

35 | 记一次线上SQL死锁事故：如何避免死锁？

2019-08-13 刘超



你好，我是刘超。今天我们来聊聊死锁，开始之前，先分享个小故事，相信你可能遇到过，或能从中获得一点启发。

之前我参与过一个项目，在项目初期，我们是没有将读写表分离的，而是基于一个主库完成读写操作。在业务量逐渐增大的时候，我们偶尔会收到系统的异常报警信息，DBA通知我们数据库出现了死锁异常。

按理说业务开始是比较简单的，就是新增订单、修改订单、查询订单等操作，那为什么会出现死锁呢？经过日志分析，我们发现是作为幂等性校验的一张表经常出现死锁异常。我们和DBA讨论之后，初步怀疑是索引导致的死锁问题。后来我们在开发环境中模拟了相关操作，果然重现了该死锁异常。

接下来我们就通过实战来重现下该业务死锁异常。首先，创建一张订单记录表，该表主要用于校验订单重复创建：

```
CREATE TABLE `order_record` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `order_no` int(11) DEFAULT NULL,  
  `status` int(4) DEFAULT NULL,  
  `create_date` datetime(0) DEFAULT NULL,  
  PRIMARY KEY (`id`) USING BTREE,  
  INDEX `idx_order_status` (`order_no`, `status`) USING BTREE  
) ENGINE = InnoDB
```

为了能重现该问题，我们先将事务设置为手动提交。这里要注意一下，MySQL数据库和Oracle提交事务不太一样，MySQL数据库默认情况下是自动提交事务，我们可以通过以下命令行查看自动提交事务是否开启：

```
mysql> show variables like 'autocommit';  
  
+-----+-----+  
| Variable_name | Value |  
+-----+-----+  
| autocommit    | ON    |  
+-----+-----+  
1 row in set (0.01 sec)
```

下面就操作吧，先将MySQL数据库的事务提交设置为手动提交，通过以下命令行可以关闭自动提交事务：

```
mysql> set autocommit = 0;  
  
Query OK, 0 rows affected (0.00 sec)
```

订单在做幂等性校验时，先是通过订单号检查订单是否存在，如果不存在则新增订单记录。知道具体的逻辑之后，我们再来模拟创建产生死锁的运行SQL语句。首先，我们模拟新建两个订单，并按照以下顺序执行幂等性校验SQL语句（垂直方向代表执行的时间顺序）：

事务A	事务B
BEGIN;	BEGIN;
SELECT id FROM `order_record` where `order_no` = 4 for update;//检查是否存在 order_no等于4的订单	
	SELECT id FROM `order_record` where `order_no` = 5 for update;//检查是否存在 order_no等于5的订单
INSERT INTO `order_record`(`order_no`, `status`, `create_date`) VALUES (4, 1, '2019-07-13 10:57:03');//如果没有，则插入 信息 此时，锁等待中.....	
	INSERT INTO `order_record`(`order_no`, `status`, `create_date`) VALUES (5, 1, '2019-07-13 10:57:03');//如果没有，则插入 信息 此时，锁等待中.....
COMMIT;(未完成)	COMMIT;(未完成)

此时，我们会发现两个事务已经进入死锁状态。我们可以在`information_schema`数据库中查询到具体的死锁情况，如下图所示：

```
mysql> use information_schema
Database changed
mysql> select * from innodb_lock_waits;
+-----+-----+-----+-----+
| requesting_trx_id | requested_lock_id | blocking_trx_id | blocking_lock_id |
+-----+-----+-----+-----+
| 369579            | 369579:144:5:1   | 369576          | 369576:144:5:1   |
+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql> select * from innodb_locks;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| lock_id          | lock_trx_id | lock_mode | lock_type | lock_table          | lock_index | lock_space | lock_page | lock_rec | lock_data          |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 369579:144:5:1   | 369579      | X         | RECORD   | `demo`.`order_record` | idx_order  | 144        | 5         | 1        | supremum pseudo-record |
| 369576:144:5:1   | 369576      | X         | RECORD   | `demo`.`order_record` | idx_order  | 144        | 5         | 1        | supremum pseudo-record |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

看到这，你可能会想，为什么SELECT要加for update排他锁，而不是使用共享锁呢？试想下，如果是两个订单号一样的请求同时进来，就有可能出现幻读。也就是说，一开始事务A中的查询没有该订单号，后来事务B新增了一个该订单号的记录，此时事务A再新增一条该订单号记录，就会创建重复的订单记录。面对这种情况，我们可以使用锁间隙算法来防止幻读。

死锁是如何产生的？

上面我们说到了锁间隙，在[第33讲](#)中，我已经讲过了并发事务中的锁机制以及行锁的具体实现算法，不妨回顾一下。

行锁的具体实现算法有三种：**record lock**、**gap lock**以及**next-key lock**。**record lock**是专门对索引项加锁；**gap lock**是对索引项之间的间隙加锁；**next-key lock**则是前面两种的组合，对索引项以其之间的间隙加锁。

只在可重复读或以上隔离级别下的特定操作才会取得**gap lock**或**next-key lock**，在**Select**、**Update**和**Delete**时，除了基于唯一索引的查询之外，其它索引查询时都会获取**gap lock**或**next-key lock**，即锁住其扫描的范围。主键索引也属于唯一索引，所以主键索引是不会使用**gap lock**或**next-key lock**。

在MySQL中，**gap lock**默认是开启的，即**innodb_locks_unsafe_for_binlog**参数值是**disable**的，且MySQL中默认的是RR事务隔离级别。

当我们执行以下查询SQL时，由于**order_no**列为非唯一索引，此时又是RR事务隔离级别，所以**SELECT**的加锁类型为**gap lock**，这里的**gap**范围是**(4,+∞)**。

