45 | 自增id用完怎么办

**表定义自增id**

表定义的自增id达到上限后的逻辑是：再申请下一个id时，得到的值保持不变

create table t(id int unsigned auto\_increment primary key) auto\_increment=4294967295;

insert into t values(null);

//成功插入一行 4294967295

show create table t;

/\* CREATE TABLE `t` (

`id` int(10) unsigned NOT NULL AUTO\_INCREMENT,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=4294967295;

\*/

insert into t values(null);

//Duplicate entry '4294967295' for key 'PRIMARY'

第一个insert语句插入数据成功后，这个表的AUTO\_INCREMENT没有改变（还是4294967295）这导致了第二个语句又拿到相同的自增id值，再试图执行插入语句，报主键冲突错误。

2的32次方-1,(4294967295)不是一个特别大的数，对于一个频繁插入删除数据的表来说，是可能会被用完的。因此在建表时需要考虑表示否有可能达到这个上限值，如果有可能，就应该创建成8个字节的bigint unsigned.

InnoDB系统自增row\_id.

如果创建的InnoDB表没有指定主键，那么InnoDB会创建一个不见的，长度为6个字节的row\_id,InnoDB维护了一个全局的dict\_sys.row\_id的值，所有无主键的InnoDB表，没插入一行数据，都将当前的dict\_sys.row\_id值作为要插入数据的row\_id,然后把dict\_sys.row\_id的值加1.

实际上，在代码实现时row\_id是一个长度为8个字节的无符号长整型（bigint unsigned）。但是，InnoDB在设计时，给row\_id留的只是6个字节的长度，这样写到数据库表中只放了最后6个字节，所以row\_id能写到数据表中的值，就有两个特征。

1. row\_id写入表中的范围，是从0到2的48次方-1
2. 当dict\_sys.row\_id=2的48次方时，如果再有插入数据的行为要来申请row\_id，拿到以后再去最后6个字节的话就是0；

也就是说，写入表的row\_id是从0到2的48次方-1，达到上限后，下一个值就是0，然后就绪循环。

当然2的48次方-1这个值本身已经很大了，但是如果一个MySQL实例跑的足够久的话，还是可能达到这个上限的。在InnoDB逻辑里，申请到row\_id=N后，将这行数据写入表中；如果表中已经存在row\_id=N的行，新写入的行就会覆盖原有的行。

