**Mysql锁详解**

**一、概述**

数据库锁定机制简单来说，就是数据库为了保证数据的一致性，而使各种共享资源在被并发访问变得有序所设计的一种规则。对于任何一种数据库来说都需要有相应的锁定机制，所以MySQL自然也不能例外。MySQL数据库由于其自身架构的特点，存在多种数据存储引擎，每种存储引擎所针对的应用场景特点都不太一样，为了满足各自特定应用场景的需求，每种存储引擎的锁定机制都是为各自所面对的特定场景而优化设计，所以各存储引擎的锁定机制也有较大区别。MySQL各存储引擎使用了三种类型（级别）的锁定机制：表级锁定，行级锁定和页级锁定。  
**1.表级锁定（table-level）**  
表级别的锁定是MySQL各存储引擎中最大颗粒度的锁定机制。该锁定机制最大的特点是实现逻辑非常简单，带来的系统负面影响最小。所以获取锁和释放锁的速度很快。由于表级锁一次会将整个表锁定，所以可以很好的避免困扰我们的死锁问题。  
当然，锁定颗粒度大所带来最大的负面影响就是出现锁定资源争用的概率也会最高，致使并大度大打折扣。  
使用表级锁定的主要是MyISAM，MEMORY，CSV等一些非事务性存储引擎。  
**2.行级锁定（row-level）**  
行级锁定最大的特点就是锁定对象的颗粒度很小，也是目前各大数据库管理软件所实现的锁定颗粒度最小的。由于锁定颗粒度很小，所以发生锁定资源争用的概率也最小，能够给予应用程序尽可能大的并发处理能力而提高一些需要高并发应用系统的整体性能。  
虽然能够在并发处理能力上面有较大的优势，但是行级锁定也因此带来了不少弊端。由于锁定资源的颗粒度很小，所以每次获取锁和释放锁需要做的事情也更多，带来的消耗自然也就更大了。此外，行级锁定也最容易发生死锁。  
使用行级锁定的主要是InnoDB存储引擎。  
**3.页级锁定（page-level）**  
页级锁定是MySQL中比较独特的一种锁定级别，在其他数据库管理软件中也并不是太常见。页级锁定的特点是锁定颗粒度介于行级锁定与表级锁之间，所以获取锁定所需要的资源开销，以及所能提供的并发处理能力也同样是介于上面二者之间。另外，页级锁定和行级锁定一样，会发生死锁。  
在数据库实现资源锁定的过程中，随着锁定资源颗粒度的减小，锁定相同数据量的数据所需要消耗的内存数量是越来越多的，实现算法也会越来越复杂。不过，随着锁定资源颗粒度的减小，应用程序的访问请求遇到锁等待的可能性也会随之降低，系统整体并发度也随之提升。  
使用页级锁定的主要是BerkeleyDB存储引擎。  
总的来说，MySQL这3种锁的特性可大致归纳如下：  
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。  
适用：从锁的角度来说，表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如Web应用；而行级锁则更适合于有大量按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（OLTP）系统。

**二、表级锁定**

由于MyISAM存储引擎使用的锁定机制完全是由MySQL提供的表级锁定实现，所以下面我们将以MyISAM存储引擎作为示例存储引擎。  
**1.MySQL表级锁的锁模式**  
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）。锁模式的兼容性：  
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操作；  
MyISAM表的读操作与写操作之间，以及写操作之间是串行的。当一个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。  
**2.如何加表锁**  
MyISAM在执行查询语句（SELECT）前，会自动给涉及的所有表加读锁，在执行更新操作（UPDATE、DELETE、INSERT等）前，会自动给涉及的表加写锁，这个过程并不需要用户干预，因此，用户一般不需要直接用LOCK TABLE命令给MyISAM表显式加锁。  
**3.MyISAM表锁优化建议**  
对于MyISAM存储引擎，虽然使用表级锁定在锁定实现的过程中比实现行级锁定或者页级锁所带来的附加成本都要小，锁定本身所消耗的资源也是最少。但是由于锁定的颗粒度比较到，所以造成锁定资源的争用情况也会比其他的锁定级别都要多，从而在较大程度上会降低并发处理能力。所以，在优化MyISAM存储引擎锁定问题的时候，最关键的就是如何让其提高并发度。由于锁定级别是不可能改变的了，所以我们首先需要尽可能让锁定的时间变短，然后就是让可能并发进行的操作尽可能的并发。  
（1）查询表级锁争用情况  
MySQL内部有两组专门的状态变量记录系统内部锁资源争用情况：

