mysql insert锁机制

一、前言

上周遇到一个因insert而引发的死锁问题，其成因比较令人费解。
于是想要了解一下insert加锁机制，但是发现网上介绍的文章比较少且零散，挖掘过程比较忙乱。

本以为只需要系统学习一个较完全的逻辑，但是实际牵扯很多innodb锁相关知识及加锁方式。我好像并没有那么大的能耐，把各种场景的加锁过程一一列举并加之分析；亦没有太多的精力验证网上的言论的准确性。

只好根据现在了解的内容，参考官方文档，说说自己当前的理解。
本文仅供参考，如有误导，概不负责。

二、现场状态

不同的mysql版本，不同的参数设置，都可能对加锁过程有影响。
分析加锁机制还是应当尽可能多地列举一下关键参数，例如：当前mysql版本、事务隔离级别等。
如下，仅仅只列出个别比较重要的参数。

1.数据库版本

```
mysql> select version();
+-----------+
| version() |
+-----------+
| 5.6.27    |
+-----------+
```

2. 数据库引擎

```
mysql> show variables like '%engine%';
```

```
+---------------------------+--------+
| Variable_name             | Value  |
+---------------------------+--------+
| default_storage_engine    | InnoDB |
| default_tmp_storage_engine | InnoDB |
| storage_engine            | InnoDB |
+---------------------------+--------+
```

注：InnoDB支持事务，Myisam不支持事务；InnoDB支持行锁和表锁；Myisam不支持行锁。

3. 事务隔离级别

```
mysql> select @@global.tx_isolation, @@session.tx_isolation,
@@tx_isolation;
+----------------------+-----------------------+----------------+
| @@global.tx_isolation | @@session.tx_isolation | @@tx_isolation  |
+----------------------+-----------------------+----------------+
| REPEATABLE-READ       | REPEATABLE-READ        | REPEATABLE-READ |
+----------------------+-----------------------+----------------+
```

注：几种事务隔离级别：READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE

4. 查看gap锁开启状态

```
mysql> show variables like 'innodb_locks_unsafe_for_binlog';
+--------------------------------+-------+
| Variable_name                  | Value |
+--------------------------------+-------+
| innodb_locks_unsafe_for_binlog | OFF   |
+--------------------------------+-------+
```

innodb_locks_unsafe_for_binlog：默认值为0，即启用gap lock。
最主要的作用就是控制innodb是否对gap加锁。
但是，这一设置变更并不影响外键和唯一索引（含主键）对gap进行加锁的需要。
开启innodb_locks_unsafe_for_binlog的REPEATABLE-READ事务隔离级别，很大程度上已经蜕变成了READ-COMMITTED。

参见官方文档[^1]：

> By default, the value of innodb_locks_unsafe_for_binlog is 0 (disabled), which means that gap locking is enabled: InnoDB uses next-key locks for searches and index scans. To enable the variable, set it to 1. This causes gap locking to be disabled: InnoDB uses only index-record locks for searches and index scans.
>
> Enabling innodb_locks_unsafe_for_binlog does not disable the use of gap locking for foreign-key constraint checking or duplicate-key checking.
>
> The effect of enabling innodb_locks_unsafe_for_binlog is similar to but not identical to setting the transaction isolation level to READ COMMITTED.

5. 查看自增锁模式

```
mysql> show variables like 'innodb_autoinc_lock_mode';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| innodb_autoinc_lock_mode | 1     |
+--------------------------+-------+
```

innodb_autoinc_lock_mode有3种配置模式：0、1、2，分别对应”传统模式”,“连续模式”,“交错模式”。[^8]
传统模式：涉及auto-increment列的插入语句加的表级AUTO-INC锁，只有插入执行结束后才会释放锁。这是一种兼容MySQL 5.1之前版本的策略。
连续模式：可以事先确定插入行数的语句(包括单行和多行插入)，分配连续

的确定的auto-increment值；对于插入行数不确定的插入语句，仍加表锁。
这种模式下，事务回滚，auto-increment值不会回滚，换句话说，自增列内容会不连续。
交错模式：同一时刻多条SQL语句产生交错的auto-increment值。

