

一个最不可思议的MySQL死锁分析

1 死锁问题背景 1

1.1 一个不可思议的死锁 1

1.1.1 初步分析 3

1.2 如何阅读死锁日志 3

2 死锁原因深入剖析 4

2.1 Delete操作的加锁逻辑 4

2.2 死锁预防策略 5

2.3 剖析死锁的成因 6

3 总结 7

1. 死锁问题背景

做MySQL代码的深入分析也有些年头了，再加上自己10年左右的数据库内核研发经验，自认为对于MySQL/InnoDB的加锁实现了如指掌，正因如此，前段时间，还专门写了一篇洋洋洒洒的文章，专门分析MySQL的加锁实现细节：《[MySQL加锁处理分析](#)》。

但是，昨天”润洁”同学在《[MySQL加锁处理分析](#)》这篇博文下咨询的一个MySQL的死锁场景，还是彻底把我给难住了。此死锁，完全违背了本人原有的锁知识体系，让我百思不得其解。本着机器不会骗人，既然报出死锁，那么就一定存在死锁的原则，我又重新深入分析了InnoDB对应的源码实现，进行多次实验，配合恰到好处的灵光一现，还真让我分析出了这个死锁产生的原因。这篇博文的余下部分的内容安排，首先是给出”润洁”同学描述的死锁场景，然后再给出我的剖析。对个人来说，这是一篇十分有必要的总结，对此博文的读者来说，希望以后碰到类似的死锁问题时，能够明确死锁的原因所在。

死锁日志：

```
-----  
LATEST DETECTED DEADLOCK  
-----  
140122 18:11:58  
*** (1) TRANSACTION:  
TRANSACTION 930F9, ACTIVE 0 sec starting index read  
mysql tables in use 1, locked 1  
LOCK WAIT 2 lock struct(s), heap size 376, 1 row lock(s)  
MySQL thread id 2096, OS thread handle 0x7f3570976700, query id 1485879 localhost 127.0.0.1 rj  
updating  
delete from dltask where a='b' and b='b' and c='a'  
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:  
RECORD LOCKS space id 0 page no 12713 n bits 96 index `uniq_a_b_c` of table `dltst`.`dltask` trx id  
930F9 lock_mode X waiting  
*** (2) TRANSACTION:  
TRANSACTION 930F3, ACTIVE 0 sec starting index read  
mysql tables in use 1, locked 1  
3 lock struct(s), heap size 376, 2 row lock(s)  
MySQL thread id 2101, OS thread handle 0x7f3573d88700, query id 1485872 localhost 127.0.0.1 rj  
updating  
delete from dltask where a='b' and b='b' and c='a'  
*** (2) HOLDS THE LOCK(S):  
RECORD LOCKS space id 0 page no 12713 n bits 96 index `uniq_a_b_c` of table `dltst`.`dltask` trx id  
930F3 lock_mode X locks rec but not gap  
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:  
RECORD LOCKS space id 0 page no 12713 n bits 96 index `uniq_a_b_c` of table `dltst`.`dltask` trx id  
930F3 lock_mode X waiting  
*** WE ROLL BACK TRANSACTION (1)
```

1. 初步分析

并发事务，每个事务只有一条SQL语句：给定唯一的二级索引键值，删除一条记录。每个事务，最多只会删除一条记录，为什么会产生死锁？这绝对是不可能的。但是，事实上，却真的是发生了死锁。产生死锁的两个事务，删除的是同一条记录，这应该是死锁发生的一个潜在原因，但是，即使是删除同一条记录，从原理上来说，也不应该产生死锁。因此，经过初步分析，这个死锁是不可能产生的。这个结论，远远不够！

1. 如何阅读死锁日志

在详细给出此死锁产生的原因之前，让我们先来看看，如何阅读MySQL给出的死锁日志。

以上打印出来的死锁日志，由InnoDB引擎中的lock0lock.c::lock_deadlock_recursive()函数产生。死锁中的事务信息，通过调用函数lock_deadlock_trx_print()处理；而每个事务持有、等待的锁信息，由lock_deadlock_lock_print()函数产生。

