MySQL是目前世界上最流行的数据库，InnoDB是MySQL最流行的存储引擎，它在大数据量高并发量的业务场景下，有着非常良好的性能表现，之所以如此，是和InnoDB的**锁机制**相关。

总的来说，InnoDB共有**七种类型的锁**：

（1）自增锁(Auto-inc Locks)；

（2）共享/排它锁(Shared and Exclusive Locks)；

（3）意向锁(Intention Locks)；

（4）插入意向锁(Insert Intention Locks)；

（5）记录锁(Record Locks)；

（6）间隙锁(Gap Locks)；

（7）临键锁(Next-key Locks)；

今天和大家来逐一介绍。

文章较长，案例较多，请大家提前收藏，点赞，转发，再看。

第一种，自增锁（Auto-inc Locks）**

【*案例说明*】

MySQL，InnoDB，默认的隔离级别(RR)，假设有数据表：

t(id AUTO_INCREMENT, name);

数据表中有数据：

1, shenjian

2, zhangsan

3, lisi

事务A**先**执行，还**未提交**：

insert into t(name) values(xxx);

事务B**后**执行：

insert into t(name) values(ooo);

问：事务B会不会被阻塞？

【*案例分析*】

InnoDB在RR隔离级别下，尝试解决幻读问题，上面这个案例中：

（1）事务A先执行insert，会得到一条(4, xxx)的记录，由于是自增列，故不用显示指定id为4，InnoDB会自动增长，注意此时事务并未提交；

（2）事务B后执行insert，**假设不会被阻塞**，那会得到一条(5, ooo)的记录；

此时，并未有什么不妥，但如果，

（3）事务A继续insert：

insert into t(name) values(xxoo);

会得到一条(6, xxoo)的记录。

（4）事务A再select：

*select * from t where id>3;*

得到的结果是：

4, xxx

6, xxoo

画外音：不可能查询到5的记录，在RR的隔离级别下，不可能读取到还未提交事务生成的数据。

这对于事务A来说，就很奇怪了，AUTO_INCREMENT的列，**连续插入了两条记录**，一条是4，接下来一条变成了6，就像莫名其妙的幻影。

【*自增锁*】

自增锁是一种特殊的**表级别锁**（table-level lock），专门针对事务插入AUTO_INCREMENT类型的列。

最简单的情况，如果一个事务正在往表中插入记录，所有其他事务的插入必须等待，以便第一个事务插入的行，是连续的主键值。

画外音：官网是这么说的

An AUTO-INC lock is a special table-level lock taken by transactions inserting into tables with AUTO_INCREMENT columns. In the simplest case, if one transaction is inserting values into the table, any other transactions must wait to do their own inserts into that table, so that rows inserted by the first transaction receive consecutive primary key values.

与此同时，InnoDB提供了innodb_autoinc_lock_mode配置，可以调节与改变该锁的模式与行为。

【*假如不是自增列*】

上面的案例，**假设不是自增列**，又会是什么样的情形呢？

t(id unique PK, name);

数据表中有数据：

10, shenjian

20, zhangsan

30, lisi

事务A**先**执行，在10与20两条记录中插入了一行，还**未提交**：

insert into t values(11, xxx);

事务B**后**执行，也在10与20两条记录中插入了一行：

insert into t values(12, ooo);

这里，便不再使用自增锁，那：

（1）会使用什么锁？

（2）事务B会不会被阻塞呢？

先卖个关子，下文再解答。

第二种，共享/排它锁(Shared and Exclusive Locks)

**

**

《InnoDB并发如此高，原因竟然在这？》一文介绍了通用的共享/排它锁，在InnoDB里当然也实现了**标准的行级锁**(row-level locking)，共享/排它锁：