[复制代码](javascript:void(0);)

mysql> show status like 'table%';

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

| Variable\_name | Value |

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

| Table\_locks\_immediate | **100** |

| Table\_locks\_waited | **11** |

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

[复制代码](javascript:void(0);)

这里有两个状态变量记录MySQL内部表级锁定的情况，两个变量说明如下：  
Table\_locks\_immediate：产生表级锁定的次数；  
Table\_locks\_waited：出现表级锁定争用而发生等待的次数；  
两个状态值都是从系统启动后开始记录，出现一次对应的事件则数量加1。如果这里的Table\_locks\_waited状态值比较高，那么说明系统中表级锁定争用现象比较严重，就需要进一步分析为什么会有较多的锁定资源争用了。  
（2）缩短锁定时间  
如何让锁定时间尽可能的短呢？唯一的办法就是让我们的Query执行时间尽可能的短。  
a)尽两减少大的复杂Query，将复杂Query分拆成几个小的Query分布进行；  
b)尽可能的建立足够高效的索引，让数据检索更迅速；  
c)尽量让MyISAM存储引擎的表只存放必要的信息，控制字段类型；  
d)利用合适的机会优化MyISAM表数据文件。  
（3）分离能并行的操作  
说到MyISAM的表锁，而且是读写互相阻塞的表锁，可能有些人会认为在MyISAM存储引擎的表上就只能是完全的串行化，没办法再并行了。大家不要忘记了，MyISAM的存储引擎还有一个非常有用的特性，那就是ConcurrentInsert（并发插入）的特性。  
MyISAM存储引擎有一个控制是否打开Concurrent Insert功能的参数选项：concurrent\_insert，可以设置为0，1或者2。三个值的具体说明如下：  
concurrent\_insert=2，无论MyISAM表中有没有空洞，都允许在表尾并发插入记录；  
concurrent\_insert=1，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。这也是MySQL的默认设置；  
concurrent\_insert=0，不允许并发插入。  
可以利用MyISAM存储引擎的并发插入特性，来解决应用中对同一表查询和插入的锁争用。例如，将concurrent\_insert系统变量设为2，总是允许并发插入；同时，通过定期在系统空闲时段执行OPTIMIZE TABLE语句来整理空间碎片，收回因删除记录而产生的中间空洞。  
（4）合理利用读写优先级  
MyISAM存储引擎的是读写互相阻塞的，那么，一个进程请求某个MyISAM表的读锁，同时另一个进程也请求同一表的写锁，MySQL如何处理呢？  
答案是写进程先获得锁。不仅如此，即使读请求先到锁等待队列，写请求后到，写锁也会插到读锁请求之前。  
这是因为MySQL的表级锁定对于读和写是有不同优先级设定的，默认情况下是写优先级要大于读优先级。  
所以，如果我们可以根据各自系统环境的差异决定读与写的优先级：  
通过执行命令SET LOW\_PRIORITY\_UPDATES=1，使该连接读比写的优先级高。如果我们的系统是一个以读为主，可以设置此参数，如果以写为主，则不用设置；  
通过指定INSERT、UPDATE、DELETE语句的LOW\_PRIORITY属性，降低该语句的优先级。  
虽然上面方法都是要么更新优先，要么查询优先的方法，但还是可以用其来解决查询相对重要的应用（如用户登录系统）中，读锁等待严重的问题。  
另外，MySQL也提供了一种折中的办法来调节读写冲突，即给系统参数max\_write\_lock\_count设置一个合适的值，当一个表的读锁达到这个值后，MySQL就暂时将写请求的优先级降低，给读进程一定获得锁的机会。  
这里还要强调一点：一些需要长时间运行的查询操作，也会使写进程“饿死”，因此，应用中应尽量避免出现长时间运行的查询操作，不要总想用一条SELECT语句来解决问题，因为这种看似巧妙的SQL语句，往往比较复杂，执行时间较长，在可能的情况下可以通过使用中间表等措施对SQL语句做一定的“分解”，使每一步查询都能在较短时间完成，从而减少锁冲突。如果复杂查询不可避免，应尽量安排在数据库空闲时段执行，比如一些定期统计可以安排在夜间执行。

