驾驭MySQL中的锁——以InnoDB为例

最简单的没有锁的场景

	会话A	会话B
1	更新记录r为v1	
2		更新记录r为v2
3	读取r = v2	

在会话A看来，自己更新的记录r的值不见了，这叫做丢失更新

解决的方式是对行或者其他粗粒度级别对象加锁，直到一次完整的操作完成，才释放锁

这样，会话A在1时刻得到锁，会话B在2时刻没有得到锁，只能阻塞

这样的“一次完整的操作”称为事务，以BEGIN标志开始，以COMMIT标志完成

共享锁（S Lock）与排他锁（X Lock）

InnoDB存储引擎实现了如下两种标准的行级锁:

$$共享锁(SLock)，允许事务读一行数据。排他锁(XLock)，允许事务删除或更新一行数据。$$

如果一个事务T1已经获得了行r的共享锁，那么另外的事务T2可以立即获得行r的共享锁，因为读取并没有改变行r的数据，称这种情况为锁兼容（Lock Compatible)。但若有其他的事务T3想获得行r的排他锁，则其必须等待事务T1、T2释放行r上的共享锁——这种情况称为锁不兼容。

	X	S
X	不兼容	不兼容
S	不兼容	兼容

多粒度锁——意向共享锁（IS Lock）与意向排他锁（IX Lock)

其设计意图在于：支持多粒度(granular）锁定，这种锁定允许事务在行级上的锁和表级上的锁同时存在。为了支持在不同粒度上进行加锁操作，InnoDB存储引擎支持一种额外的锁方式，称之为意向锁(Intention Lock)。意向锁是将锁定的对象分为多个**层次**，意向锁意味着事务希望在更细粒度（fine granularity）上进行加锁。

若将上锁的对象看成一棵树，那么对最下层的对象上锁，也就是对最细粒度的对象进行上锁，那么首先需要对粗粒度的对象上锁。

举例：如果需要对页上的记录r进行上X锁，那么分别需要对数据库A、表、页上意向锁IX，最后对记录r上X锁。若其中任何一个部分导致等待，那么该操作需要等待粗粒度锁的完成。如果在对记录r加锁之前，已经有事务对表1进行了S表锁，那么表1上已存在S锁，之后事务需要对记录r在表1上加上IX，由于不兼容，所以该事务需要等待表锁操作的完成。

层次 be like:

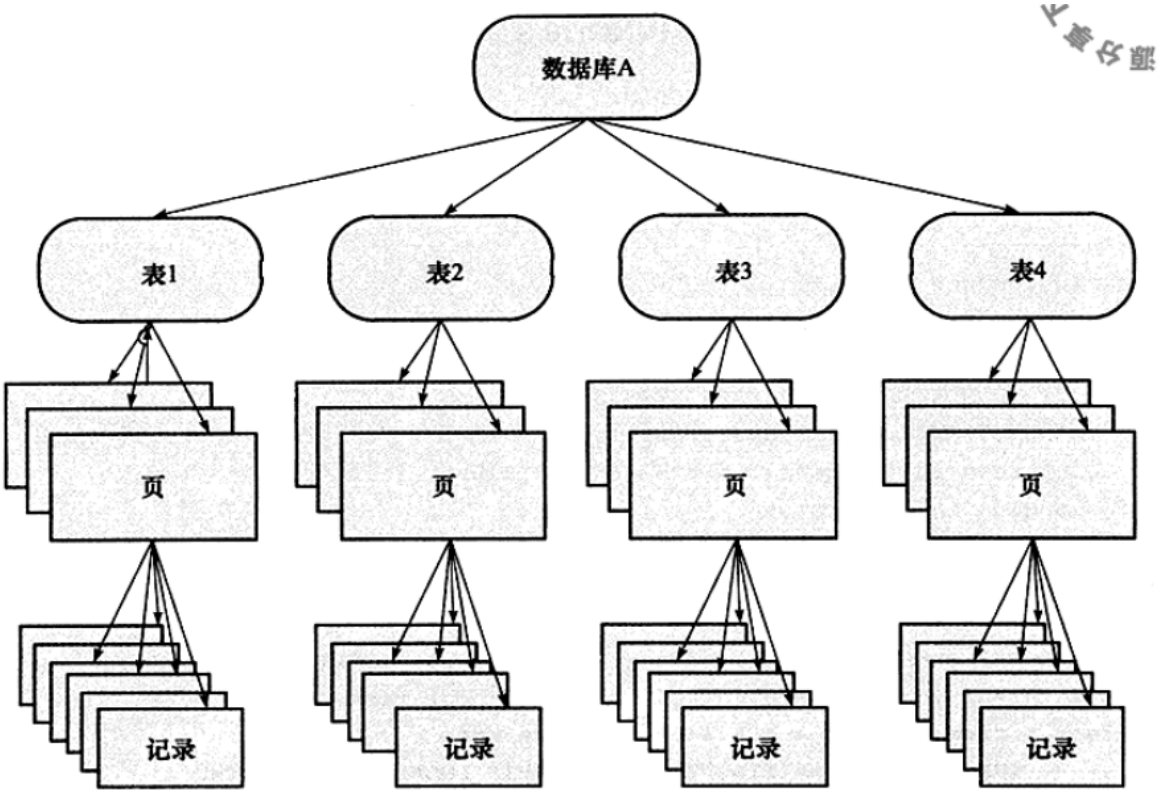


图 6-3　层次结构

表 6-4　InnoDB 存储引擎中锁的兼容性

	IS	IX	S	X
IS	兼容	兼容	兼容	不兼容
IX	兼容	兼容	不兼容	不兼容
S	兼容	不兼容	兼容	不兼容
X	不兼容	不兼容	不兼容	不兼容

SELECT的不同加锁级别

不加锁：一致性非锁定读

InnoDB借助文件系统的快照（在linux文件系统中以写时复制机制实现）实现多版本控制

当读取的行正在执行DELETE或UPDATE操作的时候，不会去等待行上的锁释放，InnoDB引擎会去直接读取行的一个快照数据

读取快照不需要上锁，因为没有事务需要对历史数据进行修改操作

这样并行度最高

这样的读是REPEATABLE READ和READ COMMITTED模式下的默认读

但REPEATABLE READ读最早的快照

READ COMMITTED读最新的快照

有时，用户需要进行一些显示地加锁

加S锁

在如下场景中：

```
mysql> create table ex.parent(parent_key integer primary key);
mysql> create table ex.child(child_key integer primary key,
                             parent_key integer,
                             foreign key (parent_key) REFERENCES
ex.parent(parent_key));
```

外键值的插入或更新，首先需要查询父表中的记录，即SELECT父表。

如果使用一致性非锁定读，如果父表上加有X锁，即便是在READ COMMITTED模式下读最新的快照，仍然会造成数据不一致的问题。读到的外键值可能在X锁释放后已经发生改变。

这时，应该使用

```
SELECT...LOCK IN SHARE MODE
```

方式，即主动对父表加一个S锁，保证在事务期间父表不可修改。如果这时父表上已经加上这样的X锁，子表上的操作会被阻塞。

时间	会话A	会话B
1	BEGIN	
2	DELETE FROM parent WHERE parent_key=3	
3		BEGIN
4		INSERT INTO child SELECT 2,3 # 因为child表的第二列是外键，执行该句时被阻塞

会话A未提交，会话B被阻塞：

```
mysql> select * from ex.parent;
+------------+
| parent_key |
+------------+
|          1 |
|          3 |
+------------+
2 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> delete from ex.parent where parent_key=3
;
Query OK, 1 row affected (0.00 sec)

mysql>
```

```
mysql> begin
    -> ;
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id:    16
Current database: *** NONE ***

Query OK, 0 rows affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into ex.child select 2,3;
```

此时若查看INNODB_LOCKS表，会看到如下结果：

```
mysql> SELECT * FROM information_schema.INNODB_LOCKS\G;
*************************** 1. row ***************************
```

```
    lock_id: 1949:28:3:3
lock_trx_id: 1949
  lock_mode: S
  lock_type: RECORD
 lock_table: `ex`.`parent`
 lock_index: PRIMARY
 lock_space: 28
  lock_page: 3
   lock_rec: 3
  lock_data: 3
*************************** 2. row ***************************
    lock_id: 1948:28:3:3
lock_trx_id: 1948
  lock_mode: X
  lock_type: RECORD
 lock_table: `ex`.`parent`
 lock_index: PRIMARY
 lock_space: 28
  lock_page: 3
   lock_rec: 3
  lock_data: 3
2 rows in set, 1 warning (0.00 sec)
```

