**MySQL 有哪些锁？**

在 MySQL 里，根据加锁的范围，可以分为**全局锁、表级锁和行锁**三类。

## 全局锁

全局锁是怎么用的？

使用全局锁命令：



执行后，**整个数据库就处于只读状态了**，这时其他线程执行以下操作，都会被**阻塞**：

* **对数据的增删改操作**，比如 insert、delete、update等语句；
* **对表结构的更改操作**，比如 alter table、drop table 等语句。

释放全局锁：

 以及会话断开时自动释放全局锁

全局锁应用场景是什么？

全局锁主要应用于做**全库逻辑备份**，这样在备份数据库期间，不会因为数据或表结构的更新，而出现备份文件的数据与预期的不一样。

备份数据库数据时使用全局锁会影响业务，那有什么其他方式可以避免？

如果数据库的引擎**支持可重复读的隔离级别（例如InnoDB）**，那么在备份数据库之前先开启事务，会先创建 Read View，然后整个事务执行期间都在用这个 Read View，而且由于 MVCC 的支持，备份期间业务依然可以对数据进行更新操作。

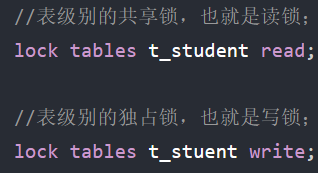
如果数据库的引擎**不支持可重复读（例如MyISAM）**，在备份时候只能使用全局锁

## 表级锁

MySQL 里面表级锁包括：**表锁**、**元数据锁（MDL）**、**意向锁**、**AUTO-INC锁**

### 表锁

对学生表（t\_student）加表锁可以使用下面的命令：



注意：表锁除了会**限制别的线程的读写**外，也会**限制本线程接下来的读写**操作

释放当前会话的所有表锁的命令：

 以及会话断开时自动释放表锁

尽量**避免在使用 InnoDB 引擎的表使用表锁**，因为表锁的颗粒度太大，会**影响并发性能**

**InnoDB 引擎有更细粒度的行级锁**

### 元数据锁（Meta Data Lock, MDL）

不需要显式地使用 MDL，对数据库表进行操作时，会自动给这个表加上 MDL：

* 对一张表进行 CRUD（增删改查）操作时，加的是 **MDL 读锁**；
* 对一张表做结构变更操作的时候，加的是 **MDL 写锁**；

MDL 是为了保证当用户对表执行 CRUD 操作时，防止其他线程变更这个表的结构。

MDL 是在**事务提交后才会释放**，这意味着**事务执行期间，MDL 是一直持有的**。

MDL是一个**写优先**的读写锁，申请 MDL 锁的操作会形成一个队列，队列中**写锁获取优先级高于读锁**，一旦出现 MDL 写锁等待，会阻塞后续该表的所有 CRUD 操作。

### 意向锁

**意向锁的目的是为了添加表锁时快速判断表里是否有记录被加锁**

* 在使用 InnoDB 引擎的表里**对某些记录加上「共享锁」之前**，需要先**在表级别加上一个「意向共享锁」**；
* 在使用 InnoDB 引擎的表里**对某些纪录加上「独占锁」之前**，需要先**在表级别加上一个「意向独占锁」**；

普通的 select 语句利用 MVCC 实现一致性读，是无锁的。如果要对记录加共享锁和独占锁，具体方式如下：



**意向共享锁和意向独占锁是表级锁，不会和行级的共享锁和独占锁发生冲突，而且意向锁之间也不会发生冲突，只会和共享表锁（lock tables ... read）和独占表锁（lock tables ... write）发生冲突。**

如果**没有「意向锁」**，那么加「独占表锁」时，就**需要遍历表里所有记录**，查看是否有记录存在独占锁，这样**效率会很慢**。

那么**有了「意向锁」**，由于在对记录加独占锁前，先会加上表级别的意向独占锁，那么在加「独占表锁」时，**直接查该表是否有意向独占锁，如果有就意味着表里已经有记录被加了独占锁，这样就不用去遍历表里的记录**。