例如，以上的死锁，有两个事务。事务1，当前正在操作一张表(mysql tables in use 1)，持有两把锁(2 lock structs，一个表级意向锁，一个行锁(1 row lock))，这个事务，当前正在处理的语句是一条delete语句。同时，这唯一的一个行锁，处于等待状态(WAITING FOR THIS LOCK TO BE GRANTED)。

事务1等待中的行锁，加锁的对象是唯一索引uniq_a_b_c上页面号为12713页面上的一行(注：具体是哪一行，无法看到。但是能够看到的是，这个行锁，一共有96个bits可以用来锁96个行记录，n bits 96: lock_rec_print()方法)。同时，等待的行锁模式为next key锁(lock_mode X)。(注：关于InnoDB的锁模式，可参考我早期的一篇PPT：《[InnoDB 事务/锁/多版本实现分析](#)》。简单来说，next key锁有两层含义，一是对当前记录加X锁，防止记录被并发修改，同时锁住记录之前的GAP，防止有新的记录插入到此记录之前。)

同理，可以分析事务2。事务2上有两个行锁，两个行锁对应的也都是唯一索引uniq_a_b_c上页面号为12713页面上的某一条记录。一把行锁处于持有状态，锁模式为X lock with no gap(注：记录锁，只锁记录，但是不锁记录前的GAP，no gap lock)。一把行锁处于等待状态，锁模式为next key锁(注：与事务1等待的锁模式一致。同时，需要注意的一点是，事务2的两个锁模式，并不是一致的，不完全相容。持有的锁模式为X lock with no gap，等待的锁模式为next key lock X。因此，并不能因为持有了X lock with no gap，就可以说next key lock X就一定能够加上。))。

分析这个死锁日志，就能发现一个死锁。事务1的next key lock X正在等待事务2持有的X lock with no gap(行锁X冲突)，同时，事务2的next key

lock X，却又在等待事务1正在等待中的next key锁(注：这里，事务2等待事务1的原因，在于公平竞争，杜绝事务1发生饥饿现象。)，形成循环等待，死锁产生。

死锁产生后，根据两个事务的权重，事务1的权重更小，被选为死锁的牺牲者，回滚。

根据对于死锁日志的分析，确认死锁确实存在。而且，产生死锁的两个事务，确实都是在运行同样的基于唯一索引的等值删除操作。既然死锁确实存在，那么接下来，就是抓出这个死锁产生原因。

1. 死锁原因深入剖析

1. Delete操作的加锁逻辑

在《[MySQL加锁处理分析](#)》一文中，我详细分析了各种SQL语句对应的加锁逻辑。例如：Delete语句，内部就包含一个当前读(加锁读)，然后通过当前读返回的记录，调用Delete操作进行删除。在此文的 组合六：id唯一索引+RR 中，可以看到，RR隔离级别下，针对于满足条件的查询记录，会对记录加上排它锁(X锁)，但是并不会锁住记录之前的GAP(no gap lock)。对应到此文上面的死锁例子，事务2所持有的锁，是一把记录上的排它锁，但是没有锁住记录前的GAP(lock_mode X locks rec but not gap)，与我之前的加锁分析一致。

其实，在《MySQL加锁处理分析》一文中的 组合七：id非唯一索引+RR 部分的最后，我还提出了一个问题：如果组合五、组合六下，针对SQL：
*select * from t1 where id = 10 for update;* 第一次查询，没有找到满足查询条件的记录，那么GAP锁是否还能够省略？针对此问题，参与的朋友在做过试验之后，给出的正确答案是：此时GAP锁不能省略，会在第一个不满足查询条件的记录上加GAP锁，防止新的满足条件的记录插入。

其实，以上两个加锁策略，都是正确的。以上两个策略，分别对应的是：
1) 唯一索引上满足查询条件的记录存在并且有效；
2) 唯一索引上满足查询条件的记录不存在。但是，除了这两个之外，其实还有第三种：
3) 唯一索引上满足查询条件的记录存在但是无效。众所周知，InnoDB上删除一条记录，并不是真正意义上的物理删除，而是将记录标识为删除状态。

