**目录**

[锁的定义](#6848-1610965678885)

[锁的分类](#4059-1610900794798)

[表锁](#8162-1610961652037)

[行锁](#5258-1610961681984)

[行锁的特点](#8393-1610961760006)

[事务](#1890-1610961722178)

[间隙锁](#6550-1610962214727)

[手动锁行](#3655-1610964897446)

[行锁优化](#5734-1610965577033)

[页锁](#8844-1610965631157)

**锁的定义**

锁是计算机协调**多个进程或线程并发访问某一资源的机制。**

**锁的分类**

1. 从操作类型分：

**读锁**（共享锁）：针对同一份数据，多个读操作可以同时进行而不会互相影响

**写锁**（排它锁）：当前写操作没有完成前，它会阻断其他写锁和读锁。

2. 从操作颗粒度:

**表锁**

**行锁**

**表锁**

**表锁的特点：偏向MyISAM存储引擎**，开销小，加锁快，无死锁，锁定粒度大，**发生锁冲突的概率最高，并发最低。**

**读锁总结：**

当前 session 和其他 session 均可以读取加了读锁的表

当前 session 不能读取其他表，并且不能修改加了读锁的表

其他 session 想要修改加了读锁的表，必须等待其读锁释放

**写锁总结**：

当前 session 可以读取和修改加了写锁的表

当前 session 不能读取其他表

其他 session 想要读取加了写锁的表，必须等待其读锁释放

案例总结：

MyISAM在执行查询语句（SELECT）前，会自动给涉及的所有表加**读锁**，在执行增删改操作前，会自动给涉及的表加**写锁**。

MySQL的表级锁有两种模式：

表共享读锁（Table Read Lock）

表独占写锁（Table Write Lock）

对MyISAM表进行操作，会有以下情况：

对MyISAM表的读操作（加读锁），不会阻塞其他进程对同一表的读请求，但会阻塞对同一表的写请求。只有当读锁释放后，才会执行其它进程的写操作。

对MyISAM表的写操作（加写锁），会阻塞其他进程对同一表的读和写操作，只有当写锁释放后，才会执行其它进程的读写操作

简而言之，就是**读锁会阻塞写操作，但是不会堵塞读操作。而写锁则会把读和写都堵塞**。

查看哪些表被锁了，0 表示未锁，1 表示被锁

show open tables;

【如何分析表锁定】

可以通过检查table\_locks\_waited和table\_locks\_immediate状态变量来分析系统上的表锁定，通过 show status like 'table%'; 命令查看

Table\_locks\_immediate：产生表级锁定的次数，表示可以立即获取锁的查询次数，每立即获取锁值加1；

Table\_locks\_waited：出现表级锁定争用而发生等待的次数（不能立即获取锁的次数，每等待一次锁值加1），此值高则说明存在着较严重的表级锁争用情况；

mysql> show status like 'table%';
+----------------------------+--------+
| Variable\_name | Value |
+----------------------------+--------+
| Table\_locks\_immediate | 500440 |
| Table\_locks\_waited | 1 |
| Table\_open\_cache\_hits | 500070 |
| Table\_open\_cache\_misses | 5 |
| Table\_open\_cache\_overflows | 0 |
+----------------------------+--------+
5 rows in set (0.00 sec)

此外，Myisam的读写锁调度是写优先，这也是myisam不适合做写为主表的引擎。因为写锁后，其他线程不能做任何操作，**大量的更新会使查询很难得到锁**，从而造成阻塞。

**行锁**

**行锁的特点**

**偏向InnoDB存储引擎**，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**。

InnoDB与MyISAM的最大不同有两点：**一是支持事务（TRANSACTION）；二是采用了行级锁。**

**没有索引或者索引失效时，InnoDB 的行锁会升级为表锁！**

**原因：Mysql 的行锁是通过索引实现的！**

**事务**

**事务（Transation）及其ACID属性**

事务是由一组SQL语句组成的逻辑处理单元，事务具有以下4个属性，通常简称为事务的ACID属性。

**原子性**（Atomicity）：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。

**一致性**（Consistent）：在事务开始和完成时，数据都必须保持一致状态。这意味着所有相关的数据规则都必须应用于事务的修改，以保持数据的完整性；事务结束时，所有的内部数据结构（如B树索引或双向链表）也都必须是正确的。

**隔离性**（Isolation）：数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。这意味着事务处理过程中的中间状态对外部是不可见的，反之亦然。

**持久性**（Durability）：事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

**并发事务处理带来的问题**

**更新丢失**（Lost Update）：

当两个或多个事务选择同一行，然后基于最初选定的值更新该行时，由于每个事务都不知道其他事务的存在，就会发生丢失更新问题一一最后的更新覆盖了由其他事务所做的更新。

例如，两个程序员修改同一java文件。每个程序员独立地更改其副本，然后保存更改后的副本，这样就覆盖了原始文档。最后保存其更改副本的编辑人员覆盖前一个程序员所做的更改。