### AUTO-INC 锁

通过对主键字段声明 AUTO\_INCREMENT 属性，可以实现自增的主键。之后可以在插入数据时，可以不指定主键的值

AUTO-INC 锁是特殊的表锁机制，锁**不是再一个事务提交后才释放，而是在执行完插入语句后就会立即释放**。对大量数据进行插入的时候，会**影响插入性能**，因为另一个事务中的插入会被阻塞。

 MySQL 5.1.22 版本开始，InnoDB 存储引擎提供了一种**轻量级的锁**来实现自增。在插入数据的时候，会为被 AUTO\_INCREMENT 修饰的字段加上轻量级锁，**然后给该字段赋值一个自增的值，就把这个轻量级锁释放了，而不需要等待整个插入语句执行完后才释放锁**。

InnoDB 存储引擎提供了个 **innodb\_autoinc\_lock\_mode** 的系统变量，是用来控制选择用 AUTO-INC 锁，还是轻量级的锁。

* 当 innodb\_autoinc\_lock\_mode = 0，就采用 AUTO-INC 锁，**语句执行结束后才释放锁**；
* 当 innodb\_autoinc\_lock\_mode = 2，就采用轻量级锁，**申请自增主键后就释放锁**，并不需要等语句执行后才释放。
* 当 innodb\_autoinc\_lock\_mode = 1：
  + **普通 insert** 语句，自增锁在**申请之后就马上释放**；
  + 类似 insert … select （将一个select的结果插入到表中，或者是create table t2 like t1等）这样的**批量插入数据**的语句，自增锁还是要**等语句结束后才被释放**；

innodb\_autoinc\_lock\_mode = **2** 与 binlog 的日志格式是 **statement** 一起使用的时候，在「主从复制的场景」中会发生**数据不一致的问题**。

举例：



A 往表 t 中插入了 4 行数据，然后B创建了一个相同结构的表 t2（相当于批量插入数据）。由于innodb\_autoinc\_lock\_mode = **2**，插入语句还未执行完就释放锁，可能出现下面情况：

* session B create语句先插入了两个记录，(1,1,1)、(2,2,2)；
* 然后，**session A 来申请自增 id 得到 id=3，插入了（3,5,5)**；
* 之后，session B 继续执行，插入两条记录 (4,3,3)、 (5,4,4)。

**session B 的 insert 语句，生成的 id 不连续**

但是当「主库」发生了这种情况，**binlog 面对 t2 表的更新只会记录这两个 session 的 insert 语句**。如果 binlog\_format=statement，**记录的语句就是原始语句**。记录的顺序要么先记 session A 的 insert 语句，要么先记 session B 的 insert 语句。binlog 拿去「从库」执行时，**从库上「不会」发生像主库那样两个 session 「同时」执行向表 t2 中插入数据的场景。所以，在备库上执行了 session B 的 insert 语句，生成的结果中id 都是连续的。这时，主从库就发生了数据不一致**。

解决方法：**binlog 日志格式要设置为 row**，这样在 binlog 里面记录的是主库分配的自增值，到备库执行的时候，主库的自增值是什么，从库的自增值就是什么。

**当 innodb\_autoinc\_lock\_mode = 2 时，并且 binlog\_format = row，既能提升并发性，又不会出现数据一致性问题**。

## 行级锁

InnoDB 引擎是支持行级锁的， MyISAM 引擎并不支持行级锁。

普通的 select 语句是不会对记录加锁的，因为它属于快照读。如果要在查询时对记录加行锁，可以使用下面这两个方式，这种查询会加锁的语句称为**锁定读：**



**当事务提交了，锁就会被释放**

共享锁（S锁）和独占锁（X锁）的兼容情况：



行级锁的类型主要有三类：

* **Record Lock，记录锁**，也就是仅仅把一条记录锁上；
* **Gap Lock，间隙锁**，锁定一个范围，但是不包含记录本身；
* **Next-Key Lock，临键锁：Record Lock + Gap Lock**，锁定一个范围，并且锁定记录本身。
* **还有一个特殊的间隙锁：插入意向锁**

