# MySQL数据库锁

数据库事务ACID中的隔离性是通过锁和MVCC实现的，锁用于并发写操作，MVCC用于并发读操作。因此，数据库锁是控制并发的一种手段。

## 概述

### latch和lock

在数据库中，lock和latch都可以称为“锁”。

latch一般称为闩锁（轻量级的锁），因为其要求锁定的时间必须非常短。若持续的时间长，则应用的性能会非常差。在InnoDB存储引擎中，latch又可以分为mutex（互斥量）和rwlock（读写锁）。其目的是用来保证**并发线程**操作临界资源的正确性，并且通常没有死锁检测的机制。

lock的对象是**事务**，用来锁定的是数据库中的对象，如表、页、行。并且一般lock的对象仅在事务commit或rollback后进行释放（不同事务隔离级别释放的时间可能不同）。此外，lock正如在大多数数据库中一样，是有死锁检测机制的。



怎么查看latch和lock的工作状况：

在Debug版本下，通过命令SHOW ENGINE INNODB MUTEX可以看到latch的更多信息：



列Type显示的总是InnoDB，列Name显示的是latch的信息以及所在源码的位置（行数）。列Status比较复杂，在Debug模式下，除了显示os\_waits，还会显示count、spin\_waits、spin\_round、os\_yields、os\_wait\_times等信息。



相对于latch的查看，lock信息显得直观多了。用户可以通过命令SHOW ENGINE INNODB STATUS以及information\_schema架构下的表INNODB\_TRX、INNODB\_LOCKS、INNODB\_LOCK\_WAITS来观察锁的信息。

### 自增长与锁

参考：

<https://blog.csdn.net/miyatang/article/details/78019063>

AUTO-INC锁是当向使用含有AUTO\_INCREMENT列的表中插入数据时需要获取的一种特殊的表级锁 。

在最简单的情况下，如果一个事务正在向表中插入值，则任何其他事务必须等待对该表执行自己的插入操作，以便第一个事务插入的行的值是连续的。

innodb\_autoinc\_lock\_mode配置选项控制用于自动增量锁定的算法。它允许您选择如何在可预测的自动递增值序列和插入操作的最大并发性之间进行权衡。

### 外键和锁

## 分类

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



### 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；

注：如果想要判断一个语句是否加了排它锁，可以通过select lock in shared mode判断，如果报冲突则存在排它锁，否则没有。

**意向锁**

**为了允许行锁和表锁共存，实现多粒度锁机制，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**：

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。之所以叫意向锁，是因为防止上来就加锁造成影响太大了，先执行一个假的加锁，即意向锁，然后再确定是否真正加锁。

比如对于INSERT INTO SELECT就会加IX锁，其他事务再次执行INSERT操作的时候会获取X锁。

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。

## 8.0新特性

### FOR SHARE

参考：

<https://mp.weixin.qq.com/s/mOZS75lgT4VhLmiS8BiV3A>

### sharded 锁系统优化

参考：<http://mysql.taobao.org/monthly/2021/02/04/>

### 锁观测方式

参考：<https://mp.weixin.qq.com/s/w7OovGTZe6ypw6obKtZaMg>

## 锁与并发

**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

## 锁问题

事务隔离性通过锁实现（还有不加锁的MVCC），使得事务可以并发。锁虽然提高了并发性能，但是却会带来潜在的问题。但是，由于事务隔离级别的要求，锁只可能带来三种问题，如果有效防止这三种情况发生，则不会产生并发问题。

### 脏读

脏数据是指事务对缓冲池中的行记录的修改，并且还没有被提交（commit）。

注：脏页与脏数据不同，脏页指的是在缓冲池中已经被修改的页（已经commit但是没有刷盘），但是还没有刷新到磁盘中，即数据库实例内存中的页和磁盘中的页数据不一致，当然在刷新到磁盘之前，日志都已经被写入到重做日志文件中。

对于脏页的读取，是非常正常的。脏页是因为数据库实例内存和磁盘的异步造成的，这并不影响数据的一致性（或者说两者最终都会达到一致性，即当脏页都刷回到磁盘）。并且因为脏页的刷新是异步的，不影响数据库的可用性，因此可以带来性能的提升。

脏数据是指未提交的数据，如果读到了脏数据，即一个事务可以读到另一个事务中未提交的数据，则显然违反了数据库的隔离性。

脏读指的就是在不同的事务下，当前事务可以读到另外事务未提交的数据，简单来说就是可以读到脏数据。

脏读现象在生产环境中并不常发生，脏读发生的条件是需要事务的隔离级别为READ UNCOMMITTED，而**目前绝大部分的数据库都至少设置成READ COMMITTED**。**InnoDB存储引擎默认的事务隔离级别为READ REPEATABLE，SQL Server数据库和Oracle为READ COMMITTED**。

在一些比较特殊的情况下还是可以将事务隔离级别设置为READ UNCOMMITTED。例如，replication环境中的salve节点，并且在该slave节点的查询并不需要特别精确的返回值。

### 不可重复读

不可重复读是指在一个事务中多次读取同一数据集合。在这个事务还没有结束时，另外一个事务也访问该同一数据集合，并做了一些DML操作。因此，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的情况这种情况称为不可重复读。

不可重复读和脏读的区别是：脏读是读到未提交的数据，而不可重复读读到的却是已经提交的数据，但是违反了数据库事务一致性的要求。

一般来说，不可重复读的问题是可以接受的，因为其读到的是已经提交的数据，本身并不会带来很大的问题。因此，很多数据库厂商（Oracle、SQL Server）将其数据库事务隔离级别设置为READ COMMITTED，在这种隔离级别下允许不可重复读的现象。

在InnoDB存储引擎中，通过使用Next-Key Lock算法来避免不可重复读。在MySQL官方文档中将不可重复读的问题定义为Phantom Problem，即幻像问题。在Next-Key Lock算法下，对于索引的扫描，不仅是锁住扫描到的索引，而且还锁住这些索引覆盖的范围（gap）。因此在这个范围内的插入都是不允许的。这样就避免了另外的事务在这个范围内插入数据导致的不可重复读的问题。因此，InnoDB存储引擎默认的事务隔离级别是READ REPEATABLE，采用Next-Key Lock算法，避免了不可重复读的现象（同时也把幻读解决了）。

### 丢失更新

更新丢失是另一个锁导致的问题，简单来说其就是一个事务的更新操作会被另一个事务的更新操作所覆盖，从而导致数据的不一致。

例如：

1. 事务T1将行记录r更新为v1，但是事务T1并未提交。
2. 与此同时，事务T2将其记录r更新为v2，事务T2未提交。
3. 事务T1提交。
4. 事务T2提交。

但是，在当前数据库的任何隔离级别下，都不会导致数据库理论意义上的任何丢失。这是因为，即使是READ UNCOMMITTED的事务隔离级别，对于行的DML操作，需要对其或其他粒度级别的对象加锁。因此在上述步骤2）中，事务T2并不能对记录r进行更新操作，其会被阻塞，直到事务T1提交。

虽然数据库能阻止丢失更新的产生，但是在生产应用中还存在另一种逻辑意义的丢失更新问题，而导致该问题的并不是因为数据库本身的问题。实际上，在所有多用户计算机系统环境下都会可能产生这个问题。简单地说，出现下面的情况，就会发丢失更新；

1. 事务T1查询一行数据，放入本地内存，并显示给一个终端用户User1。
2. 事务T2特查询该行数据，并将取得的数据显示给终端用户User2。
3. User1修改这行记录，更新数据库并提交。
4. User2修改这行记录，更新数据库并提交。

显然，这个过程中用户User1的修改更新操作“丢失”了，而这可能导致严重问题。要避免丢失更新发生，需要让事务在这种情况下的操作变成串行化，而不是并行的操作。

## 阻塞

因为不同锁之间的兼容性关系，在有些时刻一个事务汇总的锁需要等待另一个事务中的锁释放它所占用的资源，这就是阻塞。阻塞并不是一件坏事，它是为了确保事务可以并发且正常运行。

在InnoDB存储引擎中，参数innodb\_lock\_wait\_timeout用来控制等待的时间（默认50秒），innodb\_rollback\_on\_timeout用来设定是否在等待超时时对进行中的事务进行回滚操作（默认是OFF，代表不回滚）。

参数innodb\_lock\_wait\_timeout是动态的，可以在MySQL数据库运行时进行调整。而innodb\_rollback\_on\_timeout是静态的，不可以在启动时进行修改。

在默认情况下，InnoDB存储引擎不会回滚超时引发的错误异常，其实InnoDB存储引擎在大部分情况下都不会对异常进行回滚。

## 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。可以这样理解，在加行锁的时候，InnoDB总会尽可能降低加锁的范围，避免全表锁定，此时只能借助索引区快速定位具体行。

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

数据库事务的隔离性通过锁实现，锁又可以通过表锁或者行锁实现。

## 锁与存储引擎

相对其他数据库而言，MySQL的锁机制比较简单，其最 显著的特点是不同的存储引擎支持不同的锁机制。比如，MyISAM和MEMORY存储引擎采用的是表级锁（table-level locking）；BDB存储引擎采用的是页面锁（page-level locking），但也支持表级锁；InnoDB存储引擎既支持行级锁（row-level locking），也支持表级锁，但默认情况下是采用行级锁。

表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。

行级锁：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。

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。

# 实现原理

参考：

MySQL8.0 innodb锁相关：

<http://mysql.taobao.org/monthly/2021/02/08/>

scope\_guard：

<http://mysql.taobao.org/monthly/2021/10/04/>

timed\_mutexes：

<http://mysql.taobao.org/monthly/2014/08/01/>

## Lock manager

参考：<http://mysql.taobao.org/monthly/2020/04/09/>

## InnoDB latch持有分析

参考：

InnoDB latch持有分析：<http://mysql.taobao.org/monthly/2020/03/07/>

## InnoDB mutex

参考：

InnoDB mutex实现分析：<http://mysql.taobao.org/monthly/2020/03/05/>

## InnoDB读写锁

参考：

InnoDB读写锁实现分析：<http://mysql.taobao.org/monthly/2020/04/02/>

MySQL读写锁rwlock：<http://blog.chinaunix.net/uid-26896862-id-3381803.html>

# 锁调度

参考：<http://mysql.taobao.org/monthly/2021/09/01/>

# 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

## 全局读锁定(FTWRL锁)

FLUSH TABLES WITH READ LOCK这个命令是全局读锁定，执行了命令之后**所有库所有表**都被锁定只读。一般都是用在数据库联机备份，这个时候数据库的写操作将被阻塞，读操作顺利进行。

解锁的语句是unlock tables。

flush table read lock与lock tables区别：

1. flush table read lock是全局读锁，解锁是unlock tables
2. lock table table\_name read/write：执行表锁，解锁：unlock table table\_name/unlock tables table\_names，每次只能持有一个锁

## 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。

注：如果事务T1执行insert into A select \* from B where PK=’’，事务T2执行相同的语句，则T1会对这一行加锁，T2执行的时候会发现事务ID不对应产生排它锁。

