**[mysql死锁问题分析](http://www.cnblogs.com/LBSer/p/5183300.html)**

　　线上某服务时不时报出如下异常（大约一天二十多次）：“Deadlock found when trying to get lock;”。

      Oh, My God! 是死锁问题。尽管报错不多，对性能目前看来也无太大影响，但还是需要解决，保不齐哪天成为性能瓶颈。     为了更系统的分析问题，本文将从死锁检测、索引隔离级别与锁的关系、死锁成因、问题定位这五个方面来展开讨论。

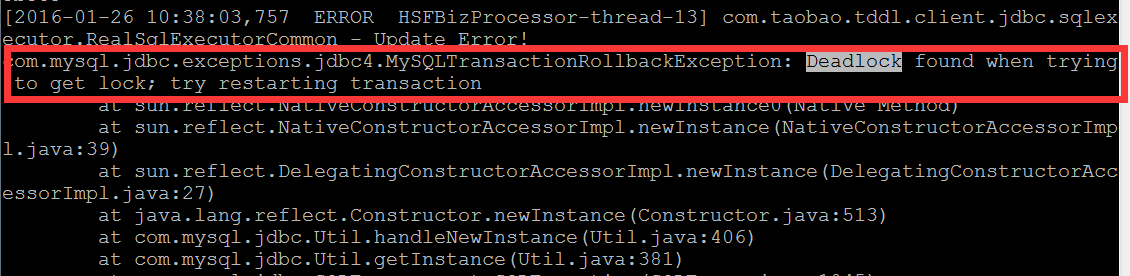


图1 应用日志

## **1 死锁是怎么被发现的？**

## **1.1 死锁成因&&检测方法**

     左图那两辆车造成死锁了吗？不是！右图四辆车造成死锁了吗？是！

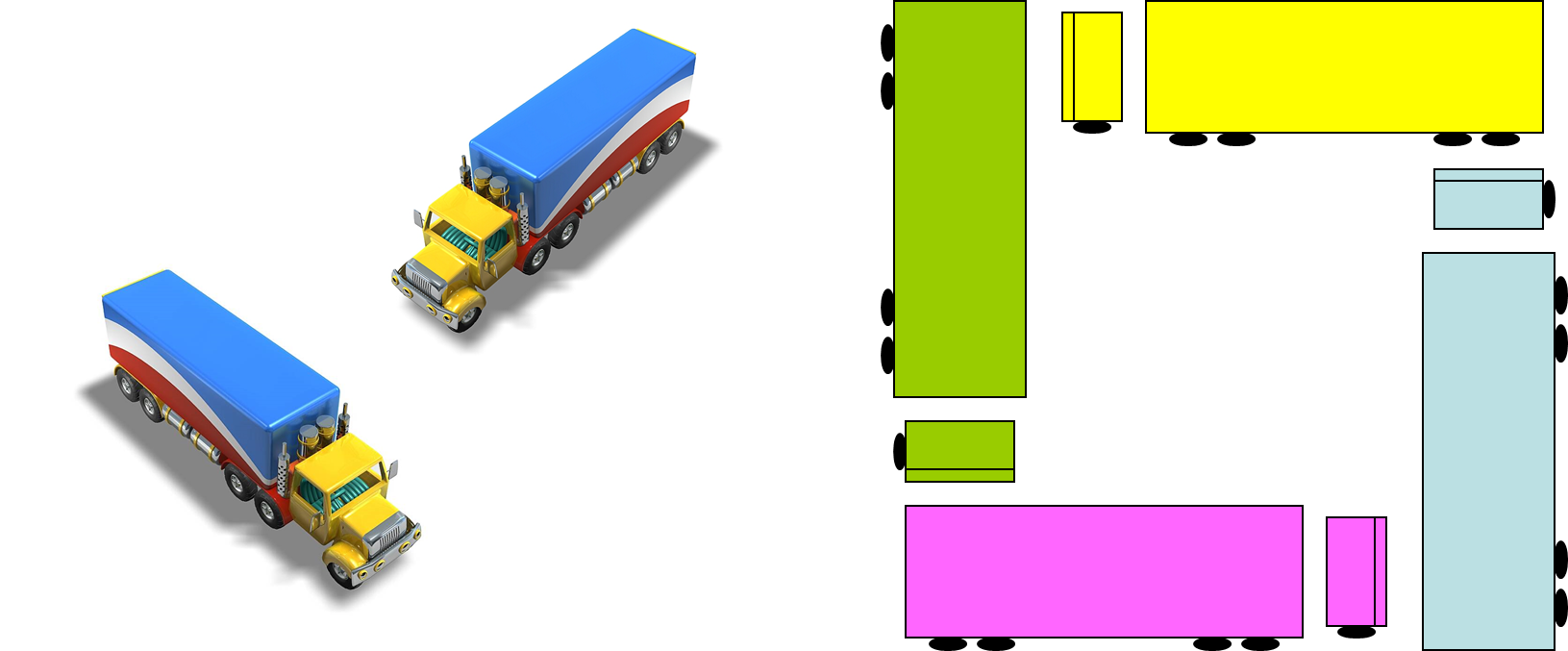


                                                                      图2 死锁描述

      我们mysql用的存储引擎是innodb，从日志来看，innodb主动探知到死锁，并回滚了某一苦苦等待的事务。问题来了，innodb是怎么探知死锁的？

     直观方法是在两个事务相互等待时，当一个等待时间超过设置的某一阀值时，对其中一个事务进行回滚，另一个事务就能继续执行。这种方法简单有效，在innodb中，参数innodb\_lock\_wait\_timeout用来设置超时时间。