会话B的任何插入操作都会被阻塞，因为会话B想要在parent表上加一个S锁，而会话A已经加了一个X锁

如下所以，insert操作被阻塞：

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> delete from ex.parent where parent_key=3
;
Query OK, 1 row affected (0.00 sec)

mysql>
```

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into ex.child select 1,1;
```

加X锁

在如下一个场景中：

时间	User1	User2
1	事务T1查询一行数据r，放入本地内存，并显示给用户User1	
2		事务T2查询一行数据r，放入本地内存，并显示给用户User2
3	User1根据查询结果修改这行记录，更新数据库并提交	
4		User2根据查询结果修改这行记录，更新数据库并提交

在时刻3中，数据库的记录r得到了更新，而User2却依旧根据时刻2的r记录对数据库更新。

再举一个具体的例子

时间	User1	User2
1	查询账户a的余额 =10000	
2		查询账户a的余额 = 10000
3	转出9000，将账户余额更新为1000	
4		转出1，将账户余额更新为9999

我们希望在这种情况下，在SELECT时对改行记录加一个X锁。如果其他的事务也需要根据查询的结果对记录进行更新，那么它就同样需要获得一个X锁，就会被阻塞。

此时，应使用

```
SELECT...FOR UPDATE
```

表锁和行锁的权衡——间隙锁

“幻读”是指在同一个事务中连续进行两次同样的查询却看到了不同的结果，就像出现了幻觉一样，这也称作不可重复读。关于幻读和相关的概念，后文将会详细总结，在此，让我们看一个幻读的例子：

表t只有一列名为a的列，初始只有一个值为1的行

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t; a:1	
3		BEGIN
4		INSERT INTO t SELECT 2;
5		COMMIT;
6	SELECT * FROM t; a:1 a:2	

因为有新的事务提交，会话A两次一致的查询看到了不同的结果。

当然我们可以给整个表t加上一个S锁

使用SELECT * FROM t LOCK IN SHARE MODE

但，给整张表加锁会降低并行度，有没有代价更低的操作呢?

在以下这种情况中：

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t WHERE a = 1; a:1	
3		BEGIN
4		INSERT INTO t SELECT 2;
5		COMMIT;
6	SELECT * FROM t WHERE a = 1; a:1	

尽管在会话A的第二次SELECT之前有另外的事务对表t插入了新的数据，但查询结果并未受到影响。那是因为符合我们查询的内容没有变化。

此时，为了避免出现幻读，我们希望第一次查询时满足a=1的记录不要被其他的事务修改，也不要有新的a=1的记录新增。

也就是说，我们希望“锁住”a=1这个条件，而不是锁住整张表。

另一种情况：

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t WHERE a >=1; a:1	
3		BEGIN
4		INSERT INTO t SELECT 2;
5		COMMIT;
6	SELECT * FROM t WHERE a >= 1; a:1 a:2	

如果不做任何措施，我们会看到上面的结果。现在，因为查询条件是一个范围，所以我们希望锁住这个范围。间隙锁可以理解为这样的“范围锁”。

在讨论间隙锁的详细定义和使用场景之前，先介绍InnoDB的锁锁住的内容是什么、在什么情况下才能对范围、行、特定的值（也许）加锁。

加在索引上的行锁

查找一列的时候，我们希望使用诸如二分查找的方式而非顺序查找来加快速度。比如按照主键查找的时候，我们希望主键之间有”大小关系“，记录的存储有顺序，以便可以快速确定主键的范围，找到主键后，可以直接找到这行记录。这时候我们在主键上建立了索引。

以表u为例：

```
CREATE TABLE `u` (
  `a` int(11) NOT NULL,
  `b` int(11) DEFAULT NULL,
  PRIMARY KEY (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=ascii |
```

创建表的时候，没有说明索引是什么，但指明了主键。很自然地想到此时有一个默认索引是主键。

```
mysql> show index from ex.u; #查看索引
```

```
mysql> show index from ex.u;
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| u     |          0 | PRIMARY  |            1 | a           | A         |           0 |     NULL | NULL   |      | BTREE      |         |               |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
1 row in set (0.00 sec)
```

可以看到，此时在主键上有一个索引。类型是BTREE。（BTREE包含B-TREE和B+TREE。InnoDB使用的是B+TREE。B+TREE的结构使得在记录数量很多的时候仍能在较少次数内查找到特定行所在的最小范围。）