# 粒度锁/范围锁

参考：<https://zhuanlan.zhihu.com/p/565505911>

## FTWRL锁

FTWRL锁（flush tables with read lock）想当于对全局增加读锁，会对数据更新定义和提交的进行阻塞，在做全库备份时。

## 表锁

表级锁是MySQL锁中粒度最大的一种锁，表示当前的操作对整张表加锁，资源开销比行锁少，不会出现死锁的情况，但是发生锁冲突的概率很大。被大部分的mysql引擎支持，MyISAM和InnoDB都支持表级锁，但是InnoDB默认的是行级锁。

**表锁由MySQL Server实现**，一般在执行DDL语句时会对整个表进行加锁，比如说 ALTER TABLE 等操作。在执行SQL语句时，也可以明确指定对某个表进行加锁。

表锁使用的是一次性锁技术，也就是说，在会话开始的地方使用 lock 命令将后续需要用到的表都加上锁，在表释放前，只能访问这些加锁的表，不能访问其他表，直到最后通过unlock tables释放所有表锁。

除了使用unlock tables显示释放锁之外，会话持有其他表锁时执行lock table 语句会释放会话之前持有的锁；会话持有其他表锁时执行 start transaction 或者 begin 开启事务时，也会释放之前持有的锁。

### 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。

对于MyISAM引擎，其锁是表锁设计。并发情况下的读没有问题，但是并发插入性能要差一些。如果插入是在“底部”，MyISAM存储引擎还是可以有一定的并发写入操作。

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

LOCK TABLE table\_name [ AS alias\_name ] READ

#### 表独占写锁

使用lock table/unlock table进行加锁/解锁。

lock tables table\_name read：没有会话能DML read\_locked表（只读不能修改）

lock tables table\_name write：除了持有锁会话之外，没有会话能访问write-locked表（其他会话不能读写）

在表读锁和表写锁的环境下：读读不阻塞，读写阻塞，写写阻塞！

读读不阻塞：当前用户在读数据，其他的用户也在读数据，不会加锁

读写阻塞：当前用户在读数据，其他的用户不能修改当前用户读的数据，会加锁！

写写阻塞：当前用户在修改数据，其他的用户不能修改当前用户正在修改的数据，会加锁！



从上面已经看到了：读锁和写锁是互斥的，读写操作是串行。

如果某个进程想要获取读锁，同时另外一个进程想要获取写锁。在mysql里边，写锁是优先于读锁的！

写锁和读锁优先级的问题是可以通过参数调节的： max\_write\_lock\_count和 low-priority-updates。

MyISAM可以支持查询和插入操作的并发进行。可以通过系统变量 concurrent\_insert来指定哪种模式，在MyISAM中它默认是：如果MyISAM表中没有空洞（即表的中间没有被删除的行），MyISAM允许在一个进程读表的同时，另一个进程从表尾插入记录。

但是InnoDB存储引擎是不支持的！

### 查询

**查看是否存在表锁：**

SHOW OPEN TABLES [FROM db\_name] [LIKE ‘pattern’]

SHOW OPEN TABLES列举在表缓存中当前被打开的非TEMPORARY表。

SHOW OPEN TABLES会返回以下字段：

database：含有该表的数据库。

table：表名称。

in\_use：表当前被查询使用的次数。如果该数为零，则表是打开的，但是当前没有被使用。

name\_locked：表名称是否被锁定。名称锁定用于取消表或对表进行重命名等操作。

mysql> show open tables where in\_use >=1;

+----------+-------+--------+-------------+

| Database | Table | In\_use | Name\_locked |

+----------+-------+--------+-------------+

| MyDB | test | 1 | 0 |

+----------+-------+--------+-------------+

1 row in set (0.00 sec)

如果您没有表的权限，则它不会显示在输出中 SHOW OPEN TABLES。

**表级锁可以通过两个变量的查询：**

Table\_locks\_immediate，产生表级锁的次数。

Table\_locks\_waited，数显表级锁而等待的次数。

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

在MyISAM存储引擎中，当执行SQL语句的时候是自动加的。

在InnoDB存储引擎中，如果没有使用索引，表锁也是自动加的。

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。

你可能会想，日常开发中，我们所有表都使用AUTO\_INCREMENT作主键，所以会非常频繁的使用到该锁。不过，事情可能并不像你想的那样。在介绍AUTO-INC表级锁之前，我们先来看下和它密切相关的SQL语句以及系统变量innodb\_autoinc\_lock\_mode。

## 行锁

行级锁是Mysql中锁定粒度最细的一种锁，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。有可能会出现死锁的情况。行级锁按照使用方式分为共享锁和排他锁。

不同存储引擎的行锁实现不同，后续没有特别说明，则行锁特指 InnoDB 实现的行锁。

### 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/读锁

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/写锁

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。

为了允许行锁和表锁共存，实现多粒度锁机制，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：

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

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

意向锁也是数据库隐式帮我们做了，不需要程序员操心！

#### 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）

### 锁问题

#### 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。**

#### 锁超时/等待

##### 概述

###### 产生

Mysql造成锁的情况有很多，下面我们就列举一些情况：

1. 执行DML操作没有commit，再执行删除操作就会锁表。
2. 在同一事务内先后对同一条数据进行插入和更新操作。
3. 表索引设计不当，导致数据库出现死锁。
4. 长事物，阻塞DDL，继而阻塞所有同表的后续操作。

###### 锁等待与死锁

但是要区分的是Lock wait timeout exceeded与Dead Lock是不一样。

Lock wait timeout exceeded：后提交的事务等待前面处理的事务释放锁，但是在等待的时候超过了mysql的锁等待时间，就会引发这个异常。

Dead Lock：两个事务互相等待对方释放相同资源的锁，从而造成的死循环，就会引发这个异常。

###### innodb\_lock\_wait\_timeout与lock\_wait\_timeout

还有一个要注意的是innodb\_lock\_wait\_timeout与lock\_wait\_timeout也是不一样的。

innodb\_lock\_wait\_timeout：innodb的dml操作的**行级锁**的等待时间，默认50s。

lock\_wait\_timeout：数据结构ddl操作的锁的等待时间

说明：说到lock\_wait\_timeout我们就不得不提到 metadata locks，我们称之为元数据锁。实质上就是进行修改字段数据类型、字符集等线上DDL时容易阻塞对该表的其它DML操作，lock\_wait\_timeout这个参数的默认时间是8760小时非常长，如果生产环境出现metadata locks无疑是灾难性的，会严重影响正常的业务操作。所以我们需要将之设为50秒或者更短。

##### 配置

如何查看innodb\_lock\_wait\_timeout的具体值？

SHOW VARIABLES LIKE 'innodb\_lock\_wait\_timeout'

如何修改innode lock wait timeout的值？

参数修改的范围有Session和Global，并且支持动态修改，可以有两种方法修改：

方法一：

通过下面语句修改

set innodb\_lock\_wait\_timeout=100;  
 set global innodb\_lock\_wait\_timeout=100;

注意global的修改对当前线程是不生效的，只有建立新的连接才生效。

方法二：

修改参数文件/etc/my.cnf innodb\_lock\_wait\_timeout = 50

注：innodb\_lock\_wait\_timeout指的是事务等待获取资源等待的最长时间，超过这个时间还未分配到资源则会返回应用失败；当锁等待超过设置时间的时候，就会报如下的错误；

ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction。 其参数的时间单位是秒，最小可设置为1s(一般不会设置得这么小)，最大可设置1073741824秒，默认安装时这个值是50s(默认参数设置)。

##### 排查思路

锁问题分析：

1. 查看存储引擎总体状况

show engine innodb status;

1. 查看是否有表锁

show open tables where in\_use>0;

1. 查看在执行的事务

select \* from information\_schema.innodb\_trx;

**处理锁超时问题思路：**

1. 查找锁数据的线程号

select thread\_id from performance\_schema.data\_locks;

1. 查找具体的SQL

select processlist\_info from threads where thread\_id=;（或者用show processlists）

1. 锁等待

select \* from performance\_schema.data\_lock\_waits;

1. 具体SQL线程

show processlists;

1. Kill

上述方法对于目前还存在的锁等待是有效的，但是对于已经发生过的锁等待，则不起作用，但是可以通过历史列表查询。

1. 事务历史列表：events\_transaction\_history\_long
2. SQL历史列表：events\_statement\_history\_long

##### 故障排查

**现象描述：**

数据更新或新增后数据经常自动回滚。

表操作总报 Lock wait timeout exceeded 并长时间无反应

**解决方法：**

应急方法：show full processlist; kill掉出现问题的进程。

注：有的时候通过processlist是看不出哪里有锁等待的，当两个事务都在commit阶段是无法体现在processlist上

根治方法：select \* from innodb\_trx;查看有是哪些事务占据了表资源。

注：通过这个办法就需要对innodb有一些了解才好处理

说起来很简单找到它杀掉它就搞定了，但是实际上并没有想象的这么简单，当问题出现要分析问题的原因，通过原因定位业务代码可能某些地方实现的有问题，从而来避免今后遇到同样的问题。

#### 死锁

参考：

<https://www.cnblogs.com/remcarpediem/p/13843180.html>

<https://www.cnblogs.com/duanxz/p/4394641.html>

<http://mysql.taobao.org/monthly/2020/01/03/>

##### 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。

##### 检查死锁

在实际开发中无法避免数据被锁的问题，那么我们可以通过哪些手段来查询锁呢？

1. 查看正在进行的事务

SELECT \* FROM INFORMATION\_SCHEMA.INNODB\_TRX;

1. 查看正在锁的事务

SELECT \* FROM INFORMATION\_SCHEMA.INNODB\_LOCKS;

//MySQL8.0已经更改为performance\_schema.data\_lock

1. 查看等待锁的事务

SELECT \* FROM INFORMATION\_SCHEMA.INNODB\_LOCK\_WITS;

//MySQL8.0已经更改

1. 查看是否存在表锁

SHOW OPEN TABLES WHERE in\_use>0;

1. 查看最近死锁的日志

SHOW ENGINE INNODB STATUS;

注：也可以查看当前执行的线程ID，然后通过SELECT \* FROM process\_list where thread\_id=;查询对应线程执行了什么SQL

当然可以通过binlog获取死锁发生时的具体事件和SQL。

###### 表锁

**查看是否存在表锁：**

SHOW OPEN TABLES [FROM db\_name] [LIKE ‘pattern’]

SHOW OPEN TABLES列举在表缓存中当前被打开的非TEMPORARY表。

SHOW OPEN TABLES会返回以下字段：

database：含有该表的数据库。

table：表名称。

in\_use：表当前被查询使用的次数。如果该数为零，则表是打开的，但是当前没有被使用。

name\_locked：表名称是否被锁定。名称锁定用于取消表或对表进行重命名等操作。

mysql> show open tables where in\_use >=1;

+----------+-------+--------+-------------+

| Database | Table | In\_use | Name\_locked |

+----------+-------+--------+-------------+

| MyDB | test | 1 | 0 |

+----------+-------+--------+-------------+

1 row in set (0.00 sec)

