全局锁、表级锁以及行锁（中）

意向锁

行级锁

InnoDB 引擎实现了两种标准的行级锁

- 共享行级锁，Shared Lock，简写为 S Lock，可以理解为 Read Lock
- 排他行级锁，Exclusive Lock，简写为 X Lock，可以理解为 Write Lock

-	S Lock	X Lock
S Lock	兼容	不兼容
X Lock	不兼容	不兼容

两种行级锁的兼容性如图所示，只有读锁和读锁之间不互斥

我们从一个简单的问题开始，InnoDB 引擎支持表级锁以及行锁这两种不同粒度的锁，那么当事务 A 在某一行数据上添加了行锁，事务 B 想要更改表结构。为了保证数据的一致性，B 需要确定当前表上没有其它的锁，以及数据行没有其它的锁，那么 InnoDB 是怎么做的呢？

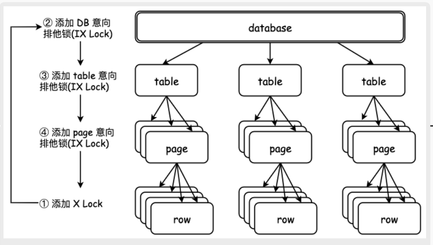
标志位

我们只需要 4 个标志位就可以解决这个问题：table_s_lock, table_x_lock, row_s_lock, row_x_lock。这四个标志位分别代表级别共享锁，表级别排他锁，行级共享锁以及行级排他锁

当某一行添加 X 锁时，将 row_x_lock 置为 true，若其余事务想要添加表级别的 X 锁，则必须等待 row_x_lock 更新为false。反之若事务已经添加了表级别的 X 锁，将 table_x_lock 置为 true，事务 B 若想在某一行添加 X 锁，则需要等待table_s_lock 以及 table_x_lock 均更新为 false

这么做的确能解决问题，但是锁的粒度比较是不同的。表级锁需要和行级锁进行判断，有些不合理

为了解决上面不同锁粒度比较的问题，InnoDB 引入了意向锁（Intention Lock），将锁的粒度分为多个等级。当想要对细粒度的数据加锁时，首先要对粗粒度的对象添加意向锁



如图所示，当我们想要对某一行添加 X Lock 时，需要分别对 DB 对象、表对象以及页对象添加意向排他锁，成功后才能真正的添加行锁

-	IX	IS	X(表级别)	S(表级别)
IX	兼容	兼容	不兼容	不兼容
IS	兼容	兼容	不兼容	兼容
X(表级别)	不兼容	不兼容	不兼容	不兼容
S(表级别)	不兼容	兼容	不兼容	兼容

表级别的意向锁以及 X Lock、S Lock 的兼容性如左图所示

需要特别注意的是，意向锁和意向锁之间没有任何冲突关系

select * from t1 where id = 10 for update

1. 分别获取 DB、table 以及 page 的意向排他锁，即 IX Lock，若这些对象上没有 X Lock 或者是 S Lock，则取锁成功
2. 由于 id 为主键，并且为 unique，故不会添加 Gap Lock，若此时 id = 10 这一行没有 X Lock 或者是 S Lock，那么获取 X Lock 成功

幻读

幻读与事务隔离级别有关，在 READ UNCOMMITTED，READ COMMITTED，REPEATABLE READ 这三个事务隔离级别下，若没有其它限制手段，都有可能出现幻读

从官方文档上来看，幻读就是在同一事务中两次 select 返回了不同的数据集合，比如第一次 select 返回了 10 行，第二次同样的 select 却返回了 11 行，而该行数据并不由当前事务所插入

官方释义

https://dev.mysql.com/doc/refman/8.0/en/innodb-next-key-locking.html

The so-called phantom problem occurs within a transaction when the same query produces different sets of rows at different times. For example, if a SELECT is executed twice, but returns a row the second time that was not returned the first time, the row is a "phantom" row.

更准确的说，幻读表示的是某一次 select 所获取的数据特征无法支撑后续的业务操作

Example-1

比如 select 查询某一行数据，发现结果为空，准备插入数据，但是在执行 insert 时发现数据已经存在了，此时就产生了幻读

transaction A	transaction B
mysql> begin; mysql> select * from user where id = 1; Empty set (0.00 sec)	
	mysql> begin; mysql> insert into user values(1, "Buz"); Query OK, 1 row affected (0.04 sec) mysql> commit; Query OK, 0 rows affected (0.00 sec)
mysql> insert into user values(1, "Buz"); ERROR 1062 (23000): Duplicate entry '1' for key 'user.PRIMARY' mysql> select * from user where id = 1; Empty set (0.00 sec)	

