# 分布式锁

## 背景

在单机时代，虽然不需要分布式锁，但也面临过类似的问题，只不过在单机的情况下，如果有多个线程要同时访问某个共享资源的时候，我们可以采用线程间加锁的机制，即当某个线程获取到这个资源后，就立即对这个资源进行加锁，当使用完资源之后，再解锁，其它线程就可以接着使用了。例如，在JAVA中，甚至专门提供了一些处理锁机制的一些API（synchronize/Lock等）。

但是到了分布式系统的时代，这种线程之间的锁机制，就没作用了，系统可能会有多份并且部署在不同的机器上，这些资源已经不是在线程之间共享了，而是属于进程之间共享的资源。

因此，为了解决这个问题，我们就必须引入「分布式锁」。

分布式锁，是指在分布式的部署环境下，通过锁机制来让多客户端互斥的对共享资源进行访问。

综上，需要加锁的场景需要满足以下条件：

1. 共享资源
2. 共享资源互斥
3. 多任务环境

我们先来看一个业务场景：

系统A是一个电商系统，目前是一台机器部署，系统中有一个用户下订单的接口，但是用户下订单之前一定要去检查一下库存，确保库存足够了才会给用户下单。

由于系统有一定的并发，所以会预先将商品的库存保存在redis中，用户下单的时候会更新redis的库存。

此时系统架构如下：



但是这样一来会产生一个问题：假如某个时刻，redis里面的某个商品库存为1，此时两个请求同时到来，其中一个请求执行到上图的第3步，更新数据库的库存为0，但是第4步还没有执行。

而另外一个请求执行到了第2步，发现库存还是1，就继续执行第3步。

这样的结果，是导致卖出了2个商品，然而其实库存只有1个。

很明显不对啊！这就是典型的**库存超卖问题**

此时，我们很容易想到解决方案：用锁把2、3、4步锁住，让他们执行完之后，另一个线程才能进来执行第2步。



按照上面的图，在执行第2步时，使用Java提供的synchronized或者ReentrantLock来锁住，然后在第4步执行完之后才释放锁。

这样一来，2、3、4 这3个步骤就被“锁”住了，多个线程之间只能串行化执行。

但是好景不长，整个系统的并发飙升，一台机器扛不住了。现在要增加一台机器，如下图：



假设此时两个用户的请求同时到来，但是落在了不同的机器上，那么这两个请求是可以同时执行了，还是会出现库存超卖的问题。

为什么呢？因为上图中的两个A系统，运行在两个不同的JVM里面，他们加的锁只对属于自己JVM里面的线程有效，对于其他JVM的线程是无效的。

因此，这里的问题是：Java提供的原生锁机制在多机部署场景下失效了

这是因为两台机器加的锁不是同一个锁(两个锁在不同的JVM里面)。

那么，我们只要保证两台机器加的锁是同一个锁，问题不就解决了吗？

此时，就该分布式锁隆重登场了，分布式锁的思路是：

在整个系统提供一个全局、唯一的获取锁的“东西”，然后每个系统在需要加锁时，都去问这个“东西”拿到一把锁，这样不同的系统拿到的就可以认为是同一把锁。

至于这个“东西”，可以是Redis、Zookeeper，也可以是数据库。

文字描述不太直观，我们来看下图：



通过上面的分析，我们知道了库存超卖场景在分布式部署系统的情况下使用Java原生的锁机制无法保证线程安全，所以我们需要用到分布式锁的方案。

## 特点

分布式锁的实现由多种方式，但是不管怎样，分布式锁一般要有以下特点：

**排他性：**任意时刻，只能有一个client能获取到锁

**容错性：**分布式锁服务一般要满足AP，也就是说，只要分布式锁服务集群节点大部分存活，client就可以进行加锁解锁操作

**避免死锁：**分布式锁一定能得到释放，即使client在释放之前崩溃或者网络不可达

除了以上特点之外，分布式锁最好也能满足可重入、高性能、阻塞锁特性（AQS这种，能够及时从阻塞状态唤醒）等。

## 方案

用到分布式锁说明遇到了多个进程共同访问同一个资源的问题，一般是在两个场景下会防止对同一个资源的重复访问：

1、提高效率。比如多个节点计算同一批任务，如果某个任务已经有节点在计算了，那其他节点就不用重复计算了，以免浪费计算资源。不过重复计算也没事，不会造成其他更大的损失。也就是允许偶尔的失败。

2、保证正确性。这种情况对锁的要求就很高了，如果重复计算，会对正确性造成影响。这种不允许失败。

引入分布式锁势必要引入一个第三方的基础设施，比如MySQL，Redis，Zookeeper等，这些实现分布式锁的基础设施出问题了，也会影响业务。

针对分布式锁的实现，目前比较常用的方案：

1. 基于数据库实现分布式锁
2. 基于缓存（redis、memcache、tair）实现分布式锁
3. 基于Zookeeper实现分布式锁

### DB锁

#### 方案

1、唯一约束

2、基于数据库来做分布式锁的话，通常有两种做法：

基于数据库的乐观锁（lock in share mode）

基于数据库的悲观锁（select ... for update）

#### 缺点

存在的问题：

1. 可用性差，数据库挂掉会导致业务系统不可用；
2. 数据库性能存在瓶颈，不适合高并发场景；
3. 锁的失效时间难以控制，删除锁失败容易导致死锁。

### Redis锁

#### 方案

加锁和解锁的锁必须是同一个，常见的解决方案是给每个锁一个钥匙（唯一ID），加锁时生成，解锁时判断。

##### redis原子操作

1、基于Redis实现的锁机制，主要是依赖redis自身的原子操作。

加锁

setnx命令加锁，并设置锁的有效时间和持有人标识：

**SET user\_key user\_value NX PX 100**

redis从2.6.12版本开始，SET命令才支持这些参数：

NX：只在在键不存在时，才对键进行设置操作，SET key value NX 效果等同于 SETNX key value

PX millisecond：设置键的过期时间为millisecond毫秒，当超过这个时间后，设置的键会自动失效

解锁

检查是否持有锁，然后删除锁

delete values命令删除锁

value具有唯一性，这是避免了一种情况：假设A获取了锁，过期时间30s，此时35s之后，锁已经自动释放了，A去释放锁，但是此时可能B获取了锁。A客户端就不能删除B的锁了。



**为什么这个命令可以帮我们实现锁机制呢？**

因为这个命令是只有在某个key不存在的时候，才会执行成功。那么当多个进程同时并发的去设置同一个key的时候，就永远只会有一个进程成功。

当某个进程设置成功之后，就可以去执行业务逻辑了，等业务逻辑执行完毕之后，再去进行解锁。

解锁很简单，只需要删除这个key就可以了，不过删除之前需要判断，这个key对应的value是当初自己设置的那个。

Redlock

除了要考虑客户端要怎么实现分布式锁之外，还需要考虑redis的部署问题。

redis有3种部署方式：

单机模式

master-slave + sentinel选举模式

redis cluster模式

使用redis做分布式锁的缺点在于：如果采用单机部署模式，会存在单点问题，只要redis故障了。加锁就不行了。

采用master-slave模式，加锁的时候只对一个节点加锁，即便通过sentinel做了高可用，但是如果master节点故障了，发生主从切换，此时就会有可能出现锁丢失的问题。