     仅用上述方法来检测死锁太过被动，innodb还提供了wait-for graph算法来主动进行死锁检测，每当加锁请求无法立即满足需要并进入等待时，wait-for graph算法都会被触发。

### **1.2 wait-for graph原理**

     我们怎么知道上图中四辆车是死锁的？他们相互等待对方的资源，而且形成环路！我们将每辆车看为一个节点，当节点1需要等待节点2的资源时，就生成一条有向边指向节点2，最后形成一个有向图。我们只要检测这个有向图是否出现环路即可，出现环路就是死锁！这就是wait-for graph算法。

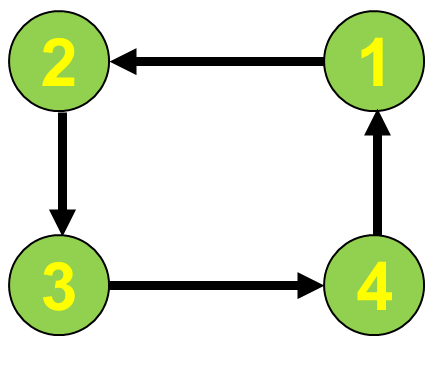


                                                                                            图3 wait for graph

     innodb将各个事务看为一个个节点，资源就是各个事务占用的锁，当事务1需要等待事务2的锁时，就生成一条有向边从1指向2，最后行成一个有向图。

## **1.2 innodb隔离级别、索引与锁**

      死锁检测是死锁发生时innodb给我们的救命稻草，我们需要它，但我们更需要的是避免死锁发生的能力，如何尽可能避免？这需要了解innodb中的锁。

### **1.2.1 锁与索引的关系**

       假设我们有一张消息表（msg），里面有3个字段。假设id是主键，token是非唯一索引，message没有索引。

|  |  |  |
| --- | --- | --- |
| id: bigint | token: varchar(30) | message: varchar(4096) |