使用联合主键的v表：

```
CREATE TABLE `v` (
  `a` int(11) NOT NULL,
  `b` int(11) NOT NULL,
  PRIMARY KEY (`a`,`b`)
) ENGINE=InnoDB DEFAULT CHARSET=ascii |
```

```
mysql> show index from ex.v;
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| v     |          0 | PRIMARY  |            1 | a           | A         |           0 |     NULL | NULL   |      | BTREE      |         |               |
| v     |          0 | PRIMARY  |            2 | b           | A         |           0 |     NULL | NULL   |      | BTREE      |         |               |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
2 rows in set (0.00 sec)
```

没有主键的w表：

```
CREATE TABLE `w` (
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=ascii |
```

```
mysql> show index from ex.w;
Empty set (0.00 sec)
```

聚焦索引

而聚集索引(clustered index）就是按照每张表的主键构造一棵B+树，同时叶子节点中存放的即为整张表的行记录数据，也将聚集索引的叶子节点称为数据页。聚集索引的这个特性决定了索引组织表中数据也是索引的一部分。同B+树数据结构一样，每个数据页都通过一个双向链表来进行链接。
由于实际的数据页只能按照一棵B+树进行排序，因此每张表只能拥有一个聚集索引。在多数情况下，查询优化器倾向于采用聚集索引。因为聚集索引能够在B+树索引的叶子节点上直接找到数据。此外，由于定义了数据的逻辑顺序，聚集索引能够特别快地访问针对范围值的查询。

辅助索引

对于辅助索引(Secondary Index，也称非聚集索引)，叶子节点并不包含行记录的全部数据。叶子节点除了包含键值以外，每个叶子节点中的索引行中还包含了一个书签(bookmark)。该书签用来告诉InnoDB存储引擎哪里可以找到与索引相对应的行数据。由于InnoDB存储引擎表是索引组织表，因此InnoDB存储引擎的辅助索引的书签就是相应行数据的聚集索引键。

行锁、间隙锁、Next-Key锁

行锁：加在一行记录上

间隙锁：锁定一个范围（开区间）

Next-Key锁：锁定一个左开右闭区间或左闭右开区间 结合了间隙锁和行锁

举例说明，一个索引是聚焦索引（因此是唯一索引），有10，11，13，20这四个值

此时的行锁：10，11，13，20

此时的间隙锁：（-∞，10）（10，11）（11，13）（13，20）（20，+∞）间隙锁也可能按情况合并

此时的Next-Key锁：（-∞，10]（10，11]（11，13]（13，20]（20，+∞）或者（-∞，10）[10，11）[11，13）[13，20）[20，+∞）

