慎用，Mybatis-Plus这个方法可能导致死锁

Original 谢星 转转技术 2023-06-14 08:30 Posted on 广东

收录于合集

#Mybatis-Plus 1 #数据库 3 #死锁 1

Scan to Follow

1 场景还原

1.1 版本信息

```
MySQL版本：5.6.36-82.1-log
Mybatis-Plus的starter版本：3.3.2
存储引擎：InnoDB
```

1.2 死锁现象

> A同学在**生产环境**使用了Mybatis-Plus提供的**com.baomidou.mybatisplus.extension.service.IService#saveOrUpdate(T, com.baomidou.mybatisplus.core.conditions.Wrapper)** 方法（以下简称**B方法**），并发场景下，数据库报了如下错误

```
get lock; try restarting transaction
; Deadlock found when trying to get lock; try restarting transaction; nested exception is java.sql.BatchUpdateException: Deadlock f
ound when trying to get lock; try restarting transaction
For more information, please visit the url, http://docs.aliyun.com/cn#/pub/ons/faq/exceptions&unexpected_exception
        at com.bj58.zhuanzhuan.zzmq.spring.listener.AbstractMessageListener.doConsume(AbstractMessageListener.java:50) ~[rocketmq-s
pring-boot-starter-3.6.7.jar!/:?]
        at com.bj58.zhuanzhuan.zzmq.spring.listener.AutoConcurrentlyListener.consumeMessage(AutoConcurrentlyListener.java:22) ~[roc
ketmq-spring-boot-starter-3.6.7.jar!/:?]
        at com.alibaba.rocketmq.client.impl.consumer.ConsumeMessageConcurrentlyService$ConsumeRequest.run(ConsumeMessageConcurrentl
yService.java:146) [rocketmq-client-3.6.9.jar!/:?]
        at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511) [?:1.8.0_191]
        at java.util.concurrent.FutureTask.run(FutureTask.java:266) [?:1.8.0_191]
        at com.alibaba.ttl.TtlRunnable.run(TtlRunnable.java:55) [?:?]
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149) [?:1.8.0_191]
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624) [?:1.8.0_191]
        at java.lang.Thread.run(Thread.java:748) [?:1.8.0_191]
```

转转技术

---

2 为什么是间隙锁死锁？

> 如上图示，数据库报了死锁，那死锁场景千万种，为什么确定B方法是由于间隙锁导致的死锁？

2.1 什么是死锁？

两个事务互相等待对方持有的锁，导致互相阻塞，从而导致死锁。

2.2 什么是间隙锁？

- 间隙锁是MySQL行锁的一种，与Record lock不同的是间隙锁锁定的是一个间隙。
- 锁定规则如下：

MySQL会向左找第一个比当前索引值小的值，向右找第一个比当前索引值大 的值（没有则为正无穷），将此区间锁住，从而阻止其他事务在此区间插入数据。

2.3 MySQL为什么要引入间隙锁？

与Record lock组合成Next-key lock，在可重复读这种隔离级别下一起工作避免幻读。

2.4 间隙锁死锁分析

理论上一款开源的框架，经过了多年打磨，提供的方法不应该造成如此严重的错误，但理论仅仅是理论上，事实就是发生了死锁，于是我们开始了一轮深度排查。首先我们从这个方法的源码入手，源码如下：

```
    default boolean saveOrUpdate(T entity, Wrapper<T> updateWrapper) {
        return this.update(entity, updateWrapper) || this.saveOrUpdate(entity);
    }
```

从源码上看此方法就没有按套路出牌，正常逻辑应该是首先执行查询，存在则修改，不存在则新增，但此方法上来就执行了修改。我们就猜想是不是MySQL在修改时增加了什么锁导致了死锁，于是我们找到了DBA获取了最新的死锁日志，即执行**show engine innodb status**，我们发现了两项关键信息如下：

```
*** (1) TRANSACTION:
...省略日志
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 347 n bits 80 index `PRIMARY` of table `database_name`.`table_name` 

*** (2) TRANSACTION:
...省略日志
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 347 n bits 80 index `PRIMARY` of table `database_name`.`table_name` 
```

简单翻译一下，就是事务一在获取插入意向锁时，需要等待间隙锁（事务二添加）释放，同时事务二在获取插入意向锁时，也在等待间隙锁释放（事务一添加），**（本文不讨论MySQL在修改与插入时添加的锁，我们把修改时添加间隙锁，插入时获取插入意向锁为已知条件）**那我们回到B方法，并发场景下，是不是就很大几率会满足事务一和事务二相互等待对方持有的间隙锁，从而导致死锁。