如果您没有表的权限，则它不会显示在输出中 SHOW OPEN TABLES。

**表级锁可以通过两个变量的查询：**

Table\_locks\_immediate，产生表级锁的次数。

Table\_locks\_waited，数显表级锁而等待的次数。

###### 行锁

**行级锁可以通过下面几个变量查询：**

Innodb\_row\_lock\_current\_waits，当前正在等待锁定的数量。

Innodb\_row\_lock\_time（重要），从系统启动到现在锁定总时长。

Innodb\_row\_lock\_time\_avg（重要），每次等待所花平均时间。

Innodb\_row\_lock\_time\_max，从系统启动到现在等待最长的一次花费时间。

Innodb\_row\_lock\_waits（重要），从系统启动到现在总共等待的次数。

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。

InnoDB存储引擎能检测到死锁的循环依赖并立即返回一个错误。

##### 死锁处理

有两种主要的方法用于处理死锁的问题。我们可以使用死锁预防（deadlock prevention）协议来保证系统永不进入死锁状态。另一种可选的方法是，我们可以允许系统进入死锁状态，然后试着用死锁检测（deadlock detection）与死锁恢复（deadlock recovery）机制进行恢复。这两种方法都有可能引起事务回滚。如果系统进入死锁状态的概率相对较高，则通常使用的是预防机制；否则，使用检测与恢复机制会更有效。

###### 死锁预防

预防死锁有两种方法。一种方法是通过对封锁请求进行排序，或要求同时获得所有的锁来保证不会发生循环等待。另一种方法更接近于死锁恢复，每当等待有可能导致死锁时，它就执行事务回滚而不是等待锁。

**避免死锁/预防死锁的一些建议：**

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。

死锁检测与恢复

**死锁检测**

死锁可以用一种称为等待图（wait-for-graph）的有向图来精确描述。

**从死锁中恢复**

当检测算法判定存在死锁时，系统必须从死锁中恢复。解除死锁最常用的解决方案是回滚一个或多个事务。需要请求的动作有三个：

1. 选择牺牲者

给定一组死锁的事务，我们必须决定回滚哪一个（或哪一些）事务以打破死锁。我们应该回滚那些将产生最低代价的事务。遗憾的是，最低代价这个术语并不准确。有很多因素会影响回滚的代价，其中包括：

1. 事务已经计算了多久，以及在完成其指定任务之前该事务还将计算多长时间；
2. 该事务已经使用了多少数据项；
3. 为了完成事务还需要使用多少数据项；
4. 回滚时将牵涉多少事务。
5. 回滚
6. 饿死

在一个系统中，如果选择牺牲者主要是基于代价因素，那么有可能发生同一个事务总是被选为牺牲者的情况。这样一来，这个事务就总是不能完成其指定的任务，这样就发生了饿死。我们必须保证一个事务被选为牺牲者的次数是有限（较少）的。最常用的解决方案是在代价因素中包含回滚的次数。

**MySQL恢复/解除死锁：**

死锁发生以后，只有部分或完全回滚其中一个事务（可以通过锁超时机制自动结束或kill强制结束），才能打破死锁。

基本操作步骤：

1. 查看当前正在运行的进程

show processlist;

或者：

select \* from information\_schema.INNODB\_TRX;

1. 杀掉进程ID对应的进程

kill id

1. 验证（kill后看是否还有锁存在）

SHOW OPEN TABLES where in\_use>0;

InnoDB方法是，将持有最少行级排他锁的事务回滚。在应用程序设计时必须考虑处理死锁，多数情况下重新执行因死锁回滚的事务即可。

##### 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**；

确定事务是否能够在更低的隔离级别上运行，执行提交读允许事务读取另一事务已读取（未提交）的数据，而不必等待第一个事务完成。使用较低的事务隔离级别（例如提交读）而不使用较高的隔离级别（例如可串行读）可以缩短持有共享锁的时间，从而降低锁争夺。

1. 为表添加合理的索引。可以看到如果不走索引将会为表的每一行记录添加上锁，死锁的概率大大增大。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**；

在同一DB中并发执行多个需要长时间运行的事务时，发生死锁的概率较大，事务运行时间越长，其持有排它锁（exclusive锁）或者更新锁（update锁）的时间就越久，从而阻塞了其他活动并可能导致死锁。保持事务在一个批处理中，可以最小化事务的网络通信往返量，减少完成事务可能的延迟并释放锁。同时，涉及多个表的查询更新操作，若比较耗时，尽量不要放在一个事务内处理，能分割尽量分割，若不能分割，便尽可能使之在业务量较小的时间执行。

1. 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；
2. 在事务开始时，如果有记录要修改，先使用 SELECT... FOR UPDATE 语句获取锁，即使这些修改语句是在后面执行。

在事务中，如果要更新记录，直接申请排他锁。而不是查询时申请共享锁、更新时再申请排他锁。

这样做会导致，当申请排他锁时，其他事务可能已经获得了相同记录的共享锁，从而造成锁冲突，甚至死锁。

简单来说，如果你要更新记录要做两步操作，第一步查询，第二步更新。就不要第一步上共享锁，第二部上排他锁了，直接在第一步就上排他锁，抢占先机。

通过 SELECT ... LOCK INSHARE MODE（共享锁）获取行的读锁后，如果当前事务再需要对该记录进行更新操作，则很有可能造成死锁。所以，如果要对行记录进行修改，直接上排他锁。

1. **以固定的顺序访问表和行（即应该按照同一顺序访问数据对象）**。比如对两个job批量更新的情形，简单方法是对id列表先排序，后执行，这样就避免了交叉等待锁的情形；将两个事务的sql顺序调整为一致，也能避免死锁。

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；

1. 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；
2. 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；
3. 可考虑体系结构的优化与代码重构，提高系统整理的运行效率。

例如尽可能不采用效率低下的计算模型，复杂的业务应采用异步调度处理。

1. 可通过程序控制事务提交的时机。

如果一次检索出10万条记录但只更改其中的100条，就可以通过代码来执行100个update，或者用分段提交，即所有的修改使用多个事务进行提交，但这样会使事务不完整，应该酌情使用。

1. **宜将经常更新的数据库和查询数据库分开**，定期将不改变的数据导入查询数据库中，这样查询和更新可以分开进行，而降低死锁几率；
2. **在进行数据库模式设计时，应注意外键引用的完整性，并对外键加索引**。

如果更新了父表的主键，由于外键上没有索引，所以子表会被锁定，如果删除了父表中的一行，整个子表也会被锁定。

##### 故障排查

<https://blog.csdn.net/simongeek/article/details/79802059>

<https://www.iteye.com/blog/825635381-2339503>

<https://blog.csdn.net/kezhong_wxl/article/details/76682710>

### 行锁与MVCC

#### 概述

乐观并发控制和悲观并发控制都是通过延迟或者终止相应的事务来解决事务之间的竞争条件来保证事务的可串行化；虽然前面的两种并发控制机制确实能够从根本上解决并发事务的可串行化的问题，但是其实都是在解决写冲突的问题，两者区别在于对写冲突的乐观程度不同(悲观锁也能解决读写冲突问题，但是性能就一般了)。而在实际使用过程中，数据库读请求是写请求的很多倍，我们如果能解决读写并发的问题的话，就能更大地提高数据库的读性能，而这就是多版本并发控制所能做到的事情。

与悲观并发控制和乐观并发控制不同的是，MVCC是为了解决读写锁造成的多个、长时间的读操作**饿死**写操作问题，也就是解决读写冲突的问题。MVCC 可以与前两者中的任意一种机制结合使用，以提高数据库的读性能。

数据库的悲观锁基于提升并发性能的考虑，一般都同时实现了多版本并发控制。不仅是MySQL，包括Oracle、PostgreSQL等其他数据库系统也都实现了MVCC，但各自的实现机制不尽相同，因为MVCC没有一个统一的实现标准。

总的来说，MVCC的出现就是数据库不满用悲观锁去解决读-写冲突问题，因性能不高而提出的解决方案。

数据库事务有不同的隔离级别，不同的隔离级别对锁的使用是不同的，锁的应用最终导致不同事务的隔离级别。

MVCC(Multi-Version Concurrency Control)多版本并发控制，可以简单地认为：**MVCC就是行级锁的一个变种(升级版)**。

**事务的隔离级别就是通过锁的机制来实现，只不过隐藏了加锁细节**。

在表锁中我们读写是阻塞的，基于提升并发性能的考虑，**MVCC一般读写是不阻塞的**(所以说MVCC很多情况下避免了加锁的操作)

MVCC实现的读写不阻塞正如其名：多版本并发控制--->通过一定机制生成一个数据请求时间点的一致性数据快照（Snapshot)，并用这个快照来提供一定级别（语句级或事务级）的一致性读取。从用户的角度来看，好像是数据库可以提供同一数据的多个版本。

#### 快照

MySQL中MVCC只能在Repeatable Read（读可重复读）、Read Committed（读可提交）这两个隔离级别下工作。

快照有两个级别：

1、语句级

针对于**Read committed**隔离级别

2、事务级别

针对于**Repeatable read**隔离级别

**事务的隔离级别有4种：**

Read uncommitted

会出现脏读，不可重复读，幻读

Read committed

会出现不可重复读，幻读

Repeatable read

会出现幻读(但在Mysql实现的Repeatable read配合gap锁不会出现幻读！)

Serializable

串行，避免以上的情况！

Readuncommitted会出现的现象--->脏读：一个事务读取到另外一个事务未提交的数据

例子：A向B转账，A执行了转账语句，但A还没有提交事务，B读取数据，发现自己账户钱变多了！B跟A说，我已经收到钱了。A回滚事务rollback，等B再查看账户的钱时，发现钱并没有多。

出现脏读的原因是因为在读的时候没有加读锁，导致可以读取出还没释放锁的记录。

**Read uncommitted过程：**

事务A读取记录(没有加任何的锁)

事务B修改记录(此时加了写锁，并且还没有commit-->也就没有释放掉写锁)

事务A再次读取记录(此时因为事务A在读取时没有加任何锁，所以可以读取到事务B还没提交的(没释放掉写锁)的记录

Read committed避免脏读的做法其实很简单：

在读取的时候生成一个版本号，直到事务其他commit被修改了之后，才会有新的版本号

**Read committed过程：**

事务A读取了记录(生成版本号)

事务B修改了记录(此时加了写锁)

事务A再读取的时候，是依据最新的版本号来读取的(当事务B执行commit了之后，会生成一个新的版本号)，如果事务B还没有commit，那事务A读取的还是之前版本号的数据。

但Read committed出现的现象--->不可重复读：一个事务读取到另外一个事务已经提交的数据，也就是说一个事务可以看到其他事务所做的修改

注：A查询数据库得到数据，B去修改数据库的数据，导致A多次查询数据库的结果都不一样

危害：A每次查询的结果都是受B的影响的，那么A查询出来的信息就没有意思了

上面也说了，**Read committed是语句级别的快照**！每次读取的都是当前最新的版本！

**Repeatable read避免不可重复读是事务级别的快照**！每次读取的都是当前事务的版本，即使被修改了，也只会读取当前事务版本的数据。



至于虚读(幻读)：是指在一个事务内读取到了别的事务插入的数据，导致前后读取不一致。

注：和不可重复读类似，但虚读(幻读)会读到其他事务的插入的数据，导致前后读取不一致

MySQL的Repeatable read隔离级别加上GAP间隙锁已经处理了幻读了。

### 对比

表锁VS行锁

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。

Microsoft SQL Server 2005版本之前都是页锁，到2005版本，开始支持乐观并发和悲观并发，在乐观并发下开始支持行级锁，但是其实现方式与InnoDB存储引擎的实现方式完全不同。