     innodb对于主键使用了聚簇索引，这是一种数据存储方式，表数据是和主键一起存储，主键索引的叶结点存储行数据。对于普通索引，其叶子节点存储的是主键值。

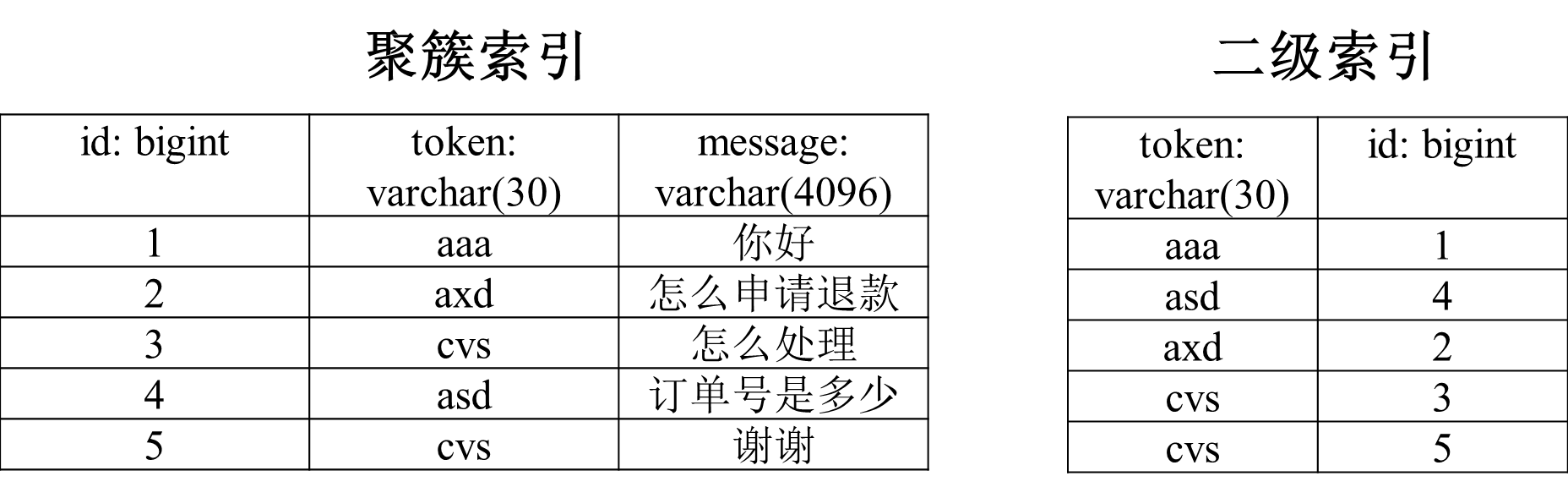


                                                                                 图4 聚簇索引和二级索引

     下面分析下索引和锁的关系。

1）delete from msg where id=2；

     由于id是主键，因此直接锁住整行记录即可。

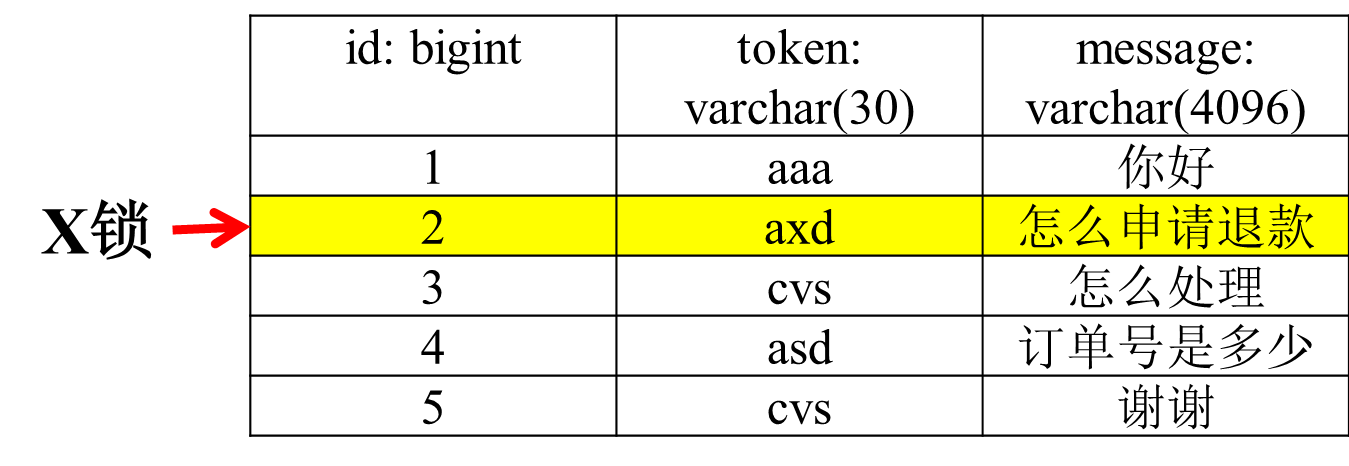


                                                                               图5

2）delete from msg where token=’ cvs’;