验证这个结论，通过修改gdb修改系统的自增row\_id来实现，注意，用gdb该变量这个操作是为了便于复现问题，只能在测试环境使用

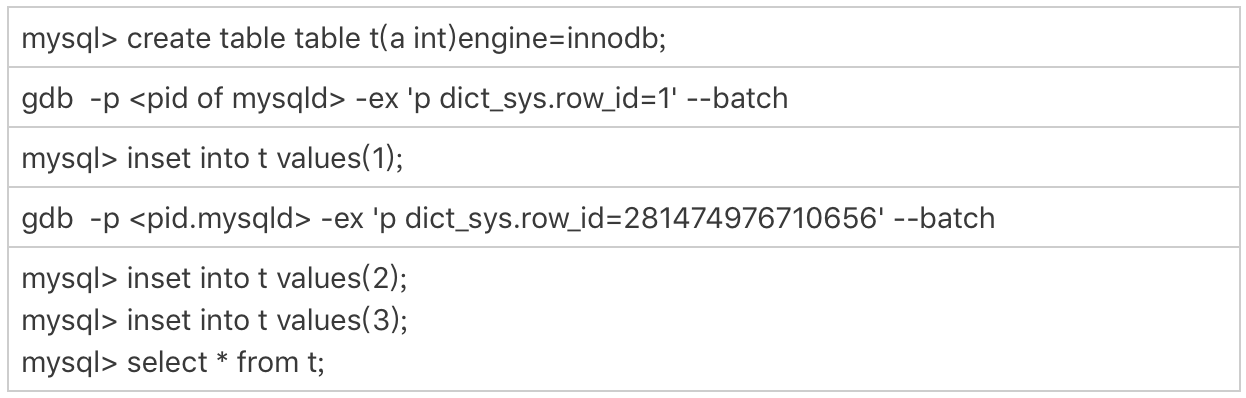


图1 row\_id用完的验证序列

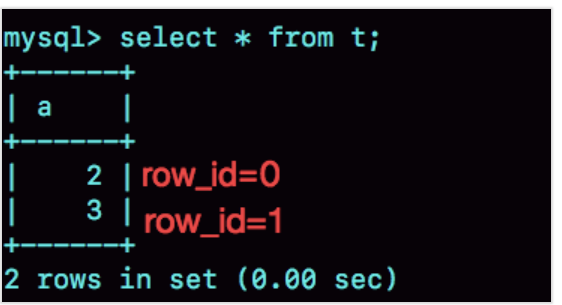


图2 row\_id用完的效果验证

在用gdb将dict\_sys.row\_id设置为2的48次方后，在插入的a=2的行会出现在表t的第一行，因为这个值的row\_id=0，之后在插入a=3的行，由于row-id=1,就覆盖了之前a=1的行，因为a=1这一行row\_id也是1.

从这个角度看，就意味着，影响的数据可靠性，报主键冲突，是插入失败，影响的是可用性。而一般情况下，可靠性优先于可用性。

Xid

那么，Xid在MySQL内部是怎么生成的呢？

MySQL内部维护了一个全局变量global\_query\_id,每次执行语句时将它赋值给Quyer\_id，然后给这个变量价加1，如果当前语句是这个事务执行的第一条语句，那么MySQL还会同时把Query\_id赋值给这个事务的Xid。

而global\_query\_id一个纯内存变量，重启之后就清零了，所以，在同一个数据库实例中，不同事务的Xid也是有可能相同的。

但是MySQL重启不会导致同一个binlog里面出现两个相同的Xid，但是如果global\_query\_id达到上限后，就会继续从0开始计数。从理论上讲，还是会出现同一个binlog里面出现相同的Xid的场景。

因为global\_query\_id定义的长度是8个字节，这个自增值的上限是2的64次方-1.要出现这种情况，必须是下面这样的过程：

1. 执行一个事务，假设Xid是A
2. 接下来执行2的64次方次查询语句，让global\_query\_id返回到A.
3. 再启动一个事务，这个事务的Xid也是A.

不过，2的64次方值太大了，大到可以认为理论上

**InnoDB trx\_id**

Xid 和InnoDB的trx\_id是两个容易混淆的概念

Xid是有server层维护的。InnoDB内部使用的Xid，就是为了能够在InnoDB事务和server之间做关联，但是，InnoDB自己的trx\_id，是另外维护的。

InnoDB内部维护了一个max\_trx\_id全局变量，每次需要申请一个新的trx\_id时，就获得max\_trx\_id的当前值，然后并将max\_trx\_id加1.

InnoDB数据可见行的核心思想是：每一行数据都记录了更新它的trx\_id,当一个事务读到一行数据时，判断这个数据是否可见的方法，就是通过事务的一致性视图与这行数据的trx\_id作对比。

对于正在执行的事务，可以从information\_schema.innodb\_trx表中看到事务的trx\_id

上一篇文章的思考题，就是从innodb\_trx表里面查到的trx\_id的，看事务现场

SessionB里，从innodb\_trx表里查出的这个两个字段，第二个字段trx\_mysql\_thread\_id就是线程id，显示线程id，是为了说明这两次查询看到事务对应的线程id都是5，也就是sessionA所在的线程。

可以看到，T2时刻显示的trx\_id是一个很大的数，T4时刻显示的trx\_id是1289，看上去是一个比较正常的数字，这是什么愿意呢？

实际上，在T1时刻，sessionA还没有涉及到更新，是一个只读事务，而对于只读事务，InnoDB并不会分配trx\_id,也就是说

1. 在T1时刻，trx\_id的值其实是0，而这个很大的数，只是用来显示的。
2. 直到sessionA在T3时刻执行insert语句的时候，InnoDB才真正分配了trx\_id,所以T4时刻，sessionB查到的这个trx\_d的值就是1289