### 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

## 锁定情况

在实际开发中无法避免数据被锁的问题，那么我们可以通过哪些手段来查询锁呢？

表级锁可以通过两个变量的查询：

table\_locks\_immediate，产生表级锁的次数。

table\_locks\_waited，数显表级锁而等待的次数。

行级锁可以通过下面几个变量查询：

innodb\_row\_lock\_current\_waits，当前正在等待锁定的数量。

innodb\_row\_lock\_time（重要），从系统启动到现在锁定总时长。

innodb\_row\_lock\_time\_avg（重要），每次等待所花平均时间。

innodb\_row\_lock\_time\_max，从系统启动到现在等待最长的一次花费时间。

innodb\_row\_lock\_waits（重要），从系统启动到现在总共等待的次数。

**基本排查思路：**

1、查看当前线程处理情况，如果不使用full关键字，信息字段中只会显示每个语句的前100个字符。  
 show processlist;   
 show full processlist;

2、查询表级锁争用情况：

Table\_locks\_immediate指的是能够立即获得表级锁的次数

Table\_locks\_waited指的是不能立即获取表级锁而需要等待的次数  
 show status like 'Table%';

3、获取锁定次数、锁定造成其他线程等待次数，以及锁定等待时间信息  
 show status like '%lock%';

4、查看正在被锁定的的表  
 show OPEN TABLES where In\_use > 0;

5、查看被锁住的信息  
 SELECT \* FROM INFORMATION\_SCHEMA.INNODB\_LOCKS;

6、等待锁定  
 SELECT \* FROM INFORMATION\_SCHEMA.INNODB\_LOCK\_WAITS;

7、查看表索引信息  
 SHOW INDEX FROM account;

**总结：**先确定哪些线程在执行加锁操作->确定哪些表被锁定->确定被锁定表的锁情况->索引信息

## 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/InnoDB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。

**InnoDB存储引擎实现了两种标准的行级锁：共享锁（S Lock）和排他锁（X Lock）**。

要解决幻读的问题有两种方案，一种是采用SERIALIZABLE数据隔离级别，但是这种方案会强制把所有事务排序，来达到事务之间不互相冲突产生幻读的问题，当事务并发高的时候，很容易产生大量超时和锁竞争的情况，所以一般不太建议采用这种方案。另一种方案是采用在RR数据隔离级别下，手动给select操作加上x锁（排它锁）或者s锁（共享锁）。其实，**共享锁和排他锁就是为了解决幻读引入的（因为serializable基本不具备实际应用价值）**。

## 共享锁

### 概述

共享锁（SELECT ... LOCK IN SHARE MODE）即一个事务获取一条记录共享锁的同时，其他事务也可以获得这条记录的共享锁，但是如果同时有多个事务获得这条记录的共享锁，谁也无法修改这条记录，直到都释放掉共享锁，只剩下一个事务拥有这条记录的锁为止。

共享锁Shared Locks（简称***S锁***，又称**读锁**，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；非独占的，允许多个并发事务读取其锁定的资源。

**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。

**共享锁：**

|  |  |
| --- | --- |
| 事务1 | 事务2 |
| start transaction;  select \* from tb where id = 1 lock in share mode;  +-----+------------+  | id | name |  +-----+------------+  | 1 | aaa |  +-----+------------+ |  |
|  | start transaction;  select \* from tb where id = 1 lock in share mode;  +-----+------------+  | id | name |  +-----+------------+  | 1 | aaa |  +-----+------------+ |
| update tb set name = 'ccc' where id = 1;  等待事务2释放锁 |  |
| 执行成功  Query OK， 1 row affected (0.02 sec) | update tb set name = 'ddd' where id = 1;  产生死锁，回滚释放锁 |
|  | update tb set name = 'ddd' where id = 1;  等待事务1释放锁 |
| commit; |  |
|  | update tb set name = 'ddd' where id = 1;  Query OK， 1 row affected (0.02 sec) |

### 性质

1. 多个事务可以封锁同一个共享页；
2. 任何事务都不能修改该页；
3. 通常该页被读取完毕后，S锁会被立即释放。

**总结：**

允许不同事务之前共享加锁读取，但是不允许其他事务修改或加入排它锁。

如果有修改必须等待一个事务提交完成，才可以执行，容易出现死锁。

### 区别

**共享锁和排他锁的异同点：**

相同点：一个事务在获得一条记录的共享锁或者排它锁的同时，其他事务都不能修改这条记录，直到这个事务释放掉锁为止。

不同点：排它锁比共享锁多阻塞了其他事务对相同记录的共享锁，但是不影响快照读。

## 排他锁

排它锁（SELECT ... FOR UPDATE）即一个事务获得了一条记录的排它锁的同时，其他事务就不能获得这条记录的共享锁和排它锁，也无法修改这条记录，直到这个事务释放掉锁为止。

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。

**排他锁：**

|  |  |
| --- | --- |
| 事务1 | 事务2 |
| start transaction;  select \* from tb where id = 1 for update;  +-----+------------+  | id | name |  +-----+------------+  | 1 | aaa |  +-----+------------+ |  |
|  | start transaction;  select \* from tb where id = 1 for update;  等待事务1释放锁 |
| update tb set name = 'ccc' where id = 1;  Query OK， 1 row affected (0.02 sec) |  |
| commit; |  |
|  | select \* from tb where id = 1 for update;  +-----+------------+  | id | name |  +-----+------------+  | 1 | ccc |  +-----+------------+ |

## 意向锁

### 背景

**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。

### 概述

意向锁的目的是告知其他事务，某事务已经锁定了或即将锁定某个/些数据行。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）。

### 分类

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。

### 兼容性

表级锁(table-level lock)的兼容性矩阵如下：



对于上面的兼容性矩阵，一定注意两点：

1、在上面的兼容性矩阵中，S是表的(不是行的)共享锁，X是表的(不是行的)排它锁。

2、意向锁IS和IX 和任何行锁 都兼容（即：和行的X锁或行的S锁都兼容）。

所以，意向锁只会阻塞全表请求（例如：LOCK TABLES ... WRITE），不会阻塞其他任何东西。因为LOCK TABLES ... WRITE需要设置X表锁，这会被意向锁IS或IX所阻塞。

### 查询

怎么查看表级IS锁？只需要查看 performance\_schema.data\_locks 表就可以了。另一个表performance\_schema..metadata\_locks 表可以查看MDL锁的详情。



查询结果例如下面这样：

[root@\*\*] [(none)]>select \* from performance\_schema.data\_locks\G  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
               ENGINE: INNODB  
       ENGINE\_LOCK\_ID: 140701134495048:1350:140701396637648  
ENGINE\_TRANSACTION\_ID: 422176111205704  
            THREAD\_ID: 87  
             EVENT\_ID: 95  
        OBJECT\_SCHEMA: yejr  
          OBJECT\_NAME: t1  
       PARTITION\_NAME: NULL  
    SUBPARTITION\_NAME: NULL  
           INDEX\_NAME: NULL  
OBJECT\_INSTANCE\_BEGIN: 140701396637648  
            LOCK\_TYPE: TABLE  
            LOCK\_MODE: IS  
          LOCK\_STATUS: GRANTED  
            LOCK\_DATA: NULL  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
               ENGINE: INNODB  
       ENGINE\_LOCK\_ID: 140701134495048:267:4:9:140701409130528  
ENGINE\_TRANSACTION\_ID: 422176111205704  
            THREAD\_ID: 87  
             EVENT\_ID: 95  
        OBJECT\_SCHEMA: yejr  
          OBJECT\_NAME: t1  
       PARTITION\_NAME: NULL  
    SUBPARTITION\_NAME: NULL  
           INDEX\_NAME: PRIMARY  
OBJECT\_INSTANCE\_BEGIN: 140701409130528  
            LOCK\_TYPE: RECORD  
            LOCK\_MODE: S，REC\_NOT\_GAP  
          LOCK\_STATUS: GRANTED  
            LOCK\_DATA: 1

此时我们能看到t1表上共有两个锁，一个是表级IS锁，另一个是c1=1上的共享锁。

同样地，我们也可以观察IX锁或其他锁。

- session1执行下面的SQL  
 [root@\*\*] [yejr]>begin; update t1 set c4=rand()\*1024 where c1=1;  
  
 - session2查询PFS.data\_locks  
 [root@\*\*] [(none)]>select \* from performance\_schema.data\_locks\G  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
               ENGINE: INNODB  
       ENGINE\_LOCK\_ID: 140701134495888:1350:140701396639728  
ENGINE\_TRANSACTION\_ID: 104536  
            THREAD\_ID: 89  
             EVENT\_ID: 43  
        OBJECT\_SCHEMA: yejr  
          OBJECT\_NAME: t1  
       PARTITION\_NAME: NULL  
    SUBPARTITION\_NAME: NULL  
           INDEX\_NAME: NULL  
OBJECT\_INSTANCE\_BEGIN: 140701396639728  
            LOCK\_TYPE: TABLE  
            LOCK\_MODE: IX  <-- 这个就是IX锁了  
          LOCK\_STATUS: GRANTED  
            LOCK\_DATA: NULL  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
               ENGINE: INNODB  
       ENGINE\_LOCK\_ID: 140701134495888:267:4:9:140701409135136  
ENGINE\_TRANSACTION\_ID: 104536  
            THREAD\_ID: 89  
             EVENT\_ID: 43  
        OBJECT\_SCHEMA: yejr  
          OBJECT\_NAME: t1  
       PARTITION\_NAME: NULL  
    SUBPARTITION\_NAME: NULL  
           INDEX\_NAME: PRIMARY  
OBJECT\_INSTANCE\_BEGIN: 140701409135136  
            LOCK\_TYPE: RECORD  
            LOCK\_MODE: X，REC\_NOT\_GAP  
          LOCK\_STATUS: GRANTED  
            LOCK\_DATA: 1

进一步，我们简单看下MDL锁。加共享行锁：

- session1加一个共享行锁  
 [root@\*\*] [yejr]>begin; select \* from t1 where c1=1 for share;  
  
 - session2查询表上有哪些MDL锁  
 [root@\*\*] [(none)]>select \* from performance\_schema.metadata\_locks\G  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
          OBJECT\_TYPE: TABLE  
        OBJECT\_SCHEMA: yejr  
          OBJECT\_NAME: t1  
          COLUMN\_NAME: NULL  
 OBJECT\_INSTANCE\_BEGIN: 140701215694512  
            LOCK\_TYPE: SHARED\_READ  <- 共享读锁，可以同时加多个共享行锁  
        LOCK\_DURATION: TRANSACTION  
          LOCK\_STATUS: GRANTED  
               SOURCE: sql\_parse.cc:5761  
      OWNER\_THREAD\_ID: 87  
       OWNER\_EVENT\_ID: 100

也看下加排他行锁：

- session1加一个排他行锁  
 [root@\*\*] [yejr]>begin; update t1 set c4=rand()\*1024 where c1=1;  
  