基于以上的考虑，其实redis的作者也考虑到这个问题，他提出了一个RedLock的算法，这个算法的意思大概是这样的：

假设redis的部署模式是redis cluster，总共有5个master节点，通过以下步骤获取一把锁：

1、获取当前时间戳，单位是毫秒

2、轮流尝试在每个master节点上创建锁，过期时间设置较短，一般就几十毫秒

3、尝试在大多数节点上建立一个锁，比如5个节点就要求是3个节点（n / 2 +1）

4、客户端计算建立好锁的时间，如果建立锁的时间小于超时时间，就算建立成功了

5、要是锁建立失败了，那么就依次删除这个锁

6、只要别人建立了一把分布式锁，你就得不断轮询去尝试获取锁

但是这样的这种算法还是颇具争议的，可能还会存在不少的问题，无法保证加锁的过程一定正确。



另外，针对redis集群模式的分布式锁，可以采用redis的Redlock机制。

Redlock为了解决单机的问题，需要多个（大于2）redis的master节点，多个master节点互相独立，没有数据同步。

Redlock的实现如下：

1、获取当前时间。

2、依次获取N个节点的锁。每个节点加锁的实现方式同上。这里有个细节，就是每次获取锁的时候的过期时间都不同，需要减去之前获取锁的操作的耗时，

比如传入的锁的过期时间为500ms，

获取第一个节点的锁花了1ms，那么第一个节点的锁的过期时间就是499ms，

获取第二个节点的锁花了2ms，那么第二个节点的锁的过期时间就是497ms

如果锁的过期时间小于等于0了，说明整个获取锁的操作超时了，整个操作失败

1. 判断是否获取锁成功。

如果client在上述步骤中获取到了(N/2 + 1)个节点锁，并且每个锁的过期时间都是大于0的，则获取锁成功，否则失败。失败时释放锁。

1. 释放锁。

对所有节点发送释放锁的指令，每个节点的实现逻辑和上面的简单实现一样。为什么要对所有节点操作？因为分布式场景下从一个节点获取锁失败不代表在那个节点上加速失败，可能实际上加锁已经成功了，但是返回时因为网络抖动超时了。

##### Redisson

实现Redis的分布式锁，除了自己基于redis client原生api来实现之外，还可以使用开源框架：Redission。

Redisson是一个企业级的开源Redis Client，也提供了分布式锁的支持。

#### 缺点

1、单点问题。这里的单点指的是单master，就算是个集群，如果加锁成功后，锁从master复制到slave的时候挂了，也是会出现同一资源被多个client加锁的。

2、执行时间超过了锁的过期时间。为了不出现一直上锁的情况，加了一个兜底的过期时间，时间到了锁自动释放，但是，如果在这期间任务并没有做完怎么办？由于GC或者网络延迟导致的任务时间变长，很难保证任务一定能在锁的过期时间内完成。

即，它获取锁的方式简单粗暴，获取不到锁直接不断尝试获取锁，比较消耗性能。

注：在实际开发中，引入redis机制，其实就是引入一个新的网元，必然存在集群涉及问题，所以DB锁有其优点的地方就是DB本身具备锁，不需要引入过多的网元。

3、redis的设计定位决定了它的数据并不是强一致性的，在某些极端情况下，可能会出现问题。锁的模型不够健壮。

4、即便使用redlock算法来实现，在某些复杂场景下，也无法保证其实现100%没有问题。

但是另一方面使用redis实现分布式锁在很多企业中非常常见，而且大部分情况下都不会遇到所谓的“极端复杂场景”

所以使用redis作为分布式锁也不失为一种好的方案，最重要的一点是redis的性能很高，可以支撑高并发的获取、释放锁操作。

#### 存在问题

##### 高并发场景下的问题

以下问题不是说在并发不高的场景下不容易出现，只是在高并发场景下出现的概率更高些而已。

**性能问题**

性能问题来自于两个方面：

1. 获取锁的时间上。

如果redlock运用在高并发的场景下，存在N个master节点，一个一个去请求，耗时会比较长，从而影响性能。这个好解决。通过上面描述不难发现，从多个节点获取锁的操作并不是一个同步操作，可以是异步操作，这样可以多个节点同时获取。即使是并行处理的，还是得预估好获取锁的时间，保证锁的TTL > 获取锁的时间+任务处理时间。

2、被加锁的资源太大。加锁的方案本身就是会为了正确性而牺牲并发的，牺牲和资源大小成正比。这个时候可以考虑对资源做拆分，拆分的方式有两种：

3、从业务上将锁住的资源拆分成多段，每段分开加锁。比如，我要对一个商户做若干个操作，操作前要锁住这个商户，这时我可以将若干个操作拆成多个独立的步骤分开加锁，提高并发。

4、用分桶的思想，将一个资源拆分成多个桶，一个加锁失败立即尝试下一个。比如批量任务处理的场景，要处理200w个商户的任务，为了提高处理速度，用多个线程，每个线程取100个商户处理，就得给这100个商户加锁，如果不加处理，很难保证同一时刻两个线程加锁的商户没有重叠，这时可以按一个维度，比如某个标签，对商户进行分桶，然后一个任务处理一个分桶，处理完这个分桶再处理下一个分桶，减少竞争。

**重试的问题**

无论是简单实现还是redlock实现，都会有重试的逻辑。如果直接按上面的算法实现，是会存在多个client几乎在同一时刻获取同一个锁，然后每个client都锁住了部分节点，但是没有一个client获取大多数节点的情况。解决的方案也很常见，在重试的时候让多个节点错开，错开的方式就是在重试时间中加一个随机时间。这样并不能根治这个问题，但是可以有效缓解问题，亲试有效。

##### 节点宕机

对于单master节点且没有做持久化的场景，宕机就挂了，这个就必须在实现上支持重复操作，自己做好幂等。

对于多master的场景，比如redlock，我们来看这样一个场景：

1、假设有5个redis的节点：A、B、C、D、E，没有做持久化。

2、client1从A、B、C 3个节点获取锁成功，那么client1获取锁成功。

3、节点C挂了。

4、client2从C、D、E获取锁成功，client2也获取锁成功，那么在同一时刻client1和client2同时获取锁，redlock被玩坏了。

怎么解决呢？最容易想到的方案是打开持久化。持久化可以做到持久化每一条redis命令，但这对性能影响会很大，一般不会采用，如果不采用这种方式，在节点挂的时候肯定会损失小部分的数据，可能我们的锁就在其中。

另一个方案是延迟启动。就是一个节点挂了修复后，不立即加入，而是等待一段时间再加入，等待时间要大于宕机那一刻所有锁的最大TTL。

但这个方案依然不能解决问题，如果在上述步骤3中B和C都挂了呢，那么只剩A、D、E三个节点，从D和E获取锁成功就可以了，还是会出问题。那么只能增加master节点的总量，缓解这个问题了。增加master节点会提高稳定性，但是也增加了成本，需要在两者之间权衡。