### 记录锁（Record Lock）

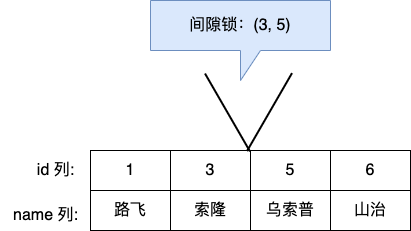
记录锁锁住的是**一条记录**

记录锁是有 S 锁和 X 锁之分：

* 当一个事务对一条记录加了 S 型记录锁后，其他事务也可以继续对该记录加 S 型记录锁（S 型与 S 锁兼容），但是不可以对该记录加 X 型记录锁（S 型与 X 锁不兼容）;
* 当一个事务对一条记录加了 X 型记录锁后，其他事务既不可以对该记录加 S 型记录锁（S 型与 X 锁不兼容），也不可以对该记录加 X 型记录锁（X 型与 X 锁不兼容）。

### 间隙锁（Gap Lock）

只存在于可重复读隔离级别，目的是为了解决可重复读隔离级别下幻读的现象。



间隙锁虽然存在 X 型间隙锁和 S 型间隙锁，但是并没有什么区别，**间隙锁之间是兼容的，即两个事务可以同时持有包含共同间隙范围的间隙锁，并不存在互斥关系，因为间隙锁的目的是防止幻读而提出的**。

### 临键锁（Next-Key Lock）

 Record Lock + Gap Lock 的组合，**锁定一个范围，并且锁定记录本身**。

**如果一个事务获取了 X 型的 next-key lock，那么另外一个事务在获取范围存在重叠的 X 型的 next-key lock 时，是会被阻塞的**。

### 插入意向锁

一个事务在插入一条记录的时候，需要**判断插入位置是否已被其他事务加了间隙锁。**如果有的话，插入操作就会发生**阻塞**，直到拥有间隙锁的那个事务提交为止（释放间隙锁的时刻），在此期间会生成一个**插入意向锁**，表明有事务想在某个区间插入新记录，**但是现在处于等待状态**。

注意：MySQL 加锁时，是先生成锁结构，然后设置锁的状态，**如果锁状态是等待状态，并不是意味着事务成功获取到了锁**，只有当锁状态为正常状态时，才代表事务成功获取到了锁

插入意向锁名字虽然有意向锁，但它并**不是意向锁，而是一种特殊的间隙锁，属于行级别锁**。

插入意向锁与间隙锁的区别：

① 间隙锁锁住的是一个**区间**，插入意向锁锁住的就是一个**点**

② 尽管「插入意向锁」也属于间隙锁，但两个事务却不能在同一时间内，一个拥有间隙锁，另一个拥有该间隙区间内的插入意向锁

# MySQL 是怎么加锁的？

## MySQL 是怎么加行级锁的？

**加锁的对象是索引，加锁的基本单位是 next-key lock**，它是由记录锁和间隙锁组合而成的，**next-key lock 是前开后闭区间，而间隙锁是前开后开区间**。

**next-key lock 在一些场景下会退化成记录锁或间隙锁。**

总结：**在能使用记录锁或者间隙锁就能避免幻读现象的场景下， next-key lock 就会退化成记录锁或间隙锁**。

### 唯一索引等值查询

下面的**「唯一索引」是用「主键索引」**作为案例说明的，加锁只加在主键索引项上。

* 当查询的**记录是「存在」**的，在索引树上定位到这一条记录后，将该记录的索引中的 next-key lock 会**退化成「记录锁」**。
* 当查询的**记录是「不存在」**的，该记录的索引中的 next-key lock 会**退化成「间隙锁」**，**间隙锁的范围为(a,b)**，其中a是小于该查询记录的第一条记录，b是大于该查询记录的第一条记录，如果a、b不存在则为-∞或+∞。

