# SQL语句执行得很慢

## 分类讨论

一条 SQL 语句执行的很慢，那是每次执行都很慢呢？还是大多数情况下是正常的，偶尔出现很慢呢？所以我觉得，我们还得分以下两种情况来讨论。

1、大多数情况是正常的，只是偶尔会出现很慢的情况。

2、在数据量不变的情况下，这条SQL语句一直以来都执行的很慢。

针对这两种情况，我们来分析下可能是哪些原因导致的。

## 针对偶尔很慢的情况

一条 SQL 大多数情况正常，偶尔才能出现很慢的情况，针对这种情况，我觉得这条SQL语句的书写本身是没什么问题的，而是其他原因导致的，那会是什么原因呢？

### 刷新脏页

数据库在刷新脏页（flush）我也无奈啊

当我们要往数据库插入一条数据、或者要更新一条数据的时候，我们知道数据库会在内存中把对应字段的数据更新了，但是更新之后，这些更新的字段并不会马上同步持久化到磁盘中去，而是把这些更新的记录写入到 redo log 日记中去，等到空闲的时候，在通过 redo log 里的日记把最新的数据同步到磁盘中去。

当内存数据页跟磁盘数据页内容不一致的时候，我们称这个内存页为“脏页”。内存数据写入到磁盘后，内存和磁盘上的数据页的内容就一致了，称为“干净页”。

刷脏页有下面4种场景（后两种不用太关注“性能”问题）：

redolog写满了：redo log 里的容量是有限的，如果数据库一直很忙，更新又很频繁，这个时候 redo log 很快就会被写满了，这个时候就没办法等到空闲的时候再把数据同步到磁盘的，只能暂停其他操作，全身心来把数据同步到磁盘中去的，而这个时候，就会导致我们平时正常的SQL语句突然执行的很慢，所以说，数据库在在同步数据到磁盘的时候，就有可能导致我们的SQL语句执行的很慢了。

内存不够用了：如果一次查询较多的数据，恰好碰到所查数据页不在内存中时，需要申请内存，而此时恰好内存不足的时候就需要淘汰一部分内存数据页，如果是干净页，就直接释放，如果恰好是脏页就需要刷脏页。

MySQL 认为系统“空闲”的时候：这时系统没什么压力。

MySQL 正常关闭的时候：这时候，MySQL 会把内存的脏页都 flush 到磁盘上，这样下次 MySQL 启动的时候，就可以直接从磁盘上读数据，启动速度会很快。

### 获取锁失败

拿不到锁我能怎么办

这个就比较容易想到了，我们要执行的这条语句，刚好这条语句涉及到的表，别人在用，并且加锁了，我们拿不到锁，只能慢慢等待别人释放锁了。或者，表没有加锁，但要使用到的某个一行被加锁了，这个时候，我也没办法啊。

如果要判断是否真的在等待锁，我们可以用 show processlist这个命令来查看当前的状态哦，这里我要提醒一下，有些命令最好记录一下，反正，我被问了好几个命令，都不知道怎么写，呵呵。

下来我们来访分析下第二种情况，我觉得第二种情况的分析才是最重要的

## 针对一直都这么慢的情况

如果在数据量一样大的情况下，这条 SQL 语句每次都执行的这么慢，那就就要好好考虑下你的 SQL 书写了，下面我们来分析下哪些原因会导致我们的 SQL 语句执行的很不理想。

我们先来假设我们有一个表，表里有下面两个字段,分别是主键 id，和两个普通字段 c 和 d。

mysql> CREATE TABLE `t` (

`id` int(11) NOT NULL,

`c` int(11) DEFAULT NULL,

`d` int(11) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB;

1、扎心了，没用到索引

没有用上索引，我觉得这个原因是很多人都能想到的，例如你要查询这条语句

select \* from t where 100 <c and c < 100000;

（1）、字段没有索引

刚好你的 c 字段上没有索引，那么抱歉，只能走全表扫描了，你就体验不会索引带来的乐趣了，所以，这回导致这条查询语句很慢。

（2）、字段有索引，但却没有用索引

好吧，这个时候你给 c 这个字段加上了索引，然后又查询了一条语句

select \* from t where c - 1 = 1000;

我想问大家一个问题，这样子在查询的时候会用索引查询吗？

答是不会，如果我们在字段的左边做了运算，那么很抱歉，在查询的时候，就不会用上索引了，所以呢，大家要注意这种字段上有索引，但由于自己的疏忽，导致系统没有使用索引的情况了。

正确的查询应该如下

select \* from t where c = 1000 + 1;

有人可能会说，右边有运算就能用上索引？难道数据库就不会自动帮我们优化一下，自动把 c - 1=1000 自动转换为 c = 1000+1。

不好意思，确实不会帮你，所以，你要注意了。

（3）、函数操作导致没有用上索引

如果我们在查询的时候，对字段进行了函数操作，也是会导致没有用上索引的，例如

select \* from t where pow(c,2) = 1000;

