MySQL发生死锁有哪些原因，你又是怎么避免的

PHP开源社区 2022-10-06 18:00 发表于湖南

PHP开源社区
免费提供10GPHP进阶架构师学习教程，关注即可免费领取。每天推送PHP最新资讯技...
83篇原创内容

公众号

一、MySQL锁类型和加锁分析

锁类型介绍：

MySQL 有三种锁的级别：页级、表级、行级。

- 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。
- 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。
- 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

算法：

- next KeyLocks 锁，同时锁住记录 (数据)，并且锁住记录前面的 Gap
- Gap 锁，不锁记录，仅仅记录前面的 Gap
- Recordlock 锁（锁数据，不锁 Gap）
- 所以其实 Next-KeyLocks=Gap 锁 + Recordlock 锁

二、死锁产生原因和示例

1、产生原因：

所谓死锁 <DeadLock>：是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去. 此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。表级锁不会产生死锁. 所以解决死锁主要还是针对于最常用的 InnoDB。

死锁的关键在于：两个 (或以上) 的 Session 加锁的顺序不一致。

那么对应的解决死锁问题的关键就是：让不同的 session 加锁有次序

2、产生示例：

案例一

需求：将投资的钱拆成几份随机分配给借款人。

起初业务程序思路是这样的：

投资人投资后，将金额随机分为几份，然后随机从借款人表里面选几个，然后通过一条条 `select for update` 去更新借款人表里面的余额等。

例如两个用户同时投资，A 用户金额随机分为 2 份，分给借款人 1，2

B 用户金额随机分为 2 份，分给借款人 2，1

由于加锁的顺序不一样，死锁当然很快就出现了。

对于这个问题的改进很简单，直接把所有分配到的借款人直接一次锁住就行了。

```
Select * from xxx where id in (xx,xx,xx) for update
```

在 in 里面的列表值 mysql 是会自动从小到大排序，加锁也是一条条从小到大加的锁

例如（以下会话 id 为主键）：

Session1:

```
mysql> select * from t3 where id in (8,9) for update;
+----+--------+------+---------------------+
| id | course | name | ctime               |
+----+--------+------+---------------------+
|  8 | WA     | f    | 2016-03-02 11:36:30 |
|  9 | JX     | f    | 2016-03-01 11:36:30 |
+----+--------+------+---------------------+
rows in set (0.04 sec)
```

Session2:

```
select * from t3 where id in (10,8,5) for update;
```

锁等待中……

其实这个时候 id=10 这条记录没有被锁住的，但 id=5 的记录已经被锁住了，锁的等待在 id=8 的这里 不信请看

Session3:

```
mysql> select * from t3 where id=5 for update;
```

锁等待中

Session4:

```
mysql> select * from t3 where id=10 for update;
+----+--------+------+---------------------+
| id | course | name | ctime               |
+----+--------+------+---------------------+
| 10 | JB     | g    | 2016-03-10 11:45:05 |
+----+--------+------+---------------------+
row in set (0.00 sec)
```

在其它 session 中 id=5 是加不了锁的，但是 id=10 是可以加上锁的。

案例二

在开发中，经常会做这类的判断需求：根据字段值查询（有索引），如果不存在，则插入；否则更新。

以 id 为主键为例，目前还没有 id=22 的行

Session1:

```
select * from t3 where id=22 for update;
Empty set (0.00 sec)
```

session2:

```
select * from t3 where id=23  for update;
Empty set (0.00 sec)
```

Session1:

```
insert into t3 values(22,'ac','a',now());
```

锁等待中……

Session2:

```
insert into t3 values(23,'bc','b',now());
```

```
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
```

当对存在的行进行锁的时候 (主键)，mysql 就只有行锁。

当对未存在的行进行锁的时候 (即使条件为主键)，mysql 是会锁住一段范围（有 gap 锁）

锁住的范围为：

(无穷小或小于表中锁住 id 的最大值，无穷大或大于表中锁住 id 的最小值)