##### 任务执行时间超过锁的TTL

之前产线上出现过因为网络延迟导致任务的执行时间远超预期，锁过期，被多个线程执行的情况。

这个问题是所有分布式锁都要面临的问题，包括基于zookeeper和DB实现的分布式锁，这是锁过期了和client不知道锁过期了之间的矛盾。

在加锁的时候，我们一般都会给一个锁的TTL，这是为了防止加锁后client宕机，锁无法被释放的问题。但是所有这种姿势的用法都会面临同一个问题，就是没发保证client的执行时间一定小于锁的TTL。虽然大多数程序员都会乐观的认为这种情况不可能发生，我也曾经这么认为，直到被现实一次又一次的打脸。

Martin Kleppmann也质疑过这一点，这里直接用他的图：



1、Client1获取到锁

2、Client1开始任务，然后发生了STW的GC，时间超过了锁的过期时间

3、Client2 获取到锁，开始了任务

4、Client1的GC结束，继续任务，这个时候Client1和Client2都认为自己获取了锁，都会处理任务，从而发生错误。

Martin Kleppmann举的是GC的例子，我碰到的是网络延迟的情况。不管是哪种情况，不可否认的是这种情况无法避免，一旦出现很容易懵逼。

如何解决呢？一种解决方案是不设置TTL，而是在获取锁成功后，给锁加一个watchdog，watchdog会起一个定时任务，在锁没有被释放且快要过期的时候会续期。

这里也提下Martin Kleppmann的解决方案，他的方案是让加锁的资源自己维护一套保证不会因加锁失败而导致多个client在同一时刻访问同一个资源的情况。



在客户端获取锁的同时，也获取到一个资源的token，这个token是单调递增的，每次在写资源时，都检查当前的token是否是较老的token，如果是就不让写。对于上面的场景，Client1获取锁的同时分配一个33的token，Client2获取锁的时候分配一个34的token，在client1 GC期间，Client2已经写了资源，这时最大的token就是34了，client1 从GC中回来，再带着33的token写资源时，会因为token过期被拒绝。这种做法需要资源那一边提供一个token生成器。

对于这种fencing的方案，我有几点问题：

1、无法保证事务。示意图中画的只有34访问了storage，但是在实际场景中，可能出现在一个任务内多次访问storage的情况，而且必须是原子的。如果client1带着33token在GC前访问过一次storage，然后发生了GC。client2获取到锁，带着34的token也访问了storage，这时两个client写入的数据是否还能保证数据正确？如果不能，那么这种方案就有缺陷，除非storage自己有其他机制可以保证，比如事务机制；如果能，那么这里的token就是多余的，fencing的方案就是多此一举。

2、高并发场景不实用。因为每次只有最大的token能写，这样storage的访问就是线性的，在高并发场景下，这种方式会极大的限制吞吐量，而分布式锁也大多是在这种场景下用的，很矛盾的设计。

3、这是所有分布式锁的问题。这个方案是一个通用的方案，可以和Redlock用，也可以和其他的lock用。所以我理解仅仅是一个和Redlock无关的解决方案。

##### 系统时钟漂移

理论上是可能出现的，实际应用中不一定会出现。

redis的过期时间是依赖系统时钟的，如果时钟漂移过大时会影响到过期时间的计算。

**为什么系统时钟会存在漂移呢？**

先简单说下系统时间，linux提供了两个系统时间：clock realtime和clock monotonic。clock realtime也就是xtime/wall time，这个时间时可以被用户改变的，被NTP改变，gettimeofday拿的就是这个时间，redis的过期计算用的也是这个时间。

clock monotonic，直译过来时单调时间，不会被用户改变，但是会被NTP改变。

最理想的情况时，所有系统的时钟都时时刻刻和NTP服务器保持同步，但这显然时不可能的。导致系统时钟漂移的原因有两个：

1、系统的时钟和NTP服务器不同步，这个目前没有特别好的解决方案。

2、clock realtime被人为修改。在实现分布式锁时，不要使用clock realtime。

### Zookeeper分布式锁

Zookeeper是一种提供配置管理、分布式协同以及命名的中心化服务。

zk的模型是这样的：zk包含一系列的节点，叫做znode，就好像文件系统一样每个znode表示一个目录，然后znode有一些特性：

**有序节点**：假如当前有一个父节点为/lock，我们可以在这个父节点下面创建子节点；

zookeeper提供了一个可选的有序特性，例如我们可以创建子节点“/lock/node-”并且指明有序，那么zookeeper在生成子节点时会根据当前的子节点数量自动添加整数序号

也就是说，如果是第一个创建的子节点，那么生成的子节点为/lock/node-0000000000，下一个节点则为/lock/node-0000000001，依次类推。

**临时节点**：客户端可以建立一个临时节点，在会话结束或者会话超时后，zookeeper会自动删除该节点。

**事件监听**：在读取数据时，我们可以同时对节点设置事件监听，当节点数据或结构变化时，zookeeper会通知客户端。当前zookeeper有如下四种事件：

节点创建

节点删除

节点数据修改

子节点变更

其实基于ZooKeeper，就是使用它的临时有序节点来实现的分布式锁。

注：zookeeper基于Paxos协议改造的ZAB协议实现选举。

#### 原理

当某客户端要进行逻辑的加锁时，就在zookeeper上的某个指定节点的目录下，去生成一个唯一的临时有序节点，然后判断自己是否是这些有序节点中序号最小的一个，如果是，则算是获取了锁。如果不是，则说明没有获取到锁，那么就需要在序列中找到比自己小的那个节点，并对其调用exist()方法，对其注册事件监听，当监听到这个节点被删除了，那就再去判断一次自己当初创建的节点是否变成了序列中最小的。如果是，则获取锁，如果不是，则重复上述步骤。

当释放锁的时候，只需将这个临时节点删除即可。



如图，locker是一个持久节点，node\_1/node\_2/…/node\_n 就是上面说的临时节点，由客户端client去创建的。

client\_1/client\_2/…/clien\_n 都是想去获取锁的客户端。以client\_1为例，它想去获取分布式锁，则需要跑到locker下面去创建临时节点（假如是node\_1）创建完毕后，看一下自己的节点序号是否是locker下面最小的，如果是，则获取了锁。如果不是，则去找到比自己小的那个节点（假如是node\_2），找到后，就监听node\_2，直到node\_2被删除，那么就开始再次判断自己的node\_1是不是序列中最小的，如果是，则获取锁，如果还不是，则继续找一下一个节点。

#### 方案

基于以上的一些zk的特性，我们很容易得出使用zk实现分布式锁的落地方案：

1、使用zk的临时节点和有序节点，每个线程获取锁就是在zk创建一个临时有序的节点，比如在/lock/目录下。

2、创建节点成功后，获取/lock目录下的所有临时节点，再判断当前线程创建的节点是否是所有的节点的序号最小的节点

3、如果当前线程创建的节点是所有节点序号最小的节点，则认为获取锁成功。

4、如果当前线程创建的节点不是所有节点序号最小的节点，则对节点序号的前一个节点添加一个事件监听。

比如当前线程获取到的节点序号为/lock/003,然后所有的节点列表为[/lock/001,/lock/002,/lock/003],则对/lock/002这个节点添加一个事件监听器。

如果锁释放了，会唤醒下一个序号的节点，然后重新执行第3步，判断是否自己的节点序号是最小。

比如/lock/001释放了，/lock/002监听到时间，此时节点集合为[/lock/002,/lock/003],则/lock/002为最小序号节点，获取到锁。