由于insert语句常常涉及自增列的加锁过程，会涉及到AUTO-INC Locks加锁过程。
为了分步了解insert加锁过程，本文暂不讨论任何涉及自增列的加锁逻辑。
这一参数设置相关内容可能会出现在我的下一篇文章里。

n. etc

相关的参数配置越详情越好。

三、InnoDB锁类型[^2]

1. 基本锁

基本锁：共享锁(Shared Locks：S锁)与排他锁(Exclusive Locks：X锁)

mysql允许拿到S锁的事务读一行，允许拿到X锁的事务更新或删除一行。
加了S锁的记录，允许其他事务再加S锁，不允许其他事务再加X锁；
加了X锁的记录，不允许其他事务再加S锁或者X锁。

mysql对外提供加这两种锁的语法如下：
加S锁：select…lock in share mode
加X锁：select…for update

2. 意向锁(Intention Locks)

InnoDB为了支持多粒度(表锁与行锁)的锁并存，引入意向锁。
意向锁是表级锁，可分为意向共享锁(IS锁)和意向排他锁(IX锁)。

> InnoDB supports multiple granularity locking which permits coexistence of row-level locks and locks on entire tables. To make locking at multiple granularity levels practical, additional types of locks called intention locks are used. Intention locks are table-level locks in InnoDB that indicate which type

> of lock (shared or exclusive) a transaction will require later for a row in that table. There are two types of intention locks used in InnoDB (assume that transaction T has requested a lock of the indicated type on table t):
> Intention shared (IS): Transaction T intends to set S locks on individual rows in table t.
> Intention exclusive (IX): Transaction T intends to set X locks on those rows.

事务在请求S锁和X锁前，需要先获得对应的IS、IX锁。

> Before a transaction can acquire an S lock on a row in table t, it must first acquire an IS or stronger lock on t. Before a transaction can acquire an X lock on a row, it must first acquire an IX lock on t.

意向锁产生的主要目的是为了处理行锁和表锁之间的冲突，用于表明“某个事务正在某一行上持有了锁，或者准备去持有锁”。

> The main purpose of IX and IS locks is to show that someone is locking a row, or going to lock a row in the table.

共享锁、排他锁与意向锁的兼容矩阵如下：

	X	IX	S	IS
X	冲突	冲突	冲突	冲突
IX	冲突	兼容	冲突	兼容
S	冲突	冲突	兼容	兼容
IS	冲突	兼容	兼容	兼容

思考

从官方文档字面意思上看意向锁是表级锁，但是大牛不认为“Intention lock 是表级锁”[^5]?
另外，由于意向锁主要用于解决行锁与表锁间冲突问题，鉴于平时表级操作特别少，在分析加锁过程是否可以不用过多考虑意向锁的问题?

3. 行锁

记录锁(Record Locks)

记录锁, 仅仅锁住索引记录的一行。
单条索引记录上加锁，record lock锁住的永远是索引，而非记录本身，即使该表上没有任何索引，那么innodb会在后台创建一个隐藏的聚集主键索引，那么锁住的就是这个隐藏的聚集主键索引。所以说当一条sql没有走任何索引时，那么将会在每一条聚集索引后面加X锁，这个类似于表锁，但原理上和表锁应该是完全不同的。

参见官方文档[^3]:

> If the table has no PRIMARY KEY or suitable UNIQUE index, InnoDB internally generates a hidden clustered index on a synthetic column containing row ID values. The rows are ordered by the ID that InnoDB assigns to the rows in such a table. The row ID is a 6-byte field that increases monotonically as new rows are inserted. Thus, the rows ordered by the row ID are physically in insertion order.

