我的Mysql死锁排查过程（案例分析）

原创 Joker_Ye 发布于2017-02-21 09:12:44 阅读数 3582 收藏 展开

以前接触到的数据库死锁，都是批量更新时加锁顺序不一致而导致的死锁，但是上周却遇到了一个很难理解的死锁。借着这个机会又重新学习了一下mysql的死锁知识以及常见的死锁场景。在多方调研以及和同事们的讨论下终于发现了这个死锁问题的成因，收获颇多。虽然是后端程序员，我们不需要像DBA一样深入地去分析与锁相关的源码，但是如果我们能够掌握基本的死锁排查方法，对我们的日常开发还是大有裨益的。

死锁起因

先介绍一下数据库和表情况，因为涉及到公司内部真是的数据，所以以下都做了模拟，不会影响具体的分析。

我们采用的是5.5版本的mysql数据库，事务隔离级别是默认的RR（Repeatable-Read），采用innodb引擎。假设存在test表：

```
CREATE TABLE `test` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `a` int(11) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `a` (`a`)
)
```

表的结构很简单，一个主键id，另一个唯一索引a。表里的数据如下：

```
mysql> select * from test;
+----+------+
| id | a    |
+----+------+
|  1 |    1 |
|  2 |    2 |
|  4 |    4 |
+----+------+
3 rows in set (0.00 sec)
```

出现死锁的操作如下：

步骤	事务1	事务2
1		begin
2		delete from test where a = 2;
3	begin	
4	delete from test where a = 2; （事务1卡住）	
5	提示出现死锁：ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction	insert into test (id, a) values (10, 2);

然后我们可以通过 SHOW ENGINE INNODB STATUS; 来查看死锁日志：

```
------------------------
LATEST DETECTED DEADLOCK
------------------------
170219 13:31:31
*** (1) TRANSACTION:
TRANSACTION 2A8BD, ACTIVE 11 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 448218, OS thread handle 0x2abe5fb5d700, query id 18923238 renjun.fangcloud.net 121.41.41.92 root updating
delete from test where a = 2
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table `oauthdemo`.`test` trx id 2A8BD lock_mode X waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;

*** (2) TRANSACTION:
TRANSACTION 2A8BC, ACTIVE 18 sec inserting
mysql tables in use 1, locked 1
```

```
MySQL thread id 448217, OS thread handle 0x2abe5fd03700, query id 18923239 renjun.fangcloud.net 121.41.41.92 root update22
insert into test (id,a) values (10,2)23
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table `oauthdemo`.`test` trx id 2A8BC lock_mode X locks rec but not gap
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table `oauthdemo`.`test` trx id 2A8BC lock mode S waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;

*** WE ROLL BACK TRANSACTION (1)
```

分析

阅读死锁日志

遇到死锁，第一步就是阅读死锁日志。死锁日志通常分为两部分，上半部分说明了事务1在等待什么锁：

```
170219 13:31:31
*** (1) TRANSACTION:
TRANSACTION 2A8BD, ACTIVE 11 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 448218, OS thread handle 0x2abe5fb5d700, query id 18923238 renjun.fangcloud.net 121.41.41.92 root updating
delete from test where a = 2
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table `oauthdemo`.`test` trx id 2A8BD lock_mode X waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;
```

从日志里我们可以看到事务1当前正在执行 delete from test where a = 2 ，该条语句正在申请索引a的X锁，所以提示 lock_mode X waiting 。

然后日志的下半部分说明了事务2当前持有的锁以及等待的锁：

```
*** (2) TRANSACTION:
TRANSACTION 2A8BC, ACTIVE 18 sec inserting
mysql tables in use 1, locked 1
4 lock struct(s), heap size 1248, 3 row lock(s), undo log entries 2
MySQL thread id 448217, OS thread handle 0x2abe5fd65700, query id 18923239 renjun.fangcloud.net 121.41.41.92 root update
insert into test (id,a) values (10,2)
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table `oauthdemo`.`test` trx id 2A8BC lock_mode X locks rec but not gap
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table `oauthdemo`.`test` trx id 2A8BC lock mode S waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;
```

从日志的 HOLDS THE LOCKS(S) 块中我们可以看到事务2持有索引a的X锁，并且是记录锁（Record Lock）。该锁是通过事务2在步骤2执行的delete语句申请的。由于是RR隔离模式下的基于唯一索引的等值查询（Where a = 2），所以会申请一个记录锁，而非next-key锁。

从日志的 WAITING FOR THIS LOCK TO BE GRANTED 块中我们可以看到事务2正在申请S锁，也就是共享锁。该锁是 insert into test (id,a) values (10,2) 语句申请的。 insert语句在普通情况下是会申请排他锁，也就是X锁，但是这里出现了S锁。这是因为a字段是一个唯一索引，所以insert语句会在插入前进行一次duplicate key的检查，为了使这次检查成功，需要申请S锁防止其他事务对a字段进行修改。

那么为什么该S锁会失败呢？这是 **对同一个字段的锁的申请是需要排队的** 。S锁前面还有一个未申请成功的X锁，所以S锁必须等待，所以形成了循环等待，死锁出现了。

死锁形成流程图

为了让大家更好地理解死锁形成的原因，我们再通过表格的形式阐述死锁形成的流程：

步骤	事务1	事务2
1		begin
2		delete from test where a = 2; 执行成功，事务2占有a=2下的X锁，类型为记录锁。
3	begin	
4	delete from test where a = 2; 事务1希望申请a=2下的X锁，但是由于事务2已经申请了一把X锁，两把X锁互斥，所以X锁申请进入锁请求队列。	
5	出现死锁，事务1权重较小，所以被选择回滚（成为牺牲品）。	insert into test (id, a) values (10, 2); 由于a字段建立了唯一索引，所以需要申请S锁以便检查duplicate key，由于插入的a的值还是2，所以排在X锁后面。但是前面的X锁的申请只有在事务2commit或者rollback之后才能成功，此时形成了循环等待，死锁产生。

拓展

在排查死锁的过程中，有个同事还发现了上述场景会产生另一种死锁，该场景无法通过手工复现，只有高并发场景下才有可能复现。

该死锁对应的日志这里就不贴出了，与上一个死锁的核心差别是事务2等待的锁从S锁换成了X锁，也就是 lock_mode X locks gap before rec insert intention waiting 。我们还是通过表格来详细说明该死锁产生的流程：

步骤	事务1	事务2
1		begin
2		delete from test where a = 2; 执行成功，事务2占有a=2下的X锁，类型为记录锁。

3	begin	
4		【insert第1阶段】insert into test (id, a) values (10, 2); 事务2申请S锁进行duplicate key进行检查。检查成功。
5	delete from test where a = 2; 事务1希望申请a=2下的X锁，但是由于事务2已经申请了一把X锁，两把X锁互斥，所以X锁申请进入锁请求队列。	
6	出现死锁，事务1权重较小，所以被选择回滚（成为牺牲品）。	【insert第2阶段】insert into test (id, a) values (10, 2); 事务2开始插入数据，S锁升级为X锁，类型为insert intention。同理，X锁进入队列排队，形成循环等待，死锁产生。

总结

排查死锁时，首先需要根据死锁日志分析循环等待的场景，然后根据当前各个事务执行的SQL分析出加锁类型以及顺序，逆向推断出如何形成循环等待，这样就能找到死锁产生的原因了。