 - session2查询表上有哪些MDL锁  
 [root@\*\*] [(none)]>select \* from performance\_schema.metadata\_locks\G  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
          OBJECT\_TYPE: TABLE  
        OBJECT\_SCHEMA: yejr  
          OBJECT\_NAME: t1  
          COLUMN\_NAME: NULL  
OBJECT\_INSTANCE\_BEGIN: 140701215694640  
            LOCK\_TYPE: SHARED\_WRITE  <- 共享写锁，可以同时加多个排他行锁（不同数据行）  
        LOCK\_DURATION: TRANSACTION  
          LOCK\_STATUS: GRANTED  
               SOURCE: sql\_parse.cc:5761  
      OWNER\_THREAD\_ID: 89  
       OWNER\_EVENT\_ID: 43

## 应用场景

共享锁举例：

譬如一个工会活动，我们会设计一张存放工会信息的总表teamInfo，还会设计一张存放操作工会日志的表teamLog，当用户操作工会的时候，我们如果直接insert一条日志到日志表，其实是有一定的风险的，如果这时会长删除了工会，那么就可能出现数据不一致的情况，所以我们在插入数据时先select \* from teamInfo where teamId = xx lock in share mode;再insert into teamLog xxxxxxx;这样就可以了。

排它锁举例：

譬如我们常见的秒杀活动，一般秒杀活动参与秒杀的物品都是有数量限制的，我们在判断用户是否能购买时会判断，是否物品还有剩余，有剩余的情况下再把剩余数量减1，具体sql为1.select \* from goods where id = xxx;2.update num = num-1 from goods where id = xxx;当多个事务并发的时候就会出现秒杀物品超卖的情况。如果我们改成select \* from goods where id = xxx for update;这样就可以了（这种用法的性能不是很高，这里只是举例排它锁的使用场景，使用时要考虑具体场景是否适合用）。

那共享锁和排它锁是否能互相代替呢，这要看具体的场景，像上面两个例子就不行，第一个例子如果用了排它锁就会造成一个用户在操作工会的时候，其他用户就不能获取这条记录共享锁的情况。第二个例子如果使用共享锁的话，其他事务都能获得goods表这条记录的共享锁，会导致谁也更新不了剩余数量这个值的情况。所以共享锁和排它锁都有各自的作用，不能互相替代。

# 实现/锁模式

实现并发控制的主要手段大致可以分为乐观并发控制（乐观锁）和悲观并发控制（悲观锁，Pessimistic Concurrency Control，缩写“PCC”）两种。

无论是悲观锁还是乐观锁，都是人们定义出来的概念，可以认为是一种思想。其实不仅仅是关系型数据库系统中有乐观锁和悲观锁的概念，像memcache、hibernate、tair等都有类似的概念。所以，**不应该拿乐观锁、悲观锁和其他的数据库锁等进行对比**。

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（Optimistic Concurrency Control）的“乐观情绪”体现在，它认为数据的变动不会太频繁。因此，它允许多个事务同时对数据进行变动。

乐观锁（Optimistic Locking）是相对悲观锁而言的，乐观锁假设数据一般情况下不会造成冲突，所以**在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做（可以重试）**。

相对于悲观锁，在对数据库进行处理的时候，乐观锁并不会使用数据库提供的锁机制。一般的实现乐观锁的方式就是记录数据版本。

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。

即数据库乐观锁加锁的一个原则就是尽量想办法减少锁的范围。锁的范围越大，性能越差，数据库的锁就是把锁的范围减小到了最小。

注：在分布式数据库中，乐观锁的实现可以是普通的SQL语句+GTID字段（类似版本号）。

### 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

### 特点

#### 优点

乐观并发控制相信事务之间的数据竞争(data race)的概率是比较小的，因此尽可能直接做下去，直到提交的时候才去锁定，所以不会产生任何锁和死锁。即乐观锁并未真正加锁，效率高。

乐观锁不在数据库上加锁，任何事务都可以对数据进行操作，在更新时才进行校验，这样就避免了悲观锁造成的吞吐量下降的劣势。

#### 缺点

一旦锁的粒度掌握不好，更新失败的概率就会比较高，容易发生业务失败。

乐观锁因为是通过我们人为实现的，它仅仅适用于我们自己业务中，如果有外来事务插入，那么就可能发生错误。

### 实现

乐观锁的概念中其实已经阐述了他的具体实现细节：主要就是两个步骤：冲突检测和数据更新。其实现方式有一种比较典型的就是Compare and Swap(CAS)。

CAS是项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

#### 版本号/CAS

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timestamp）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。其中，采用版本号最常用。在分布式数据库中，可以采用自增的GTID字段。

使用乐观锁的解决思路是，我们认为数据修改产生冲突的概率并不大，多个事务在修改数据的之前先查出版本号，在修改时把当前版本号作为修改条件，只会有一个事务可以修改成功，其他事务则会失败。

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

}

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：

update table set column='value' for update

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。

#### ABA问题

比如扣减库存问题，通过乐观锁可以实现如下：

//查询出商品库存信息，quantity = 3

select quantity from items where id=1

//修改商品库存为2

update items set quantity=2 where id=1 and quantity = 3;

以上，我们在更新之前，先查询一下库存表中当前库存数（quantity），然后在做update的时候，以库存数作为一个修改条件。当我们提交更新的时候，判断数据库表对应记录的当前库存数与第一次取出来的库存数进行比对，如果数据库表当前库存数与第一次取出来的库存数相等，则予以更新，否则认为是过期数据。

以上更新语句存在一个比较重要的问题，即传说中的ABA问题。

比如说一个线程one从数据库中取出库存数3，这时候另一个线程two也从数据库中库存数3，并且two进行了一些操作变成了2，然后two又将库存数变成3，这时候线程one进行CAS操作发现数据库中仍然是3，然后one操作成功。尽管线程one的CAS操作成功，但是不代表这个过程就是没有问题的。



有一个比较好的办法可以解决ABA问题，那就是通过一个单独的可以顺序递增的version字段（分布式数据库中采用GTID）。改为以下方式即可：

//查询出商品信息，version = 1

select version from items where id=1

//修改商品库存为2

update items set quantity=2，version = 3 where id=1 and version = 2;

乐观锁每次在执行数据的修改操作时，都会带上一个版本号，一旦版本号和数据的版本号一致就可以执行修改操作并对版本号执行+1操作，否则就执行失败。因为每次操作的版本号都会随之增加，所以不会出现ABA问题，因为版本号只会增加不会减少。



除了version以外，还可以使用时间戳，因为时间戳天然具有顺序递增性。

以上SQL其实还是有一定的问题的，就是一旦发上高并发的时候，就只有一个线程可以修改成功，那么就会存在大量的失败。

对于像淘宝这样的电商网站，高并发是常有的事，总让用户感知到失败显然是不合理的。所以，还是要想办法减少乐观锁的粒度的。

有一条比较好的建议，可以减小乐观锁力度，最大程度的提升吞吐率，提高并发能力！如下：

//修改商品库存

update item

set quantity=quantity - 1

where id = 1 and quantity - 1 > 0

以上SQL语句中，如果用户下单数为1，则通过quantity - 1 > 0的方式进行乐观锁控制。

以上update语句，在执行过程中，会在一次原子操作中自己查询一遍quantity的值，并将其扣减掉1。

高并发环境下锁粒度把控是一门重要的学问，选择一个好的锁，在保证数据安全的情况下，可以大大提升吞吐率，进而提升性能。

### 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（因为会影响系统吞吐量），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。

**注：**在分布式数据库中，乐观锁采用最后校验活跃GTID（在语句最后添加WHERE GTID IN (活跃GTID列表)）+重试的机制。

## 悲观锁

### 概述

之所以叫做悲观锁，是因为这是一种对数据的修改抱有悲观态度的并发控制方式。我们一般认为数据被并发修改的概率比较大，所以需要在修改之前先加锁。

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。

**说明：**

**数据库中的行锁，表锁，读锁，写锁，以及syncronized实现的锁均为悲观锁**。

**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）**。

mysql中最常用的引擎是Innodb，Innodb默认使用的是行锁。而行锁是基于索引的，因此要想加上行锁，在加锁时必须命中索引，否则将使用表锁。

### 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。

### 特点

#### 优点

悲观并发控制实际上是“先取锁再访问”的保守策略，为数据处理的安全提供了保证。

**悲观锁利用数据库中的锁机制来实现数据变化的顺序执行**，这是最有效的办法。

#### 缺点

悲观锁依赖数据库锁，效率低。更新失败的概率比较低。

但是在效率方面，处理加锁的机制会让数据库产生额外的开销，还有增加产生死锁的机会；另外，还会降低并行性，一个事务如果锁定了某行数据，其他事务就必须等待该事务处理完才可以处理那行数据。

即，一个事务用悲观锁对数据加锁之后，其他事务将不能对加锁的数据进行除了查询以外的所有操作，如果该事务执行时间很长，那么其他事务将一直等待，那势必影响我们系统的吞吐量。

### 实现

悲观锁有两种实现方式：

1. 自行写原生SQL，然后写上for update语句；
2. 在Java中，使用@Lock注解，并且设置值为LockModeType.PESSIMISTIC\_WRITE即可代表行级锁。

悲观锁的实现，往往依靠数据库提供的锁机制。在数据库中，悲观锁的流程如下：

1. 在对记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）。
2. 如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。具体响应方式由开发者根据实际需要决定。
3. 如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。
4. 其间如果有其他对该记录做修改或加排他锁的操作，都会等待我们解锁或直接抛出异常。

**注意：**

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，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。set autocommit=0;

使用select…for update会把数据给锁住，不过我们需要注意一些锁的级别，MySQL InnoDB默认行级锁。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，会使用表级锁把整张表锁住，这点需要注意。

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。

## 区别

1. 乐观锁是一种思想，具体实现是，表中有一个版本字段，第一次读的时候，获取到这个字段。处理完业务逻辑开始更新的时候，需要再次查看该字段的值是否和第一次的一样。如果一样更新，反之拒绝。之所以叫乐观，因为这个模式没有从数据库加锁，等到更新的时候再判断是否可以更新。

乐观锁不是数据库层面上的锁，是需要自己手动去加的锁。

1. 悲观锁是数据库层面加锁，都会阻塞去等待锁。

**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

## 选择

在乐观锁与悲观锁的选择上面，主要看下两者的区别以及适用场景就可以了。

1、乐观锁并未真正加锁，效率高。一旦锁的粒度掌握不好，更新失败的概率就会比较高，容易发生业务失败。

2、悲观锁依赖数据库锁，效率低。更新失败的概率比较低。

随着互联网三高架构（高并发、高性能、高可用）的提出，悲观锁已经越来越少的被使用到生产环境中了，尤其是并发量比较大的业务场景。

**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**

1、用来解决读-写冲突和写-写冲突的的加锁并发控制

2、适用于写多读少，写冲突严重的情况，因为悲观锁是在读取数据的时候就加锁的，读多的场景会需要频繁的加锁和很多的的等待时间，而在写冲突严重的情况下使用悲观锁可以保证数据的一致性