整个过程如下：



#### 特点

1. 统一视图（多个客户端连接访问不同节点，数据保持一致）；
2. Zookeeper可以存储数据；
3. Zookeeper目录类型分为4种：
4. 持久化目录：客户端断开连接后不会删除
5. 临时目录：客户端断开连接后自动删除
6. 持久有顺序目录
7. 临时有顺序目录
8. Zookeeper支持事件回调

zk节点目录下的文件会注册一个事件，如果该事件发生，则会调用客户端的回调函数执行相应的操作。

#### 缺点

对于zk分布式锁而言:

zookeeper天生设计定位就是分布式协调，强一致性。锁的模型健壮、简单易用、适合做分布式锁。

如果获取不到锁，只需要添加一个监听器就可以了，不用一直轮询，性能消耗较小。

但是zk也有其缺点：如果**有较多的客户端频繁的申请加锁、释放锁，对于zk集群的压力会比较大**。

## 选择

比较推崇zk实现的锁：

因为**redis是有可能存在隐患的，可能会导致数据不对的情况**。但是，怎么选用要看具体在公司的场景了。

如果公司里面有zk集群条件，优先选用zk实现，但是如果说公司里面只有redis集群，没有条件搭建zk集群。

那么其实用redis来实现也可以，另外还可能是系统设计者考虑到了系统已经有redis，但是又不希望再次引入一些外部依赖的情况下，可以选用redis。

这个是要系统设计者基于架构的考虑了。

# MySQL数据库锁

锁的分类：

宏观：数据库锁（粒度小，方便用于集群环境），代码锁（粒度大，需要封装）；

微观：行锁表锁

## 分类

### 粒度锁

**根据锁的粒度，可以分为：**

记录锁：锁住行

表锁：锁住表

页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

数据库锁：锁住整个库

### 算法锁

算法锁（基于行锁的算法）：**记录锁，间隙锁，临键锁**。

注：记录锁、间隙锁、临键锁，都属于排它锁。

### 属性锁

**在MySQL中，锁可以分为两类：**

共享锁Shared Locks（简称S锁，属于行锁）：共享锁是将对象数据变为只读形式，不能进行更新，所以也称为读取锁定；

排他锁Exclusive Locks（简称X锁，属于行锁）：排他锁是当执行INSERT/UPDATE/DELETE的时候，其他事务不能读取该数据，因此也称为写入锁定；

**意向锁**

当一个事务试图对整个表进行加锁（共享锁或排它锁）之前，首先需要获得对应类型的意向锁（意向共享锁或意向共享锁）。

意向共享锁Intention Shared Locks（简称IS锁，属于表锁），表示事务准备给数据行加上共享锁，也就是说一个数据行在加共享锁之前必须先取得该表的IS锁；

意向排他锁Intention Exclusive Locks（简称IX锁，属于表锁），表示事务准备给数据行加上排他锁，也就是说一个数据行加排他锁之前必须先取得该表的IX锁。

**注：**意向锁是InnoDB数据操作之前自动加的，不需要用户干涉。

自增锁AUTO-INC Locks，针对自增列。

### 状态锁

基于属性锁的状态：意向共享锁，意向排它锁。

## 并发

**基于锁的并发控制流程：**

1、事务根据自己对数据项进行的操作类型申请相应的锁（读申请共享锁，写申请排他锁）；

2、申请锁的请求被发送给锁管理器，锁管理器根据当前数据项是否已经有锁以及申请的和持有的锁是否存在冲突，决定是否为该请求授予锁；

3、若锁被授予，则申请锁的事务可以继续执行；若被拒绝，则申请锁的事务将进行等待，直到锁被其他事务释放。

**可能出现的问题：**

1、死锁：多个事务持有锁并互相循环等待其他事务的锁导致所有事务都无法继续执行。

2、饥饿：数据项A一直被加共享锁，导致事务一直无法获取A的排他锁。

对于可能发生冲突的并发操作，锁使它们由并行变为串行执行，是一种悲观的并发控制。

**拓展：**除了锁可以实现并发控制之外，还有其他策略：

基于时间戳的并发控制

基于有效性检查的并发控制

基于快照隔离MVCC的并发控制

## 读写锁

根据读写行为可以分为读锁和写锁。

读锁：共享锁、shared locks、S锁

写锁：排他锁、exclusive locks、X锁

Select：不加锁

|  |  |  |
| --- | --- | --- |
|  | X锁 | S锁 |
| X锁 | 冲突 | 冲突 |
| S锁 | 冲突 | 不冲突 |

读写操作中涉及的锁：

### 读锁

对于普通SELECT语句，InnoDB不会加任何锁；

select ... lock in share mode：共享锁

select ... for update：排他锁

### 写锁

1、DELETE：删除一条数据时，先对记录加X锁，再执行删除操作；

2、INSERT：插入一条记录时，会先加“隐式锁”来保护这条新插入的记录在本事务提交前不被别的事务访问；

3、UPDATE：

如果被更新的列，修改前后没有导致存储空间变化，那么会先给记录加X锁，再直接对记录进行修改；

如果被更新的列，修改前后导致存储空间发生了变化，那么会先给记录加X锁，然后将记录删除，再INSERT一条新记录。

**隐式锁：**一个事务插入一条记录后，还未提交，这条记录会保存本次事务id，而其他事务如果想来对这个记录加锁时（比如执行update、delete操作）会发现事务id不对应，这时会产生X锁，所以相当于再插入一条记录时，隐式地给这条记录加了一把隐式X锁。

## 粒度锁/范围锁

### 表锁

#### 概述

表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。

最常使用的MYISAM与INNODB都支持表级锁定（行锁失效时使用表锁）。

**注：表锁是非索引字段**，即全表扫描，全表扫描时锁定整张表，sql语句可以通过执行计划看出扫描了多少条记录。

比如，先执行操作A：update tb set A=A+1 where B=1;，然后执行操作B：update tb set A=A+2 where B=2;字段B为非索引字段，此时A是独占锁，锁住整个表，而不是锁住B=1的行，所以此时操作B是阻塞的。

如果将字段B建立索引，则执行A锁住行，B不会阻塞，会正常执行。

**使用表锁的情况：**

**主键不明确：**select \* from table where pk <>1 for update;

**where字段不是索引：**select \* from table where normalkey=1;

即，**当where条件中的字段没有加索引时，会锁住整张表；在有索引的情况下，更新不同的行，InnoDB默认的行锁是不会阻塞的**。

#### 兼容性

对ＭyISAM表的读操作，不会阻塞其他用户对同一表的读请求，但会阻塞对同一表的写请求；

对ＭyISAM表的写操作，则会阻塞其他用户对同一表的读和写请求；

ＭyISAM表的读和写操作之间，以及写和写操作之间是串行的！(当某一线程获得对一个表的写锁后，只有持有锁的线程可以对表进行更新操作。其他线程的读、写操作都会等待，直到锁被释放为止)