间隙锁(Gap Locks)

区间锁, 仅仅锁住一个索引区间(开区间)。
在索引记录之间的间隙中加锁，或者是在某一条索引记录之前或者之后加锁，并不包括该索引记录本身。

next-key锁(Next-Key Locks)

record lock + gap lock, 左开右闭区间。

> A next-key lock is a combination of a record lock on the index record and a gap lock on the gap before the index record.
>
> By default, InnoDB operates in REPEATABLE READ transaction isolation level and with the innodb_locks_unsafe_for_binlog system variable disabled. In this case, InnoDB uses next-key locks for searches and index scans, which prevents phantom rows。

默认情况下，innodb使用next-key locks来锁定记录。
但当查询的索引含有唯一属性的时候，Next-Key Lock 会进行优化，将其降

级为Record Lock，即仅锁住索引本身，不是范围。

插入意向锁(Insert Intention Locks)

Gap Lock中存在一种插入意向锁（Insert Intention Lock），在insert操作时产生。在多事务同时写入不同数据至同一索引间隙的时候，并不需要等待其他事务完成，不会发生锁等待。
假设有一个记录索引包含键值4和7，不同的事务分别插入5和6，每个事务都会产生一个加在4-7之间的插入意向锁，获取在插入行上的排它锁，但是不会被互相锁住，因为数据行并不冲突。

> An insert intention lock is a type of gap lock set by INSERT operations prior to row insertion. This 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. Suppose that there are index records with values of 4 and 7. Separate transactions that attempt to insert values of 5 and 6, respectively, each lock the gap between 4 and 7 with insert intention locks prior to obtaining the exclusive lock on the inserted row, but do not block each other because the rows are nonconflicting.

注：插入意向锁并非意向锁，而是一种特殊的间隙锁。

4. 行锁的兼容矩阵[^4]

	Gap	Insert Intention	Record	Next-Key
Gap	兼容	兼容	兼容	兼容
Insert Intention	冲突	兼容	兼容	冲突
Record	兼容	兼容	冲突	冲突
Next-Key	兼容	兼容	冲突	冲突

表注：横向是已经持有的锁，纵向是正在请求的锁。

由于S锁和S锁是完全兼容的，因此在判别兼容性时只考虑持有的锁与请求的锁是这三种组合情形：X、S和S、X和X、X。
另外，需要提醒注意的是进行兼容判断也只是针对于加锁涉及的行有交集的

情形。

分析兼容矩阵可以得出如下几个结论：

- INSERT操作之间不会有冲突。
- GAP,Next-Key会阻止Insert。
- GAP和Record,Next-Key不会冲突
- Record和Record、Next-Key之间相互冲突。
- 已有的Insert锁不阻止任何准备加的锁。

5. 自增锁(AUTO-INC Locks)

AUTO-INC锁是一种特殊的表级锁，发生涉及AUTO_INCREMENT列的事务性插入操作时产生。

官方解释如下[^3]：

> 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.

四、insert加锁过程

官方文档[^6]对于insert加锁的描述如下：

> INSERT sets an exclusive lock on the inserted row. This lock is an index-record lock, not a next-key lock (that is, there is no gap lock) and does not prevent other sessions from inserting into the gap before the inserted row.
>
> Prior to inserting the row, a type of gap lock called an insert intention gap lock is set. This 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. Suppose that there are index records with values of 4 and 7. Separate transactions that attempt to insert values of 5 and 6 each lock the gap between 4 and 7 with insert

> intention locks prior to obtaining the exclusive lock on the inserted row, but do not block each other because the rows are nonconflicting.
>
> If a duplicate-key error occurs, a shared lock on the duplicate index record is set. This use of a shared lock can result in deadlock should there be multiple sessions trying to insert the same row if another session already has an exclusive lock.

简单的insert会在insert的行对应的索引记录上加一个排它锁，这是一个record lock，并没有gap，所以并不会阻塞其他session在gap间隙里插入记录。

不过在insert操作之前，还会加一种锁，官方文档称它为insertion intention gap lock，也就是意向的gap锁。这个意向gap锁的作用就是预示着当多事务并发插入相同的gap空隙时，只要插入的记录不是gap间隙中的相同位置，则无需等待其他session就可完成，这样就使得insert操作无须加真正的gap lock。
假设有一个记录索引包含键值4和7，不同的事务分别插入5和6，每个事务都会产生一个加在4-7之间的插入意向锁，获取在插入行上的排它锁，但是不会被互相锁住，因为数据行并不冲突。

假设发生了一个唯一键冲突错误，那么将会在重复的索引记录上加读锁。当有多个session同时插入相同的行记录时，如果另外一个session已经获得该行的排它锁，那么将会导致死锁。

