第四节-Mysql事务以及锁原理讲解

鲁班学院-周瑜

曾参与大型电商平台、互联网金融产品等多家互联网公司的开发，曾就职于大众点评，任项目经理等职位，参与并主导千万级并发电商网站与系统架构搭建

事务（ACID）

场景：小明向小强转账10元

原子性（Atomicity）

转账操作是一个不可分割的操作，要么转失败，要么转成功，不能存在中间的状态，也就是转了一半的这种情况。我们把这种要么全做，要么全不做的规则称之为原子性。

隔离性（Isolation）

另外一个场景：

1. 小明向小强转账10元
2. 小明向小红转账10元

隔离性表示上面两个操作是不能相互影响的

一致性（Consistency）

对于上面的转账场景，一致性表示每一次转账完成后，都需要保证整个系统的余额等于所有账户的收入减去所有账户的支出。

如果不遵循原子性，也就是如果小明向小强转账10元，但是只转了一半，小明账户少了10元，小强账户并没有增加，所以没有满足一致性了。

同样，如果不满足隔离性，也有可能导致破坏一致性。

所以说，数据库某些操作的原子性和隔离性都是保证一致性的一种手段，在操作执行完成后保证符合所有既定的约束则是一种结果。

实际上我们也可以对表建立约束来保证一致性。

持久性（Durability）

对于转账的交易记录，需要永久保存。

事务的概念

我们把需要保证原子性、隔离性、一致性和持久性的一个或多个数据库操作称之为一个事务。

事务的使用

开启事务

BEGIN [WORK];

BEGIN语句代表开启一个事务，后边的单词WORK可有可无。开启事务后，就可以继续写若干条语句，这些语句都属于刚刚开启的这个事务。

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

mysql> sql...
```

START TRANSACTION;

START TRANSACTION语句和BEGIN语句有着相同的功效，都标志着开启一个事务，比如这样：

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

mysql> sql...
```

提交事务

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

mysql> UPDATE account SET balance = balance - 10 WHERE id = 1;
Query OK, 1 row affected (0.02 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE account SET balance = balance + 10 WHERE id = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

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

手动中止事务

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

mysql> UPDATE account SET balance = balance - 10 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE account SET balance = balance + 1 WHERE id = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

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

这里需要强调一下，ROLLBACK语句是我们程序员手动的去回滚事务时才去使用的，如果事务在执行过程中遇到了某些错误而无法继续执行的话，事务自身会自动的回滚。

自动提交

```
mysql> SHOW VARIABLES LIKE 'autocommit';
```

默认情况下，如果我们不显式的使用START TRANSACTION或者BEGIN语句开启一个事务，那么每一条语句都算是一个独立的事务，这种特性称之为事务的自动提交

如果我们想关闭这种自动提交的功能，可以使用下边两种方法之一：

- 显式的的使用START TRANSACTION或者BEGIN语句开启一个事务。这样在本次事务提交或者回滚前会暂时关闭掉自动提交的功能。
- 把系统变量autocommit的值设置为OFF，就像这样：`SET autocommit = OFF;`这样的话，我们写入的多条语句就算是属于同一个事务了，直到我们显式的写出COMMIT语句来把这个事务提交掉，或者显式的写出ROLLBACK语句来把这个事务回滚掉。

隐式提交

当我们使用START TRANSACTION或者BEGIN语句开启了一个事务，或者把系统变量autocommit的值设置为OFF时，事务就不会进行自动提交，但是如果我们输入了某些语句之后就会悄悄的提交掉，就像我们输入了COMMIT语句了一样，这种因为某些特殊的语句而导致事务提交的情况称为隐式提交，这些会导致事务隐式提交的语句包括：