（1）事务拿到某一行记录的共享S锁，才可以读取这一行；

（2）事务拿到某一行记录的排它X锁，才可以修改或者删除这一行；

其**兼容互斥表**如下：

	S	X
S	兼容	互斥
X	互斥	互斥

即：

（1）多个事务可以拿到一把S锁，读读可以并行；

（2）而只有一个事务可以拿到X锁，写写/读写必须互斥；

共享/排它锁的潜在问题是，不能充分的并行，解决思路是**数据多版本**，具体思路在《InnoDB并发如此高，原因竟然在这？》介绍过，这里不再深入展开。

第三种，意向锁(Intention Locks)**

InnoDB支持多粒度锁(multiple granularity locking)，它允许行级锁与表级锁共存，实际应用中，InnoDB使用的是意向锁。

意向锁是指，未来的某个时刻，事务可能要加共享/排它锁了，先提前声明一个意向。

意向锁有这样一些特点：

（1）首先，意向锁，是一个表级别的锁(table-level locking)；

（2）意向锁分为：

- **意向共享锁**(intention shared lock, IS)，它预示着，事务有意向对表中的某些行加共享S锁
- **意向排它锁**(intention exclusive lock, IX)，它预示着，事务有意向对表中的某些行加排它X锁

举个例子：

select ... lock in share mode，要设置**IS锁**；

select ... for update，要设置**IX锁**；

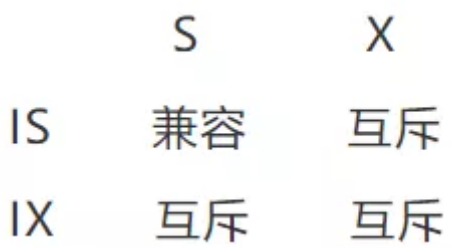
（3）意向锁协议(intention locking protocol)并不复杂：

- 事务要获得某些行的S锁，必须先获得表的IS锁
- 事务要获得某些行的X锁，必须先获得表的IX锁

（4）由于意向锁仅仅表明意向，它其实是比较弱的锁，意向锁之间并不相互互斥，而是可以并行，其**兼容互斥表**如下：

	IS	IX
IS	兼容	兼容
IX	兼容	兼容

（5）额，既然意向锁之间都相互兼容，那其意义在哪里呢？它会与共享锁/排它锁互斥，其**兼容互斥表**如下：



画外音：排它锁是很强的锁，不与其他类型的锁兼容。这也很好理解，修改和删除某一行的时候，必须获得强锁，禁止这一行上的其他并发，以保障数据的一致性。

第四种，插入意向锁**(Insert Intention Locks)**

**
**

对已有数据行的**修改与删除**，必须加强互斥锁X锁，那对于**数据的插入**，是否还需要加这么强的锁，来实施互斥呢？插入意向锁，孕育而生。

插入意向锁，是间隙锁(Gap Locks)的一种（所以，也是实施在索引上的），它是专门针对insert操作的。

画外音：有点尴尬，间隙锁下文才会介绍，暂且理解为，它是一种实施在索引上，锁定索引某个区间范围的锁。

它的玩法是：

多个事务，在同一个索引，同一个范围区间插入记录时，如果插入的位置不冲突，不会阻塞彼此。

画外音：官网的说法是

Insert Intention Lock signals the intent to insert in such a way that multiple transactions inserting into the same index gap need not wait for each other if they are not inserting at the same position within the gap.

这样，之前挖坑的例子，就能够解答了。

在MySQL，InnoDB，RR下：

t(id unique PK, name);

数据表中有数据：

10, shenjian

20, zhangsan

30, lisi

事务A先执行，在10与20两条记录中插入了一行，还未提交：

insert into t values(11, xxx);

事务B后执行，也在10与20两条记录中插入了一行：

insert into t values(12, ooo);