思考：Insert Intention Locks作用

Insert Intention Locks的引入，我理解是为了提高数据插入的并发能力。
如果没有Insert Intention Locks的话，可能就需要使用Gap Locks来代替。

五、insert死锁场景分析

接下来，带大家看几个与insert相关的死锁场景。

1. duplicate key error引发的死锁

这个场景主要发生在两个以上的事务同时进行唯一键值相同的记录插入操作。

表结构

```
CREATE TABLE `aa` (
  `id` int(10) unsigned NOT NULL COMMENT '主键',
  `name` varchar(20) NOT NULL DEFAULT '' COMMENT '姓名',
  `age` int(11) NOT NULL DEFAULT '0' COMMENT '年龄',
  `stage` int(11) NOT NULL DEFAULT '0' COMMENT '关卡数',
  PRIMARY KEY (`id`),
  UNIQUE KEY `udx_name` (`name`),
  KEY `idx_stage` (`stage`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

表数据

```
mysql> select * from aa;
+----+------+-----+-------+
| id | name | age | stage |
+----+------+-----+-------+
|  1 | yst  |  11 |     8 |
|  2 | dxj  |   7 |     4 |
|  3 | lb   |  13 |     7 |
|  4 | zsq  |   5 |     7 |
|  5 | lxr  |  13 |     4 |
+----+------+-----+-------+
```

事务执行时序表

T1(36727)	T2(36728)	T3(36729)
begin;	begin;	begin;
insert into aa values(6, ‘test’, 12, 3);		

	insert into aa values(6, ‘test’, 12, 3);	
		insert into aa values(6, ‘test’, 12, 3);
rollback;		
		ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
	Query OK, 1 row affected (13.10 sec)	

如果T1未rollback，而是commit的话，T2和T3会报唯一键冲突：ERROR 1062 (23000): Duplicate entry ‘6’ for key ‘PRIMARY’

事务锁占用情况

T1 rollback前，各事务锁占用情况：

```
mysql> select * from information_schema.innodb_locks;

+-------------+-------------+-----------+-----------+-------------+------
------+-----------+-----------+----------+-----------+
| lock_id       | lock_trx_id | lock_mode | lock_type | lock_table  |
lock_index | lock_space | lock_page | lock_rec | lock_data |
+-------------+-------------+-----------+-----------+-------------+------
------+-----------+-----------+----------+-----------+
| 36729:24:3:7 | 36729       | S         | RECORD    | `test`.`aa` |
PRIMARY    |         24 |         3 |        7 | 6         |
| 36727:24:3:7 | 36727       | X         | RECORD    | `test`.`aa` |
PRIMARY    |         24 |         3 |        7 | 6         |
| 36728:24:3:7 | 36728       | S         | RECORD    | `test`.`aa` |
PRIMARY    |         24 |         3 |        7 | 6         |
+-------------+-------------+-----------+-----------+-------------+------
------+-----------+-----------+----------+-----------+
```

注：mysql有自己的一套规则来决定T2与T3哪个进行回滚，本文不做讨论。

死锁日志

```
------------------------
LATEST DETECTED DEADLOCK
------------------------
2016-07-21 19:34:23 700000a3f000
*** (1) TRANSACTION:
TRANSACTION 36728, ACTIVE 199 sec inserting
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1184, 2 row lock(s)
MySQL thread id 13, OS thread handle 0x700000b0b000, query id 590
localhost root update
insert into aa values(6, 'test', 12, 3)
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 24 page no 3 n bits 80 index `PRIMARY` of table
`test`.`aa` trx id 36728 lock_mode X insert intention waiting
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info
bits 0
 0: len 8; hex 73757072656d756d; asc supremum;;
*** (2) TRANSACTION:
TRANSACTION 36729, ACTIVE 196 sec inserting
mysql tables in use 1, locked 1
4 lock struct(s), heap size 1184, 2 row lock(s)
MySQL thread id 14, OS thread handle 0x700000a3f000, query id 591
localhost root update
insert into aa values(6, 'test', 12, 3)
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 24 page no 3 n bits 80 index `PRIMARY` of table
`test`.`aa` trx id 36729 lock mode S
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info
bits 0
 0: len 8; hex 73757072656d756d; asc supremum;;
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 24 page no 3 n bits 80 index `PRIMARY` of table
`test`.`aa` trx id 36729 lock_mode X insert intention waiting
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info
bits 0
```