**三、行级锁定**

行级锁定不是MySQL自己实现的锁定方式，而是由其他存储引擎自己所实现的，如广为大家所知的InnoDB存储引擎，以及MySQL的分布式存储引擎NDBCluster等都是实现了行级锁定。考虑到行级锁定君由各个存储引擎自行实现，而且具体实现也各有差别，而InnoDB是目前事务型存储引擎中使用最为广泛的存储引擎，所以这里我们就主要分析一下InnoDB的锁定特性。  
**1.InnoDB锁定模式及实现机制**  
考虑到行级锁定君由各个存储引擎自行实现，而且具体实现也各有差别，而InnoDB是目前事务型存储引擎中使用最为广泛的存储引擎，所以这里我们就主要分析一下InnoDB的锁定特性。  
总的来说，InnoDB的锁定机制和Oracle数据库有不少相似之处。InnoDB的行级锁定同样分为两种类型，共享锁和排他锁，而在锁定机制的实现过程中为了让行级锁定和表级锁定共存，InnoDB也同样使用了意向锁（表级锁定）的概念，也就有了意向共享锁和意向排他锁这两种。  
当一个事务需要给自己需要的某个资源加锁的时候，如果遇到一个共享锁正锁定着自己需要的资源的时候，自己可以再加一个共享锁，不过不能加排他锁。但是，如果遇到自己需要锁定的资源已经被一个排他锁占有之后，则只能等待该锁定释放资源之后自己才能获取锁定资源并添加自己的锁定。而意向锁的作用就是当一个事务在需要获取资源锁定的时候，如果遇到自己需要的资源已经被排他锁占用的时候，该事务可以需要锁定行的表上面添加一个合适的意向锁。如果自己需要一个共享锁，那么就在表上面添加一个意向共享锁。而如果自己需要的是某行（或者某些行）上面添加一个排他锁的话，则先在表上面添加一个意向排他锁。意向共享锁可以同时并存多个，但是意向排他锁同时只能有一个存在。所以，可以说InnoDB的锁定模式实际上可以分为四种：共享锁（S），排他锁（X），意向共享锁（IS）和意向排他锁（IX），我们可以通过以下表格来总结上面这四种所的共存逻辑关系：  


如果一个事务请求的锁模式与当前的锁兼容，InnoDB就将请求的锁授予该事务；反之，如果两者不兼容，该事务就要等待锁释放。  
意向锁是InnoDB自动加的，不需用户干预。对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁（X)；对于普通SELECT语句，InnoDB不会加任何锁；事务可以通过以下语句显示给记录集加共享锁或排他锁。