需要注意的是，除了显而易见的修改类语句外，如果在select语句后面加上for update,这个事务也不是只读事务。

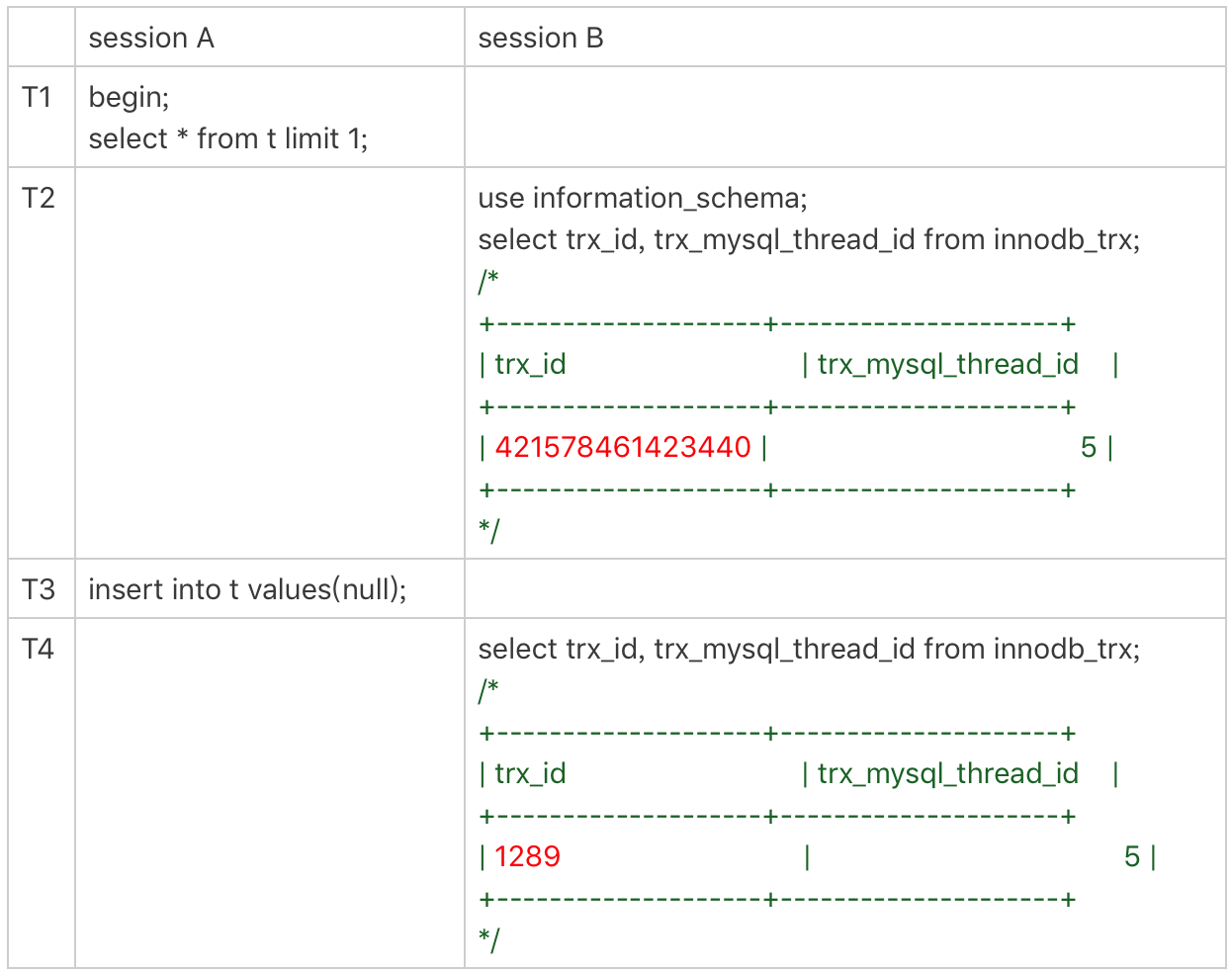


图3 事务的trx\_id

在上篇评论区，用同学提出，实验时发现不止加1，这是因为

1. update和delete语句除了事务本身，还涉及到标记删除旧数据，也就是要把数据放到purge队列里等待后续物理删除，这个操作也会把max\_trx\_id+1，因此在一个事务中至少2
2. .InnoDB的后台操作，比如表的索引信息统计这类操作，也会启动内部事务的，因此，可以看到trx\_id值并不会按照加1递增的。

那么**，在T2时刻查到的这个很大的数字是怎么来的呢？**

其实,这个数字是每次查询时候有系统临时计算出来的，它的算法是：把当前事务的trx变量指针地址转成整数，再加上2的48次方，使用这个算法，他可以保证以下两点：

1. 因为同一个只读事务在执行期间，它的指针地址是不会变的，所以不论是在innodb\_trx还是在innodb\_locks表里，同一个只读事务查出来的trx\_id就会是一样的
2. 如果有并行的多个只读事务，每个事务的trx变量的指针肯定不同。这样，不同的并发只读事务，查出来的trx\_id就是不同的。