```
 0: len 8; hex 73757072656d756d; asc supremum;;

*** WE ROLL BACK TRANSACTION (2)
```

死锁成因

事务T1成功插入记录，并获得索引id=6上的排他记录锁(LOCK_X | LOCK_REC_NOT_GAP)。
紧接着事务T2、T3也开始插入记录，请求排他插入意向锁(LOCK_X | LOCK_GAP | LOCK_INSERT_INTENTION)；但由于发生重复唯一键冲突，各自请求的排他记录锁(LOCK_X | LOCK_REC_NOT_GAP)转成共享记录锁(LOCK_S | LOCK_REC_NOT_GAP)。

T1回滚释放索引id=6上的排他记录锁(LOCK_X | LOCK_REC_NOT_GAP)，T2和T3都要请求索引id=6上的排他记录锁(LOCK_X | LOCK_REC_NOT_GAP)。
由于X锁与S锁互斥，T2和T3都等待对方释放S锁。
于是，死锁便产生了。

如果此场景下，只有两个事务T1与T2或者T1与T3，则不会引发如上死锁情况产生。

思考

- 为什么发现重复主键冲突的时候，要将事务请求的X锁转成S锁?
  (比较牵强的)个人理解，跟插入意向锁类型，也是为了提高插入的并发效率。

- 插入前请求插入意向锁的作用?
  个人认为，通过兼容矩阵来分析，Insert Intention Locks是为了减少插入时的锁冲突。

2. GAP与Insert Intention冲突引发的死锁

表结构

```
CREATE TABLE `t` (
```

```
  `a` int(11) NOT NULL,
  `b` int(11) DEFAULT NULL,
  PRIMARY KEY (`a`),
  KEY `idx_b` (`b`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

表数据

```
mysql> select * from t;
+----+------+
| a  | b    |
+----+------+
|  1 |    2 |
|  2 |    3 |
|  3 |    4 |
| 11 |   22 |
+----+------+
```

事务执行时序表

T1(36831)	T2(36832)
begin;	begin;
select * from t where b = 6 for update;	
	select * from t where b = 8 for update;
insert into t values (4,5);	
	insert into t values (4,5);
	ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
Query OK, 1 row affected (5.45 sec)	

事务锁占用情况

T2 insert前，各事务锁占用情况：

```
mysql> select * from information_schema.innodb_locks;
+-------------+------------+-----------+-----------+------------+-------
-----+-----------+-----------+----------+-----------+
| lock_id     | lock_trx_id | lock_mode | lock_type | lock_table |
lock_index | lock_space | lock_page | lock_rec | lock_data |
+-------------+------------+-----------+-----------+------------+-------
-----+-----------+-----------+----------+-----------+
| 36831:25:4:5 | 36831       | X,GAP     | RECORD    | `test`.`t` | idx_b
|         25 |         4 |        5 | 22, 11    |
| 36832:25:4:5 | 36832       | X,GAP     | RECORD    | `test`.`t` | idx_b
|         25 |         4 |        5 | 22, 11    |
+-------------+------------+-----------+-----------+------------+-------
-----+-----------+-----------+----------+-----------+
```