（1）会使用什么锁？

（2）事务B会不会被阻塞呢？

回答：虽然事务隔离级别是RR，虽然是同一个索引，虽然是同一个区间，但插入的记录并不冲突，故这里：

（1）使用的是插入意向锁；

（2）并不会阻塞事务B；

【*思路小结*】

（1）InnoDB使用**共享锁**，可以提高读读并发；

（2）为了保证数据强一致，InnoDB使用强**互斥锁**，保证同一行记录修改与删除的串行性；

（3）InnoDB使用**插入意向锁**，可以提高插入并发；

【另一个案例】

假设不是插入并发，而是读写并发，又会是什么样的结果呢？

MySQL，InnoDB，默认的隔离级别(RR)。

t(id unique PK, name);

数据表中有数据：

10, shenjian

20, zhangsan

30, lisi

事务A先执行，查询了一些记录，还未提交：

*select * from t where id>10;*

事务B后执行，在10与20两条记录中插入了一行：

insert into t values(11, xxx);

这里：

（1）会使用什么锁?

（2）事务B会不会被阻塞呢?

继续卖关子，下文解答。

【继续插入，知识铺垫**】**

InnoDB的细粒度锁，是实现在索引记录上的，如果查询没有命中索引，也将退化为表锁。

【InnoDB的索引**】**

InnoDB的索引有两类索引，**聚集索引**(Clustered Index)与**普通索引**(Secondary Index)。

InnoDB的每一个表都会有聚集索引：

（1）如果表定义了PK，则**PK**就是聚集索引；

（2）如果表没有定义PK，则**第一个非空unique列**是聚集索引；

（3）否则，InnoDB会**创建一个隐藏的row-id**作为聚集索引；

为了方便说明，后文都将以PK说明。

索引的结构**是B+树**，这里不展开B+树的细节，说几个结论：

（1）在索引结构中，非叶子节点存储key，叶子节点存储value；

（2）**聚集索引**，叶子节点存储行记录(row)；

画外音：所以，InnoDB索引和记录是存储在一起的，而MyISAM的索引和记录是分开存储的。

（3）**普通索引**，叶子节点存储了PK的值；

画外音：

所以，InnoDB的普通索引，如果未满足索引覆盖，实际上会扫描两遍：

第一遍，由普通索引找到PK；

第二遍，由PK找到行记录；

索引结构，InnoDB/MyISAM的索引结构，如果大家感兴趣，未来撰文详述。

举个例子，假设有InnoDB表：

t(id PK, name KEY, sex, flag);