```
SELECT id FROM demo.order_record where order_no = 4 for update;
```

执行查询SQL语句获取的**gap lock**并不会导致阻塞，而当我们执行以下插入SQL时，会在插入间隙上再次获取插入意向锁。插入意向锁其实也是一种**gap**锁，它与**gap lock**是冲突的，所以当其它事务持有该间隙的**gap lock**时，需要等待其它事务释放**gap lock**之后，才能获取到插入意向锁。

以上事务A和事务B都持有间隙**(4,+∞)**的**gap**锁，而接下来的插入操作为了获取到插入意向锁，都在等待对方事务的**gap**锁释放，于是就造成了循环等待，导致死锁。

```
INSERT INTO demo.order_record(order_no, status, create_date) VALUES (5, 1, '2019-07-13 10:57:03');
```

我们可以通过以下锁的兼容矩阵图，来查看锁的兼容性：

	Gap	Insert Intention	Record	Next-Key
Gap	兼容	冲突	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
备注	横向是已经持有的锁，纵向是正在请求的锁。			

避免死锁的措施

知道了死锁问题源自哪儿，就可以找到合适的方法来避免它了。

避免死锁最直观的方法就是在两个事务相互等待时，当一个事务的等待时间超过设置的某一阈值，就对这个事务进行回滚，另一个事务就可以继续执行了。这种方法简单有效，在InnoDB中，参数innodb_lock_wait_timeout是用来设置超时时间的。

另外，我们还可以将order_no列设置为唯一索引列。虽然不能防止幻读，但我们可以利用它的唯一性来保证订单记录不重复创建，这种方式唯一的缺点就是当遇到重复创建订单时会抛出异常。

我们还可以使用其它的方式来代替数据库实现幂等性校验。例如，使用Redis以及ZooKeeper来实现，运行效率比数据库更佳。

其它常见的SQL死锁问题

这里再补充一些常见的SQL死锁问题，以便你遇到时也能知道其原因，从而顺利解决。

我们知道死锁的四个必要条件：互斥、占有且等待、不可强占用、循环等待。只要系统发生死锁，这些条件必然成立。所以在一些经常需要使用互斥共用一些资源，且有可能循环等待的业务场景中，要特别注意死锁问题。

接下来，我们再来了解一个出现死锁的场景。

我们讲过，InnoDB存储引擎的主键索引为聚簇索引，其它索引为辅助索引。如果我们之前使用辅助索引来更新数据库，就需要修改为使用聚簇索引来更新数据库。如果两个更新事务使用了不同的辅助索引，或一个使用了辅助索引，一个使用了聚簇索引，就都有可能导致锁资源的循环等待。由于本身两个事务是互斥，也就构成了以上死锁的四个必要条件了。

我们还是以上面的这个订单记录表来重现下聚簇索引和辅助索引更新时，循环等待锁资源导致的死锁问题：

事务A	事务B
BEGIN;	BEGIN;
UPDATE `order_record` SET status = 1 WHERE `order_no` = 4 ;	UPDATE `order_record` SET status = 1 WHERE id = 4 ;

出现死锁的步骤：

事务A	事务B
首先获取idx_order_status非聚族索引	
	获取主键索引的行锁
根据非聚族索引获取的主键，获取主键索引的行锁	
	更新status列时，需要获取idx_order_status非聚族索引

综上所述，在更新操作时，我们应该尽量使用主键来更新表字段，这样可以有效避免一些不必要的死锁发生。

总结

数据库发生死锁的概率并不是很大，一旦遇到了，就一定要彻查具体原因，尽快找出解决方案，老实说，过程不简单。我们只有先对MySQL的InnoDB存储引擎有足够的了解，才能剖析出造成死锁的具体原因。

例如，以上我例举的两种发生死锁的场景，一个考验的是我们对锁算法的了解，另外一个考验则是对聚簇索引和辅助索引的熟悉程度。

解决死锁的最佳方式当然就是预防死锁的发生，我们平时编程中，可以通过以下一些常规手段来预防死锁的发生：

- 1.在编程中尽量按照固定的顺序来处理数据库记录，假设有两个更新操作，分别更新两条相同的记录，但更新顺序不一样，有可能导致死锁；
- 2.在允许幻读和不可重复读的情况下，尽量使用RC事务隔离级别，可以避免gap lock导致的死锁问题；
- 3.更新表时，尽量使用主键更新；
- 4.避免长事务，尽量将长事务拆解，可以降低与其它事务发生冲突的概率；