如：如果表中目前有已有的 id 为（11， 12），那么就锁住（12，无穷大）

如果表中目前已有的 id 为（11， 30），那么就锁住（11，30）

对于这种死锁的解决办法是：

```
insert into t3(xx,xx) on duplicate key update `xx`='XX';
```

用 mysql 特有的语法来解决此问题。因为 insert 语句对于主键来说，插入的行不管有没有存在，都会只有行锁。

案例三

```
mysql> select * from t3 where id=9 for update;
+----+--------+------+---------------------+
| id | course | name | ctime               |
+----+--------+------+---------------------+
|  9 | JX     | f    | 2016-03-01 11:36:30 |
+----+--------+------+---------------------+

row in set (0.00 sec)
```

Session2:

```
mysql> select * from t3 where id<20 for update;
```

锁等待中

Session1:

```
mysql> insert into t3 values(7,'ae','a',now());
```

```
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
```

这个跟案例一其它是差不多的情况，只是 session1 不按常理出牌了，

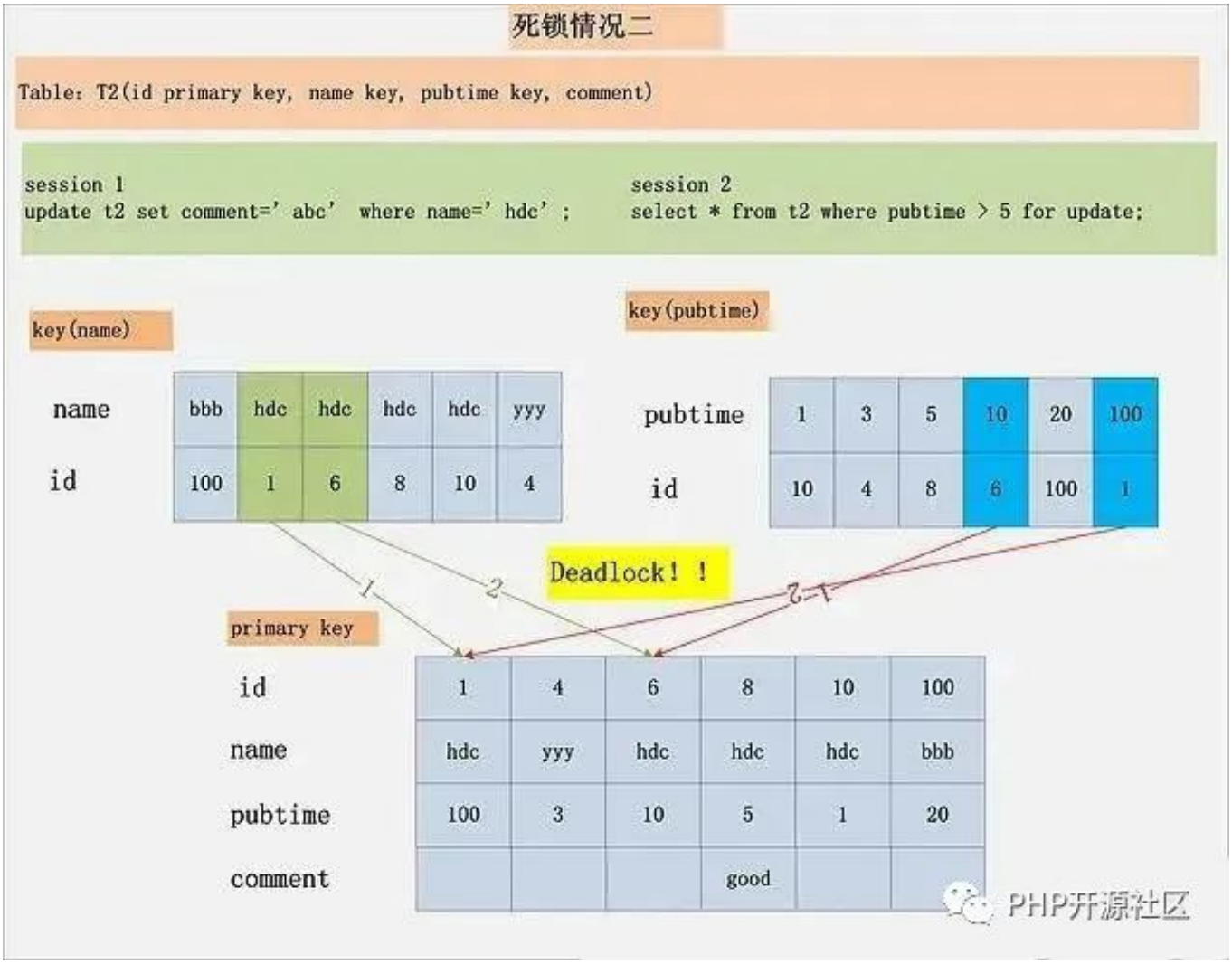
Session2 在等待 Session1 的 id=9 的锁，session2 又持了 1 到 8 的锁（注意 9 到 19 的范围并没有被 session2 锁住），最后，session1 在插入新行时又得等待 session2, 故死锁发生了。