- 定义或修改数据库对象的数据定义语言（Data definition language，缩写为：DDL）。所谓的数据库对象，指的就是数据库、表、视图、存储过程等等这些东西。当我们使用CREATE、ALTER、DROP等语句去修改这些所谓的数据库对象时，就会隐式的提交前边语句所属于的事务。
- 隐式使用或修改mysql数据库中的表：当我们使用ALTER USER、CREATE USER、DROP USER、GRANT、RENAME USER、SET PASSWORD等语句时也会隐式的提交前边语句所属于的事务。
- 事务控制或关于锁定的语句：当我们在一个事务还没提交或者回滚时就又使用START TRANSACTION或者BEGIN语句开启了另一个事务时，会隐式的提交上一个事务。或者当前的autocommit系统变量的值为OFF，我们手动把它调为ON时，也会隐式的提交前边语句所属的事务。或者使用LOCK TABLES、UNLOCK TABLES等关于锁定的语句也会隐式的提交前边语句所属的事务。
- 加载数据的语句：比如我们使用LOAD DATA语句来批量往数据库中导入数据时，也会隐式的提交前边语句所属的事务。
- 其它的一些语句：使用ANALYZE TABLE、CACHE INDEX、CHECK TABLE、FLUSH、 LOAD INDEX INTO CACHE、OPTIMIZE TABLE、REPAIR TABLE、RESET等语句也会隐式的提交前边语句所属的事务。

保存点

如果你开启了一个事务，并且已经敲了很多语句，忽然发现上一条语句有点问题，你只好使用ROLLBACK语句来让数据库状态恢复到事务执行之前的样子，然后一切从头再来，总有一种一夜回到解放前的感觉。所以MYSQL提出了一个保存点（英文：savepoint）的概念，就是在事务对应的数据库语句中打几个点，我们在调用ROLLBACK语句时可以指定会滚到哪个点，而不是回到最初的原点。定义保存点的语法如下：

```
SAVEPOINT 保存点名称;
```

当我们想回滚到某个保存点时，可以使用下边这个语句（下边语句中的单词WORK和SAVEPOINT是可有可无的）：

```
ROLLBACK [WORK] TO [SAVEPOINT] 保存点名称;
```

不过如果ROLLBACK语句后边不跟随保存点名称的话，会直接回滚到事务执行之前的状态。

如果我们想删除某个保存点，可以使用这个语句:

```
RELEASE SAVEPOINT 保存点名称;
```

隔离性详解

```
-- 修改隔离级别
mysql> set session transaction isolation level read uncommitted;


-- 查看隔离级别
mysql> select @@tx_isolation;
```

读未提交（READ UNCOMMITTED）

一个事务可以读到其他事务还没有提交的数据，会出现脏读。

> 一个事务读到了另一个未提交事务修改过的数据，这就是脏读。

读已提交（READ COMMITTED）

一个事务只能读到另一个已经提交的事务修改过的数据，并且其他事务每对该数据进行一次修改并提交后，该事务都能查询得到最新值，会出现不可重复读、幻读。

> 如果一个事务先根据某些条件查询出一些记录，之后另一个事务又向表中插入了符合这些条件的记录，原先的事务再次按照该条件查询时，能把另一个事务插入的记录也读出来，这就是幻读。

可重复读（REPEATABLE READ）

一个事务第一次读过某条记录后，即使其他事务修改了该记录的值并且提交，该事务之后再读该条记录时，读到的仍是第一次读到的值，而不是每次都读到不同的数据，这就是可重复读，这种隔离级别解决了不可重复，但是还是会出现幻读。

串行化（SERIALIZABLE）

以上3种隔离级别都允许对同一条记录同时进行读-读、读-写、写-读的并发操作，如果我们不允许读-写、写-读的并发操作，可以使用SERIALIZABLE隔离级别，这种隔离基金因为对同一条记录的操作都是串行的，所以不会出现脏读、幻读等现象。

总结

- READ UNCOMMITTED隔离级别下，可能发生**脏读**、**不可重复读**、幻读**问题。
- READ COMMITTED隔离级别下，可能发生**不可重复读**和**幻读**问题，但是不会发生**脏读**问题。
- REPEATABLE READ隔离级别下，可能发生**幻读**问题，不会发生**脏读**和**不可重复读**的问题。
- SERIALIZABLE隔离级别下，各种问题都不可以发生。
- 注意：这四种隔离级别是SQL的标准定义，不同的数据库会有不同的实现，特别需要注意的是**MySQL在REPEATABLE READ隔离级别下，是可以禁止幻读问题的发生的**。

版本链

对于使用InnoDB存储引擎的表来说，它的聚簇索引记录中都包含两个必要的隐藏列（row_id并不是必要的，我们创建的表中有主键或者非NULL唯一键时都不会包含row_id列）：

- trx_id：每次对某条记录进行改动时，都会把对应的事务id赋值给trx_id隐藏列。
- roll_pointer：每次对某条记录进行改动时，这个隐藏列会存一个指针，可以通过这个指针找到该记录修改前的信息。