**注意**：

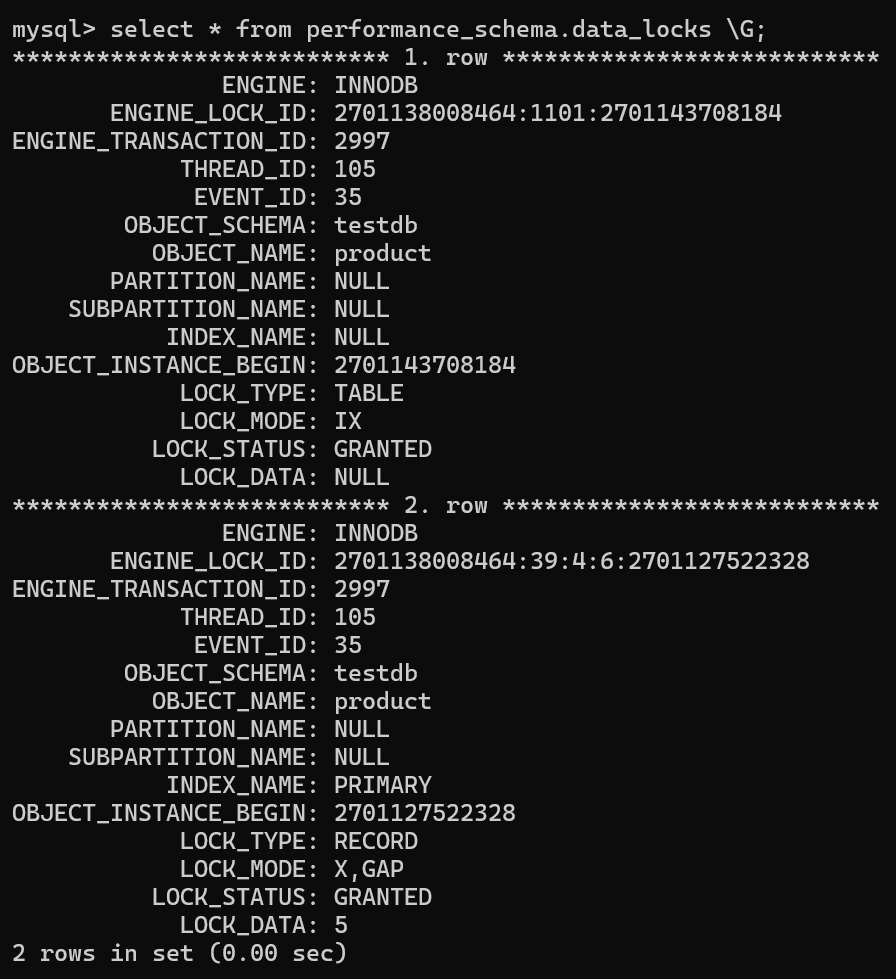
① 间隙锁之间是兼容的，因此**另一个事务可以同时当前查**该间隙内的记录

② 但是另一个事务不能同时拥有这个间隙内的插入意向锁，因此其他事务**向这个间隙内插入是会被阻塞**的。

③ 虽然间隙锁是开区间，但是a和b处也不能插入，因为存在主键约束。

有什么命令可以分析加了什么锁？

**select \* from performance\_schema.data\_locks\G;**



重点关注字段：

① **LOCK\_TYPE：表示锁的等级**

**TABLE**：表级锁

**RECORD**：行级锁（**注意不是记录锁的意思**）

② **LOCK\_MODE：表示锁的类型**

X：独占型

S：共享型

* 如果 LOCK\_MODE 为**X**，说明是 **next-key 锁**；
* 如果 LOCK\_MODE 为 **X, REC\_NOT\_GAP**，说明是**记录锁**；
* 如果 LOCK\_MODE 为**X, GAP**，说明是**间隙锁**；
* 如果 LOCK\_MODE 为**IX**，说明是**意向锁**；