共享锁（S）：SELECT \* FROM table\_name WHERE ... LOCK IN SHARE MODE

排他锁（X)：SELECT \* FROM table\_name WHERE ... FOR UPDATE

用SELECT ... IN SHARE MODE获得共享锁，主要用在需要数据依存关系时来确认某行记录是否存在，并确保没有人对这个记录进行UPDATE或者DELETE操作。  
但是如果当前事务也需要对该记录进行更新操作，则很有可能造成死锁，对于锁定行记录后需要进行更新操作的应用，应该使用SELECT... FOR UPDATE方式获得排他锁。  
**2.InnoDB行锁实现方式**  
InnoDB行锁是通过给索引上的索引项加锁来实现的，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。下面通过一些实际例子来加以说明。  
（1）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。  
（2）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。  
（3）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。  
（4）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认为全表扫描效率更高，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。  
**3.间隙锁（Next-Key锁）**  
当我们用范围条件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（Next-Key锁）。  
例：  
假如emp表中只有101条记录，其empid的值分别是 1,2,...,100,101，下面的SQL：

mysql> select \* from emp where empid > **100** for update;

是一个范围条件的检索，InnoDB不仅会对符合条件的empid值为101的记录加锁，也会对empid大于101（这些记录并不存在）的“间隙”加锁。  
InnoDB使用间隙锁的目的：  
（1）防止幻读，以满足相关隔离级别的要求。对于上面的例子，要是不使用间隙锁，如果其他事务插入了empid大于100的任何记录，那么本事务如果再次执行上述语句，就会发生幻读；  
（2）为了满足其恢复和复制的需要。  
很显然，在使用范围条件检索并锁定记录时，即使某些不存在的键值也会被无辜的锁定，而造成在锁定的时候无法插入锁定键值范围内的任何数据。在某些场景下这可能会对性能造成很大的危害。  
除了间隙锁给InnoDB带来性能的负面影响之外，通过索引实现锁定的方式还存在其他几个较大的性能隐患：  
（1）当Query无法利用索引的时候，InnoDB会放弃使用行级别锁定而改用表级别的锁定，造成并发性能的降低；  
（2）当Query使用的索引并不包含所有过滤条件的时候，数据检索使用到的索引键所只想的数据可能有部分并不属于该Query的结果集的行列，但是也会被锁定，因为间隙锁锁定的是一个范围，而不是具体的索引键；  
（3）当Query在使用索引定位数据的时候，如果使用的索引键一样但访问的数据行不同的时候（索引只是过滤条件的一部分），一样会被锁定。  
因此，在实际应用开发中，尤其是并发插入比较多的应用，我们要尽量优化业务逻辑，尽量使用相等条件来访问更新数据，避免使用范围条件。  
还要特别说明的是，InnoDB除了通过范围条件加锁时使用间隙锁外，如果使用相等条件请求给一个不存在的记录加锁，InnoDB也会使用间隙锁。  
**4.死锁**  
上文讲过，MyISAM表锁是deadlock free的，这是因为MyISAM总是一次获得所需的全部锁，要么全部满足，要么等待，因此不会出现死锁。但在InnoDB中，除单个SQL组成的事务外，锁是逐步获得的，当两个事务都需要获得对方持有的排他锁才能继续完成事务，这种循环锁等待就是典型的死锁。  