3、数据一致性要求高

4、可以解决脏读，幻读，不可重复读，第一类更新丢失，第二类更新丢失的问题

**乐观锁**

1、解决写-写冲突的无锁并发控制

2、适用于读多写少，因为如果出现大量的写操作，写冲突的可能性就会增大，业务层需要不断重试，这会大大降低系统性能

3、数据一致性要求不高，但要求非常高的响应速度

4、无法解决脏读，幻读，不可重复读，但是可以解决更新丢失问题

**MVCC**

1、解决读-写冲突的无锁并发控制

2、与上面两者结合，提升它们的读性能

3、可以解决脏读，幻读，不可重复读等事务问题，更新丢失问题除外

# MDL锁

## 背景

当我们在MySQL中执行DDL语句时，经常会发现语句没有在你预期的时间完成，这时候我们通常会使用show full processlist，来看看发生了什么状况。当你看到waiting for table metadata lock时，那就碰到元数据锁了。

为了在并发环境下维护表元数据的数据一致性，在表上有活动事务（显式或隐式）的时候，不可以对元数据进行写入操作。因此从MySQL5.5版本开始引入了MDL锁（metadata lock），来保护表的元数据信息，用于解决或者保证DDL操作与DML操作之间的一致性。

对于引入MDL，其主要解决了2个问题，一个是**事务隔离问题**，比如在可重复隔离级别下，会话A在2次查询期间，会话B对表结构做了修改，两次查询结果就会不一致，无法满足可重复读的要求；另外一个是**数据复制的问题**，比如会话A执行了多条更新语句期间，另外一个会话B做了表结构变更并且先提交，就会导致slave在重做时，先重做alter，再重做update时就会出现复制错误的现象。

所以在对表进行上述操作时，如果表上有活动事务（未提交或回滚），请求写入的会话会等待在Metadata lock wait 。例如下面的这种情形：

[](http://www.ywnds.com/wp-content/uploads/2016/11/2016112810215353.png)

若没有MDL锁的保护，则事务2可以直接执行DDL操作，并且导致事务1出错，5.1版本即是如此。5.5版本加入MDL锁就在于保护这种情况的发生，由于事务1开启了查询，那么获得了MDL锁，锁的模式为SHARED\_READ，事务2要执行DDL，则需获得EXCLUSIVE锁，两者互斥，所以事务2需要等待。

注：支持事务的InnoDB引擎表和不支持事务的MyISAM引擎表，都会出现Metadata Lock Wait等待现象。一旦出现Metadata Lock Wait等待现象，后续所有对该表的访问都会阻塞在该等待上，导致连接堆积，业务受影响。

## 概述

### 定义

MDL全称为metadata lock，即元数据锁，一般也可称为字典锁。MDL的主要作用是为了管理数据库对象的并发访问和确保元数据一致性。元数据锁适用对象包含：table、schema、procedures、functions、triggers、scheduled events、tablespaces。

注：在创建用户赋权的时候不能并发执行，否则会出现元数据锁，无法创建用户，需要解锁后才可以继续创建用户。

**这个锁等待比较特殊，在innodb\_lock\_wait和show engine innodb status 表里面都查不到**。

### 读写锁

metadata lock是表级锁，是在server层加的，适用于所有存储引擎。所有的dml操作都会在表上加一个metadata读锁；所有的ddl操作都会在表上加一个metadata写锁。读锁和写锁的阻塞关系如下：

1. 读锁和写锁之间相互阻塞，即同一个表上的dml和ddl之间互相阻塞。
2. 写锁和写锁之间互相阻塞，即两个session不能对表同时做表定义变更，需要串行操作。
3. 读锁和读锁之间不会产生阻塞。也就是增删改查不会因为metadata lock产生阻塞，可以并发执行，日常工作中大家看到的dml之间的锁等待是innodb行锁引起的，和metadata lock无关。

熟悉innodb行锁的同学这里可能有点困惑，因为行锁分类和metadata lock很类似，也主要分为读锁和写锁，或者叫共享锁和排他锁，读写锁之间阻塞关系也一致。二者最重要的区别一个是表锁，一个是行锁，且行锁中的读写操作对应在metadata lock中都属于读锁。

### 获取规则

1、语句逐个（one by one）获取元数据锁，不是同时获取，并在获取过程中执行死锁检测。

2、DML语句获取锁按照语句中table出现的顺序来获取锁。

3、DDL语句、LOCK TABLES和其他类似语句按名称顺序获取锁，对于隐式使用的表（例如外键关系中也必须锁定的表）可能会以不同的顺序获取锁。

4、DDL的写锁请求优先级高于DML

### 生命周期

表上的metadata lock的生命周期从事务中的第一条涉及自身的语句开始，到整个事务结束而结束。而5.5之前是基于语句的，事务中执行完语句就释放，如果此时另外一个session对表做了一个删字段操作，那么就会造成两个问题：

1. ddl操作如果先于事务完成，那么binlog中ddl就会排在事务之前，明显和逻辑不符，触发bug。
2. 如果是RR隔离级别，那么事务中此表第二次执行将无法返回同样的结果，无法满足可重复读的要求。

所以，如果要降低metadata lock的锁等待时间，最好要及时提交事务，同时尽量避免大事务。

那么如果发生metadata lock锁等待，等待锁的session会等待多长时间呢？大家都知道MySQL里面行锁等待有个超时时间（参数innodb\_lock\_wait\_timeout），默认50s。metadata lock也有类似参数控制：



当然，生产环境中，我们很少会等待metadata lock超时，更多的是要想办法把产生metadata lock的源头找到，快速提交或者回滚，或者想办法kill掉。

### 模拟加锁规则

两个相同表结构的表t和t\_new开始。三个线程来操作这些表：

#### 场景一：LOCK TABLE

线程1:

LOCK TABLE t WRITE， t\_new WRITE;

该语句按表名顺序在 t 和 t\_new 上获取写锁  
 线程2:

INSERT INTO t VALUES(1);

该语句处于也需要获取表t上的MDL所以处于等待状态  
 线程 3:

RENAME TABLE t TO t\_old， t\_new TO t;

该语句需要按表名顺序在t、t\_new、t\_old上获取互斥锁，所以也处于等待状态  
 线程1:

UNLOCK TABLES;

该语句释放对t和t\_new的写锁定。线程3对t加写锁的优先级高于线程2，因此线程3在t上优先获得互斥锁，然后依次在t\_new、t\_old上获取互斥锁，执行重命名后释放其锁定。线程2获得t上的写锁，执行插入操作，然后释放其锁定。rename操作在insert之前执行。

另外一种场景：

两个具有相同表结构的表t和new\_t，同样是三个线程来操作这些表

线程1:

LOCK TABLE t WRITE， new\_t WRITE;

该语句按表名顺序在new\_t和t上获取写锁                
 线程2:

INSERT INTO t VALUES(1);

该语句处于也需要获取表t上的MDL所以处于等待状态  
 线程3:

RENAME TABLE t TO old\_t， new\_t TO t;

该语句需要按表名顺序在new\_t、old\_t、t上获取互斥锁，所以也处于等待状态  
 该语句释放对t和new\_t的写锁定。对于t首先发起锁请求的是线程2 ，因此线程2优先获得了t上的元数据写锁，执行完插入操作，然后释放该锁。线程3首先获取的是new\_t 、old\_t的互斥锁，最后才会请求t上的互斥锁，所以线程3在线程2执行完毕之前都是处于等待状态的。rename操作在insert操作之后。

#### 场景二：当前有执行DML操作时执行ALTRE操作

# SESSION A

mysql> insert into sbtest2 select \* from sbtest1;

# SESSION B

mysql> alter table sbtest2 add test1 int; //等待SESSION A执行完;

# SESSION C



# SESSION D

mysql> select \* from sbtest2 limit 10; //等待元数据锁;

# SESSION E



从上述例子可以看出，我们在执行DDL语句的时候得事先看一下，进程中是否已经存在某些DML语句占用了表的元数据锁，这样会导致DDL语句处于锁等待状态。一旦出现Waiting for table metadata lock等待现象，后续所有对该表的访问都会阻塞在该等待上，包括读操作，导致连接堆积，业务受影响。

#### 场景三：当前有对表的长时间查询或使用mysqldump/mysqlpump时，使用alter会被堵住

# SESSION A

mysql> select \*，sleep(10) from sbtest2;

# SESSION B

mysql> alter table sbtest2 add test2 int; //等待SESSION A执行完;

# SESSION C



#### 场景四：显示或者隐式开启事务后未提交或回滚，比如查询完成后未提交或者回滚，使用alter会被堵住

# SESSION A

mysql> begin;

mysql> select \* from sbtest2;

# SESSION B

mysql> alter table sbtest2 add test2 int; //等待SESSION A执行完;

# SESSION C



#### 场景五：表上有失败的查询事务，比如查询不存在的列，语句失败返回，但是事务没有提交，此时alter仍然会被堵住

# SESSION A

mysql> begin;

mysql> select error from sbtest2;

ERROR 1054 (42S22): Unknown column 'error' in 'field list'

# SESSION B

mysql> alter table sbtest2 add test3 int; //等待SESSION A提交或回滚;

# SESSION C



# SESSION D

mysql> select \* from information\_schema.innodb\_trx;

Empty set (0.00 sec)

其实SESSION A中的事务并未开启，但是由于select获取表元数据的语句，语法上是有效的，虽然执行失败了，但是任然不会释放元数据锁，故而导致SESSION B的alter动作被阻塞。

通过SESSION D查看当前打开事务时，你会发现没有，从而找不到原因。所以当出现这种场景时，如何判断是哪个进程导致的呢，我们可以尝试查看表performance\_schema. events\_statements\_current，分析进程状态来进行判断。



然后找到其sid，kill掉该session，也可以kill掉DDL所在的session解决可以解决此问题。

另外，测试时SESSION A要显式开启一个事务，否则查询会隐式回滚结束，无法重现上面的场景。SESSION B执行alter后，没有立即阻塞住，而是立马开始copy to tmp table，这个过程结束后，才进行了MDL锁等待。这怎么解释呢，应该是执行alter操作主要分为创建临时新表->插入老表的数据->临时新表rename to老表三个步骤，在这种情况下，到最后一步才需要MDL锁，所以copy过程中不会阻塞。由于没有查询在进行，而且查询也没有进入innodb层 (失败返回)，所以show processlist和information\_schema.innodb\_trx没有可以参考的信息。

出现以上几种情况时，这个时候如果进行如下操作就会引起MDL：

1. 创建、删除索引。
2. 修改表结构。
3. 表维护操作（optimize table、repair table等）。
4. 删除表。
5. 获取表上表级写锁 (lock table tab\_name write)。

## 源码分析

参考：

MDL 锁的实现与获取机制：<https://zhuanlan.zhihu.com/p/515017257>

MDL之LF\_HASH：<https://kernelmaker.github.io/MySQL_lf_allocator>

LF\_HASH无锁实现：<https://zhuanlan.zhihu.com/p/452849776>

## 监控

如何监控元数据锁？

performance\_schema.metadata\_locks 表中记录了元数据锁相关的信息，开启方式如下：在线开启 metadata\_locks，操作如下：

UPDATE performance\_schema.setup\_consumers SET ENABLED = 'YES' WHERE NAME ='global\_instrumentation';--此值默认已开启了，可检查确认。  
 UPDATE performance\_schema.setup\_instruments SET ENABLED = 'YES' WHERE NAME ='wait/lock/metadata/sql/mdl';