(注：这些标识为删除状态的记录，后续会由后台的Purge操作进行回收，物理删除。但是，删除状态的记录会在索引中存放一段时间。) 在RR隔离级别下，唯一索引上满足查询条件，但是却是删除记录，如何加锁？InnoDB在此处的处理策略与前两种策略均不相同，或者说是前两种策略的组合：对于满足条件的删除记录，InnoDB会在记录上加next key lock X(对记录本身加X锁，同时锁住记录前的GAP，防止新的满足条件的记录插入。) Unique查询，三种情况，对应三种加锁策略，总结如下：

- 找到满足条件的记录，并且记录有效，则对记录加X锁，No Gap锁(lock_mode X locks rec but not gap)；
- 找到满足条件的记录，但是记录无效(标识为删除的记录)，则对记录加next key锁(同时锁住记录本身，以及记录之前的Gap：lock_mode X)；
- 未找到满足条件的记录，则对第一个不满足条件的记录加Gap锁，保证没有满足条件的记录插入(locks gap before rec)；

此处，我们看到了next key锁，是否很眼熟？对了，前面死锁中事务1，事务2处于等待状态的锁，均为next key锁。明白了这三个加锁策略，其实构造一定的并发场景，死锁的原因已经呼之欲出。但是，还有一个前提策略需要介绍，那就是InnoDB内部采用的死锁预防策略。

1. 死锁预防策略

InnoDB引擎内部(或者说是所有的数据库内部)，有多种锁类型：事务锁(行锁、表锁)，Mutex(保护内部的共享变量操作)、RWLock(又称之为Latch，保护内部的页面读取与修改)。

InnoDB每个页面为16K，读取一个页面时，需要对页面加S锁，更新一个页面时，需要对页面加上X锁。任何情况下，操作一个页面，都会对页面加锁，页面锁加上之后，页面内存储的索引记录才不会被并发修改。

因此，为了修改一条记录，InnoDB内部如何处理：

1. 根据给定的查询条件，找到对应的记录所在页面；

2. 对页面加上X锁(RWLock)，然后在页面内寻找满足条件的记录；
 3. 在持有页面锁的情况下，对满足条件的记录加事务锁(行锁：根据记录是否满足查询条件，记录是否已经被删除，分别对应于上面提到的3种加锁策略之一)；
 4. **死锁预防策略**：相对于事务锁，页面锁是一个短期持有的锁，而事务锁(行锁、表锁)是长期持有的锁。因此，为了防止页面锁与事务锁之间产生死锁。InnoDB做了死锁预防的策略：持有事务锁(行锁、表锁)，可以等待获取页面锁；但反之，持有页面锁，不能等待持有事务锁。
 5. 根据死锁预防策略，在持有页面锁，加行锁的时候，如果行锁需要等待。则释放页面锁，然后等待行锁。此时，行锁获取没有任何锁保护，因此加上行锁之后，记录可能已经被并发修改。因此，此时要重新加回页面锁，重新判断记录的状态，重新在页面锁的保护下，对记录加锁。如果此时记录未被并发修改，那么第二次加锁能够很快完成，因为已经持有了相同模式的锁。但是，如果记录已经被并发修改，那么，就有可能导致本文前面提到的死锁问题。
1. 以上的InnoDB死锁预防处理逻辑，对应的函数，是row0sel.c::row_search_for_mysql()。感兴趣的朋友，可以跟踪调试下这个函数的处理流程，很复杂，但是集中了InnoDB的精髓。

1. 剖析死锁的成因

做了这么多铺垫，有了Delete操作的3种加锁逻辑、InnoDB的死锁预防策略等准备知识之后，再回过头来分析本文最初提到的死锁问题，就会手到拈来，事半功倍。

首先，假设dltask中只有一条记录：(1, 'a', 'b', 'c', 'data')。三个并发事务，同时执行以下的这条SQL：

```
delete from dltask where a='a' and b='b' and c='c';
```

