Oracle技术嘉年华

Oracle Technology Carnival 2015

稳健●高效●云端 - 数据技术最佳实践

何登成

管中窥豹 ——MySQL/InnoDB死锁分析之道

目录

- 为什么选择“死锁”？
- **基础篇**
  - 为什么要加锁？
  - 锁的持有周期
  - 锁粒度
  - 死锁产生的原因
- **MySQL篇**
  - 哪些操作会加锁？
  - 锁模式
- **实战篇**
  - 死锁案例123

个人简介

- 何登成
- 阿里巴巴高级数据库专家
- 个人微博：@何_登成
- 个人博客：http://hedengcheng.com

为什么选择“死锁”

- **我可以选择的话题**
  - –双11
  - –异地多活
  - –OceanBase
  - –AliSQL（Alibaba MySQL 分支）
  - –阿里巴巴数据库最佳实践
  - –… …
- **但是，我选择了这个话题**
  - –死锁，Why？？？

为什么选择“死锁”

- 首先，本人自封为“死锁小王子”☺
- 但更重要的是，死锁“血泪史”
  - 公司内，帮助多个务线解决死锁问题
  - 公司外部，很多人也在找我讨论MySQL死锁
  - 死锁分析，是MySQL DBA的一个不可承之重☹

基础篇

- 本篇介绍的内容，是对所有数据库均通用的，是分析死锁的基础

- **本篇内容大纲**
  - 为什么要加锁？
  - 锁的持有周期
  - 锁粒度
  - 死锁产生的原因

基础篇——为什么要加锁

- 此锁非彼锁

- 数据库中的锁
  - 确保并发更新场景下的数据正确性
  - 常见的并发更新：商品减库存、消息产生/消费
- ACID中的I（Isolation）

基础篇——锁的持有周期

- **实际情况是**
  - **加锁**：实际访问到某个待更新的行时，对其加锁（而非一开始就将所有的锁都一次性持有）
  - **解锁**：事务提交/回滚时（而非语句结束时就释放）
- **原则之一**
  - 要分析一个死锁，必须深入业务，了解整个事务的逻辑（闭门无法造车）

基础篇——锁粒度

- 试想一个场景
  - 如果我要去图书馆里借一本《高性能MySQL》，为了防止有人提前把这本书给借走了，我可以提前进行预约（加锁），这把锁可以怎么加？
  - 1、封锁图书馆（肯定被打死 ——**数据库级别的锁**
  - 2、把数据库相关的书都锁住——**表级别的锁**
  - 3、只锁MySQL相关的书 ——**页级别的锁**
  - 4、或者干脆只锁《高性能MySQL》这本书 ——**行级别的锁**
- 锁的粒度越细，并发级别越高（实现也更复杂）
  - **传统关系型数据库，都实现了行级别的锁**

基础篇——为什么会产生死锁

- 常见的场景 →

- 产生死锁的必要条件
  - 多个并发事务（2个或者以上）
  - 每个事务都持有了锁（或者是已经在等待锁）
  - 每个事务都需要再继续持有锁（为了完成事务逻辑，还必须更新更多的行）
  - 事务之间产生加锁的循环等待，形成死锁

MySQL篇

- 不同于前面的基础篇，本篇介绍MySQL在加锁处理上的具体实现。大部分跟其他数据库相同，但是小部分有自己的特殊逻辑在里面（有时，往往是这些特殊逻辑，导致了MySQL死锁分析尤其困难）。
  - 注：本篇介绍的，都是InnoDB存储引擎的实现，非InnoDB引擎请不要照搬照抄

- 本篇内容大纲
  - MySQL有哪些操作会加锁？
  - MySQL中有哪些锁模式？
  - MySQL中的锁冲突矩阵
  - MySQL中的特殊加锁逻辑

MySQL篇——会加锁的操作

- **常见的加锁操作**
  - **Insert、Delete、Update（毫无疑问）**
  - **Select … lock in share mode、select … for update（显式加锁）**
  - Lock table … read/write （显示加表级锁）
  - Alter table … / Create Index … （DDL操作引入的加锁）
  - Flush tables … with read lock （备份常用）
  - **Primary Key/Unique Key唯一约束检查**
- **MySQL特有的加锁操作**
  - Purge操作加锁（purge是什么鬼？后面分析）