ReadView

对于使用READ UNCOMMITTED隔离级别的事务来说，直接读取记录的最新版本就好了，对于使用SERIALIZABLE隔离级别的事务来说，使用加锁的方式来访问记录。对于使用READ COMMITTED和REPEATABLE READ隔离级别的事务来说，就需要用到我们上边所说的版本链了，核心问题就是：需要判断一下版本链中的哪个版本是当前事务可见的。

ReadView中主要包含4个比较重要的内容：

1. m_ids：表示在生成ReadView时当前系统中活跃的读写事务的事务id列表。
2. min_trx_id：表示在生成ReadView时当前系统中活跃的读写事务中最小的事务id，也就是m_ids中的最小值。
3. max_trx_id：表示生成ReadView时系统中应该分配给下一个事务的id值。
4. creator_trx_id：表示生成该ReadView的事务的事务id。

> 注意max_trx_id并不是m_ids中的最大值，事务id是递增分配的。比方说现在有id为1，2，3这三个事务，之后id为3的事务提交了。那么一个新的读事务在生成ReadView时，m_ids就包括1和2，min_trx_id的值就是1，max_trx_id的值就是4。

有了这个ReadView，这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本是否可见：

- 如果被访问版本的trx_id属性值与ReadView中的creator_trx_id值相同，意味着当前事务在访问它自己修改过的记录，所以该版本可以被当前事务访问。
- 如果被访问版本的trx_id属性值小于ReadView中的min_trx_id值，表明生成该版本的事务在当前事务生成ReadView前已经提交，所以该版本可以被当前事务访问。
- 如果被访问版本的trx_id属性值大于ReadView中的max_trx_id值，表明生成该版本的事务在当前事务生成ReadView后才开启，所以该版本不可以被当前事务访问。
- 如果被访问版本的trx_id属性值在ReadView的min_trx_id和max_trx_id之间，那就需要判断一下trx_id属性值是不是在m_ids列表中，如果在，说明创建ReadView时生成该版本的事务还是活跃的，该版本不可以被访问；如果不在，说明创建ReadView时生成该版本的事务已经被提交，该版本可以被访问。

READ COMMITTED的实现方式

每次读取数据前都生成一个ReadView

REPEATABLE READ实现方式

在第一次读取数据时生成一个ReadView

MVCC总结

MVCC（Multi-Version Concurrency Control ，多版本并发控制）指的就是在使用READ COMMITTD、REPEATABLE READ这两种隔离级别的事务在执行普通的SEELCT操作时访问记录的版本链的过程。可以使不同事务的读-写、写-读操作并发执行，从而提升系统性能。READ COMMITTD、REPEATABLE READ这两个隔离级别的一个很大不同就是：生成ReadView的时机不同，READ COMMITTD在每一次进行普通SELECT操作前都会生成一个ReadView，而REPEATABLE READ只在第一次进行普通SELECT操作前生成一个ReadView，之后的查询操作都重复使用这个ReadView就好了。

锁

读锁与写锁

- 读锁：共享锁、Shared Locks、S锁。
- 写锁：排他锁、Exclusive Locks、X锁。
- select :不加锁

	X锁	**S锁**
X锁	冲突	冲突
S锁	冲突	不冲突

读操作

对于普通 SELECT 语句，InnoDB 不会加任何锁

select ... lock in share mode

将查找到的数据加上一个S锁，允许其他事务继续获取这些记录的S锁，不能获取这些记录的X锁（会阻塞）

使用场景：读出数据后，其他事务不能修改，但是自己也不一定能修改，因为其他事务也可以使用**select ... lock in share mode** 继续加读锁。

select ... for update

将查找到的数据加上一个X锁，不允许其他事务获取这些记录的S锁和X锁。

使用场景：读出数据后，其他事务即不能写，也不能加读锁，那么就导致只有自己可以修改数据。

写操作

- DELETE：删除一条数据时，先对记录加X锁，再执行删除操作。
- INSERT：插入一条记录时，会先加**隐式锁**来保护这条新插入的记录在本事务提交前不被别的事务访问到。
- UPDATE
    - 如果被更新的列，修改前后没有导致存储空间变化，那么会先给记录加X锁，再直接对记录进行修改。
    - 如果被更新的列，修改前后导致存储空间发生了变化，那么会先给记录加X锁，然后将记录删掉，再Insert一条新记录。