现在我们理论有了，我们现在用真实数据来验证此场景。

2.5 验证间隙锁死锁

- 准备如下表结构（以下简称验证一）

```
create table t_gap_lock(
id int auto_increment primary key comment '主键ID',
name varchar(64) not null comment '名称',
age int not null comment '年龄'
) comment '间隙锁测试表';
```

- 准备如下表数据

```
mysql> select * from t_gap_lock;
+----+------+-----+
| id | name | age |
+----+------+-----+
|  1 | 张三 |  18 |
|  5 | 李四 |  19 |
|  6 | 王五 |  20 |
|  9 | 赵六 |  21 |
| 12 | 孙七 |  22 |
+----+------+-----+
```

- 我们开启事务一，并执行如下语句，**注意这个时候我们还没有提交事务**

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> update t_gap_lock t set t.age = 25 where t.id = 4;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

- 同时我们开启事务二，并执行如下语句，**事务二我们同样不提交事务**

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> update t_gap_lock t set t.age = 25 where t.id = 7;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

- 接下来我们在事务一中执行如下语句

```
mysql> insert into t_gap_lock(id, name, age) value (7,'间隙锁7',27);
```

- 我们会发现事务一被阻塞了，然后我们执行以下语句看下当前正在锁的事务。

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS \G;
*************************** 1. row ***************************
    lock_id: 749:0:360:3
lock_trx_id: 749
  lock_mode: X,GAP
  lock_type: RECORD
 lock_table: `test`.`t_gap_lock`
 lock_index: `PRIMARY`
 lock_space: 0
  lock_page: 360
   lock_rec: 3
  lock_data: 5
*************************** 2. row ***************************
    lock_id: 74A:0:360:3
lock_trx_id: 74A
  lock_mode: X,GAP
  lock_type: RECORD
 lock_table: `test`.`t_gap_lock`
 lock_index: `PRIMARY`
 lock_space: 0
  lock_page: 360
   lock_rec: 3
  lock_data: 5
2 rows in set (0.00 sec)
```

根据lock_type和lock_mode我们可以很清晰的看到锁类型是行锁，锁模式是间隙锁。

- 与此同时我们在事务二中执行如下语句

```
insert into t_gap_lock(id, name, age) value (4,'间隙锁4',24);
```

- 一执行以上语句，数据库就立马报了死锁，并且回滚了事务二（可以在死锁日志中看到*** WE ROLL BACK TRANSACTION (2)）

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

到这里，细心的同学就会发现，诶，你这上面故意造了一个间隙，并且让两个事务分别在对方的间隙中插入数据，太刻意了，生产环境基本上不会有这种场景，是的，生产环境怎么会有这种场景呢，上面的数据只是为了让大家直观的看到间隙锁的死锁过程，接下来那我们再来一组数据，我们简称验证二。

- 我们还是以验证一的表结构与数据，我们来执行这样一个操作。首先我们开始开启事务一并且执行如下操作，依然不提交事务

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> update t_gap_lock t set t.age = 25 where t.id = 4;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

- 同时我们开启事务二，执行与事务一一样的操作，我们会惊奇的发现，竟然也成功了。

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> update t_gap_lock t set t.age = 25 where t.id = 4;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

- 于是乎我们在事务一执行如下操作，我们又惊奇的发现事务一被阻塞了。

```
insert into t_gap_lock(id, name, age) value (4,'间隙锁4',24);
```

- 在事务一被阻塞的同时，我们在事务二执行同样的语句，我们发现数据库立马就报了死锁。