#### 特点

开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低。

#### 分类

MySQL的表级锁有两种模式：表共享读锁（Table Read Lock）和表独占写锁（Table Write Lock）。

##### 表共享读锁

##### 表独占写锁

使用lock table/unlock table进行加锁/解锁。

#### 查询

检查table\_locks\_waited和table\_locks\_immediate状态变量分析

1、table\_locks\_immediate：可以立即获取锁的次数

2、table\_locks\_waited：不能立即获取锁，需要等待锁的次数

table\_locks\_waited 的值越高，则说明存在严重的表级锁的争用情况。

#### 加锁

执行select前，会自动给涉及的所有表加读

执行更新（update，delete，insert)会自动给涉及到的表加写

不需要用户直接显式用lock table命令

对于给MyISAM显式加锁，一般是为了在一定程度上模拟事务操作，实现对某一个时间点多个表一致性读取。

#### 算法

##### 意向锁/升级机制

当一个事务带着表锁去访问一个被加了行锁的资源，那么，此时，这个行锁就会升级成意向锁，将表锁住。

事务A（升级表锁）：select \* from user where id=1 for update;

事务B（表锁）：select \* from user where name like = ‘kkk’ for update;

注：**意向锁必然是表锁！**

##### 自增锁

事务插入自增类型的列时，获取自增锁（比如oracle sequence，可以将表和锁分离）。

如果一个事务正在往表中插入自增记录，其他事务都必须等待。

### 行锁

#### 概述

行级锁是Mysql中锁定**粒度最细**的一种锁（***该锁是对索引记录进行加锁，锁是在加索引上而不是行上的***。innodb一定存在聚簇索引，因此**行锁最终都会落到聚簇索引上！**），表示只针对当前操作的行（某一行或多行）进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。

#### 特点

开销大，加锁慢；**会出现死锁**；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

#### 分类

行级锁分为共享锁（lock in shared mode）和排他锁（for update）。

InnoDB实现了两种类型的行锁：

##### 共享锁

共享锁（S）：又称**读锁**。允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。

所谓的共享锁，就是多个事务只能读数据不能修改数据。加共享锁可以使用select…lock in share mode语句。

若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其他事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁。这保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。

select…lock in share mode

将查找到的数据加上一个S锁，允许其他事务继续获取这些记录的S锁，不能获取这些记录的X锁（会阻塞）。

**使用场景：**读出数据后，其他事务不能修改，但是**自己也不一定能修改，因为其他事务也可以使用“select…lock in share mode”继续加读锁**。

注：MySQL8.0以上，for share代替了lock in share mode，仍然支持lock in share mode。但是，存在跳锁skip locked，等待nowait，配合自旋锁，可以高效实现一个等待队列。

##### 排他锁

排他锁（X）：又称**写锁**。允许获取排他锁的事务更新数据，阻止其他事务获取相同的数据集共享读锁和排他写锁。

若事务T对数据对象A加上X锁，事务T可以读A也可以修改A，其他事务不能再对A加任何锁，知道T释放A上的锁。

排他锁指的是一个事务在一行数据加上排他锁后，其他事务不能再在其上加上其他的锁。**InnoDB引擎默认的修改数据语句：update、delete、insert都会自动给涉及到的数据加上排他锁**。所以加过排他锁的数据行在其他事务中是不能修改数据的，也不能通过for update和lock in share mode锁的方式查询数据，但是可以直接通过select…from…查询数据，因为普通查询没有任何锁限制。

我们通过update、delete等语句加上的锁都是行级别的锁。只有LOCK TABLE…READ和LOCK TABLE…WRITE才能申请表级别的锁。

另外，为了允许行锁和表锁共存，实现多粒度的锁机制，InnoDB还有两种内部使用的意向锁（Intention Locks），这两种意向锁都是表锁。

**对于普通的SELECT语句，InnoDB不会加任何锁！**

1、SELECT：select…for update

将读到的数据加上一个X锁，不允许其他事务获取这些记录的S锁和X锁。

使用场景：读出数据后，其他事务既不能写，也不能加读锁，那么就导致只有自己可以修改数据。

2、DELETE：删除一条数据时，先对记录加X锁，再执行删除操作

3、INSERT：插入一条记录时，会先加“隐式锁”来保护这条新插入的记录在本事务提交前不被别的事务访问到。

4、UPDATE：

如果被更新的列，修改前后没有导致存储空间变化，那么会先给记录加X锁，再直接对记录进行修改。

如果被更新的列，修改前后导致存储空间发生了变化，那么会先给记录加X锁，然后将记录删掉，再INSERT一条新纪录。

注：隐式锁：一个事务插入一条记录后，还未提交，这条记录会保存本地事务id，而其他事务如果想来对这个记录加锁时会发现事务id不响应，这时会产生X锁，所以相当于再插入一条记录时，隐式的给这条记录加了一把隐式X锁。

根据锁生效范围可以分为：行级锁、表级锁、页级锁（这都是理论上的锁，不是实际真正语法上的锁）。

InnoDB存储引擎既支持行级锁（row-level locking），也支持表级锁，但默认情况下采用行级锁。

乐观锁和悲观锁，不管是什么锁都需要加失败重试。

##### 意向共享锁

意向共享锁（IS）：事务打算给数据行共享锁，事务在给一个数据行加共享锁前必须先取得该表的IS锁。IS属于表锁。

##### 意向排他锁

意向排他锁（IX）：事务打算给数据行加排他锁，事务在给一个数据行加排他锁前必须先取得该表的IX锁。IX锁属于表锁。

**意向共享锁**

意向共享锁（IS锁）：一个事务在获取（任何一行/或者全表）S锁之前，一定会先在所在的表上加IS锁。

**意向排他锁**

意向排他锁（IX锁）：一个事务在获取（任何一行/或者全表）X锁之前，一定会先在所在的表上加IX锁。

意向锁存在的目的：假设事务T1，用X锁来锁住了表上的几条记录，那么此时表上存在IX锁，即意向排他锁。那么此时事务T2要进行LOCK TABLE … WRITE的表级别锁的请求，可以直接根据意向锁是否存在而判断是否有锁冲突。

**意向锁也是数据库隐式帮我们做了，不需要程序员操心！**

#### 算法

行锁根据具体算法衍生出记录锁、间隙锁、临键锁。

##### Record Lock（普通行锁）

##### Gap Lock（间隙锁）

##### Next Key（行&间隙锁）

#### 行锁失效

**行级锁都是基于索引的，如果一条SQL语句用不到索引是不会使用行级锁的，会使用表锁**。

**只有InnoDB存储引擎存在行锁！**

注：**只有明确指定主键/索引，才会执行行锁，否则执行表锁**。

1. 无锁

主键不存在：select \* from table where pk=1 for update;

注：for update的不一定都是行锁，有可能是无锁。

1. 行锁
2. 表锁

主键不明确：select \* from table where pk <>1 for update;

where字段不是索引：select \* from table where normalkey=1;

注：**当where条件查询中字段没有加索引时，会锁住整张表。**

**在有索引的情况下，更新不同的行，InnoDB默认的行锁是不会阻塞的。**

**存在索引，但是索引失效，行锁转表锁。**

#### 死锁

##### 概述

**MyISAM中是不会产生死锁的**，因为MyISAM总是一次性获得所需的全部锁，要么全部满足，要么全部等待。而**在InnoDB中，锁是逐步获得的，就造成了死锁的可能**。

在MySQL中，行级锁并不是直接锁记录，而是**锁索引**。索引分为主键索引和非主键索引两种，如果一条sql语句操作了主键索引，MySQL就会锁定这条主键索引；如果一条语句操作了非主键索引，MySQL会先锁定该非主键索引，再锁定相关的主键索引。在UPDATE、DELETE操作时，MySQL不仅锁定WHERE条件扫描过的所有索引记录，而且会锁定相邻的键值，即所谓的next-key locking。