Question: 各操作分别加什么锁？

MySQL篇——锁模式

- **常规锁模式**
  - LOCK_S（读锁，共享锁，**2**）← 这个数字有啥用？⊗
  - LOCK_X（写锁，排它锁，**3**）

  - 最容易理解的锁模式，读加共享锁，写加排它锁

- **锁的属性**
  - LOCK_REC_NOT_GAP（锁记录，**1024**）
  - LOCK_GAP（锁记录前的GAP，**512**）
  - LOCK_ORDINARY（**同时锁记录+记录前的GAP，0。传说中的Next Key锁**）
  - LOCK_INSERT_INTENTION（插入意向锁，**2048**）

  - 加上LOCK_GAP，一切难以理解的源头（**后面重点分析**）

- **锁组合（属性+模式）**
  - 锁的属性可以与锁模式任意组合。例如：LOCK_REC_NOT_GAP（1024）+ LOCK_X（3）

MySQL篇——锁冲突矩阵

列：存在锁 行：待加锁	S（Not Gap）	S（Gap）	S（Ordinary）	X（Not Gap）	X（Gap）	X（Ordinary）	Insert Intention
S（Not Gap）				冲突		冲突	
S（Gap）							冲突
S（Ordinary）				冲突		冲突	冲突
X（Not Gap）	冲突		冲突	冲突		冲突	
X（Gap）							冲突
X（Ordinary）	冲突		冲突	冲突		冲突	冲突
Insert Intention							

MySQL篇——操作与加锁的对照关系

- 注：以RC隔离级别为例（后面的事例，均为RC隔离级别）

- **Insert**
  - 无Unique Key：LOCK_X + LOCK_REC_NOT_GAP （3 + 1024 = 1027）
  - 有Unique Key：
    - 唯一性约束检查：LOCK_S + LOCK_ORDINARY（2 + 0 = 2）
    - 插入的位置有Gap锁：LOCK_INSERT_INTENTION（2048）
    - 新数据插入：LOCK_X + LOCK_REC_NOT_GAP （3 + 1024 = 1027）

MySQL篇——操作与加锁的对照关系

- Delete
  - 满足删除条件的所有记录，**LOCK_X + LOCK_REC_NOT_GAP**

MySQL篇——操作与加锁的对照关系

- **Update操作分解**
  - Step 1：定位到下一条满足查询条件的记录（查询过程，类似于Select/Delete）
  - Step 2：删除当前定位到的记录（标记为删除状态）
  - Step 3：拼装更新后项，根据更新后项定位到新的插入位置
  - Step 4：在新的插入位置，判断是否存在 Unique 冲突（存在Unique Key时）
  - Step 5：插入更新后项（不存在Unique冲突时）
  - Step 6：重复Step 1到Step 5的操作，直至扫描完整个查询范围
- **Update操作分析**
  - Step 1，Step 2：**Delete**
  - Step 3，Step 4，Step 5：**Insert**

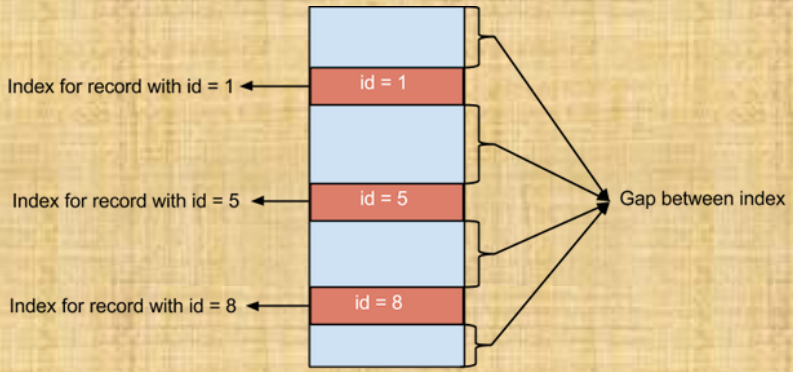
MySQL篇——操作与加锁的对照关系

- **Update**
  - 无**Unique Key**：查询范围中的所有记录， **LOCK_X + LOCK_REC_NOT_GAP**
  - 有**Unique Key**：
    - 查找满足条件的记录：查询范围内的所有记录， **LOCK_X + LOCK_REC_NOT_GAP**
    - 更新后项存在唯一性冲突：冲突项上的加锁，**LOCK_S + LOCK_ORDINARY**
    - 更新后项不存在唯一性冲突： 更新位置后项加锁，**LOCK_S + LOCK_GAP（省略）**
    - 实际更新操作：可看做插入了一条新纪录，**LOCK_X + LOCK_REC_NOT_GAP**
    - **很复杂，我相信大家都晕了……**
    - **记住一点：Unique 索引的加锁处理，非常复杂（☹）**