死锁日志

```
------------------------
LATEST DETECTED DEADLOCK
------------------------
2016-07-28 12:28:34 700000a3f000
*** (1) TRANSACTION:
TRANSACTION 36831, ACTIVE 17 sec inserting
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1184, 3 row lock(s), undo log
entries 1
MySQL thread id 38, OS thread handle 0x700000b0b000, query id 953
localhost root update
insert into t values (4,5)
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 25 page no 4 n bits 72 index `idx_b` of table
`test`.`t` trx id 36831 lock_mode X locks gap before rec insert intention
waiting
```

```
Record lock, heap no 5 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 80000016; asc     ;;
 1: len 4; hex 8000000b; asc     ;;
*** (2) TRANSACTION:
TRANSACTION 36832, ACTIVE 13 sec inserting
mysql tables in use 1, locked 1
3 lock struct(s), heap size 360, 2 row lock(s)
MySQL thread id 39, OS thread handle 0x700000a3f000, query id 954 localhost root update
insert into t values (4,5)
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 25 page no 4 n bits 72 index `idx_b` of table `test`.`t` trx id 36832 lock_mode X locks gap before rec
Record lock, heap no 5 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 80000016; asc     ;;
 1: len 4; hex 8000000b; asc     ;;
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 25 page no 3 n bits 72 index `PRIMARY` of table `test`.`t` trx id 36832 lock mode S locks rec but not gap waiting
Record lock, heap no 5 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 4; hex 80000004; asc     ;;
 1: len 6; hex 000000008fdf; asc       ;;
 2: len 7; hex 8d000001d00110; asc        ;;
 3: len 4; hex 80000005; asc     ;;
*** WE ROLL BACK TRANSACTION (2)
```

死锁成因

事务T1执行查询语句，在索引b=6上加排他Next-key锁(LOCK_X | LOCK_ORDINARY)，会锁住idx_b索引范围(4, 22)。
事务T2执行查询语句，在索引b=8上加排他Next-key锁(LOCK_X | LOCK_ORDINARY)，会锁住idx_b索引范围(4, 22)。由于请求的GAP与已持

有的GAP是兼容的，因此，事务T2在idx_b索引范围(4, 22)也能加锁成功。

事务T1执行插入语句，会先加排他Insert Intention锁。由于请求的Insert Intention锁与已有的GAP锁不兼容，则事务T1等待T2释放GAP锁。
事务T2执行插入语句，也会等待T1释放GAP锁。
于是，死锁便产生了。

注：LOCK_ORDINARY拥有LOCK_GAP一部分特性。

思考：Insert Intention锁在加哪级索引上?

这个排他锁加在PK上，还是二级索引上?

六、课后思考

1. 无主键的加锁过程
   无PK时，会创建一个隐式聚簇索引。加锁在这个隐式聚簇索引会有什么不同?
2. 复合索引加锁过程
3. 多条件(where condition)加锁过程
4. 隐式锁与显式锁，隐式锁什么情况下会转换成显式锁
5. 如果插入意向锁不阻止任何锁，这个锁还有必要存在吗?
   目前看到的作用是，通过加锁的方式来唤醒等待线程。
   但这并不意味着，被唤醒后可以直接做插入操作了。需要再次判断是否有锁冲突。

七、补充知识

1. 查看事务隔离级别

```
SELECT @@global.tx_isolation;
SELECT @@session.tx_isolation;
SELECT @@tx_isolation;
```

2. 设置隔离级别

SET [SESSION | GLOBAL] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE}
例如：set session transaction isolation level read uncommitted;

3. 查看auto_increment机制模式

show variables like ‘innodb_autoinc_lock_mode’;

4. 查看表状态

show table status like ‘plan_branch’\G;
show table status from test like ‘plan_branch’\G;

5. 查看SQL性能

show profiles
show profile for query 1;

6. 查看当前最新事务ID

每开启一个新事务，记录当前最新事务的id，可用于后续死锁分析。
show engine innodb status\G;

7. 查看事务锁等待状态情况

select from information_schema.innodb_locks;
select from information_schema.innodb_lock_waits;
select * from information_schema.innodb_trx;

8. 查看innodb状态(包含最近的死锁日志)

show engine innodb status;

八、参考文档

[^1]: InnoDB Startup Options and System Variables

[^2]: InnoDB Locking

[^3]: Clustered and Secondary Indexes

[^4]: [MySQL] gap lock/next-key lock浅析

[^5]: Intention Lock是否表级锁

[^6]: Locks Set by Different SQL Statements in InnoDB

[^8]: AUTO_INCREMENT lock Handing in InnoDB

[^9]: 深入理解innodb的锁(record,gap,Next-Key lock)

[^11]: The INFORMATION_SCHEMA INNODB_LOCKS Table

[^12]: The INFORMATION_SCHEMA INNODB_TRX Table

[^13]: mysql innodb插入意向锁