当两个事务同时执行，一个锁住了主键索引，在等待其他相关索引。另一个锁定了非主键索引，在等待主键索引。这样就会发生死锁。发生死锁后，InnoDB一般都可以检测到，并使一个事务释放锁回退，另一个获取锁完成事务。但在涉及外部锁，或涉及锁的情况下，InnoDB并不能完全自动检测到死锁，这需要通过设置锁等待超时参数innodb\_lock\_wait\_timeout来解决。需要说明的是，这个参数并不是只用来解决死锁问题，在并发访问比较高的情况下，如果大量事务因无法立即获取所需的锁而挂起，会占用大量计算机资源，造成严重性能问题，甚至拖垮数据库。我们通过设置合适的锁等待超时阈值，可以避免这种情况发生。

##### 检查

如果出现死锁，可以用SHOW INNODB STATUS命令来确定最后一个死锁产生的原因和改进措施。

##### 解决

通常来说，死锁都是应用设计的问题，通过调整业务流程、数据库对象设计、事务大小、以及访问数据库的SQL语句，绝大部分都可以避免。

有多种方法可以避免死锁，这里只介绍常见的三种：

1. 如果不同程序会并发存取多个表，**尽量约定以相同的顺序访问表**，可以大大降低死锁机会；
2. 在事务中，要更新记录，应直接申请排他锁,而不应该先申请共享锁；
3. 在同一个事务中，**尽可能做到一次锁定所需要的所有资源**，减少死锁产生概率；
4. 对于非常容易产生死锁的业务部分，**可以尝试使用升级锁定颗粒度**，通过表级锁定来减少死锁产生的概率；
5. 当事务隔离级别为READ REPEATED可重复读时，如果两个线程同时对相同条件记录用SELECT...ROR UPDATE加排他写锁，在没有符合该记录情况下，两个线程都会加锁成功，程序发现记录尚不存在，就试图插入一条新记录，如果两个线程都这么做，就会出现死锁，这种情况下，将隔离级别改成READ COMMITTED，就可以避免问题；
6. 当隔离级别为READ COMMITED时，如果两个线程都先执行SELECT...FOR UPDATE 判断是否存在符合条件的记录，没有则插入记录；此时，只有一个线程能插入成功，另一个线程会出现锁等待。当第１个线程提交后，第２个线程会因主键重出错，但虽然这个线程出错了，却会获得一个排他锁！这时如果有第３个线程又来申请排他锁，也会出现死锁。对于这种情况，可以直接做插入操作，然后再捕获主键重异常，或者在遇到主键重错误时，总是执行ROLLBACK释放获得的排他锁。

##### 使用建议

在了解InnoDB锁特性后，用户可以通过设计和SQL调整等措施减少锁冲突和死锁，包括：

1. 尽量使用较低的隔离级别；精心设计索引，并尽量使用索引访问数据，使加锁更精确，从而减少锁冲突的机会；
2. 选择合理的事务大小，小事务发生锁冲突的几率也更小；
3. 给记录集显式加锁时，最好一次性请求足够级别的锁。比如要修改数据的话，最好直接申请排他锁，而不是先申请共享锁，修改时再请求排他锁，这样容易产生死锁；
4. 不同的程序访问一组表时，应尽量约定以相同的顺序访问各表，对一个表而言，尽可能以固定的顺序存取表中的行。这样可以大大减少死锁的机会；
5. 尽量用相等条件访问数据，这样可以避免间隙锁对并发插入的影响； 不要申请超过实际需要的锁级别；除非必须，查询时不要显示加锁；
6. 对于一些特定的事务，可以使用表锁来提高处理速度或减少死锁的可能。

#### 对比

什么时候使用行锁和表锁？

InnoDB行锁是通过给索引上的索引项加锁来实现的，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！

在实际应用中，要特别注意InnoDB行锁的这一特性，不然的话，可能导致大量的锁冲突，从而影响并发性能。

在不通过索引条件查询的时候,InnoDB 确实使用的是表锁,而不是行锁。

由于 MySQL 的行锁是针对索引加的锁,不是针对记录加的锁,所以虽然是访问不同行的记录,但是如果是使用相同的索引键,是会出现锁冲突的。应用设计的时候要注意这一点。

当表有多个索引的时候,不同的事务可以使用不同的索引锁定不同的行,另外,不论 是使用主键索引、唯一索引或普通索引,InnoDB 都会使用行锁来对数据加锁。

即便在条件中使用了索引字段,但是否使用索引来检索数据是由 MySQL 通过判断不同 执行计划的代价来决定的,如果 MySQL 认为全表扫效率更高,比如对一些很小的表,它就不会使用索引,这种情况下 InnoDB 将使用表锁,而不是行锁。因此,在分析锁冲突时, 别忘了检查 SQL 的执行计划,以确认是否真正使用了索引。

### 页锁

#### 概述

页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。

BDB支持页级锁。

#### 特点

开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

#### 分类

### 使用

MyISAM和MEMORY采用表级锁(table-level locking)。

BDB采用页面锁(page-level locking)或表级锁，默认为页面锁。

InnoDB支持行级锁(row-level locking)和表级锁，默认为行级锁。

#### 锁选择

Read Uncommited(RU)：读未提交，一个事务可以读到另一个事务未提交的数据。

Read Committed (RC)：读已提交，一个事务可以读到另一个事务已提交的数据。

Repeatable Read (RR):可重复读，加入间隙锁，一定程度上避免了幻读的产生！注意了，只是一定程度上，并没有完全避免!另外就是记住从该级别才开始加入间隙锁。

Serializable：串行化，该级别下读写串行化，且所有的select语句后都自动加上lock in share mode，即使用了共享锁。因此在该隔离级别下，使用的是当前读，而不是快照读。

那么关于是表锁还是行锁，一个说法是这样的，InnoDB行锁是通过给索引上的索引项加锁来实现的，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。 InnoDB这种行锁实现特点意味着：**只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！**

#### 性能对比

行级锁：开销大，加锁慢，会出现死锁，锁粒度小，发生锁冲突的概率最低，并发度最高。

表级锁：开销小，加锁快，不会产生死锁，锁粒度大，发生锁冲突的概率最高，并发度最低。

页面锁：开销和加锁时间介于表锁和行锁之间，会产生死锁，锁粒度介于表锁和行锁之间，并发度一般。

综上所述，很难笼统地说哪种锁更好，只能就具体应用的特点选择合适的锁！仅从锁的角度来看，表级锁更适合查询为主，只有少量按索引更新数据的应用场景，如web应用；而行级锁则更适合于有大量按照索引条件并发更新少量不同数据，同时又有并发查询的应用，如一些在线事务处理（OLAP）系统。