在InnoDB的事务管理和锁定机制中，有专门检测死锁的机制，会在系统中产生死锁之后的很短时间内就检测到该死锁的存在。当InnoDB检测到系统中产生了死锁之后，InnoDB会通过相应的判断来选这产生死锁的两个事务中较小的事务来回滚，而让另外一个较大的事务成功完成。  
那InnoDB是以什么来为标准判定事务的大小的呢？MySQL官方手册中也提到了这个问题，实际上在InnoDB发现死锁之后，会计算出两个事务各自插入、更新或者删除的数据量来判定两个事务的大小。也就是说哪个事务所改变的记录条数越多，在死锁中就越不会被回滚掉。  
但是有一点需要注意的就是，当产生死锁的场景中涉及到不止InnoDB存储引擎的时候，InnoDB是没办法检测到该死锁的，这时候就只能通过锁定超时限制参数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得所需的锁而挂起，会占用大量计算机资源，造成严重性能问题，甚至拖跨数据库。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。  
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，绝大部分死锁都可以避免。下面就通过实例来介绍几种避免死锁的常用方法：  
（1）在应用中，如果不同的程序会并发存取多个表，应尽量约定以相同的顺序来访问表，这样可以大大降低产生死锁的机会。  
（2）在程序以批量方式处理数据的时候，如果事先对数据排序，保证每个线程按固定的顺序来处理记录，也可以大大降低出现死锁的可能。  
（3）在事务中，如果要更新记录，应该直接申请足够级别的锁，即排他锁，而不应先申请共享锁，更新时再申请排他锁，因为当用户申请排他锁时，其他事务可能又已经获得了相同记录的共享锁，从而造成锁冲突，甚至死锁。  
（4）在REPEATABLE-READ隔离级别下，如果两个线程同时对相同条件记录用SELECT...FOR UPDATE加排他锁，在没有符合该条件记录情况下，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，就可避免问题。  
（5）当隔离级别为READ COMMITTED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，如果没有，就插入记录。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。  
**5.什么时候使用表锁**  
对于InnoDB表，在绝大部分情况下都应该使用行级锁，因为事务和行锁往往是我们之所以选择InnoDB表的理由。但在个别特殊事务中，也可以考虑使用表级锁：  
（1）事务需要更新大部分或全部数据，表又比较大，如果使用默认的行锁，不仅这个事务执行效率低，而且可能造成其他事务长时间锁等待和锁冲突，这种情况下可以考虑使用表锁来提高该事务的执行速度。  
（2）事务涉及多个表，比较复杂，很可能引起死锁，造成大量事务回滚。这种情况也可以考虑一次性锁定事务涉及的表，从而避免死锁、减少数据库因事务回滚带来的开销。  
当然，应用中这两种事务不能太多，否则，就应该考虑使用MyISAM表了。  
在InnoDB下，使用表锁要注意以下两点。  
（1）使用LOCK TABLES虽然可以给InnoDB加表级锁，但必须说明的是，表锁不是由InnoDB存储引擎层管理的，而是由其上一层──MySQL Server负责的，仅当autocommit=0、InnoDB\_table\_locks=1（默认设置）时，InnoDB层才能知道MySQL加的表锁，MySQL Server也才能感知InnoDB加的行锁，这种情况下，InnoDB才能自动识别涉及表级锁的死锁，否则，InnoDB将无法自动检测并处理这种死锁。  
（2）在用 LOCK TABLES对InnoDB表加锁时要注意，要将AUTOCOMMIT设为0，否则MySQL不会给表加锁；事务结束前，不要用UNLOCK TABLES释放表锁，因为UNLOCK TABLES会隐含地提交事务；COMMIT或ROLLBACK并不能释放用LOCK TABLES加的表级锁，必须用UNLOCK TABLES释放表锁。正确的方式见如下语句：  
例如，如果需要写表t1并从表t读，可以按如下做：