该查询语句结果为空，满足 REPEATABLE READ 隔离级别的性质

Example-2

```
+----+------+
| id | name |
+----+------+
|  1 | Buz  |
|  2 | Fuz  |
+----+------+
```

transaction A	transaction B
mysql> begin; mysql> select * from user ;	
	mysql> begin; mysql> insert into user values(3, "Fuz"); Query OK, 1 row affected (0.00 sec) mysql> commit; Query OK, 0 rows affected (0.00 sec)
mysql> update user set name = "Puz" where name = "Fuz"; Query OK, 2 rows affected (0.00 sec) Rows matched: 2 Changed: 2 Warnings: 0 mysql> select * from user;	

```
+----+------+
| id | name |
+----+------+
|  1 | Buz  |
|  2 | Puz  |
|  3 | Puz  |
+----+------+
```

此时，id = 3 这一行并不是由事务 A 插入的，但是在查询时依旧返回了，也就是第一个 select 的结果并不具备准确性，因此产生了幻读

快照读与当前读

InnoDB 通过 undo log 实现的 MVCC 来实现快照读，通俗的来说就是比较数据的事务版本号。快照读又称之为一致性非锁定读，读取的数据并不一定是最新的，但是可以保证在同一个事务内对同一行数据进行多次读取的结果是一样的，以及在范围读取时不会多出几行数据

当前读表示每一次的读取都是数据库中最新的数据，比如 update、insert 以及 delete，都是当前读，即首先获取 X Lock，然后再进行修改、插入和删除动作

一定要注意的是，幻读一定发生在多个事务都执行了“当前读”，包括 insert/update/delete，单一的快照读（select）可通过 MVCC 解决幻读问题

幻读与间隙锁

幻读产生的原因就在于另一个事务插入了满足当前事务搜索条件的数据，并且当前事务执行了“当前读”。也就是说，行锁只能锁住某一行或者多行，但是插入数据这个动作，要更新的是数据和数据之间的间隙。因此，InnoDB 为了解决幻读问题，不得不引入新的锁机制，也就是间隙锁（Gap Lock）

间隙锁本质上锁定的是一个范围，当事务 A 对 (X, Y] 这一区间添加间隙锁时，事务 B 将无法在该范围内插入数据，必须等待 A 结束

幻读的解决

```
+----+------+----------+
| id | name | user_id  |
+----+------+----------+
|  5 | Buz  | 100002   |
+----+------+----------+
```

```
*************************** 4. row ***************************
                    ENGINE: INNODB
ENGINE_TRANSACTION_ID: 15422
              OBJECT_NAME: user
               INDEX_NAME: name
                LOCK_TYPE: RECORD
                LOCK_MODE: X,GAP
              LOCK_STATUS: GRANTED
                LOCK_DATA: 'Cuz', 10
```

transaction A	transaction B
mysql> begin; mysql> select * from user where name = "Buz" for update; mysql> select * from performance_schema.data_locks\G;	
	mysql> begin; mysql> insert into user values(1, "Buz", "800004"); Blocked
mysql> update user set name = "Puz" where name = "Buz"; Query OK, 1 rows affected (0.00 sec) Rows matched: 1 Changed: 1 Warnings: 0 mysql> commit;	
	Query OK, 1 row affected (8.92 sec) mysql> commit;

如上图所示，当我们执行 select * for update 时，不仅仅在 name = "Buz" 的数据行上添加了行锁，同时也添加了一个间隙锁。此时就可以防止其它事务插入满足当前事务搜索条件的数据了，因而可以解决幻读

间隙锁

间隙锁和行锁一起形成 Next-Key Lock，专门用于解决 InnoDB RR 隔离级别下的幻读问题。正因为 Next-Key Lock 的存在，锁住的数据行要比 RC 隔离级别下更多，其并发性能会略微降低

间隙锁的冲突关系与 X Lock 的冲突关系是不一样的，间隙锁和间隙锁之间是不会有冲突的，和间隙锁存在冲突关系的是“往这个间隙中插入一条记录”这个操作

```
*************1. row ***************
    LOCK_TYPE: TABLE
   LOCK_MODE: IX

*************2. row ***************
   INDEX_NAME: name
    LOCK_TYPE: RECORD
   LOCK_MODE: X
 LOCK_STATUS: GRANTED
    LOCK_DATA: supremum pseudo-record
```

transaction A	transaction B
mysql> begin; mysql> select * from user where name = "Quz" for update; Empty set (0.00 sec) mysql> select * from performance_schema.data_locks\G;	
	mysql> begin; mysql> select * from user where name = "Quz" for update; Empty set (0.00 sec) mysql> insert into user values(25, "Quz", "800010"); Blocked
mysql> insert into user values(25, "Quz", "800010"); ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction	
	Query OK, 1 row affected (18.83 sec) mysql> commit; Query OK, 0 rows affected (0.00 sec)

死锁发生，事务 A 将回滚

如上图所示，transaction B 中的 select ... for update 并不会被阻塞，因为此时 name = "Quz" 这一行记录并不存在，所以只是添加了间隙锁。但是，一旦事务 B 执行了 insert 语句，那么死锁就发生了，这也是一个比较典型的死锁案列

间隙锁的具体加锁规则放在下篇整理中，欲知后事如何，且听下回分解