时分布式锁会工作在主引擎上。一旦所依赖的主引擎出现故障，那么使用方可以通过自动或者手动方式调用该切换引擎接口，平滑的将分布式锁切换到另一个引擎上以将风险降到最低。自动切换方式可以利用Hystrix实现。手动切换推荐的一个方案则是使用美团点评基于Zookeeper的基础组件MCC，通过监听MCC配置项更改，来达到手动将分布式系统所有主机同步切换引擎的目的。需要注意的是，切换引擎目前并不会迁移原引擎已有的锁。这样做的目的是出于必要性、系统复杂度和可靠性的综合考虑。在实际情况下，引擎故障到切换引擎，尤其是手动切换引擎的时间，要远大于分布式锁的存活时间。作为较轻量级的Cerberus来说，迁移锁会带来不必要的开销以及较高的系统复杂度。鉴于此，如果想要保证在引擎故障后的绝对可靠，那么则需要结合其他方案来进行处理。

除此以外，Cerberus还提供了内置公用集群，免去搭建和配置集群的烦恼。Cerberus也有一套完善的应用授权机制，以此防止业务方未经评估使用，对集群造成影响。

目前，Cerberus分布式锁已经持续迭代了8个版本，先后在美团点评多个项目中稳定运行。

幂等性问题

所谓幂等，简单地说，就是对接口的多次调用所产生的结果和调用一次是一致的。扩展一下，这里的接口，可以理解为对外发布的HTTP接口或者Thrift接口，也可以是接收消息的内部接口，甚至是一个内部方法或操作。

那么我们为什么需要接口具有幂等性呢？设想一下以下情形：

- 在App中下订单的时候，点击确认之后，没反应，就又点击了几次。在这种情况下，如果无法保证该接口的幂等性，那么将会出现重复下单问题。
- 在接收消息的时候，消息推送重复。如果处理消息的接口无法保证幂等，那么重复消费消息产生的影响可能会非常大

会非常大。

在分布式环境中，网络环境更加复杂，因前端操作抖动、网络故障、消息重复、响应速度慢等原因，对接口的重复调用概率会比集中式环境下更大，尤其是重复消息在分布式环境中很难避免。Tyler Treat也在《You Cannot Have Exactly-Once Delivery》(http://bravenewgeek.com/you-cannot-have-exactly-once-delivery/)一文中提到：

> Within the context of a distributed system, you cannot have exactly-once message delivery.

分布式环境中，有些接口是天然保证幂等性的，如查询操作。有些对数据的修改是一个常量，并且无其他记录和操作，那也可以说是具有幂等性的。其他情况下，所有涉及对数据的修改、状态的变更就都有必要防止重复性操作的发生。通过间接的实现接口的幂等性来防止重复操作所带来的影响，成为了一种有效的解决方案。

GTIS

GTIS就是这样的一个解决方案。它是一个轻量的重复操作关卡系统，它能够确保在分布式环境中操作的唯一性。我们可以用它来间接保证每个操作的幂等性。它具有如下特点：

- 高效：低延时，单个方法平均响应时间在2ms内，几乎不会对业务造成影响；
- 可靠：提供降级策略，以应对外部存储引擎故障所造成的影响；提供应用鉴权，提供集群配置自定义，降低不同业务之间的干扰；
- 简单：接入简捷方便，学习成本低。只需简单的配置，在代码中进行两个方法的调用即可完成所有的接入工作；
- 灵活：提供多种接口参数、使用策略，以满足不同的业务需求。

实现原理

基本原理

GTIS的实现思路是将每一个不同的业务操作赋予其唯一性。这个唯一性是通过对不同操作所对应的唯一的内容特性生成一个唯一的全局ID来实现的。基本原则为：相同的操作生成相同的全局ID；不同的操作生成不同的全局ID。

生成的全局ID需要存储在外部存储引擎中，数据库、Redis亦或是Tair等等均可实现。考虑到Tair天生分布式和持久化的优势，目前的GTIS存储在Tair中。其相应的key和value如下：

- key：将对于不同的业务，采用APP_KEY+业务操作内容特性生成一个唯一标识trans_contents。然后对唯一标识进行加密生成全局ID作为Key。

- value：current_timestamp + trans_contents，current_timestamp用于标识当前的操作线程。

判断是否重复，主要利用Tair的SETNX方法，如果原来没有值则set且返回成功，如果已经有值则返回失败。

内部流程

GTIS的内部实现流程为：

1. 业务方在业务操作之前，生成一个能够唯一标识该操作的transContents，传入GTIS；
2. GTIS根据传入的transContents，用MD5生成全局ID；
3. GTIS将全局ID作为key，current_timestamp+transContents作为value放入Tair进行setNx，将结果返回给业务方；
4. 业务方根据返回结果确定能否开始进行业务操作；
5. 若能，开始进行操作；若不能，则结束当前操作；
6. 业务方将操作结果和请求结果传入GTIS，系统进行一次请求结果的检验；
7. 若该次操作成功，GTIS根据key取出value值，跟传入的返回结果进行比对，如果两者相等，则将该全局ID的过期时间改为较长时间；
8. GTIS返回最终结果。

实现难点

GTIS的实现难点在于如何保证其判断重复的可靠性。由于分布式环境的复杂度和业务操作的不确定性，在上一章节分布式锁的实现中考虑的网络断开或主机宕机等等问题，同样需要在GTIS中设法解决。这里列出几个典型的场景：

- 如果操作执行失败，理想的情况应该是另一个相同的操作可以立即进行。因此，需要对业务方的操作结果进行判断，如果操作失败，那么就需要立即删除该全局ID；
- 如果操作超时或主机宕机，当前的操作无法告知GTIS操作是否成功。那么我们必须引入超时机制，一旦长时间获取不到业务方的操作反馈，那么也需要该全局ID失效；
- 结合上两个场景，既然全局ID会失效并且可能会被删除，那就需要保证删除的不是另一个相同操作的全局ID。这就需要将特殊的标识记录下来，并由此来判断。这里所用的标识为当前时间戳。