    由于token是二级索引，因此首先锁住二级索引（两行），接着会锁住相应主键所对应的记录；

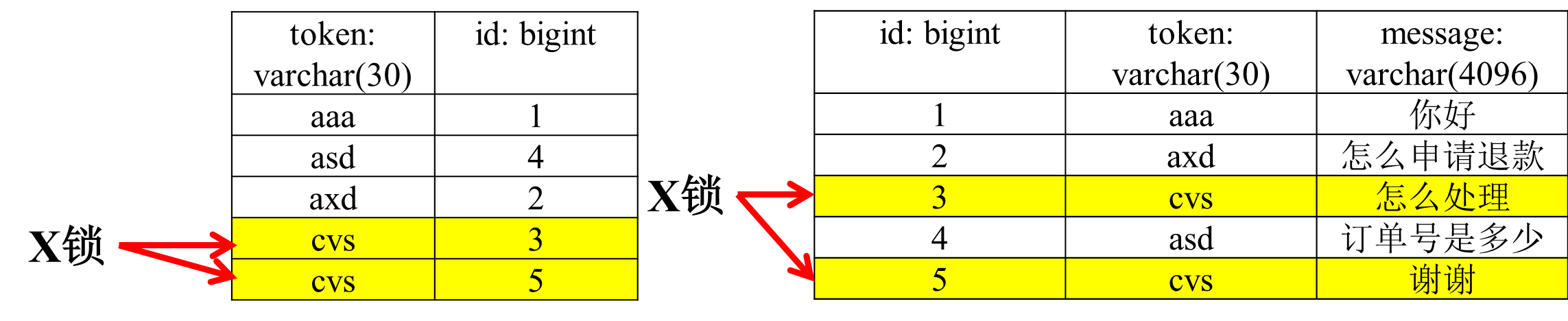


                                                                       图6

3）delete from msg where message=订单号是多少’；

     message没有索引，所以走的是全表扫描过滤。这时表上的各个记录都将添加上X锁。

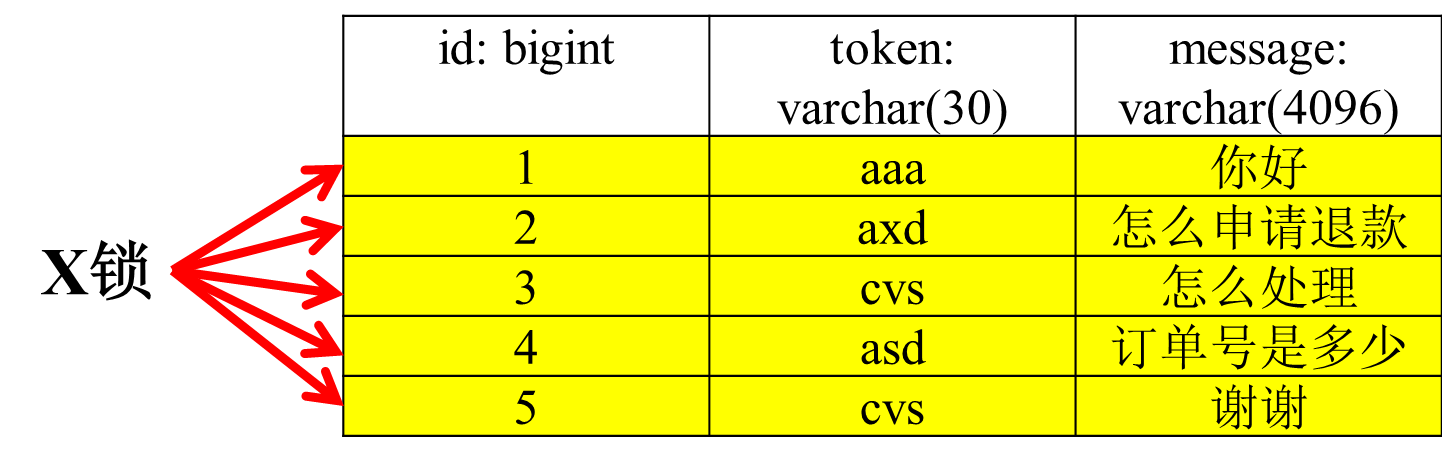


                                                                        图7

### **1.2.2 锁与隔离级别的关系**

     大学数据库原理都学过，为了保证并发操作数据的正确性，数据库都会有事务隔离级别的概念：1）未提交读（Read uncommitted）；2）已提交读（Read committed（RC））；3）可重复读（Repeatable read（RR））；4）可串行化（Serializable）。我们较常使用的是**RC和RR**。

     提交读(RC)：只能读取到已经提交的数据。

     可重复读(RR)：在同一个事务内的查询都是事务开始时刻一致的，InnoDB默认级别。

     我们在1.2.1节谈论的其实是RC隔离级别下的锁，它可以防止不同事务版本的数据修改提交时造成数据冲突的情况，但当别的事务插入数据时可能会出现问题。

       如下图所示，事务A在第一次查询时得到1条记录，在第二次执行相同查询时却得到两条记录。从事务A角度上看是见鬼了！这就是幻读，RC级别下尽管加了行锁，但还是避免不了幻读。