```
insert into t_gap_lock(id, name, age) value (4,'间隙锁4',24);
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
```

验证二完整的复现了线上死锁的过程，也就是事务一先执行了更新语句，事务二在同一时刻也执行了更新语句，然后事务一发现没有更新到就去执行主键查询语句，发现确实没有，所以执行了插入语句，但是插入要先获取插入意向锁，在获取插入意向锁的时候发现这个间隙已经被事务二加锁了，所以事务一开始等待事务二释放间隙锁，同理，事务二也执行上述操作，最终导致事务一与事务二互相等待对方释放间隙锁，最终导致死锁。

验证二还说明了一个问题，就是间隙锁加锁是非互斥的，也就是事务一对间隙A加锁后，事务二依然可以给间隙A加锁。

3 如何解决？

3.1 关闭间隙锁（不推荐）

- 降低隔离级别，例如降为提交读。
- 直接修改my.cnf，将开关，innodb_locks_unsafe_for_binlog改为1，默认为0即开启

PS：以上方法仅适用于当前业务场景确实不关心幻读的问题。

3.2 自定义saveOrUpdate方法（推荐）

建议**自己编写**一个saveOrUpdate方法，当然也可以直接采用Mybatis-Plus提供的saveOrUpdate方法，但是根据源码发现，会有很多额外的反射操作，并且还添加了事务，大家都知道，MySQL单表操作完全不需要开事务，会增加额外的开销。

```
@Transactional(
      rollbackFor = {Exception.class}
  )
  public boolean saveOrUpdate(T entity) {
      if (null == entity) {
          return false;
      } else {
          Class<?> cls = entity.getClass();
          TableInfo tableInfo = TableInfoHelper.getTableInfo(cls);
          Assert.notNull(tableInfo, "error: can not execute. because can not find cache of TableIn
          String keyProperty = tableInfo.getKeyProperty();
          Assert.notEmpty(keyProperty, "error: can not execute. because can not find column for id
          Object idVal = ReflectionKit.getFieldValue(entity, tableInfo.getKeyProperty());
          return !StringUtils.checkValNull(idVal) && !Objects.isNull(this.getById((Serializable)id
      }
  }
```

4 拓展

4.1 如果两个事务同时修改存在的行会发生什么？

在验证二中两个事务修改的都是不存在的行，都能加间隙锁成功，那如果两个事务修改的是存在的行，MySQL还会加间隙锁吗？或者说把间隙锁从锁间隙降为锁一行？带着疑问，我们执行以下数据验证，我们还是使用验证一的表和数据。

- 首先我们开启事务一执行以下语句

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> update t_gap_lock t set t.age = 25 where t.id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

- 我们再开启事务二，执行同样的语句，发现事务二已经被阻塞

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> update t_gap_lock t set t.age = 25 where t.id = 1;
```

- 这个时候我们执行以下语句看下当前正在锁的事务。

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS \G;
*************************** 1. row ***************************
    lock_id: 75C:0:360:2
lock_trx_id: 75C
  lock_mode: X
  lock_type: RECORD
 lock_table: `test`.`t_gap_lock`
 lock_index: `PRIMARY`
 lock_space: 0
  lock_page: 360
   lock_rec: 2
  lock_data: 1
*************************** 2. row ***************************
    lock_id: 75B:0:360:2
lock_trx_id: 75B
  lock_mode: X
  lock_type: RECORD
 lock_table: `test`.`t_gap_lock`
 lock_index: `PRIMARY`
 lock_space: 0
  lock_page: 360
   lock_rec: 2
  lock_data: 1
2 rows in set (0.00 sec)
```

根据lock_type和lock_mode我们看到事务一和二加的锁变成了Record Lock，并没有再添加间隙锁，根据以上数据验证MySQL在修改存在的数据时会给行加上Record Lock，与间隙锁不同的是该锁是互斥的，即不同的事务不能同时对同一行记录添加Record Lock。

5 结语

虽然Mybatis-Plus提供的这个方法可能会造成死锁，但是依然不可否认它是一款非常优秀的增强框架，其提供的lambda写法在日常工作中极大的提高了我们的开发效率，所以凡事都有两面性，我们应该秉承辩证的态度，熟悉的方法尝试用，陌生的方法谨慎用。

以上就是我们在生产环境间隙锁死锁分析的全过程，如果大家觉得本文让你对间隙锁，以及间隙锁死锁有一点的了解，别忘记一键三连，多多支持转转技术，转转技术在未来将会给大家带来更多的生产实践与探索。

作者简介

> 谢星，转转金融技术部后端工程师。热爱编程，热爱分享，拥抱开源。

想了解更多转转公司的业务实践，欢迎点击关注下方公众号：

大转转FE
定期分享一些团队对前端的想法与沉淀
340篇原创内容
公众号

转转QA
知识分享时代， 带你从新视角认识「转转QA」.
151篇原创内容
公众号

People who liked this content also liked

规则引擎技术在转转钱包的实践
转转技术

转转商品到手价设计
转转技术

All in ECP，转转一站式ES数据清洗解决方案
转转技术