# 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。

## 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

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

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

基于数据库的悲观锁

### Redis锁

基于Redis实现的锁机制，主要是依赖redis自身的原子操作，例如：

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

上述代码示例是指，当redis中不存在user\_key这个键的时候，才会去设置一个user\_key键，并且给这个键的值设置为 user\_value，且这个键的存活时间为100ms。

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。

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

### Zookeeper分布式锁

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。

原理就是：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。

# MySQL数据库锁

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 |

## 分类

**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

**根据锁的粒度，可以分为：**

记录锁：锁住行

表锁：锁住表

数据库锁：锁住整个库

## 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

基于快照隔离的并发控制

## 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。

表级锁定分为表共享读锁与表独占写锁。

### 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。

### 算法

#### 意向锁

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。

#### 自增锁

事务插入自增类型的列时，获取自增锁。

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。

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

注：只有明确指定主键，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;

### 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

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。

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。

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。

**对于普通的SELECT语句，InnoDB不会加任何锁**

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。

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。

### 算法

#### Record Lock（普通行锁）

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

#### Gap Lock（间隙锁）

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。

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）。

#### Next Key（行&间隙锁）

在键值范围条件内，同时键值又不存在条件范围内。

#id只有50

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

注：id=50为行锁，>50为间隙锁。

### 死锁

**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。

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；

3、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；

### 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

## 总结

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，大家可以看到网上最流传的一个说法是这样的，

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。

## 实现

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

#### 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。

#### 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。