③ **LOCK\_DATA：表示锁住的数据**

### 唯一索引范围查询

#### 1、针对「大于」的范围查询（where id > m）

假设落在范围内的数据为a1，a2，a3，….. ，amax

① 对于每个an (1**<**n<max)建立一个范围为(an-1, an]的next-key锁；

② 对于a1，建立一个范围为(p, a1] 的**next-key锁**，其中p为不在范围内的最大的索引值

为什么是p不是m：个人理解是因为加锁对象是索引，而m不是索引不好加

③ 还需建立一个(amax, +∞]的next-key锁，+∞用**supremum pseudo-record**表示

(a, b]的next-key锁 = (a, b)的间隙锁（锁插入，不锁查询）+ b处的记录锁（锁删除修改查询，不锁插入（**插入由唯一索引控制**））

#### 2、针对「大于等于」的范围查询（where id >= m）

分解成一次等于查询、一次大于查询即可：

① 如果m不存在于表中，与大于查询完全一样

② 如果m存在于表中，m处建立一个记录锁，之后建立和大于查询一样的next-key锁

#### 3、针对「小于」的范围查询（where id < m）

假设落在范围内的数据为amax，……，a3，a2，a1

① 对于每个an (1**≤**n<max)建立一个范围为(an+1, an]的next-key锁；

② 建立一个范围为(a1, p) 的**间隙锁**，其中p为不在范围内的最小的索引值

③ 对于amax，建立一个范围为(-∞, amax]的next-key锁，由于查看锁时，next-key锁是用右边界表示的，因此这里-∞并不需要表示出来。

#### 4、针对「小于等于」的范围查询（where id <= m）

① 如果m不存在于表中，与小于查询完全一样

② 如果m存在于表中，小于查询中的②，即加间隙锁这一步可以直接省略。

### 非唯一索引等值查询

当我们用非唯一索引进行等值查询的时候，**因为存在两个索引，一个是主键索引，一个是非唯一索引（二级索引），所以在加锁时，同时会对这两个索引都加锁，但是对主键索引加锁的时候，只有满足查询条件的记录才会对它们的主键索引加锁**。

#### 1、记录不存在的情况

定位到第一条不符合查询条件的二级索引记录，**该二级索引的 next-key 锁会退化成间隙锁，间隙锁的范围为(**{second\_a, key\_a}**,** {second\_b, key\_b}**)**

second\_a/key\_a是**小于该查询记录的第一条记录**的二级索引/主键索引。

second\_b/key\_b是**大于该查询记录的第一条记录**的二级索引/主键索引。

判断另一个事务是否可以在间隙锁的边界插入：

① 当二级索引为second\_a：主键<key\_a可以插入；主键>=key\_a阻塞

② 当二级索引为second\_b：主键>key\_b可以插入；主键<=key\_b阻塞

#### 2、记录存在的情况

① 对于所有符合查询条件的**二级索引**，加上**next-key锁**；**主键索引**，加上**记录锁**。

② 对于大于该查询记录的**第一条记录的二级索引**，加上**间隙锁**

### 非唯一索引范围查询

① **非唯一索引范围查询，索引的 next-key lock 不会有退化为间隙锁和记录锁的情况，对于所有在范围内的非唯一索引（包括supremum pseudo-record）都加上next-key lock。**

② 范围内的非唯一索引对应的**主键索引都加上记录锁**

③ **如果是小于或小于等于，不在范围内的最小的一个记录也会建立next-key lock！其对应的主键索引也会加上记录锁！**

### 没有加索引的查询

**如果锁定读查询语句，没有使用索引列作为查询条件，或者查询语句没有走索引查询，导致扫描是全表扫描。那么，每一条记录的索引上都会加 next-key 锁，这样就相当于锁住了全表，这时如果其他事务对该表进行增、删、改操作的时候，都会被阻塞**。

因此，**在线上在执行 update、delete、select ... for update 等具有加锁性质的语句，一定要检查语句是否走了索引**