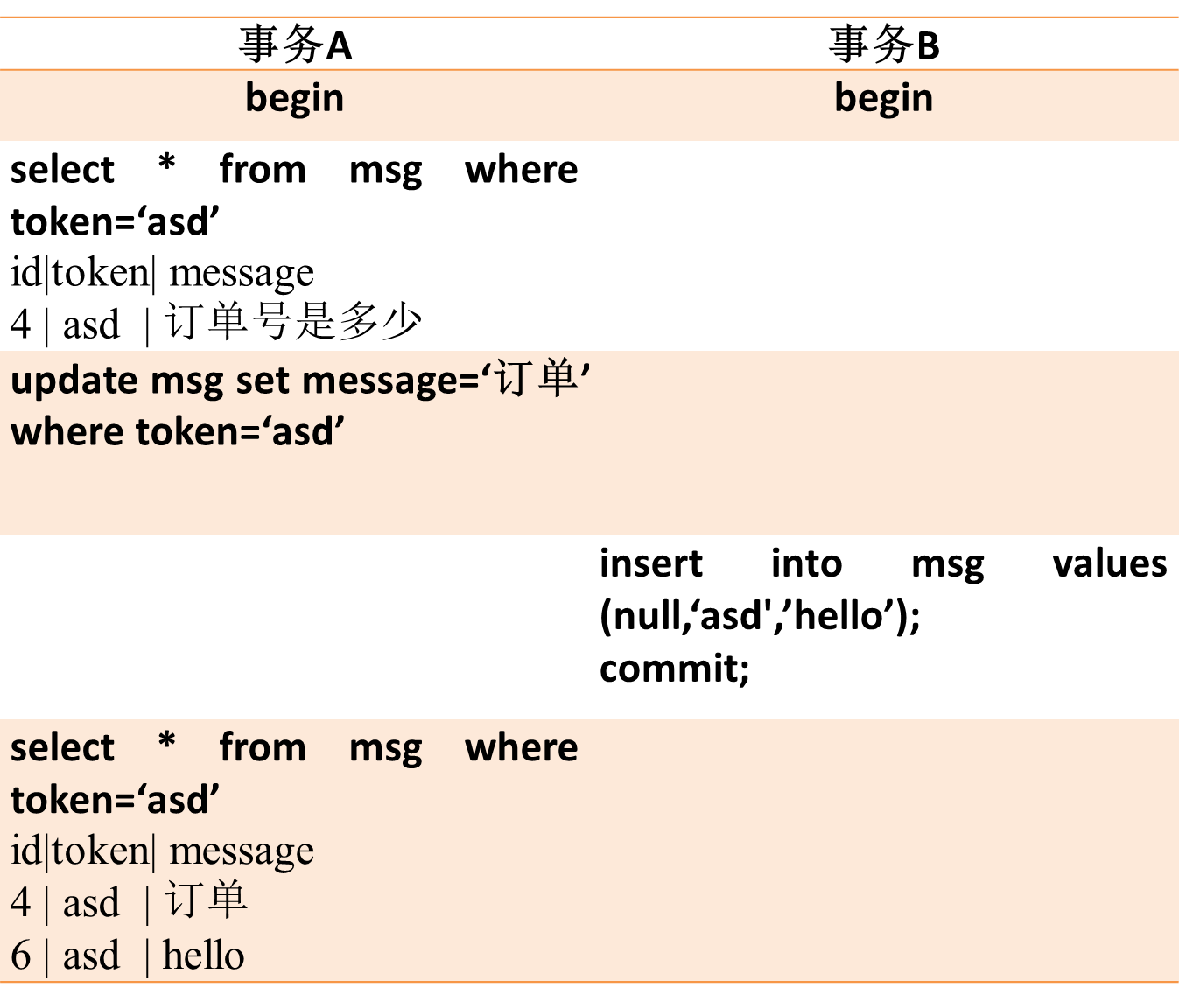


                                                                     图8

     innodb的RR隔离级别可以避免幻读发生，怎么实现？当然需要借助于锁了！

     为了解决幻读问题，innodb引入了gap锁。

      在事务A执行：update msg set message=‘订单’ where token=‘asd’;

      innodb首先会和RC级别一样，给索引上的记录添加上X锁，此外，还在非唯一索引’asd’与相邻两个索引的区间加上锁。

       这样，当事务B在执行insert into msg values (null,‘asd',’hello’); commit;时，会首先检查这个区间是否被锁上，如果被锁上，则不能立即执行，需要等待该gap锁被释放。这样就能避免幻读问题。