这种一般是在业务需求中基本不会出现，因为你锁住了 id=9，却又想插入 id=7 的行，这就有点跳了，当然肯定也有解决的方法，那就是重理业务需求，避免这样的写法。

案例四

一般的情况，两个 session 分别通过一个 sql 持有一把锁，然后互相访问对方加锁的数据产生死锁。

案例五



两个单条的 sql 语句涉及到的加锁数据相同，但是加锁顺序不同，导致了死锁。

案例六

死锁场景如下：

表结构：

```
CREATE TABLE dltask (
    id bigint unsigned NOT NULL AUTO_INCREMENT COMMENT 'auto id',
    a varchar(30) NOT NULL COMMENT 'uniq.a',
    b varchar(30) NOT NULL COMMENT 'uniq.b',
    c varchar(30) NOT NULL COMMENT 'uniq.c',
    x varchar(30) NOT NULL COMMENT 'data',
    PRIMARY KEY (id),
    UNIQUE KEY uniq_a_b_c (a, b, c)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='deadlock test';
```

a，b，c 三列，组合成一个唯一索引，主键索引为 id 列。

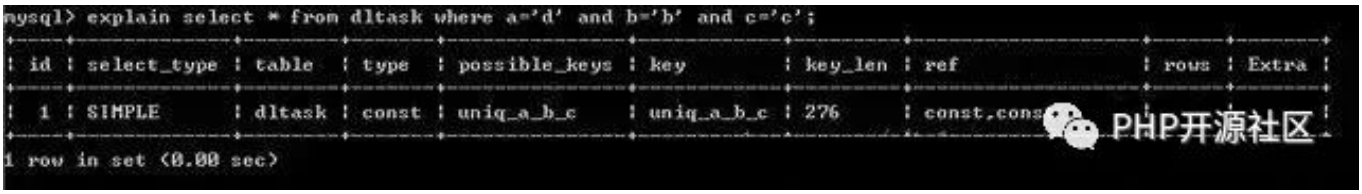
事务隔离级别：

RR (Repeatable Read)

每个事务只有一条 SQL：

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

SQL 的执行计划：



```
mysql> explain select * from dltask where a='d' and b='b' and c='c';
+----+-------------+--------+-------+---------------+-----------+---------+-------------------+------+-------+
| id | select_type | table  | type  | possible_keys | key       | key_len | ref               | rows | Extra |
+----+-------------+--------+-------+---------------+-----------+---------+-------------------+------+-------+
|  1 | SIMPLE      | dltask | const | uniq_a_b_c    | uniq_a_b_c | 276     | const,const,const |    1 |       |
+----+-------------+--------+-------+---------------+-----------+---------+-------------------+------+-------+
1 row in set (0.00 sec)
```


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)
```

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：

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。

- 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)；

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：

- 根据给定的查询条件，找到对应的记录所在页面；
- 对页面加上 X 锁 (RWLock)，然后在页面内寻找满足条件的记录；
- 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)；

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。

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。

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。

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，事半而功倍。

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：

```
MySQL/InnoDB, Repeatable Read

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
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)；		
		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；
3. 进行删除操作，将索引上的记录标识为删除状态；		
4 事务0提交，释放记录锁；		
		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；
	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	

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。

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)

▲扫描二维码-免费领取

点击“查看原文”获取更多

阅读原文

喜欢此内容的人还喜欢

Redis碎片整理原理解析与实践

Redis开发运维实战

一款Linux、数据库、Redis、MongoDB统一管理平台，有点牛逼了！

开源前线

盘点Python网络爬虫过程中xpath的联合查询定位一个案例

Python爬虫与数据挖掘