INNODB中锁和索引的关系

唯一索引才有行锁S和行锁X 也可以使用间隙锁

辅助索引只能使用间隙锁

只有唯一索引想锁住一行/一个值的时候才能使用行锁。指定这么做的时候，即便SELECT的这个记录不存在，使用的也是行锁。

而辅助索引因为并不保证索引key值是唯一的，所以只能锁住一个间隙。

当一列上不存在索引的时候，不管是锁住一个范围还是想锁住一个特定的值，此时的锁都退化为表锁

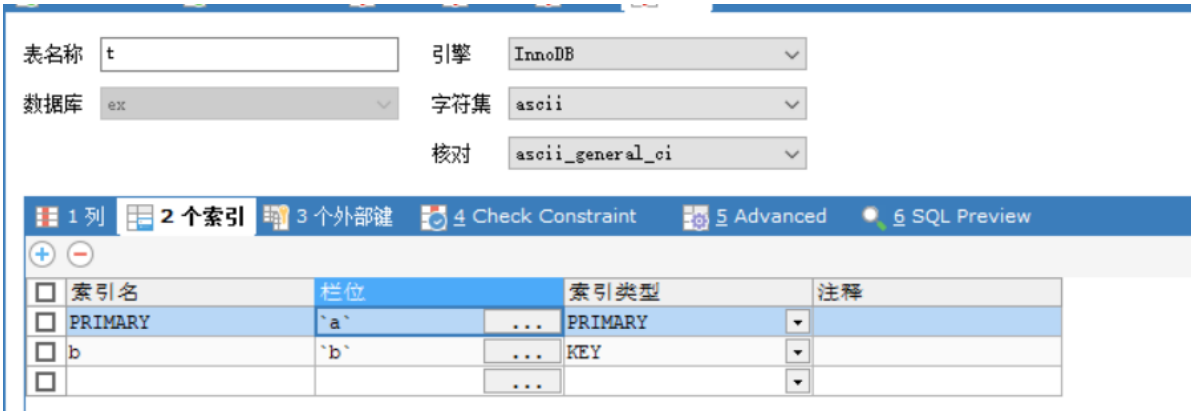
SELECT FOR UPDATE应对幻读

采取行锁、间隙锁机制

表t结构如下：

```
mysql> select * from ex.t;
+----+------+
| a  | b    |
+----+------+
|  1 |    1 |
|  5 |    5 |
| 10 |   10 |
| 15 |   15 |
+----+------+
4 rows in set (0.00 sec)
```

其中a上有唯一索引，b上有辅助索引



每个会话使用

```
SET tx_isolation='REPEATABLE-READ';
```

切换至REPEATABLE READ模式。

在该模式下：

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t WHERE a=1 for update; (1,1)	
3		BEGIN
4		INSERT INTO t SELECT 2; #阻塞
5	SELECT * FROM t WHERE a=1; (1,1)	

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t WHERE a>=2 for update; (5,5)(10,10)(15,15)三条记录	
3		BEGIN
4		INSERT INTO t SELECT 2,3; #阻塞
5	SELECT * FROM t WHERE a>=1 for update; (5,5)(10,10)(15,15)三条记录	

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t WHERE a>=2; (5,5)(10,10)(15,15)三条记录	
3		BEGIN
4		INSERT INTO t SELECT 2,3;
5	SELECT * FROM t WHERE a>=1; (5,5)(10,10)(15,15)三条记录	
6		SELECT * FROM t WHERE a>=1; (2,3)(5,5)(10,10)(15,15)三条记录

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t WHERE b=10 for update; (10,10)记录	
3		BEGIN
4		INSERT INTO t SELECT 6,6 #阻塞 此时索引b上的（5，10）和（10，15）合并加锁 #插入 20，10也是不行的
5	SELECT * FROM t WHERE b = 10 for update; (10,10)记录	

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t WHERE b=10 for update; (10,10)记录	
3		BEGIN
4		INSERT INTO t SELECT 20，20 #成功
5	SELECT * FROM t WHERE b = 10 for update; (10,10)记录 #不会造成幻读	

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t WHERE b=8 for update; empty	
3		BEGIN
4		INSERT INTO t SELECT 8，8 #阻塞 此时索引b上的（5，10）加锁<br
5	SELECT * FROM t WHERE b = 8 for update; empty	

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t WHERE b=15 for update; (15,15)	
3		BEGIN
4		SELECT FROM t WHERE a = 15 LOCK IN SHARE MODE; #阻塞 因为想获得记录a上15索引的S锁 而会话A已经持有索引a的X锁

总结：丢失更新、脏读、幻读、顺序

丢失更新

即在同一个事务中丢失更新的内容。解决方式是加锁。获得记录的X锁才可以更新。并在事务结束后释放锁。

在任何的隔离级别中，都不会出现丢失更新。

在逻辑层面上，如果利用旧的查询结果会出现逻辑上的问题，使用SELECT...FOR UPDATE来避免。

脏读

脏读与脏数据

数据库中的事务只有**提交**之后才能被更新到数据库中，除了提交，一个事务也可以选择**回滚**。

未提交的数据称为脏数据。一个事务读到另一个事务未提交的数据，称为脏读。

脏读举例

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t; a:1	
3		BEGIN
4		INSERT INTO t SELECT 2;
5	SELECT * FROM t; a:1 a:2	

要看到这样的结果需要设置

```
set tx_isolation='read-uncommitted';
```

幻读

有的地方将幻读与不可重复读做区分。将幻读定义为查询结果的数量变化，将不可重复读定义为查询记录内容的变化。其实这两种说法都是在同一个事物中，两次相同的查询语句获得的结果不一样。InnoDB将这种现象称为幻读，通过行锁和间隙锁得到解决。

幻读举例：

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t WHERE a >=1; a:1	
3		BEGIN
4		INSERT INTO t SELECT 2;
5		COMMIT;
6	SELECT * FROM t WHERE a >= 1; a:1 a:2	