MySQL篇——GAP锁

- **GAP锁的目的**
  - 实现可重复读，防止幻读的产生（与MVCC的区别？）
  - 生活中的例子

  - 数据库中的例子



MySQL篇——GAP锁

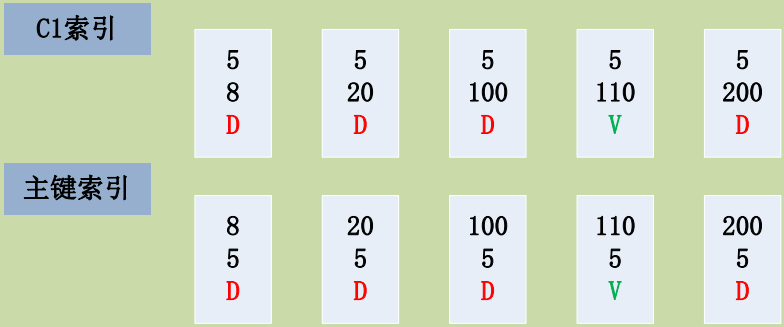
- **哪些操作会加GAP锁？**
  - **Read Committed（RC）**：Unique Key 唯一约束检查；Purge操作；
  - **Repeatable Read（RR）**：RC的基础上，所有需要加锁的索引范围扫描（Update/Delete…）

- **原则之二**
  - GAP锁很复杂，为了减少GAP锁，减少GAP导致的死锁，尽量选择Read Committed隔离级别（**RC + row based binlog**，基本上能够解决所有问题，无需使用Repeatable Read）
  - 适当的**减少Unique索引**，能够减少GAP锁导致的死锁（根据业务情况而定）

MySQL篇——特殊加锁实现分析

- **Q1**：在RC隔离级别下，唯一性约束检查，为何需要加Gap锁？
- 解答：
  - **索引的逻辑删除**

Table T1(id int primary key, c1 int unique key) engine = innodb;



1. 5条记录，C1列取值相同，但是只有一个是有效项【110,5】
2. 此时做唯一性约束检查，要确保任意一个位置，均不能插入C1 = 5的记录

- **追加Question**
  - **1. 为何采用逻辑删除机制？**
  - **2. 逻辑删除的项，如何物理删除？→ Purge**（https://bugs.mysql.com/bug.php?id=76927）

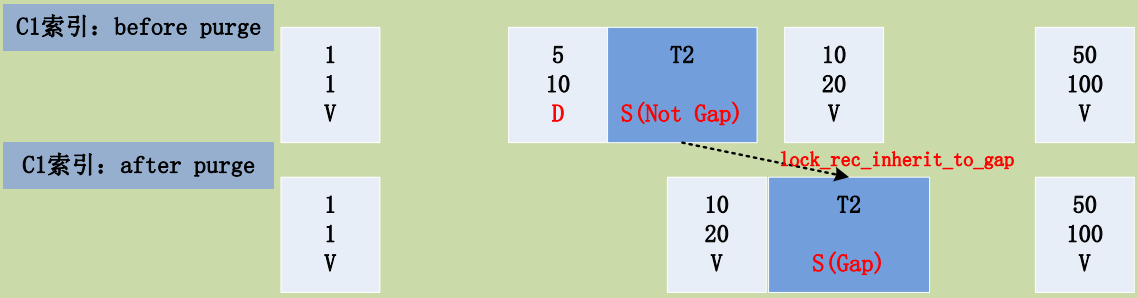
MySQL篇——特殊加锁实现分析

```
Table T1(id int primary key, c1 int unique key) engine = innodb;
```

T1表原始记录：【1,1】，【10,5】，【20,10】，【100,50】

并发的三个事务：

```
T1                          T2                          T3                          Tpurge
Begin
Delete from t1 where c1 = 5;
                            begin
                            select * from t1 where c1 = 5 lock in share mode;
Commit;
                                                                                    purge c1 = 5;
                                                        begin
                                                        insert into t1 values (300,7) ;
```