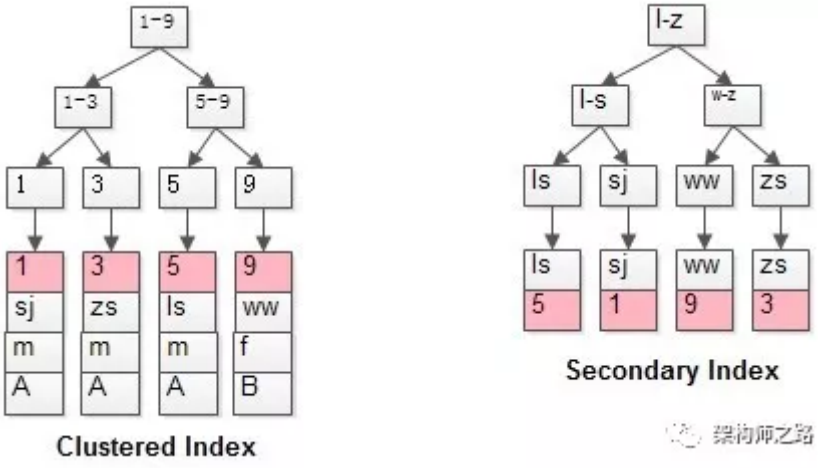
表中有四条记录：

1, shenjian, m, A

3, zhangsan, m, A

5, lisi, m, A

9, wangwu, f, B



可以看到：

（1）第一幅图，id PK的聚集索引，叶子存储了所有的行记录；

（2）第二幅图，name上的普通索引，叶子存储了PK的值；

对于：

*select * from t where name='shenjian';*

（1）会先在name普通索引上查询到PK=1；

（2）再在聚集索引上查询到(1,shenjian, m, A)的行记录；

有了上面的铺垫，下文继续介绍InnoDB剩下三种锁：

（1）记录锁(Record Locks)；

（2）间隙锁(Gap Locks)；

（3）临键锁(Next-Key Locks)；

为了方便讲述，如无特殊说明，后文中，默认的事务隔离级别为**可重复读**(Repeated Read, RR)。

第五种，记录锁**(Record Locks)**

**

**

记录锁，它封锁索引记录，例如：

*select * from t where id=1 for update;*

它会在id=1的索引记录上加锁，以阻止其他事务插入，更新，删除id=1的这一行。

需要说明的是：

*select * from t where id=1;*

是**快照读**(SnapShot Read)，它并不加锁。

第六种，间隙锁**(Gap Locks)**

**
**

间隙锁，它封锁索引记录中的间隔，或者第一条索引记录之前的范围，又或者最后一条索引记录之后的范围。

依然是上面的例子，InnoDB，RR：

t(id PK, name KEY, sex, flag);

表中有四条记录：

1, shenjian, m, A

3, zhangsan, m, A

5, lisi, m, A

9, wangwu, f, B

这个SQL语句

*select * from t*

where id between 8 and 15

for update;

会封锁区间，以阻止其他事务id=10的记录插入。

画外音：

为什么要阻止id=10的记录插入？

如果能够插入成功，头一个事务执行相同的SQL语句，会发现结果集多出了一条记录，即幻影数据。

间隙锁的**主要目的**，就是为了防止其他事务在间隔中插入数据，以导致“不可重复读”。

如果把事务的隔离级别降级为**读提交**(Read Committed, RC)，间隙锁则会自动失效。

第七种，临键锁**(Next-Key Locks)**

**

**

临键锁，是记录锁与间隙锁的组合，它的封锁范围，既包含索引记录，又包含索引区间。

更具体的，临键锁会封锁索引记录本身，以及索引记录之前的区间。

如果一个会话占有了索引记录R的共享/排他锁，其他会话不能立刻在R之前的区间插入新的索引记录。

画外音：原文是说

If one session has a shared or exclusive lock on record R in an index, another session cannot insert a new index record in the gap immediately before R in the index order.

依然是上面的例子，InnoDB，RR：

t(id PK, name KEY, sex, flag);

表中有四条记录：

1, shenjian, m, A

3, zhangsan, m, A

5, lisi, m, A

9, wangwu, f, B

PK上潜在的临键锁为：

(-infinity, 1]

(1, 3]

(3, 5]

(5, 9]

(9, +infinity)

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。

画外音：关于事务的隔离级别，以及幻读，之前的文章一直没有展开说明，如果大家感兴趣，后文详述。

【*总结*】

（1）**自增锁**(Auto-inc Locks)：表级锁，专门针对事务插入AUTO_INC的列，如果插入位置冲突，多个事务会阻塞，以保证数据一致性；

（2）**共享/排它锁**(Shared and Exclusive Locks)：行级锁，S锁与X锁，强锁；

（3）**意向锁**(Intention Locks)：表级锁，IS锁与IX锁，弱锁，仅仅表明意向；

（4）**插入意向锁**(Insert Intention Locks)：针对insert的，如果插入位置不冲突，多个事务不会阻塞，以提高插入并发；

（5）**记录锁**(Record Locks)：索引记录上加锁，对索引记录实施互斥，以保证数据一致性；

（6）**间隙锁**(Gap Locks)：封锁索引记录中间的间隔，在RR下有效，防止间隔中被其他事务插入；

（7）**临键锁**(Next-key Locks)：封锁索引记录，以及索引记录中间的间隔，在RR下有效，防止幻读；

InnoDB的锁，与索引类型，事务的隔离级别相关，更多更复杂更有趣的案例，后续和大家介绍。