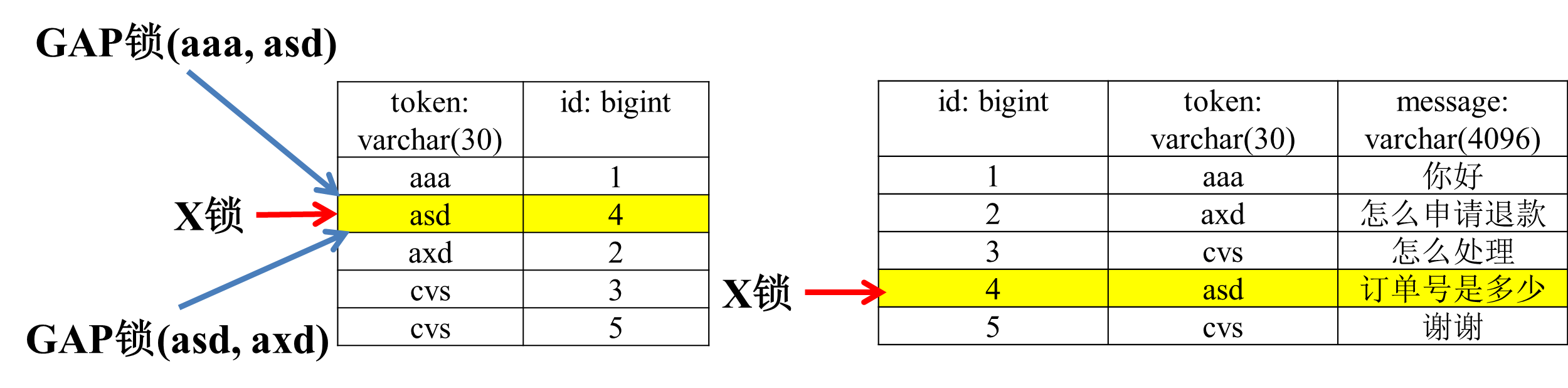


                                                                           图9

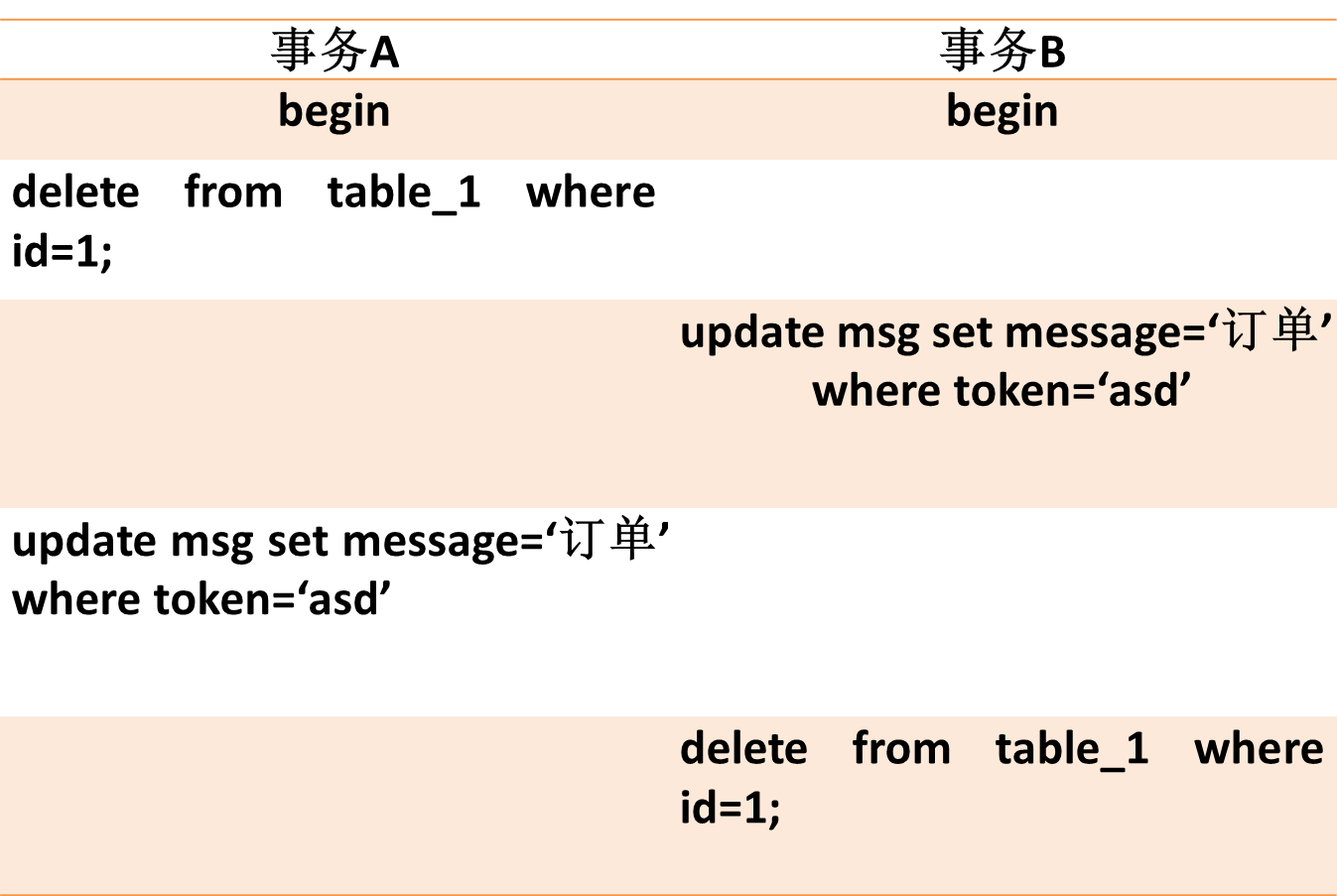
     推荐一篇好文，可以深入理解锁的原理：[http://hedengcheng.com/?p=771#\_Toc374698322](http://hedengcheng.com/?p=771" \l "_Toc374698322)

## **3 死锁成因**

     了解了innodb锁的基本原理后，下面分析下死锁的成因。如前面所说，死锁一般是事务相互等待对方资源，最后形成环路造成的。下面简单讲下造成相互等待最后形成环路的例子。

### **3.1不同表相同记录行锁冲突**

     这种情况很好理解，事务A和事务B操作两张表，但出现循环等待锁情况。



START TRANSACTION:

delete from student\_score where score\_id = 2;

update student set sex = '男' where id=3;

commit;

START TRANSACTION:

update student set sex = '男' where id=3;

delete from student\_score where score\_id = 2;

COMMIT;

                                                                       图10

### **3.2相同表记录行锁冲突**

     这种情况比较常见，之前遇到两个job在执行数据批量更新时，jobA处理的的id列表为[1,2,3,4]，而job处理的id列表为[8,9,10,4,2]，这样就造成了死锁。