## 算法锁

行锁根据算法不同衍生出记录锁、间隙锁、Next Key。

LOCK\_REC\_NOT\_GAP：单个行记录上的锁；

LOCK\_GAP：间隙锁，锁定一个范围，但不包括记录本身，GAP锁的目的，是为了防止同一事务的两次当前读，出现幻读的情况；

LOCK\_ORIINARY：锁定一个范围，并且锁定记录本身，对于行的查询，都是采用该方法，主要目的是解决幻读的问题。

### Record Lock/普通行锁

记录锁/普通行锁：事务加锁后锁住的只是表的某一条记录。



**条件：精准条件**命中，并且命中的条件字段是**唯一索引**；

满足如下条件使用普通行锁：

1. **键值在条件范围内**
2. **记录存在**

**例如：**update user\_info set name=’张三’ where id=1，这里的id是唯一索引。

**作用：**加了记录锁之后可以避免数据在查询的时候被修改的**重复读问题**，也避免了在修改的事务未提交前被其他事务读取的**脏读问题**。

### Gap Lock/间隙锁

当我们用范围条件检索数据而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合范围条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP)”。InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁。

注意：**间隙锁只会在 Repeatable read隔离级别下使用**。

对于键值不存在条件范围内，叫做“间隙”（GAP），引擎就会对这个“间隙”加锁，这种机制就是Gap机制。

**间隙锁/区间锁**：事务加锁后锁住的是表记录的某一个区间（隶属于行锁），当表的相邻ID之间出现空隙则会形成一个区间，遵循**左开右闭**原则。



比如下面的表里面的数据ID为1,4,5,7,10,那么会形成以下几个间隙区间，-n-1区间，1-4区间，7-10区间，10-n区间（-n代表负无穷大，n代表正无穷大）。

**条件：**

1. 范围查询（非等值查找）并且查询**未命中记录**；
2. 查询条件**必须命中索引；**
3. **间隙锁只会出现在REPEATABLE\_READ（重复读)的事务级别中**。

注：**MySQL默认事务隔离级别就是RR**，所以可以借助**间隙锁消除幻读（默认的行锁是临键锁，已经包含了间隙锁）**。

**例如：**对应上图的表执行select \* from user\_info where id>1 and id<4(这里的id是唯一索引) ，这个SQL查询不到对应的记录，那么此时会使用间隙锁。

**作用：防止幻读问题**，事务并发的时候，如果没有间隙锁，就会发生如下图的问题，在同一个事务里，A事务的两次查询出的结果会不一样。



注：数据库可重复读事务隔离级别中insert时采用间隙锁，防止幻读（因为插入的数据必然在记录的附近，所以需要加间隙锁）。

总结，InnoDB使用间隙锁的目的有两个：

1、为了防止幻读(Repeatableread隔离级别下再通过GAP锁即可避免了幻读)

2、满足恢复和复制的需要

MySQL的恢复机制要求：在一个事务未提交前，其他并发事务不能插入满足其锁定条件的任何记录，也就是不允许出现幻读。

### Next Key/临键锁

**临键锁（行-间隙锁）是InnoDB的行锁默认算法**，它是记录锁和间隙锁的组合，临键锁会把查询出来的记录锁住，同时也会把该范围查询内的所有间隙空间也会锁住，再之它会把相邻的下一个区间也会锁住。

注：间隙锁所锁定的区间是一个左开右闭的集合，而临键锁锁定是当前记录的区间和下一个记录的区间。

比如下面表的数据执行select \* from user\_info where id>1 and id<=13 for update ;

会锁住ID为1，5，10的记录；

同时会锁住，1至5，5至10，10至15的区间。



**条件：**

1. 范围查询并命中；
2. 查询命中了索引。

注：行锁命中索引和记录，间隙锁命中索引，未命中范围，临键锁命中索引，且命中范围查找。

**作用：**结合记录锁和间隙锁的特性，临键锁避免了在范围查询时出现脏读、重复读、幻读问题。**加了临键锁之后，在范围区间内数据不允许被修改和插入**。

**注：临键锁的主要目的，也是为了避免幻读(Phantom Read)。如果把事务的隔离级别降级为RC，临键锁则也会失效（事务隔离级别为RR）。**

注：

#id只有50

select \* from table where id>49 for update;

注：id=50为行锁，>50为间隙锁。

## 属性锁

行锁和表锁是粒度的概念，共享锁和排他锁是它们的具体实现。

### 共享锁

共享锁Shared Locks（简称S锁，属于行锁）：共享锁是将对象数据变为只读形式，不能进行更新，所以也称为读取锁定；

**注：**共享锁的特性主要是为了支持并发的读取数据，读取数据的时候不支持修改，避免出现重复读的问题。

MySQL8.0以上，for share代替了lock in share mode，但是仍然支持lock in share mode，但是可以实现nowait，skip lock，配个自旋锁，可以高效地实现一个等待队列。

### 排他锁

排他锁Exclusive Locks（简称X锁，属于行锁）：排他锁是当执行INSERT/UPDATE/DELETE的时候，其他事务不能读取该数据，因此也称为写入锁定；

注：排他锁的目的是在数据修改时候，不允许其他人同时修改，也不允许其他人读取。避免了出现脏数据和脏读的问题。

### 意向锁

当一个事务试图对整个表进行加锁（共享锁或排它锁）之前，首先需要获得对应类型的意向锁（意向共享锁或意向共享锁）。

意向共享锁Intention Shared Locks（简称IS锁，属于表锁），表示事务准备给数据行加上共享锁，也就是说一个数据行在加共享锁之前必须先取得该表的IS锁；

意向排他锁Intention Exclusive Locks（简称IX锁，属于表锁），表示事务准备给数据行加上排他锁，也就是说一个数据行加排他锁之前必须先取得该表的IX锁。

**注：**意向锁是InnoDB数据操作之前自动加的，不需要用户干涉。

自增锁AUTO-INC Locks，针对自增列。

**为什么需要意向锁？**

1. 事务A对user\_info表执行一个SQL:update user\_info set name =”张三” where id=6 加锁情况如下图：



1. 与此同时数据库又接收到事务B修改数据的请求：SQL: update user\_info set name =”李四”；

1、因为事务B是对整个表进行修改操作，那么此SQL是需要对整个表进行加排它锁的（update加锁类型为排他锁）；

2、我们首先做的第一件事是先检查这个表有没有被别的事务锁住，只要有事务对表里的任何一行数据加了共享锁或排他锁我们就无法对整个表加锁（排他锁不能与任何属性的锁兼容）。

3、因为INNODB锁的机制是基于行锁，那么这个时候我们会对整个索引每个节点一个个检查，我们需要检查每个节点是否被别的事务加了共享锁或排它锁。

4、最后检查到索引ID为6的节点被事务A锁住了，最后导致事务B只能等待事务A锁的释放才能进行加锁操作。

**思考：**

在A事务的操作过程中，后面的每个需要对user\_info加持表锁的事务都需要遍历整个索引树才能知道自己是否能够进行加锁，这种方式是不是太浪费时间和损耗数据库性能了？

所以InnoDB就加了意向锁的概念：如果当事务A加锁成功之后就设置一个状态告诉后面的人，已经有人对表里的行加了一个排他锁了，你们不能对整个表加共享锁或排它锁了，那么后面需要对整个表加锁的人只需要获取这个状态就知道自己是不是可以对表加锁，避免了对整个索引树的每个节点扫描是否加锁。