要看到这样的结果需要设置

```
set tx_isolation='read-committed';
```

在read-committed级别下，使用SELECT...FOR UPDATE语句，InnoDB将使用间隙锁（也包含行锁、Next-Key锁）避免幻读的发生。

要达到使用SELECT也不会出现幻读的情况，需要如此设置隔离级别:

```
set tx_isolation='repeatable-read';
```

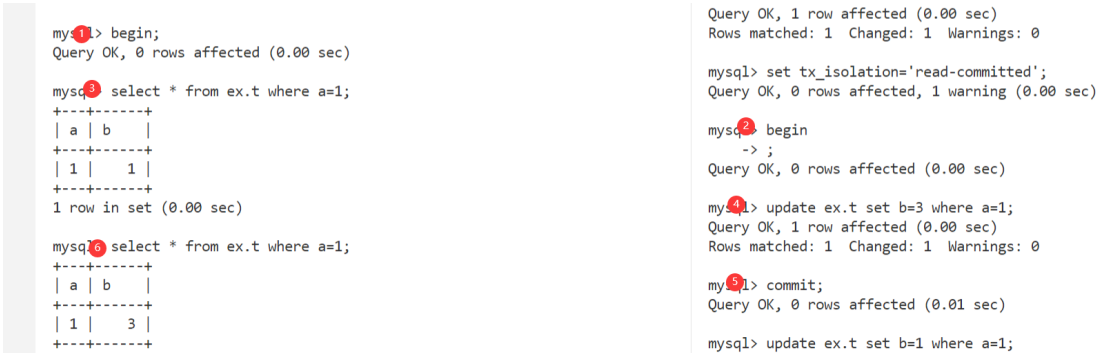
注意：

虽然在repeatable-read和read-committed隔离级别下，SELECT的默认方式都是快照读。

然而注意，因为read-committed的SELECT读取最新的快照（被更新的记录其上锁释放后，新记录就成为最新的快照），所以read-committed下使用SELECT仍然有出现幻读的可能。

举例：

read-committed下出现幻读

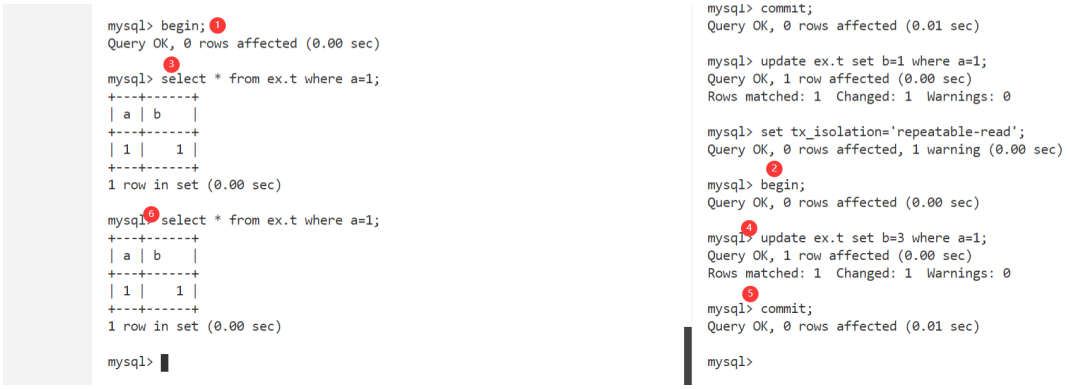


```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from ex.t where a=1;
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
+---+------+
1 row in set (0.00 sec)

mysql> select * from ex.t where a=1;
+---+------+
| a | b    |
+---+------+
| 1 |    3 |
+---+------+
```

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> set tx_isolation='read-committed';
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> begin
    -> ;
Query OK, 0 rows affected (0.00 sec)

mysql> update ex.t set b=3 where a=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql> update ex.t set b=1 where a=1;
```

repeatable-read没有幻读

在InnoDB的实现中，repeatable-read没有幻读和不可重复读