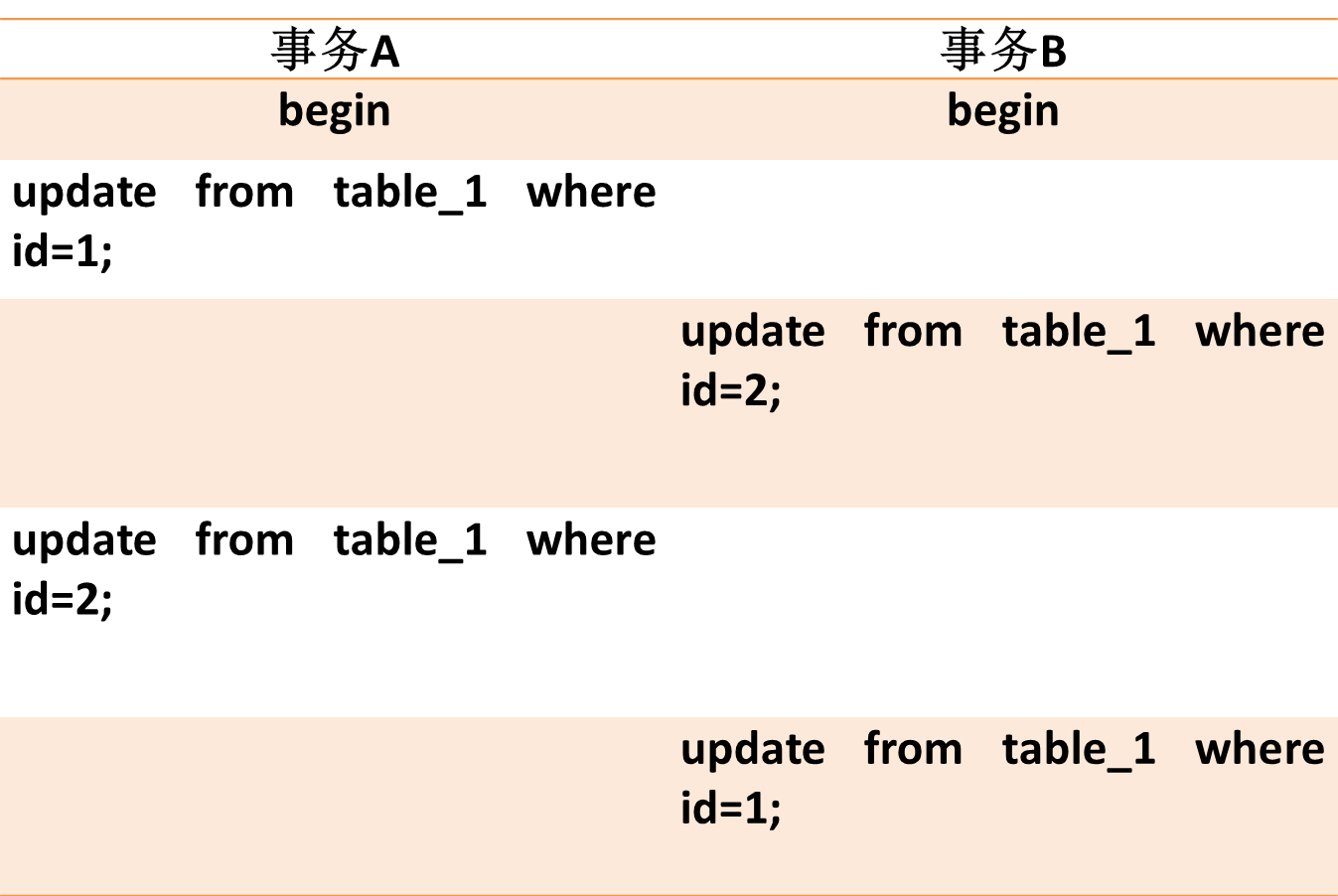


                                                                          图11

### **3.3不同索引锁冲突**

     这种情况比较隐晦，事务A在执行时，除了在二级索引加锁外，还会在聚簇索引上加锁，在聚簇索引上加锁的顺序是[1,4,2,3,5]，而事务B执行时，只在聚簇索引上加锁，加锁顺序是[1,2,3,4,5]，这样就造成了死锁的可能性。