**那么，为什么还有加上2的48次方？**

在显示值里面加上2的48次方，目的是要保证只读事务显示的trx\_id值比较大，正常情况下就会区别于读写事务的id，但是，trx\_id跟row\_id逻辑类似，定义长度也是8个字节，因此，在理论上还是可能出现一个读写事务与一个只读事务显示的trx\_id相同的情况，不过这个概率很低，并且没有什么实质危害，可以不用管。

**另外一个问题，只读事务不分配trx\_id,有什么好处呢？**

·一个好处是，这样做可以减小事务视图里面活跃事务数组的大小，因为当前正在运行的只读事务，是不影响数据的可见性判断的，所以，在创建事务的一致性试图时，InnoDB就只需要拷贝读写事务的trx\_id

·另一个好处是，可以减少trx\_id的申请次数。在InnoDB里，即使只执行一个普通的select语句，在执行过程中，也是要对应一个只读事物的。所以只读事务优化后，普通的查询语句不需要申请trx\_id,就大大减少了并发事务申请trx\_id的锁冲突。

由于只读事务不分配trx\_id,一个自然而然的结果就是trx\_id的增加速度变慢了。

但是，max\_trx\_id会持久化存储，重启也不会重置为0，从理论上，只要一个MySQL服务跑的足够久，就可能出现max\_trx\_id达到2的48次方-1的上限，然后从0开始的情况。

当到达这个状态后，MySQL就会持续出现一个脏读的bug,

首先需要把当前读的max\_trx\_id先修改成2的48次方-1.



图4 复现脏读

已经把系统的max\_trx\_id设置成了2的48次方-1，所以在sessionA事务TA的低水位就是2的48次方-1。

在T2时刻，sessionB执行第一条update语句的事务id是2的48次方-1，而第二条update语句事务id是0就是0了，这条update语句执行 后生成的的数据版本上的trx\_id就是0.

在T3时刻，sessionA执行select语句时，判断可见性发现，c=3这个数据版本的trx\_id，小于事务TA的低水位，因此认为这个数据可见。

但，这个是脏读

由于低水位会持续增加，而事务id从0开始计数，就导致了系统在这个时刻之后，所有查询都会出现脏读的。

并且，MySQL重启时，max\_trx\_id也不会清0，也就是说重启MySQL,这个bug依然存在。

**那么，这个bug也只是存在于理论上吗**

假设一个MySQL实例的TPSs是每秒50万，持续这个压力的话，在17.8年后，就会出现这个情况，如果TPS更高，这个年限自然更短。但是Mysql真正开始流行到现在，还没有实例跑到这个上限。不过，这个bug只要MySQL实例服务时间够长，就会必然出现。

**Thread\_id**

线程id才是mysql中最常见的一种自增id,平时查各种现场时，show processlist里面的第一列，就是thread\_id。

Thread\_id的逻辑很好理解，系统保存了一个全局变量thread\_id\_counter，每建一个连接，就将thread\_id\_counter赋值给这个连接的线程变量。

Thread\_id\_counter,定义了大小是4个字节，因此达到2的32次方-1，它就会重置为0，然后继续增加。但是，不会再show processlist里看到两个相同的thread\_id.

因为MySQL设计了一个唯一数组的逻辑，给新线程分配thread\_id时，逻辑代码是这样的

do {

new\_id= thread\_id\_counter++;

} while (!thread\_ids.insert\_unique(new\_id).second);

**小结**

每种自增id有各自的应用场景。达到上限后表现也不同

1. 表的自增id达到上限后，再申请他的值就不会变，进而导致继续插入数据是报主键冲突错误
2. Row\_id达到上限后，则会归0在重新递增，如果出现相同的row\_id，后写的数据会覆盖之前的数据。
3. Xid只需要不在同一个binlog文件中出现重复值即可，虽然理论上会出现重复值，但是概率极小，可以忽略不计
4. InnoDB的max\_trx\_id递增值每次MySQL重启都会被保存起来，所以我们文中提到的脏读的例子就是一个必然的bug,
5. Thread\_id时候常见的，而且也是处理最好的一个自增id逻辑。

还有table\_id,binlog文件序号等。