SET AUTOCOMMIT=**0**;

LOCK TABLES t1 WRITE, t2 READ, ...;

[do something with tables t1 and t2 here];

COMMIT;

UNLOCK TABLES;

**6.InnoDB行锁优化建议**  
InnoDB存储引擎由于实现了行级锁定，虽然在锁定机制的实现方面所带来的性能损耗可能比表级锁定会要更高一些，但是在整体并发处理能力方面要远远优于MyISAM的表级锁定的。当系统并发量较高的时候，InnoDB的整体性能和MyISAM相比就会有比较明显的优势了。但是，InnoDB的行级锁定同样也有其脆弱的一面，当我们使用不当的时候，可能会让InnoDB的整体性能表现不仅不能比MyISAM高，甚至可能会更差。  
（1）要想合理利用InnoDB的行级锁定，做到扬长避短，我们必须做好以下工作：  
a)尽可能让所有的数据检索都通过索引来完成，从而避免InnoDB因为无法通过索引键加锁而升级为表级锁定；  
b)合理设计索引，让InnoDB在索引键上面加锁的时候尽可能准确，尽可能的缩小锁定范围，避免造成不必要的锁定而影响其他Query的执行；  
c)尽可能减少基于范围的数据检索过滤条件，避免因为间隙锁带来的负面影响而锁定了不该锁定的记录；  
d)尽量控制事务的大小，减少锁定的资源量和锁定时间长度；  
e)在业务环境允许的情况下，尽量使用较低级别的事务隔离，以减少MySQL因为实现事务隔离级别所带来的附加成本。  
（2）由于InnoDB的行级锁定和事务性，所以肯定会产生死锁，下面是一些比较常用的减少死锁产生概率的小建议：  
a)类似业务模块中，尽可能按照相同的访问顺序来访问，防止产生死锁；  
b)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；  
c)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。  
（3）可以通过检查InnoDB\_row\_lock状态变量来分析系统上的行锁的争夺情况：

[复制代码](javascript:void(0);)

mysql> show status like 'InnoDB\_row\_lock%';

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

| Variable\_name | Value |

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

| InnoDB\_row\_lock\_current\_waits | **0** |

| InnoDB\_row\_lock\_time | **0** |

| InnoDB\_row\_lock\_time\_avg | **0** |

| InnoDB\_row\_lock\_time\_max | **0** |

| InnoDB\_row\_lock\_waits | **0** |

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

[复制代码](javascript:void(0);)

InnoDB 的行级锁定状态变量不仅记录了锁定等待次数，还记录了锁定总时长，每次平均时长，以及最大时长，此外还有一个非累积状态量显示了当前正在等待锁定的等待数量。对各个状态量的说明如下：  
InnoDB\_row\_lock\_current\_waits：当前正在等待锁定的数量；  
InnoDB\_row\_lock\_time：从系统启动到现在锁定总时间长度；  
InnoDB\_row\_lock\_time\_avg：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常的一次所花的时间；  
InnoDB\_row\_lock\_waits：系统启动后到现在总共等待的次数；  
对于这5个状态变量，比较重要的主要是InnoDB\_row\_lock\_time\_avg（等待平均时长），InnoDB\_row\_lock\_waits（等待总次数）以及InnoDB\_row\_lock\_time（等待总时长）这三项。尤其是当等待次数很高，而且每次等待时长也不小的时候，我们就需要分析系统中为什么会有如此多的等待，然后根据分析结果着手指定优化计划。  
如果发现锁争用比较严重，如InnoDB\_row\_lock\_waits和InnoDB\_row\_lock\_time\_avg的值比较高，还可以通过设置InnoDB Monitors 来进一步观察发生锁冲突的表、数据行等，并分析锁争用的原因。  
锁冲突的表、数据行等，并分析锁争用的原因。具体方法如下：

mysql> create table InnoDB\_monitor(a INT) engine=InnoDB;

然后就可以用下面的语句来进行查看：

mysql> show engine InnoDB status;

监视器可以通过发出下列语句来停止查看：

mysql> drop table InnoDB\_monitor;

设置监视器后，会有详细的当前锁等待的信息，包括表名、锁类型、锁定记录的情况等，便于进行进一步的分析和问题的确定。可能会有读者朋友问为什么要先创建一个叫InnoDB\_monitor的表呢？因为创建该表实际上就是告诉InnoDB我们开始要监控他的细节状态了，然后InnoDB就会将比较详细的事务以及锁定信息记录进入MySQL的errorlog中，以便我们后面做进一步分析使用。打开监视器以后，默认情况下每15秒会向日志中记录监控的内容，如果长时间打开会导致.err文件变得非常的巨大，所以用户在确认问题原因之后，要记得删除监控表以关闭监视器，或者通过使用“--console”选项来启动服务器以关闭写日志文件。