- 5.设置锁等待超时参数，我们可以通过innodb_lock_wait_timeout设置合理的等待超时阈值，特别是在一些高并发的业务中，我们可以尽量将该值设置得小一些，避免大量事务等待，占用系统资源，造成严重的性能开销。

思考题

除了设置 innodb_lock_wait_timeout 参数来避免已经产生死锁的SQL长时间等待，你还知道其它方法来解决类似问题吗？

期待在留言区看到你的见解。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请

他一起讨论。

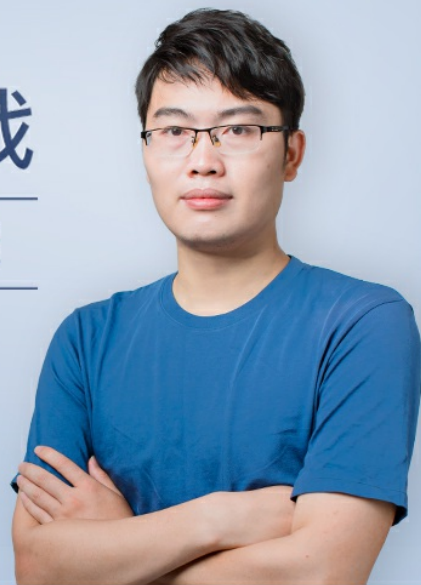


Java 性能调优实战

覆盖 80% 以上 Java 应用调优场景

刘超

金山软件西山居技术经理



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言



张学磊

4

MySQL默认开启了死锁检测机制，当检测到死锁后会选择一个最小(锁定资源最少得事务)的事务进行回滚

2019-08-13

作者回复

这个回答是我想要的。

Innodb提供了wait-for graph算法来主动进行死锁检测，我们可以通过innodb_deadlock_detect = on 打开死锁检测。

2019-08-15



我已经设置了昵称

2

老师。我们一般不会对查询加上for update，我们的组长让我们事务中不要放查询语句，只能放插入或者更新，就是提前查好，组装好，然后开始执行事务。我觉得这其实会出现重复插入（并发量一高就会出现）。请问老师事务中真的不能做查询操作吗，还有查询的时候怎么防止同时两个事务查不到相对应的数据而造成重复插入

2019-08-13

作者回复

for update是一种悲观锁实现，我们可以使用性能更好的乐观锁来实现，通过版本号来实现数据更新不丢失问题，这种方式是最佳选择。

而对于插入时防重复问题，可以对不允许重复字段设置唯一索引，进行唯一约束，这是一种不友好的实现方式。

2019-08-13



a、

👍 2

- 1.设置Transaction的超时时间
- 2.设置Transaction的级别为串行化级别

2019-08-13



新世界

👍 0

for update和**update**都是悲观锁有什么区别？

2019-08-18



K

👍 0

老师您好，麻烦问一下：“如果使用辅助索引来更新数据库，就需要使用聚簇索引来更新数据库...”这句话是什么意思啊？

2019-08-18

作者回复

如果我们之前使用辅助索引来更新数据库，就需要修改为使用聚簇索引来更新数据库。

2019-08-20



星星滴蓝天

👍 0

我们有一个很悲催的经历，更新的时候没有使用主键更新，之前还好好的，后来（服务迁移、降低配置.....处理流程变长）很悲剧的死锁了。

2019-08-17

作者回复

过来人

2019-08-20



Tomcat

👍 0

老师，使用**ON DUPLICATE KEY UPDATE**这个是不是也可以解决这种呢

2019-08-16



ok

👍 0

老师，请问事例中**insert order_record**的事务AB中，请解答下疑惑，我描述如下

- 1、事务A执行**select 4 for update**获取（4,+∞）间隙锁
- 2、图中B事务再执行**select 5 for update**获取（5,+∞）的间隙锁
- 3、事务A执行**insert 4**发现事务A自己持有（4,+∞）间隙锁，所以不用等待呀！
- 4、事务B执行**insert 5**发现事务A没有commit，持有（4,+∞）间隙锁，所以等待事务A释放锁
- 5、事务A提交，事务B **insert 5**获取到锁，commit

请指出问题...

2019-08-16



奇奇

👍 0

mysql本身自带死锁检测

超时时间在业务上是很难接受的

2019-08-15

作者回复

是的

2019-08-16



Fever

👍 0

兼容矩阵图里的Gap锁和横向的Insert Intention不兼容吧。。

2019-08-13

作者回复

是的，跟竖向是一样的

2019-08-13



陈华应

👍 0

for update基本不被允许使用，除非经过review以及测试不会有死锁风险

对于锁机制，要很好的了解索引数据结构才能明白锁导致的奇怪现象是怎么出现的

2019-08-13

作者回复

对的

2019-08-15