                                                                          图12

### **3.4 gap锁冲突**

     innodb在RR级别下，如下的情况也会产生死锁，比较隐晦。不清楚的同学可以自行根据上节的gap锁原理分析下。

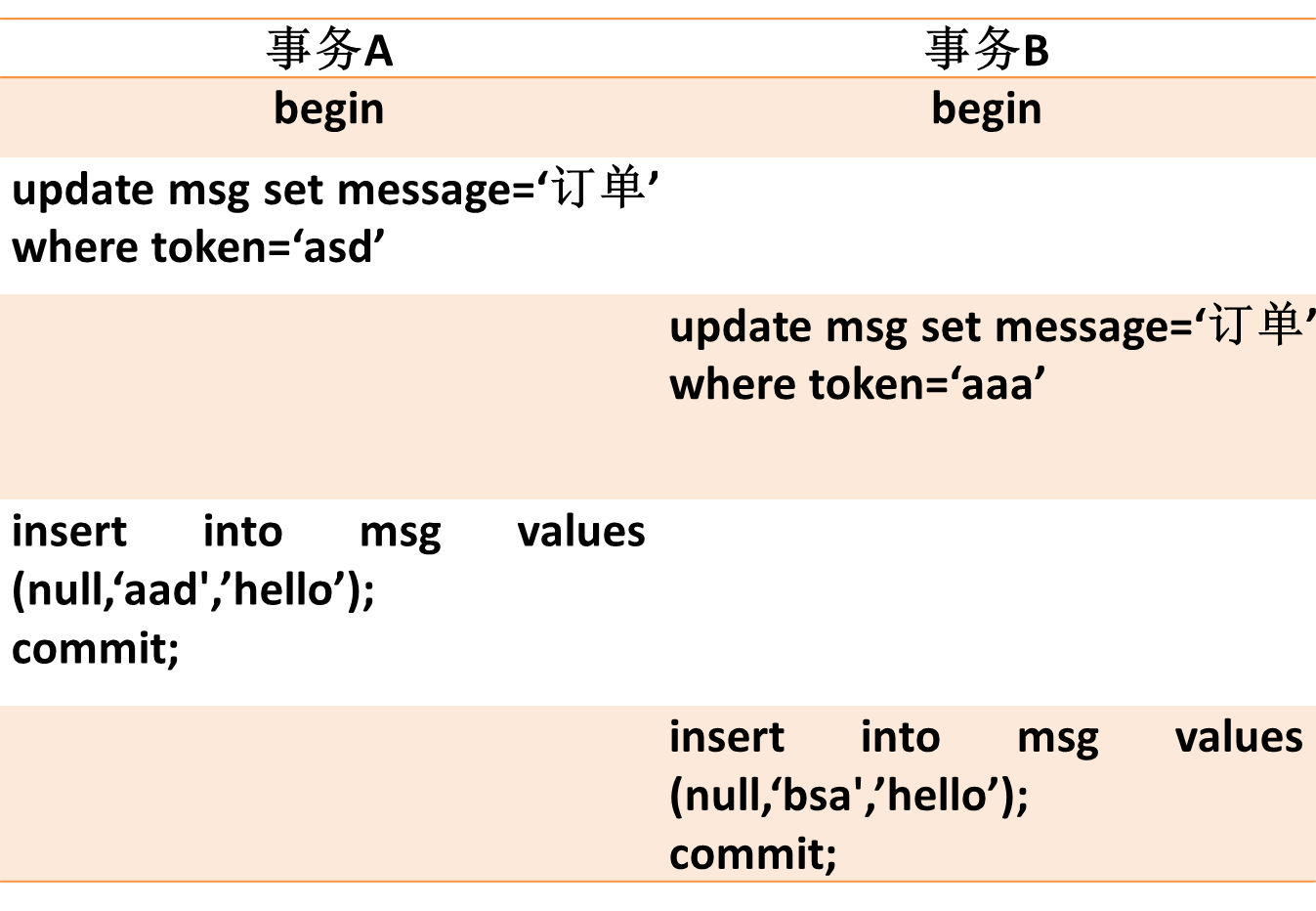


                                                                               图13

## **4 如何尽可能避免死锁**

1）以固定的顺序访问表和行。比如对第2节两个job批量更新的情形，简单方法是对id列表先排序，后执行，这样就避免了交叉等待锁的情形；又比如对于3.1节的情形，将两个事务的sql顺序调整为一致，也能避免死锁。

2）大事务拆小。大事务更倾向于死锁，如果业务允许，将大事务拆小。

3）在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁概率。

4）降低隔离级别。如果业务允许，将隔离级别调低也是较好的选择，比如将隔离级别从RR调整为RC，可以避免掉很多因为gap锁造成的死锁。

5）为表添加合理的索引。可以看到如果不走索引将会为表的每一行记录添加上锁，死锁的概率大大增大。

## **5 如何定位死锁成因**

     下面以本文开头的死锁案例为例，讲下如何排查死锁成因。

1）通过应用业务日志定位到问题代码，找到相应的事务对应的sql；

      因为死锁被检测到后会回滚，这些信息都会以异常反应在应用的业务日志中，通过这些日志我们可以定位到相应的代码，并把事务的sql给梳理出来。

|  |  |
| --- | --- |
| 1  2  3  4  5 | start tran  1 deleteHeartCheckDOByToken  2 updateSessionUser  ...  commit |

      此外，我们根据日志回滚的信息发现在检测出死锁时这个事务被回滚。

2）确定数据库隔离级别。

     执行select @@global.tx\_isolation，可以确定数据库的隔离级别，我们数据库的隔离级别是RC，这样可以很大概率排除gap锁造成死锁的嫌疑;

3）找DBA执行下show InnoDB STATUS看看最近死锁的日志。

     这个步骤非常关键。通过DBA的帮忙，我们可以有更为详细的死锁信息。通过此详细日志一看就能发现，与之前事务相冲突的事务结构如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | start tran  1 updateSessionUser  2 deleteHeartCheckDOByToken  ...  commit |

　　这不就是图10描述的死锁嘛！