若可停库维护，则在 my.cnf 中添加如下：

[mysqld]performance-schema-instrument='wait/lock/metadata/sql/mdl=ON'

## 优化

如何优化元数据锁？

MDL锁一旦发生会对业务造成极大影响，因为后续所有对该表的访问都会被阻塞，造成连接积压。我们日常要尽量避免MDL锁的发生，下面给出几点优化建议可供参考：

1. 开启metadata\_locks表记录MDL锁
2. 设置参数lock\_wait\_timeout为较小值，使被阻塞端主动停止
3. 规范使用事务，及时提交事务，避免使用大事务
4. 增强监控告警，及时发现MDL锁
5. DDL操作及备份操作放在业务低峰期执行。

注：对于这种MDL是需要及时发现并且进行业务改造避免的。

## 故障排查

### SHOW PROFILING

当mysql运行一条SQL语句时，在你预期的时间内，没有完成时，我们都会登陆到mysql数据库上想查看是不是出了什么问题，通常会使用的一个命令就是 show processlist，看看有哪些session，这些session在做什么事情。就从这个命令开始，显示如下：



图中看到了显示了几处信息：

 id：为session\_id，也就是processlist\_id

 user：该session使用什么用户登陆的mysql数据库

 host：客户端登陆的ip地址（这里我都是本地登陆的）

 db：连接了哪个数据库（这里我只是连接上了数据库，并没有其他操作，所以都是NULL）

 command：当前session执行命令的类型

 Time：处于当前命令类型持续的时间

 State：当前命令类型的状态

 Info：具体命令信息

使用Profile分析场景四：显示或者隐式开启事务后未提交或回滚，比如查询完成后未提交或者回滚，使用alter会被堵住

# SESSION A

mysql> set profiling=on;

mysql> begin;

mysql> select \* from sbtest.sbtest2 limit 1;

# SESSION B

mysql> set profiling=on;

mysql> alter table sbtest.sbtest5 add test2 int; //等待SESSION A执行完;

# SESSION C

mysql> set profiling=on;

mysql> select \* from sbtest.sbtest2 limit 1;

# SESSION D

mysql> set profiling=on;

mysql> show processlist;



然后回滚SESSION A，等待SESSION B和SESSION C执行完，查看profile。

查看SESSION A





查看SESSION C



从上述测试可以看出，SESSION C需要打开表时碰到了元数据锁。MySQL不论SESSION A执行的是select还是delete，此时alter table语句无法获取到metadata独占锁，会进行等待；所以会影响SESSION C的读取。

这是最基本的一种情形，这个和MySQL 5.6中的online ddl并不冲突。一般alter table的操作过程中，在after create步骤会获取metadata独占锁，当进行到altering table的过程时（通常是最花时间的步骤），对该表的读写都可以正常进行，这就是online ddl的表现，并不会像之前在整个alter table过程中阻塞写入，当然并不是所有ALTER语句都支持online ddl。

总之，alter table的语句是很危险的（其实他的危险其实是未提交事物或者长事务导致的），在操作之前最好确认对要操作的表没有任何进行中的操作、没有未提交事务、也没有显式事务中的报错语句。如果有alter table的维护任务，在无人监管的时候运行，最好通过lock\_wait\_timeout设置好超时时间，避免长时间的metedata锁等待。

### performance\_schema

#### INNODB\_TRX

SQL语句：

SELECT

t.PROCESSLIST\_ID，

t.PROCESSLIST\_USER，

t.PROCESSLIST\_HOST，

t.PROCESSLIST\_DB，

t.PROCESSLIST\_STATE，

t.PROCESSLIST\_COMMAND，

t.PROCESSLIST\_TIME，

t.PROCESSLIST\_INFO，

e.CURRENT\_SCHEMA，

group\_concat(e.SQL\_TEXT separator '\n') as sql\_text

FROM

`performance\_schema`.threads t ，

`information\_schema`.INNODB\_TRX trx ，

`performance\_schema`.events\_statements\_history e

WHERE

t.thread\_id = e.thread\_id and

t.PROCESSLIST\_ID = trx.trx\_mysql\_thread\_id

group by t.THREAD\_ID desc;

如果从INNODB\_TRX只能查到一个会话，那就算运气很好了，肯定是这个造成的。

那么如果碰到多于一个的情况呢，可以通过PROCESSLIST\_INFO字段里面的sql来判断出来是哪一个。但是有些情况下，持有metadata锁的会话是在sleep状态下的。也就是说造成持有这个锁的语句已经执行过了，但是由于没有提交或者回滚，导致会话还是持有着这个锁。如果碰到这种情况，PROCESSLIST\_INFO字段就可能是空的了，那就只能通过判断会话已经执行过的语句来猜了。

当然猜也不是瞎猜，是有根据的猜。mysql有一个events\_statements\_history表，可以通过连接这个表来查看会话执行过什么语句。如果有涉及到等待锁的表的语句就能大概猜出来是哪一个了。比如下面这个结果

+----------------+------------------+------------------+----------------+-------------------+---------------------+------------------+------------------+----------------+--------------------------------------------------------------------------------------------------------------------------------------------------------------+

| PROCESSLIST\_ID | PROCESSLIST\_USER | PROCESSLIST\_HOST | PROCESSLIST\_DB | PROCESSLIST\_STATE | PROCESSLIST\_COMMAND | PROCESSLIST\_TIME | PROCESSLIST\_INFO | CURRENT\_SCHEMA | sql\_text |

+----------------+------------------+------------------+----------------+-------------------+---------------------+------------------+------------------+----------------+--------------------------------------------------------------------------------------------------------------------------------------------------------------+

| 12268 | root | localhost | NULL | NULL | Sleep | 435 | NULL | NULL | select @@version\_comment limit 1;select USER();begin;select \* from t;select \* from test.t;update t set b = 4 where a = 3;update test.t set b = 4 where a = 3 |

+----------------+------------------+------------------+----------------+-------------------+---------------------+------------------+------------------+----------------+--------------------------------------------------------------------------------------------------------------------------------------------------------------+

最后的字段显示了这个会话执行过了什么语句。12268会话就执行过一个select，两个update等等语句。这很可能说明12268会话持有t表的metadata锁。一般来说杀了这个会话就可以解决问题了。

但是这个events\_statements\_history有一个限制，他不会存储所有执行过的语句，而是存储最新执行过的N个语句。这个N是由performance\_schema\_events\_statements\_history\_size控制的，该参数是只读的，只能重启mysql生效，默认是10。如果执行过的语句超过了这个限制，那么很有可能是看不到对这个表进行修改的语句的，这个时候就需要猜了。

#### metadata\_locks

从mysql5.7开始，有了performance\_schema.metadata\_locks表，用于显示等待和持有metadata锁的会话信息。有效的简化了处理metadata锁等待的方法。

5.7版本该特性不是默认开启的，需要手动启动。8.0开始是默认开启的不需要配置。

SQL语句：

SELECT

t1.OBJECT\_SCHEMA，

t1.OBJECT\_NAME，

t1.LOCK\_TYPE，

t1.LOCK\_STATUS，

t2.PROCESSLIST\_ID，

t2.PROCESSLIST\_USER，

t2.PROCESSLIST\_HOST，

t2.PROCESSLIST\_DB，

t2.PROCESSLIST\_COMMAND，

t2.PROCESSLIST\_STATE，

t2.PROCESSLIST\_INFO

FROM

`performance\_schema`.metadata\_locks t1，

`performance\_schema`.threads t2

WHERE

t1.owner\_thread\_id = t2.thread\_id

AND t1.OBJECT\_SCHEMA = 'test'

AND t1.OBJECT\_NAME = 't'

只有开启特性以后出现的持有或者等待metadata锁的会话才会被记录。

+---------------+-------------+-------------------+-------------+----------------+------------------+------------------+----------------+---------------------+---------------------------------+--------------------------------+

| OBJECT\_SCHEMA | OBJECT\_NAME | LOCK\_TYPE | LOCK\_STATUS | PROCESSLIST\_ID | PROCESSLIST\_USER | PROCESSLIST\_HOST | PROCESSLIST\_DB | PROCESSLIST\_COMMAND | PROCESSLIST\_STATE | PROCESSLIST\_INFO |

+---------------+-------------+-------------------+-------------+----------------+------------------+------------------+----------------+---------------------+---------------------------------+--------------------------------+

| test | t | SHARED\_WRITE | GRANTED | 12268 | root | localhost | test | Sleep | NULL | NULL |

| test | t | SHARED\_READ | GRANTED | 12268 | root | localhost | test | Sleep | NULL | NULL |

| test | t | SHARED\_UPGRADABLE | GRANTED | 12378 | root | localhost | test | Query | Waiting for table metadata lock | alter table t add column f int |

| test | t | EXCLUSIVE | PENDING | 12378 | root | localhost | test | Query | Waiting for table metadata lock | alter table t add column f int |

+---------------+-------------+-------------------+-------------+----------------+------------------+------------------+----------------+---------------------+---------------------------------+--------------------------------+

上面的结果中LOCK\_STATUS字段表示连接对于matadata锁的持有状态，GRANTED表示持有，PENDING表示等待。很容易可以看出来12268会话持有了锁，而12378会话正在等待这个锁。通过kill 12268语句，或者让12268提交回滚都能解决这个问题。

### 实践

快速解决问题永远是第一位的，一旦出现长时间的metadata lock，尤其是在访问频繁的业务表上产生，通常会导致表无法访问，读写全被阻塞，此时找到阻塞源头是第一位的。这里最重要的表就是前面提到过的performance\_schema.metadata\_locks表。

metadata\_locks是5.7中被引入，记录了metadata lock的相关信息，包括持有对象、类型、状态等信息。但5.7默认设置是关闭的（8.0默认打开），需要通过下面命令打开设置：

UPDATE performance\_schema.setup\_instruments SET ENABLED = 'YES'， TIMED = 'YES'WHERE NAME = 'wait/lock/metadata/sql/mdl';

如果要永久生效，需要在配置文件中加入如下内容：

[mysqld]performance-schema-instrument='wait/lock/metadata/sql/mdl=ON'

单纯查询这个表无法得出具体的阻塞关系，也无法得知什么语句造成的阻塞，这里要关联另外两个表performance\_schema.thread和performance\_schema.events\_statements\_history，thread表可以将线程id和show processlist中id关联，events\_statements\_history表可以得到事务的历史sql，关联后的完整sql如下：

SELECT

locked\_schema，

locked\_table，

locked\_type，

waiting\_processlist\_id，

waiting\_age，

waiting\_query，

waiting\_state，

blocking\_processlist\_id，

blocking\_age，

substring\_index(sql\_text，"transaction\_begin;" ，-1) AS blocking\_query，

sql\_kill\_blocking\_connection

FROM

(

SELECT

b.OWNER\_THREAD\_ID AS granted\_thread\_id，

a.OBJECT\_SCHEMA AS locked\_schema，

a.OBJECT\_NAME AS locked\_table，

"Metadata Lock" AS locked\_type，

c.PROCESSLIST\_ID AS waiting\_processlist\_id，

c.PROCESSLIST\_TIME AS waiting\_age，

c.PROCESSLIST\_INFO AS waiting\_query，

c.PROCESSLIST\_STATE AS waiting\_state，

d.PROCESSLIST\_ID AS blocking\_processlist\_id，

d.PROCESSLIST\_TIME AS blocking\_age，

d.PROCESSLIST\_INFO AS blocking\_query，

concat('KILL '， d.PROCESSLIST\_ID) AS sql\_kill\_blocking\_connection

FROM

performance\_schema.metadata\_locks a

JOIN performance\_schema.metadata\_locks b ON a.OBJECT\_SCHEMA = b.OBJECT\_SCHEMA