实战篇

- 做了这么长的前戏铺垫，就是为了现在的实战演练。本篇内容，挑选了本人过去工作中碰到的一些死锁案例，与大家分享。

- 如何解读死锁日志

- 死锁案例123

实战篇——解读死锁日志

```
*** (1) TRANSACTION:
TRANSACTION 60882C7, ACTIVE 1 sec inserting
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 376, 1 row lock(s), undo log entries 1
MySQL thread id 45, OS thread h                 834080700, query id 6553 localhost root update
insert into t values (null,10)
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 10049 page no 4 n bits 72 index `c2` of table `tt`.`t` trx id 60882C7 lock mode S
waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 0

*** (2) TRANSACTION:
TRANSACTION 60882C5, ACTIVE 32 sec inserting, thread declared inside InnoDB 500
mysql tables in use 1, locked 1
3 lock struct(s), heap size 376, 2 row lock(s), undo log entries 2
MySQL thread id 44, OS thread handle 0x2af828101700, query id 6554 localhost root update
insert into t values (null,9)
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 10049 page no 4 n bits 72 index `c2` of table `tt`.`t` trx id 60882C5 lock_mode X locks
rec but not gap
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 0

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 10049 page no 4 n bits 72 index `c2` of table `tt`.`t` trx id 60882C5 lock_mode X locks
gap before rec insert intention waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 0

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

执行SQL

哪个页面

哪个索引

锁模式

哪条记录

实战篇——实战环境

测试环境：

表定义：

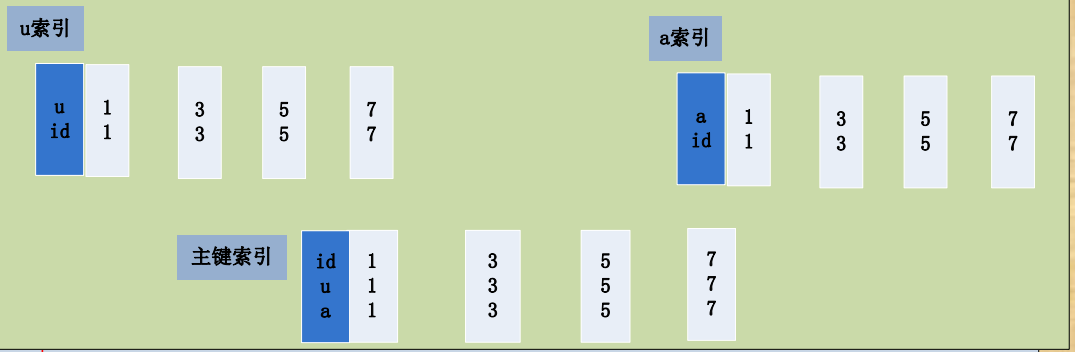
```
root@tt 10:08:55>show create table tu \G
*************************** 1. row ***************************
       Table: tu
Create Table: CR
  `id` int(11) N
  `u` int(11) DE
  `a` int(11) DE
  PRIMARY KEY (`
  UNIQUE KEY `u`
  KEY `a` (`a`)
) ENGINE=InnoDB
1 row in set (0.
```

原始数据：

```
root@tt 10:32:42
+----+------+---
| id | u    | a
+----+------+---
|  1 |    1 |
|  3 |    3 |    3 |

|  5 |    5 |    5 |
|  7 |    7 |    7 |
+----+------+------+
4 rows in set (0.01 sec)
```