```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from ex.t where a=1;
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
+---+------+
1 row in set (0.00 sec)

mysql> select * from ex.t where a=1;
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
+---+------+
1 row in set (0.00 sec)

mysql>
```

```
mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql> update ex.t set b=1 where a=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> set tx_isolation='repeatable-read';
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> update ex.t set b=3 where a=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql>
```

可串行化

所有的读写都是串行的，一般不用。

阻塞与死锁

阻塞

因为不同锁之间的兼容性关系，在有些时刻一个事务中的锁需要等待另一个事务中的锁释放它所占用的资源，这就是阻塞。阻塞并不是一件坏事，其是为了确保事务可以并发且正常地运行。
在 InnoDB存储引擎中，参数innodb_lock_wait_timeout用来控制等待的时间（默认是50秒），innodb_rollback_on_timeout用来设定是否在等待超时时对进行中的事务进行回滚操作（默认是OFF，代表不回滚）。参数innodb_lock_wait_timeout是动态的，可以在MySQL数据库运行时进行调整。

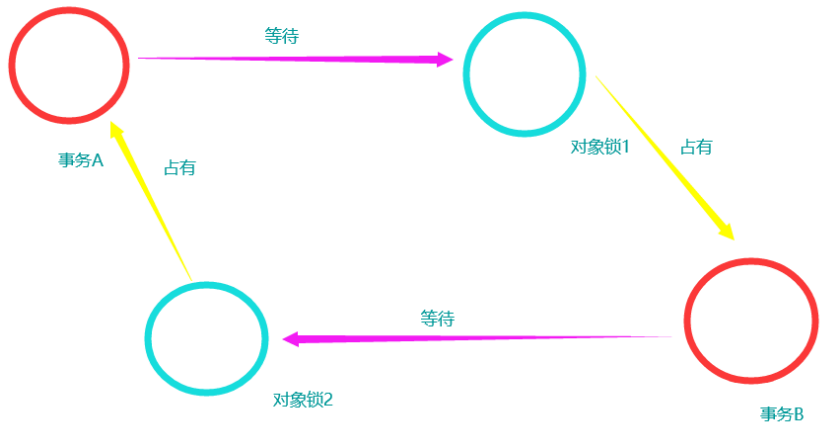
死锁

概念

死锁是指两个或两个以上的事务在执行过程中，因争夺锁资源而造成的一种互相等待的现象。若无外力作用，事务都将无法推进下去。

解决方法

- 直接回滚：解决死锁问题最简单的方式是不要有等待，将任何的等待都转化为回滚，并且事务重新开始。毫无疑问，这的确可以避免死锁问题的产生。然而在线上环境中，这可能导致并发性能的下降，甚至任何一个事务都不能进行。而这所带来的问题远比死锁问题更为严重，因为这很难被发现并且浪费资源。
- 超时：即当两个事务互相等待时，当一个等待时间超过设置的某一阈值时，其中一个事务进行回滚，另一个等待的事务就能继续进行。在InnoDB存储引擎中，参数innodb_lock_wait_timeout用来设置超时的时间。
- 主动死锁检测：超时机制虽然简单，但是其仅通过超时后对事务进行回滚的方式来处理，或者说其是根据FIFO的顺序选择回滚对象。但若超时的事务所占权重比较大，如事务操作更新了很多行，占用了较多的undo log，这时采用FIFO的方式，就显得不合适了，因为回滚这个事务的时间相对另一个事务所占用的时间可能会很多。
  因此，除了超时机制，当前数据库还都普遍采用wait-for graph（等待图）的方式来进行死锁检测。较之超时的解决方案，这是一种更为主动的死锁检测方式。



死锁示例

示例一：假设表t中a列为主键

时间	会话A	会话B
1	BEGIN	
2	SELECT * FROM t WHERE a=1 FOR UPDATE 结果：a：1 #持有记录a=1行锁X	BEGIN
3		SELECT * FROM t WHERE a=2 FOR UPDATE 结果：a：2 #持有记录a=2行锁X
4	SELECT * FROM t WHERE a=2 FOR UPDATE #需要获得记录a=2行锁X	
5		SELECT * FROM t WHERE a=1 FOR UPDATE 结果：a：1 #需要获得记录a=1行锁X #因死锁发成回滚

示例二：假设表t中a列为主键，且已有1,2,4,5这四条记录

时间	会话A	会话B
1	BEGIN	
2		BEGIN
3	SELECT * FROM t WHERE a=4 FOR UPDATE 结果：a：4 #持有记录a=4行锁X	
4		SELECT * FROM t WHERE a<=4 FOR UPDATE #持有记录a=1行锁X a=2行锁X 相应的间隙锁 等待行锁a=4
5	INSERT INTO t VALUES(3) #需要获得记录a=3行锁X #因死锁发成回滚	
6		事务获得锁 正常运行

因为会话B的undo log记录更大，所以选择回滚会话A的事务