> 隐式锁：一个事务插入一条记录后，还未提交，这条记录会保存本次事务id，而其他事务如果想来对这个记录加锁时会发现事务id不对应，这时会产生X锁，所以相当于在插入一条记录时，隐式的给这条记录加了一把隐式X锁。

行锁与表锁

查看锁情况的SQL：

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_TRX; -- 记录当前运行的事务
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS; -- 记录当前出现的锁
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS; -- 记录锁等待的对应关系
```

INNODB_TRX表字段：

- trx_id:InnoDB存储引擎内部唯一的事物ID
- trx_status:当前事务的状态， RUNNING, LOCK WAIT, ROLLING BACK or COMMITTING.
- trx_status:事务的开始时间
- trx_requested_lock_id：事务等待的锁的ID（如果事务状态不是LOCK WAIT，这个字段是NULL），详细的锁的信息可以连查INNODB_LOCKS表
- trx_wait_started：事务等待的开始时间
- trx_weight：事务的权重，反应一个事务修改和锁定的行数，当发现死锁需要回滚时，权重越小的值被回滚
- trx_mysql_thread_id：MySQL中的进程ID，与show processlist中的ID值相对应
- trx_query：事务运行的SQL语句
- trx_operation_state：事务当操作的类型 如updating or deleting，starting index read等

- trx_tables_in_use：查询用到的表的数量
- trx_tables_locked：查询加行锁的表的数量
- trx_rows_locked：事务锁住的行数（不是准确数字）
- trx_rows_modified：事务插入或者修改的行数

INNODB_LOCKS表：

- lock_id:锁ID
- lock_trx_id：拥有锁的事务 ID。可以和 INNODB_TRX 表 JOIN 得到事务的详细信息。
- lock_mode:锁的模式。
- lock_type：锁的类型。RECORD 代表行级锁，TABLE 代表表级锁。
- lock_table：被锁定的或者包含锁定记录的表的名称。
- lock_index：当LOCK_TYPE='RECORD' 时，表示索引的名称；否则为 NULL。
- lock_space：当LOCK_TYPE='RECORD' 时，表示锁定行的表空间 ID；否则为 NULL。
- lock_page：当 LOCK_TYPE='RECORD' 时，表示锁定行的页号；否则为 NULL。
- lock_rec：当 LOCK_TYPE='RECORD' 时，表示一堆页面中锁定行的数量，亦即被锁定的记录号；否则为 NULL。
- lock_data：当 LOCK_TYPE='RECORD' 时，表示锁定行的主键；否则为NULL。

INNODB_LOCK_WAITS表：

- requesting_trx_id：申请锁资源的事务ID
- requesting_lock_id：申请的锁的ID
- blocking_trx_id：租塞的事务ID
- blocking_lock_id：租塞的锁的ID

行锁

- LOCK_REC_NOT_GAP：单个行记录上的锁。
- LOCK_GAP：间隙锁，锁定一个范围，但不包括记录本身。GAP锁的目的，是为了防止同一事务的两次当前读，出现幻读的情况。
- LOCK_ORDINARY：锁定一个范围，并且锁定记录本身。对于行的查询，都是采用该方法，主要目的是解决幻读的问题。

间隙锁（LOCK_GAP、GAP锁）

READ COMMITTED级别下

查询使用的主键

session1:

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

mysql> select * from t1 where a = 1 for update;
+---+------+------+------+------+
| a | b    | c    | d    | e    |
+---+------+------+------+------+
| 1 |    1 |    1 |    1 | 1    |
+---+------+------+------+------+
1 row in set (0.00 sec)
```

session2:

```
mysql> select * from t1 where a = 1 for update; -- 会阻塞
mysql> select * from t1 where a = 2 for update;  -- 不会阻塞
```

总结：查询使用的是主键时，只需要在主键值对应的那一个条数据加锁即可。

查询使用的唯一索引

session1:

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

mysql> select * from t1 where b = 1 for update;
+---+------+------+------+------+
| a | b    | c    | d    | e    |
+---+------+------+------+------+
| 1 |    1 |    1 |    1 | 1    |
+---+------+------+------+------+
1 row in set (0.00 sec)
```

session2:

```
mysql> select * from t1 where b = 1 for update; -- 会阻塞
mysql> select * from t1 where b = 2 for update;  -- 不会阻塞
```

总结：查询使用的是唯一索引时，只需要对查询值所对应的唯一索引记录项和对应的聚集索引上的项加锁即可。

查询使用的普通索引

session1:

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

mysql> select * from t1 where e = '6' for update;
+----+------+------+------+------+
| a  | b    | c    | d    | e    |
+----+------+------+------+------+
|  6 |    6 |    1 |    4 | 6    |
| 12 |   12 |    1 |    1 | 6    |
+----+------+------+------+------+
2 rows in set (0.00 sec)
```

session2:

```
mysql> select * from t1 where a = 6 for update; -- 会阻塞
mysql> select * from t1 where a = 12 for update; -- 会阻塞
mysql> select * from t1 where a = 1 for update;  -- 不会阻塞
mysql> select * from t1 where a = 2 for update;  -- 不会阻塞
mysql> insert t1(b,c,d,e) values(20,1,1,'51'); -- 不会阻塞
mysql> insert t1(b,c,d,e) values(21,1,1,'61'); -- 不会阻塞
```

总结：查询使用的是普通索引时，会对满足条件的索引记录都加上锁，同时对这些索引记录对应的聚集索引上的项也加锁。

查询使用没有用到索引

session1:

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

mysql> select * from t1 where c = '1' for update;
+----+------+------+------+------+
| a  | b    | c    | d    | e    |
+----+------+------+------+------+
|  1 |    1 |    1 |    1 | 1    |
|  2 |    2 |    1 |    2 | 2    |
|  4 |    3 |    1 |    1 | 4    |
|  6 |    6 |    1 |    4 | 6    |
|  8 |    8 |    1 |    8 | 8    |
| 10 |   10 |    1 |    2 | 10   |
| 12 |   12 |    1 |    1 | 6    |
+----+------+------+------+------+
7 rows in set (0.00 sec)
```

session2:

```
mysql> select * from t1 where a = 1 for update; -- 会阻塞
mysql> select * from t1 where a = 2 for update; -- 会阻塞
mysql> select * from t1 where a = 3 for update;  -- 不会阻塞
mysql> select * from t1 where a = 7 for update;  -- 不会阻塞
```

总结：查询的时候没有走索引，也只会对满足条件的记录加锁。

REPEATABLE READ级别下

查询使用的主键

和RC隔离级别一样。

查询使用的唯一索引

和RC隔离级别一样。

查询使用的普通索引

session1:

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

mysql> select * from t1 where e = '6' for update;
+----+------+------+------+------+
| a  | b    | c    | d    | e    |
+----+------+------+------+------+
|  6 |    6 |    1 |    4 | 6    |
| 12 |   12 |    1 |    1 | 6    |
+----+------+------+------+------+
2 rows in set (0.00 sec)
```

session2:

```
mysql> select * from t1 where a = 6 for update; -- 会阻塞
mysql> select * from t1 where a = 12 for update; -- 会阻塞
mysql> select * from t1 where a = 1 for update;  -- 不会阻塞
mysql> select * from t1 where a = 2 for update;  -- 不会阻塞
mysql> insert t1(b,c,d,e) values(20,1,1,'51'); -- 会阻塞
mysql> insert t1(b,c,d,e) values(21,1,1,'61'); -- 会阻塞
```

总结：REPEATABLE READ级别可以解决幻读，解决的方式就是加了GAP锁。

查询使用没有用到索引

session1:

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

mysql> select * from t1 where c = '1' for update;
+----+------+------+------+------+
| a  | b    | c    | d    | e    |
+----+------+------+------+------+
|  1 |    1 |    1 |    1 | 1    |
|  2 |    2 |    1 |    2 | 2    |
|  4 |    3 |    1 |    1 | 4    |
|  6 |    6 |    1 |    4 | 6    |
|  8 |    8 |    1 |    8 | 8    |
| 10 |   10 |    1 |    2 | 10   |
| 12 |   12 |    1 |    1 | 6    |
+----+------+------+------+------+
7 rows in set (0.00 sec)
```

session2:

```
mysql> select * from t1 where a = 1 for update; -- 会阻塞
mysql> select * from t1 where a = 2 for update; -- 会阻塞
mysql> select * from t1 where a = 3 for update;  -- 不会阻塞
mysql> select * from t1 where a = 7 for update;  -- 会阻塞(在READ COMMITTED级别中不会阻塞，跟解决幻读有关系)
```

总结：查询的时候没有走索引，会对表中所有的记录以及间隙加锁。

表锁

表级别的S锁、X锁

在对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，InnoDB存储引擎是不会为这个表添加表级别的S锁或者X锁的。

在对某个表执行ALTER TABLE、DROP TABLE这些DDL语句时，其他事务对这个表执行SELECT、INSERT、DELETE、UPDATE的语句会发生阻塞，或者，某个事务对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，其他事务对这个表执行DDL语句也会发生阻塞。这个过程是通过使用的元数据锁（英文名：Metadata Locks，简称MDL）来实现的，并不是使用的表级别的S锁和X锁。

- LOCK TABLES t1 READ：对表t1加表级别的S锁。
- LOCK TABLES t1 WRITE：对表t1加表级别的X锁。

> 尽量不用这两种方式去加锁，因为InnoDB的优点就是行锁，所以尽量使用行锁，性能更高。

IS锁、IX锁

- IS锁：意向共享锁、Intention Shared Lock。当事务准备在某条记录上加S锁时，需要先在表级别加一个IS锁。
- IX锁，意向排他锁、Intention Exclusive Lock。当事务准备在某条记录上加X锁时，需要先在表级别加一个IX锁。

IS、IX锁是表级锁，它们的提出仅仅为了在之后加表级别的S锁和X锁时可以快速判断表中的记录是否被上锁，以避免用遍历的方式来查看表中有没有上锁的记录。

AUTO-INC锁

- 在执行插入语句时就在表级别加一个AUTO-INC锁，然后为每条待插入记录的AUTO_INCREMENT修饰的列分配递增的值，在该语句执行结束后，再把AUTO-INC锁释放掉。这样一个事务在持有AUTO-INC锁的过程中，其他事务的插入语句都要被阻塞，可以保证一个语句中分配的递增值是连续的。
- 采用一个轻量级的锁，在为插入语句生成AUTO_INCREMENT修饰的列的值时获取一下这个轻量级锁，然后生成本次插入语句需要用到的AUTO_INCREMENT列的值之后，就把该轻量级锁释放掉，并不需要等到整个插入语句执行完才释放锁。

系统变量innodb_autoinc_lock_mode：

- innodb_autoinc_lock_mode值为0：采用AUTO-INC锁。
- innodb_autoinc_lock_mode值为2：采用轻量级锁。
- 当innodb_autoinc_lock_mode值为1：当插入记录数不确定是采用AUTO-INC锁，当插入记录数确定时采用轻量级锁。

悲观锁

悲观锁用的就是数据库的行锁，认为数据库会发生并发冲突，直接上来就把数据锁住，其他事务不能修改，直至提交了当前事务。

乐观锁

乐观锁其实是一种思想，认为不会锁定的情况下去更新数据，如果发现不对劲，才不更新(回滚)。在数据库中往往添加一个version字段来实现。

死锁

session1:

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

mysql> select * from t1 where a =1 for update; -- 1
+---+------+------+------+------+
| a | b    | c    | d    | e    |
+---+------+------+------+------+
| 1 |    1 |    1 |    1 | 1    |
+---+------+------+------+------+
1 row in set (0.00 sec)

mysql> update t1 set c = 2 where a = 4; -- 3,一开始会阻塞
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
```

session2:

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

mysql> delete from t1 where a = 4;  -- 2
Query OK, 1 row affected (0.00 sec)

mysql> delete fromt t1 where a = 1; -- 4,那道理会阻塞，并产生死锁，但是mysql有死锁检查机制让死锁中断。
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 't1 where a = 1' at line 1
mysql> delete from t1 where a = 1;
Query OK, 1 row affected (0.01 sec)
```

死锁检测

- 系统变量innodb_deadlock_detect：控制是否打开死锁检测，默认打开。
- 系统变量innodb_lock_wait_timeout：等待锁的超时时间，默认50s。
- 系统变量innodb_print_all_deadlocks：将所有的死锁日志写入到mysql的错误日志中，默认是关闭

的。

检测到死锁时，InnoDB会在导致死锁的事务中选择一个权重比较小的事务来回滚，这个权重值可能是由该事务影响的行数(增加、删除、修改)决定的。

`SHOW ENGINE INNODB STATUS;` 看看最近死锁的日志

避免死锁

- 以固定的顺序访问表和行
- 大事务拆小，大事务更容易产生死锁
- 在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁概率
- 降低隔离级别
- 为表添加合理的索引