```
Table Tu(id int primary key, u int unique key, a int key) engine = innodb;
```



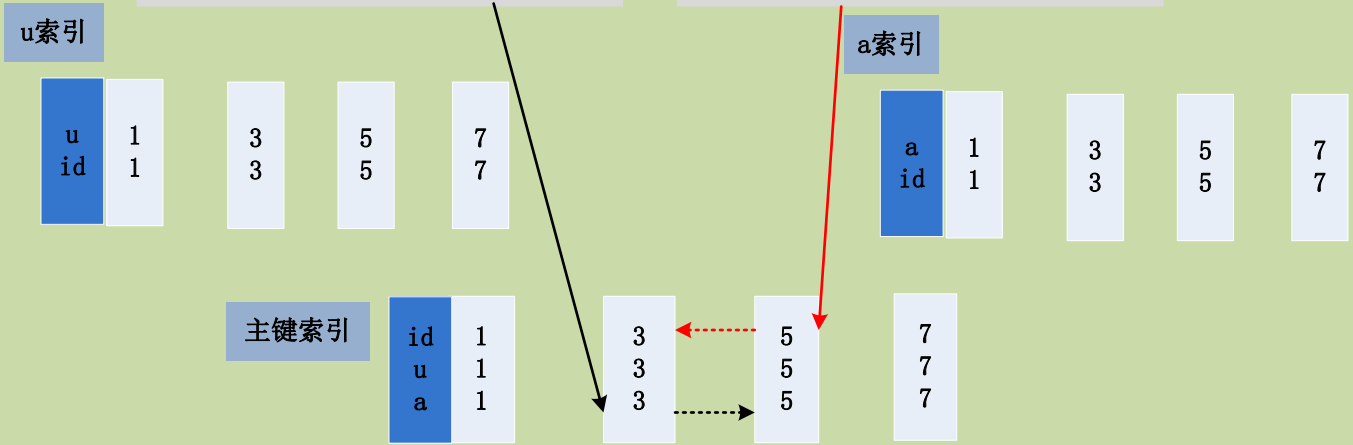
实战篇——死锁事例1

```
*** (1) TRANSACTION:
TRANSACTION 140142654803, ACTIVE 54 sec starting index read
mysql tables in use 1, locked 1
```

```
Table Tu(id int primary key, u int unique key, a int key) engine = innodb;
```

```
Session1:
 begin;
 select * from tu where id=3 for update;
 select * from tu where id=5 for update;
```

```
Session2:
 begin;
 select * from tu where id=5 for update;
 select * from tu where id=3 for update;
```



实战篇——死锁事例1

- **死锁事例1**
  - 最简单、最经典的死锁场景

- **两个并发事务**
  - Session1：

    Begin；

    select * from tu where id=3 for update;

    select * from tu where id=5 for update;

    Commit；

    Session 2

    Begin

    select * from tu where id=5 for update

    select * from tu where id=3 for update

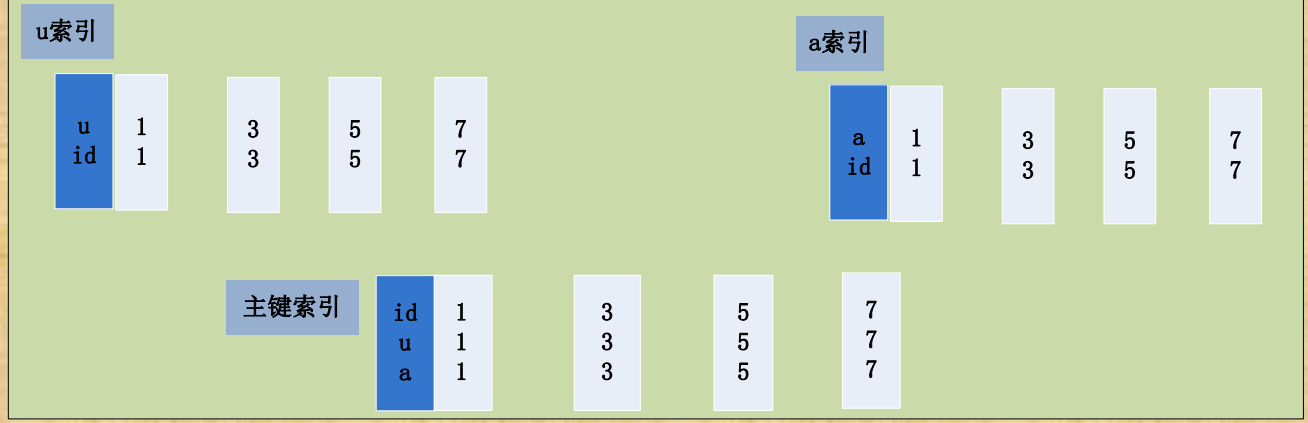
    Commit；

- 分析
  - 死锁，永远跟加锁顺序相关。事务以相反的顺序操作数据，才会产生死锁。分析死锁，就是挖掘出这个相反的顺序在哪。

实战篇——死锁事例2

- 回顾一下表结构