如果在一个程序员完成并提交事务之前，另一个程序员不能访问同一文件，则可避免此问题。

**脏读**（Dirty Reads）：

一个事务正在对一条记录做修改，在这个事务完成并提交前，这条记录的数据就处于不一致状态；这时，另一个事务也来读取同一条记录，如果不加控制，第二个事务读取了这些“脏”数据，并据此做进一步的处理，就会产生未提交的数据依赖关系。这种现象被形象地叫做”脏读”。

一句话：事务A读取到了事务B已修改但尚未提交的的数据，还在这个数据基础上做了操作。此时，如果B事务回滚，A读取的数据无效，不符合一致性要求。

**不可重复读**（Non-Repeatable Reads）：

一个事务在读取某些数据后的某个时间，再次读取以前读过的数据，却发现其读出的数据已经发生了改变、或某些记录已经被删除了！这种现象就叫做“不可重复读”。

一句话：事务A读取到了事务B已经提交的修改数据，不符合隔离性

**幻读**（Phantom Reads）

一个事务按相同的查询条件重新读取以前检索过的数据，却发现其他事务插入了满足其查询条件的新数据，这种现象就称为“幻读一句话：事务A读取到了事务B提交的新增数据，不符合隔离性。

多说一句：幻读和脏读有点类似，脏读是读到了别人修改未提交的数据（可能随时回滚），幻读是读到了其他线程新增的数据。

**事物的隔离级别**

脏读”、“不可重复读”和“幻读”，其实都是数据库读一致性问题，必须由数据库提供一定的事务隔离机制来解决。

数据库的事务隔离越严格，并发副作用越小，但付出的代价也就越大，因为事务隔离实质上就是使事务在一定程度上“串行化”进行，这显然与“并发”是矛盾的。

同时，不同的应用对读一致性和事务隔离程度的要求也是不同的，比如许多应用对“不可重复读”和“幻读”并不敏感，可能更关心数据并发访问的能力。

查看当前数据库的事务隔离级别：show variables like 'tx\_isolation'; **mysql 默认是可重复读 RR（ 实现原理（MVCC [ 多版本并发控制 ]））**



**MVCC扩展：**

InnoDB在每行记录后面保存两个隐藏的列来，分别保存了这个行的创建时间和行的更新时间。这里存储的并不是实际的时间值,而是系统版本号，当数据被修改时，版本号加1

　　在读取事务开始时，系统会给当前读事务一个版本号，事务会读取版本号<=当前版本号的数据

　　此时如果其他写事务修改了这条数据，那么这条数据的版本号就会加1，从而比当前读事务的版本号高，读事务自然而然的就读不到更新后的数据了.

**注意：** 在mysql中，提供了两种事务隔离技术，第一个是mvcc，第二个是next-key技术。这个在使用不同的语句的时候可以动态选择。不加lock in share mode之类的就使用mvcc。否则使用next-key。mvcc的优势是不加锁，并发性高。缺点是不是实时数据。next-key的优势是获取实时数据，但是需要加锁。

​

在RR级别下，**快照读**是通过MVVC(多版本控制)和undo log来实现的，**当前读**是通过加record lock(记录锁)和gap lock(间隙锁)来实现的。

所以从上面的显示来看，如果需要实时显示数据，还是需要通过加锁来实现。这个时候会使用next-key技术来实现。

1. **快照读**：

简单的select操作。

2.**当前读**

select ... lock in share mode

select ... for update

insert

update

delete

**间隙锁**

**什么是间隙锁**

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。

**间隙锁的危害**

通过范围查找会锁定整个范围内所有的索引键值，即使这个键值并不存在。

间隙锁有一个比较致命的弱点，就是当锁定一个范围键值之后，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。

**手动锁行**

**如何锁定一行**

select xxx ... for update 锁定某一行后，其它的操作会被阻塞，直到锁定行的会话提交

**结论**

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（索引失效，导致行锁变表锁），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。

**行锁分析**

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。

show status like 'innodb\_row\_lock%';

mysql> show status like 'innodb\_row\_lock%';
+-------------------------------+--------+
| Variable\_name | Value |
+-------------------------------+--------+
| Innodb\_row\_lock\_current\_waits | 0 |
| Innodb\_row\_lock\_time | 212969 |
| Innodb\_row\_lock\_time\_avg | 42593 |
| Innodb\_row\_lock\_time\_max | 51034 |
| Innodb\_row\_lock\_waits | 5 |
+-------------------------------+--------+
5 rows in set (0.00 sec)

**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）

Innodb\_row\_lock\_time（等待总时长）

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。（show

profile分析）

**行锁优化**

尽可能让所有数据检索都通过索引来完成，避免无索引行锁升级为表锁

合理设计索引，尽量缩小锁的范围

尽可能减少检索条件，避免间隙锁

尽量控制事务大小，减少锁定资源量和时间长度

尽可能低级别事务隔离

**页锁**

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。

了解即可