并且产生了以下的并发执行逻辑，就会产生死锁：

Transaction 0: delete from dltask where a=' a' and b=' b' and c=' c' ;

Transaction 1: delete from dltask where a=' a' and b=' b' and c=' c' ;

Transaction 2: delete from dltask where a=' a' and b=' b' and c=' c' ;

Transaction 0	Transaction 1	Transaction 2
<p>1. 在 uniq_a_b_c 索引上, 对 (a,b,c) 记录加锁(lock_mode X locks rec but not gap);</p> <p>3. 进行删除操作, 将索引上的记录标识为删除状态;</p> <p>4 事务0提交, 释放记录锁;</p>	<p>6. 此时事务1持有页面锁, 读取到对应的记录, 发现满足条件的记录为删除状态。根据满足条件的记录存在, 但为删除状态的加锁逻辑, 尝试对记录加next key锁(X lock plus gap lock)。但是, 由于事务2持有记录锁, 需要等待。因此释放页面锁, 等待事务2释放记录锁, 等待的锁模式为next key(lock_mode X waiting);</p>	<p>2. 在 uniq_a_b_c 索引上, 对 (a,b,c) 记录加锁(lock_mode X locks rec but not gap): 此时需要等待, 因此释放页面锁, 等待记录锁;</p> <p>5. 事务2在事务0提交后, 成功获取记录锁(X lock not gap)。此时, 由于此锁是在释放页面锁之后获取, 记录可能已经被修改, 因此需要 Restart, 重新判断记录状态;</p> <p>7. 事务2 Restart, 重新获取页面锁, 读取符合条件的记录。此时, 符合条件的记录已经变为删除状态, 因此按照标准, 需要加上next key锁。但是, 由于事务1已经在等待, 并且 next key锁与事务2已经持有的(X lock not gap)不兼容, 与事务1等待的next key锁互斥。因此, 为了防止事务1发生饥饿现象, 事务2释放页面锁, 等待获取next key锁(lock_mode X waiting);</p> <p>8. 事务2等待事务1, 事务1等待事务2, 形成死锁。此时, 要选择一个事务牺牲。事务的权重如何计算, 可参考trx0trx.c::trx_weight_ge()函数。此处, 牺牲的是事务1, 事务1的权重较低, 与前面提到的死锁情况与事务回滚情况, 一致。</p>

上面分析的这个并发流程，完整展现了死锁日志中的死锁产生的原因。其实，根据事务1步骤6，与事务0步骤3/4之间的顺序不同，死锁日志中还有可能产生另外一种情况，那就是事务1等待的锁模式为记录上的X锁 + No Gap锁(lock_mode X locks rec but not gap waiting)。这第二种情况，也是”润洁”同学给出的死锁用例中，使用MySQL 5.6.15版本测试出来的死锁产生的原因。

1. 总结

行文至此，MySQL基于唯一索引的单条记录的删除操作并发，也会产生死锁的原因，已经分析完毕。其实，分析此死锁的难点，在于理解MySQL/InnoDB的行锁模式，针对不同情况下的加锁模式的区别，以及InnoDB处理页面锁与事务锁的死锁预防策略。明白了这些，死锁的分析就会显得清晰明了。

最后，总结下此类死锁，产生的几个前提：

- Delete操作，针对的是唯一索引上的等值查询的删除；（范围下的删除，也会产生死锁，但是死锁的场景，跟本文分析的场景，有所不同）
- 至少有3个(或以上)的并发删除操作；
- 并发删除操作，有可能删除到同一条记录，并且保证删除的记录一定存在；
- 事务的隔离级别设置为Repeatable Read，同时未设置innodb_locks_unsafe_for_binlog参数(此参数默认为FALSE)；(Read Committed隔离级别，由于不会加Gap锁，不会有next key，因此也不会产生死锁)
- 使用的是InnoDB存储引擎；（废话！MyISAM引擎根本就没有行锁）