```
Table Tu(id int primary key, u int unique key, a int key) engine = innodb;
```



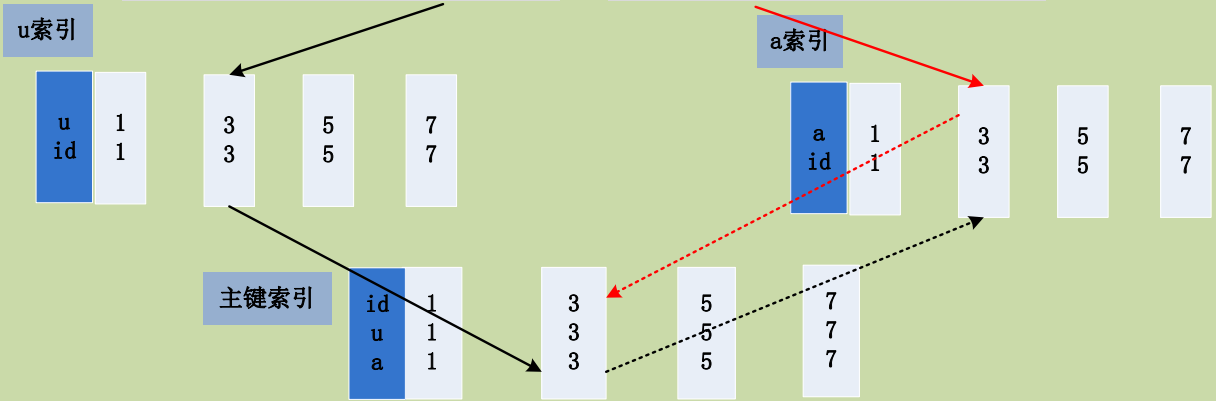
实战篇——死锁事例2

```
*** (1) TRANSACTION:
TRANSACTION 140142655258, ACTIVE 2 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 376, 2 row lock(s)
```

```
Table Tu(id int primary key, u int unique key, a int key) engine = innodb;
```

```
Session1:
 update tu set a=4 where u=3;
```

```
Session2:
 update tu set u=4 where a=3;
```




```
Record lock, heap no 7 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 80000003; asc     ;;
 0: len 4; hex 80000003; asc     ;;
```

实战篇——死锁案例2

- **死锁事例2**
  - 稍微复杂点的情况

- **两个并发事务**
  - Session1：

    Update tu set a = 4 **where u = 3**；

    Session 2

    update tu set u = 4 **where a = 3**；

- **原则之三**
  - **在MySQL中，以不同索引的过滤条件，来操作相同的记录（Update/Delete），很容易产生死锁。**（例如：如上例所示，执行计划分别走u和a索引，但是均操作了id = 3这一列，产生死锁。）

实战篇——死锁案例3

- 复杂死锁例子（Gap 锁闪亮登场）

- 新的测试环境
- 表结构：

```
CREATE TABLE `t` (
    `c1` int(11) NOT NULL AUTO_INCREMENT,
    `c2` int(11) DEFAULT NULL,
    PRIMARY KEY (`c1`),
    UNIQUE KEY `c2` (`c2`)
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=gbk
```

- SQL：T1、T2两个并发事务（这也会产生死锁… …???）

```
T2: insert into t values (null,10);
T1: insert into t values (null,10);
T2: insert into t values (null,9);
```

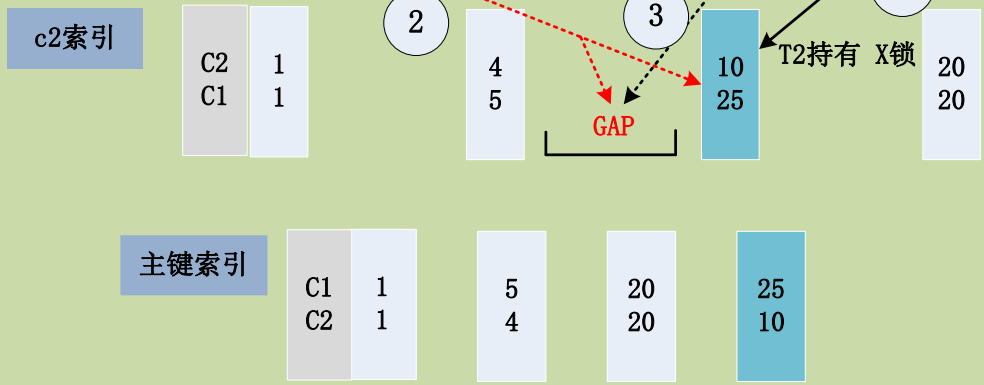
实战篇——死锁案例3