## 实现/锁模式

行锁和表锁其实是粒度上的概念，共享锁和排他锁是它们的具体实现。

乐观锁其实是一种思想，正如其名：认为不会锁定的情况下去更新数据，如果发现不对劲，才不更新(回滚)。在数据库中往往添加一个version字段来实现。

悲观锁用的就是数据库的行锁，认为数据库会发生并发冲突，直接上来就把数据锁住，其他事务不能修改，直至提交了当前事务。

### 乐观锁

#### 概述

乐观锁的特点是先进行业务操作，不到万不得已不去拿锁，即“乐观”的认为拿锁多半是成功的，因此在进行完业务操作需要实际更新数据的最后一步再去拿一下锁就好。

#### 原理

乐观锁机制其实就是在数据库表中引入一个**版本号（version）字段**来实现的。

当我们要从数据库中读取数据的时候，同时把这个version字段也读出来，如果要对读出来的数据进行更新后写回数据库，则需要将version加1，同时将新的数据与新的version更新到数据表中，且必须在更新的时候同时检查目前数据库里version值是不是之前的那个version，如果是，则正常更新。如果不是，则更新失败，说明在这个过程中有其它的进程去更新过数据了。



如图，假设同一个账户，用户A和用户B都要去进行取款操作，账户的原始余额是2000，用户A要去取1500，用户B要去取1000，如果没有锁机制的话，在并发的情况下，可能会出现余额同时被扣1500和1000，导致最终余额的不正确甚至是负数。但如果这里用到乐观锁机制，当两个用户去数据库中读取余额的时候，除了读取到2000余额以外，还读取了当前的版本号version=1，等用户A或用户B去修改数据库余额的时候，无论谁先操作，都会将版本号加1，即version=2，那么另外一个用户去更新的时候就发现版本号不对，已经变成2了，不是当初读出来时候的1，那么本次更新失败，就得重新去读取最新的数据库余额。

通过上面这个例子可以看出来，使用「乐观锁」机制，必须得满足：

（1）锁服务要有**递增**的版本号version

（2）每次更新数据的时候都必须先判断版本号对不对，然后再写入新的版本号

#### 实现

乐观锁在数据库上的实现完全是逻辑的，不需要数据库提供特殊的支持。

乐观锁采用**版本号（或者时间戳）的和CAS**方式，即当前版本号如果对应上了就可以写入数据，如果判断当前版本号不一致，那么就不会更新成功。

比如：

update table

set

column = value

where

version=${version}

and

otherKey = ${otherKey}

实现步骤：

1. select data as lod\_data,version as old\_version from…
2. 根据获取的数据进行业务操作，得到new\_data和new\_version
3. update set data=new\_data,version=new\_version where version=old\_version

if(update row > 0){

//乐观锁获取成功，操作完成

}else{

//乐观锁获取失败，回滚并重试

}

乐观锁是否在事务中其实都是无所谓的，其底层机制是这样的：在数据库内部update同一行的时候是不允许并发的，即数据库每次执行一条update语句的时候会获取被update行的写锁，直到这一行被成功更新后才释放。因此在业务操作进行前获取需要锁的数据的当前版本号，然后实际更新数据时再次对比版本号确认与之前获取的时间，并更新版本号，即可确认这之间没有发生并发的修改。如果更新失败即可认为老版本的数据已经被并发修改掉而不存在了，此时认为获取锁失败，需要回滚整个业务操作并可根据需要重试整个过程。

#### 适用场景

乐观锁适用于**写少读多**的情景，因为这种乐观锁相当于JAVA的CAS，所以多条数据同时过来的时候，不用等待，可以立即进行返回。

悲观锁适用于**写多读少**的情景，这种情况也相当于JAVA的synchronized，reentrantLock等，大量数据过来的时候，只有一条数据可以被写入，其他的数据需要等待。执行完成后下一条数据可以继续。

### 悲观锁

#### 概述

总是假设最坏的情况，每次取数据时都认为其他线程会修改，所以都会加锁（读锁、写锁、行锁等），当其他线程想要访问数据时，都需要阻塞挂起。可以根据数据库实现，如行锁、读锁和写锁等，都是在操作之前加锁。在Java中，synchronize的思想也是悲观锁。

注：要使用悲观锁，我们必须关闭MySQL数据库的自动提交属性。因为MySQL默认使用autocommit模式（自动提交事务），也就是说，当我们执行一个更新操作后，MySQL会立刻将结果进行提交。

**MySQL基本都是使用悲观锁，共享锁和排他锁都是属于悲观锁（可以理解为悲观锁的具体实现）**。

#### 原理

悲观锁也叫作排它锁，在Mysql中是基于 for update 来实现加锁的，例如：

//锁定的方法-伪代码

public boolean lock(){

connection.setAutoCommit(false)

for(){

result =

select \* from user where

id = 100 for update;

if(result){

//结果不为空，

//则说明获取到了锁

return true;

}

//没有获取到锁，继续获取

sleep(1000);

}

return false;

}

//释放锁-伪代码

connection.commit();

上面的示例中，user表中，id是主键，通过 for update 操作，数据库在查询的时候就会给这条记录加上排它锁。

（需要注意的是，在InnoDB中只有字段加了索引的，才会是行级锁，否者是表级锁，所以这个id字段要加索引）

当这条记录加上排它锁之后，其它线程是无法操作这条记录的。

那么，这样的话，我们就可以认为获得了排它锁的这个线程是拥有了分布式锁，然后就可以执行我们想要做的业务逻辑，当逻辑完成之后，再调用上述释放锁的语句即可。

#### 实现

悲观锁也叫作排它锁，悲观锁实现的机制一般是在执行更新语句的时候采用for update方式。

比如：

update table

set

column=

'value'

for

update

这种情况where条件一定要涉及到数据库对应的索引字段，这样才会是行级锁，否则会是表锁，这样执行速度会变慢。

#### 适用场景

综上所述，乐观锁在不发生取锁失败的情况下开销比悲观锁小，但是一旦发生失败回滚开销则比较大，因此适合用在取锁失败概率比较小的场景，可以提升系统并发性能。乐观锁还适用于一些比较特殊的场景，例如在业务操作过程中无法和数据库保持连接等悲观锁无法适用的地方。

# 索引与锁

行锁是基于索引实现的，如果索引失效，则行锁转表锁。

在了解InnoDB的锁特性后，用户可以通过设计和SQL调整等措施减少锁冲突和死锁：

1. 尽量使用较低的隔离级别

选择合理的事务大小，小事务发生锁冲突的几率也更小

2、精心设计索引，并尽量使用索引访问数据，使加锁更精确，从而减少锁冲突的机会

利用索引优化锁：

索引可以减少锁定的行数

索引可以加快处理速度，同时也加快锁的释放

3、给记录集显式加锁时，最好一次性请求足够级别的锁。比如要修改数据的话，最好直接申请排他锁，而不是先申请共享锁，修改时再请求排他锁，这样容易产生死锁。

4、不同的程序访问一组表时，应尽量约定以相同的顺序访问各表，对一个表而言，尽可能以固定的顺序存取表中的行。这样可以大减少死锁的机会。

5、尽量用相等条件访问数据，这样可以避免间隙锁对并发插入的影响。

6、不要申请超过实际需要的锁级别；除非必须，查询时不要显示加锁。

7、对于一些特定的事务，可以使用表锁来提高处理速度或减少死锁的可能

# 锁与事务

数据库事务隔离通过锁实现，锁又可以通过表锁或者行锁实现。