AND a.OBJECT\_NAME = b.OBJECT\_NAME

AND a.lock\_status = 'PENDING'

AND b.lock\_status = 'GRANTED'

AND a.OWNER\_THREAD\_ID <> b.OWNER\_THREAD\_ID

AND a.lock\_type = 'EXCLUSIVE'

JOIN performance\_schema.threads c ON a.OWNER\_THREAD\_ID = c.THREAD\_ID

JOIN performance\_schema.threads d ON b.OWNER\_THREAD\_ID = d.THREAD\_ID

) t1，

(

SELECT

thread\_id，

group\_concat( CASE WHEN EVENT\_NAME = 'statement/sql/begin' THEN "transaction\_begin" ELSE sql\_text END ORDER BY event\_id SEPARATOR ";" ) AS sql\_text

FROM

performance\_schema.events\_statements\_history

GROUP BY thread\_id

) t2

WHERE

t1.granted\_thread\_id = t2.thread\_id \G

对于前面的例子执行此sql，得到一个清晰的阻塞关系：

locked\_schema: db1

locked\_table: t1

locked\_type: Metadata Lock

waiting\_processlist\_id: 28

waiting\_age: 227

waiting\_query: alter table t1 add cl3 int

waiting\_state: Waiting for table metadata lock

blocking\_processlist\_id: 27

blocking\_age: 252

blocking\_query: select \* from t1

sql\_kill\_blocking\_connection: KILL 27

1 row in set， 1 warning (0.00 sec)

根据显示结果，processlist\_id为27的线程阻塞了28的线程，我们需要kill 27即可解锁。

实际上，MySQL也提供了一个类似的视图来解决metadata lock问题，视图名称为sys.schema\_table\_lock\_waits，但此视图查询结果有bug，不是很准确，建议大家还是参考上面sql。

# SELECT锁

InnoDB有两种不同的SELECT，即普通SELECT和锁定读SELECT。锁定读SELECT 又有两种，即SELECT ... FOR SHARE 和 SELECT ... FOR UPDATE；锁定读SELECT 之外的则是 普通SELECT。

不同的SELECT是否都需要加锁呢？

**1、普通SELECT 时使用一致性非锁定读（即MVCC），不加锁；**

**2、锁定读SELECT使用锁定读，加锁；**

3、此外，DML(INSERT/UPDATE/DELETE)时，需要先查询表中的记录，此时也使用锁定读，加锁；

FOR SHARE 语法是 MySQL 8.0 时加入的，FOR SHARE 和 LOCK IN SHARE MODE 是等价的，但，FOR SHARE 用于替代 LOCK IN SHARE MODE，不过，为了向后兼容，LOCK IN SHARE MODE依然可用。

## 一致性非锁定读(consistent nonlocking read)

### 背景

乐观并发控制和悲观并发控制都是通过延迟或者终止相应的事务来解决事务之间的竞争条件来保证事务的可串行化；虽然前面的两种并发控制机制确实能够从根本上解决并发事务的可串行化的问题，但是其实都是在解决写冲突的问题，两者区别在于对写冲突的乐观程度不同(悲观锁也能解决读写冲突问题，但是性能就一般了)。而在实际使用过程中，数据库读请求是写请求的很多倍，我们如果能解决读写并发的问题的话，就能更大地提高数据库的读性能，而这就是多版本并发控制所能做到的事情。  
 **与悲观并发控制和乐观并发控制不同的是，MVCC是为了解决读写锁造成的多个、长时间的读操作饿死写操作问题，也就是解决读写冲突的问题。MVCC可以与前两者中的任意一种机制结合使用，以提高数据库的读性能**。  
 数据库的悲观锁基于提升并发性能的考虑，一般都同时实现了多版本并发控制。不仅是MySQL，包括Oracle、PostgreSQL等其他数据库系统也都实现了MVCC，但各自的实现机制不尽相同，因为MVCC没有一个统一的实现标准。  
 总的来说，MVCC的出现就是数据库不满用悲观锁去解决读-写冲突问题，因性能不高而提出的解决方案。

### 概述

InnoDB采用多版本并发控制(MVCC， multiversion concurrency control)来增加读操作的并发性。MVCC是指，InnoDB使用基于时间点的快照来获取查询结果，读取时在访问的表上不设置任何锁，因此，在事务T1读取的同一时刻，事务T2可以自由的修改事务T1所读取的数据。这种读操作被称为一致性非锁定读。这里的读操作就是普通SELECT。

多版本并发控制(Multiversion concurrency control，MCC或MVCC)，是数据库管理系统常用的一种并发控制，也用于程序设计语言实现事务内存。

### MVCC与隔离级别

**隔离级别为RU和Serializable时不需要MVCC，因此，只有RC和RR时，才存在MVCC，才存在一致性非锁定读**。

一致性非锁定读在两种隔离级别RC和RR时，是否有什么不同呢？是的，两种隔离级别下，拍得快照的时间点不同

1、RC时，同一个事务内的每一个一致性读总是设置和读取它自己的最新快照。也就是说，**每次读取时，都再重新拍得一个最新的快照（所以，RC时总是可以读取到最新提交的数据）**。

2、RR时，同一个事务内的所有的一致性读总是读取同一个快照，此快照是执行该事务的第一个一致性读时所拍得的。

### 实现方式

MVCC的实现，是通过保存数据在某个时间点的快照来实现的。每个事务读到的数据项都是一个历史快照，被称为快照读，不同于当前读的是快照读读到的数据可能不是最新的，但是快照隔离能使得在整个事务看到的数据都是它启动时的数据状态。而写操作不覆盖已有数据项，而是创建一个新的版本，直至所在事务提交时才变为可见。

什么是MySQL InnoDB下的当前读和快照读?

#### 当前读

像**select lock in share mode(共享锁)，select for update ; update， insert ，delete(排他锁)这些操作都是一种当前读**，为什么叫当前读？就是它读取的是记录的最新版本，读取时还要保证其他并发事务不能修改当前记录，会对读取的记录进行加锁。

#### 快照读

像不加锁的select操作就是快照读，即不加锁的非阻塞读；**快照读的前提是隔离级别不是未提交读和串行化级别**，因为未提交读总是读取最新的数据行，而不是符合当前事务版本的数据行。而**串行化则会对所有读取的行都加锁**。

### 优缺点

**MVCC使大多数读操作都可以不用加锁，这样设计使得读数据操作很简单，性能很好，并且也能保证只会读取到符合标准的行**。不足之处是每行记录都需要额外的存储空间，需要做更多的行检查工作，以及一些额外的维护工作。

## 锁定读(locking read)

如果你先查询数据，然后，在同一个事务内插入/更新相关数据，普通的SELECT语句是不能给你足够的保护的。其他事务可以 更新/删除 你刚刚查出的数据行。InnoDB提供两种锁定读，即：SELECT ... FOR SHARE和SELECT ... FOR UPDATE。它俩都能提供额外的安全性。

这两种锁定读在搜索时所遇到的（注意：不是最终结果集中的）每一条索引记录(index record)上设置排它锁或共享锁。此外，如果当前隔离级别是RR，它还会在每个索引记录前面的间隙上设置排它的或共享的gap lock（排它的和共享的gap lock没有任何区别，二者等价）。

# INSERT锁

# UPDATE锁

# DELETE锁

# 查看锁信息

## information\_schema.innodb\_locks

## show engine innodb status

有多种方法可以查看InnoDB中锁的情况，例如：

select \* from information\_schema.innodb\_locks; #锁的概况

show engine innodb status; #InnoDB整体状态，其中包括锁的情况

下面来看一个例子：

#在事务A中执行：

start transaction;

update account SET balance = 1000 where id = 1;

#在事务B中执行：

start transaction;

update account SET balance = 2000 where id = 1;

此时查看锁的情况：



show engine innodb status查看锁相关的部分：



通过上述命令可以查看事务24052和24053占用锁的情况；其中lock\_type为RECORD，代表锁为行锁(记录锁)；lock\_mode为X，代表排它锁(写锁)。

## performance\_schema.data\_locks

## performance\_schema.data\_lock\_waits

# 锁优化

## MyISAM存储引擎

在使用MyISAM存储引擎时，执行SQL语句，会自动为SELECT语句加上共享锁，为UDI（更新，删除，插入）操作加上排他锁。

由于这个特性在多进程并发插入同一张表的时候，就会因为排他锁而进行等待。

因此可以通过配置concurrent\_insert系统变量，来控制其并发的插入行为。

1、concurrent\_insert=0时，不允许并发插入。

2、concurrent\_insert=1时，如果 MyISAM 表中没有空洞（即表中没有被删除的行），允许一个进程读表时，另一个进程向表的尾部插入记录（MySQL 默认设置）。

注：空洞是行记录被删除以后，只是被标记为“已删除”其存储空间没有被回收，也就是说没有被物理删除。由另外一个进程，异步对这个数据进行删除。

因为空间长度问题，删除以后的物理空间不能被新的记录所使用，从而形成了空洞。

3、concurrent\_insert=2时，无论MyISAM表中有没有空洞，都允许在表尾并发插入记录。

如果在数据插入的时候，没有并发删除操作的话，可以尝试把 concurrent\_insert设置为 1。

反之，在数据插入的时候有删除操作且量较大时，也就是会产生“空洞”的时候，就需要把concurrent\_insert设置为2。

另外，当一个进程请求某个MyISAM表的读锁，另一个进程也请求同一表的写锁。

即使读请求先到达，写请求后到达，写请求也会插到读请求之前。因为MySQL的默认设置认为，写请求比读请求重要。

我们可以通过low\_priority\_updates来调节读写行为的优先级：

数据库以读为主时，要优先保证查询性能时，可通过low\_priority\_updates=1设置读优先级高于写优先级。

数据库以写为主时，则不用设置 low\_priority\_updates参数。

## InnoDB优化建议

从锁机制的实现方面来说，InnoDB的行级锁带来的性能损耗可能比表级锁要高一点，但在并发方面的处理能力远远优于MyISAM的表级锁。这也是大多数公司的MySQL都是使用InnoDB模式的原因。

但是，InnoDB也有脆弱的一面，下面提出几个优化建议供大家参考：

尽可能让数据检索通过索引完成，避免InnoDB因为无法通过索引加行锁，而导致升级为表锁的情况。换句话说就是，多用行锁，少用表锁。

加索引的时候尽量准确，避免造成不必要的锁定影响其他查询。

尽量减少给予范围的数据检索（间隙锁），避免因为间隙锁带来的影响，锁定了不该锁定的记录。

尽量控制事务的大小，减少锁定的资源量和锁定时间。

尽量使用较低级别的事务隔离，减少MySQL因为事务隔离带来的成本。