```
*** (1) TRANSACTION:
TRANSACTION 60882C7, ACTIVE 1 sec inserting
mysql tables in use 1, locked 1
```

```
Table t(c1 int primary key, c2 int unique key) engine = innodb;
原始数据：【1,1】，【5,4】，【20，20】
```

```
Session1：
 begin;
 insert into t values (30, 10);
```

```
Session2：
 begin;
 insert into t values (25, 10);
 insert into t values (40, 9);
```




```
0: len 4; hex 8000000a; asc     ;;
```

实战篇——死锁案例3

- **死锁案例3**
  - 这是我们线上系统的一个真实案例
  - 简单的并发插入，会出现死锁… …
  - 死锁信息中，出现了Next Key（Gap锁）
- **分析**
  - 在存在Unique索引的表中，比较容易出现由于check unique（唯一性约束检查）而产生的死锁
- **原则之四**
  - RC隔离级别下，如果死锁中出现Next Key（Gap锁），说明表中一定存在unique索引
  - 多语句事务产生的死锁，确保每条语句操作记录的顺序性，能够极大减少死锁

实战篇——AliSQL增强

- 死锁分析为什么难？
  - 理解MySQL的加锁处理逻辑比较难
  - **MySQL的死锁日志信息不全**
- **AliSQL在这方面有所增强**
  - 打印完整的死锁信息

```
*** (1) TRANSACTION:
TRANSACTION 140142655253, ACTIVE 1 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 376, 2 row lock(s)
MySQL thread id 3, OS thread handle 0x2b4ec0080700, query id 25
localhost root Searching rows for update
update tu set u=4 where a=3
*** (1) HOLDS THE LOCK(S):
RECORD LOCKS space id 20950 page no 5 n bits 80 index `a` of
table `tt`.`tu` trx id 140142655253 lock_mode X locks rec but
not gap
Record lock, heap no 7 PHYSICAL RECORD: n_fields 2; compact
format; info bits 0
 0: len 4; hex 80000003; asc     ;;

*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 20950 page no 3 n bits 72 index `PRIMARY`
of table `tt`.`tu` trx id 140142655253 lock_mode X locks rec but
not gap waiting
Record lock, heap no 5 PHYSICAL RECORD: n_fields 5; compact
format; info bits 0

*** (2) TRANSACTION:
TRANSACTION 140142655250, ACTIVE 130 sec updating or deleting,
thread declared inside InnoDB 4999
mysql tables in use 1, locked 1
4 lock struct(s), heap size 1248, 3 row lock(s), undo log
entries 1
MySQL thread id 1, OS thread handle 0x2b4eb4080700, query id 20
localhost root updating
update tu set a=4 where u=3
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 20950 page no 3 n bits 72 index `PRIMARY`
of table `tt`.`tu` trx id 140142655250 lock_mode X locks rec but
not gap
```

总结

- **原则之一**
  - 要分析一个死锁，必须深入业务，了解整个事务的逻辑（闭门无法造车）
- **原则之二**
  - GAP锁很复杂，为了减少GAP锁，减少GAP导致的死锁，尽量选择Read Committed隔离级别（**RC + row based binlog**，基本上能够解决所有问题，无需使用Repeatable Read）
  - 适当的**减少Unique索引**，能够减少GAP锁导致的死锁（根据业务情况而定）
- **原则之三**
  - **在MySQL中，以不同索引的过滤条件，来操作相同的记录（Update/Delete），很容易产生死锁。**（例如：如上例所示，执行计划分别走u和a索引，但是均操作了id = 3这一列，产生死锁。）
- **原则之四**
  - RC隔离级别下，如果死锁中出现Next Key（Gap锁），说明表中一定存在unique索引
  - 多语句事务产生的死锁，确保每条语句操作记录的顺序性，能够极大减少死锁

THANKS