这里我只是做一个例子，假设函数 pow 是求 c 的 n 次方，实际上可能并没有 pow(c,2)这个函数。其实这个和上面在左边做运算也是很类似的。

所以呢，一条语句执行都很慢的时候，可能是该语句没有用上索引了，不过具体是啥原因导致没有用上索引的呢，你就要会分析了，我上面列举的三个原因，应该是出现的比较多的吧。

2、呵呵，数据库自己选错索引了

我们在进行查询操作的时候，例如

select \* from t where 100 < c and c < 100000;

我们知道，主键索引和非主键索引是有区别的，主键索引存放的值是整行字段的数据，而非主键索引上存放的值不是整行字段的数据，而且存放主键字段的值。不大懂的可以看这篇文章： 【思维导图-索引篇】搞定数据库索引就是这么简单 里面有说到主键索引和非主键索引的区别

也就是说，我们如果走 c 这个字段的索引的话，最后会查询到对应主键的值，然后，再根据主键的值走主键索引，查询到整行数据返回。

好吧扯了这么多，其实我就是想告诉你，就算你在 c 字段上有索引，系统也并不一定会走 c 这个字段上的索引，而是有可能会直接扫描扫描全表，找出所有符合 100 < c and c < 100000 的数据。

为什么会这样呢？

其实是这样的，系统在执行这条语句的时候，会进行预测：究竟是走 c 索引扫描的行数少，还是直接扫描全表扫描的行数少呢？显然，扫描行数越少当然越好了，因为扫描行数越少，意味着I/O操作的次数越少。

如果是扫描全表的话，那么扫描的次数就是这个表的总行数了，假设为 n；而如果走索引 c 的话，我们通过索引 c 找到主键之后，还得再通过主键索引来找我们整行的数据，也就是说，需要走两次索引。而且，我们也不知道符合 100 c < and c < 10000 这个条件的数据有多少行，万一这个表是全部数据都符合呢？这个时候意味着，走 c 索引不仅扫描的行数是 n，同时还得每行数据走两次索引。

所以呢，系统是有可能走全表扫描而不走索引的。那系统是怎么判断呢？

判断来源于系统的预测，也就是说，如果要走 c 字段索引的话，系统会预测走 c 字段索引大概需要扫描多少行。如果预测到要扫描的行数很多，它可能就不走索引而直接扫描全表了。

那么问题来了，系统是怎么预测判断的呢？这里我给你讲下系统是怎么判断的吧，虽然这个时候我已经写到脖子有点酸了。

系统是通过索引的区分度来判断的，一个索引上不同的值越多，意味着出现相同数值的索引越少，意味着索引的区分度越高。我们也把区分度称之为基数，即区分度越高，基数越大。所以呢，基数越大，意味着符合 100 < c and c < 10000 这个条件的行数越少。

所以呢，一个索引的基数越大，意味着走索引查询越有优势。

那么问题来了，怎么知道这个索引的基数呢？

系统当然是不会遍历全部来获得一个索引的基数的，代价太大了，索引系统是通过遍历部分数据，也就是通过采样的方式，来预测索引的基数的。

扯了这么多，重点的来了，居然是采样，那就有可能出现失误的情况，也就是说，c 这个索引的基数实际上是很大的，但是采样的时候，却很不幸，把这个索引的基数预测成很小。例如你采样的那一部分数据刚好基数很小，然后就误以为索引的基数很小。然后就呵呵，系统就不走 c 索引了，直接走全部扫描了。

所以呢，说了这么多，得出结论：由于统计的失误，导致系统没有走索引，而是走了全表扫描，而这，也是导致我们 SQL 语句执行的很慢的原因。

这里我声明一下，系统判断是否走索引，扫描行数的预测其实只是原因之一，这条查询语句是否需要使用使用临时表、是否需要排序等也是会影响系统的选择的。

不过呢，我们有时候也可以通过强制走索引的方式来查询，例如

select \* from t force index(a) where c < 100 and c < 100000;

我们也可以通过

show index from t;

来查询索引的基数和实际是否符合，如果和实际很不符合的话，我们可以重新来统计索引的基数，可以用这条命令

analyze table t;

来重新统计分析。

既然会预测错索引的基数，这也意味着，当我们的查询语句有多个索引的时候，系统有可能也会选错索引哦，这也可能是 SQL 执行的很慢的一个原因。

好吧，就先扯这么多了，你到时候能扯出这么多，我觉得已经很棒了，下面做一个总结。

## 总结

以上是我的总结与理解，最后一个部分，我怕很多人不大懂数据库居然会选错索引，所以我详细解释了一下，下面我对以上做一个总结。

一个 SQL 执行的很慢，我们要分两种情况讨论：

1、大多数情况下很正常，偶尔很慢，则有如下原因

(1)、数据库在刷新脏页，例如 redo log 写满了需要同步到磁盘。

(2)、执行的时候，遇到锁，如表锁、行锁。

2、这条 SQL 语句一直执行的很慢，则有如下原因。

(1)、没有用上索引：例如该字段没有索引；由于对字段进行运算、函数操作导致无法用索引。

(2)、数据库选错了索引。

# 线上IO问题

## 问题描述

某业务CDB实例，每天在特地时间段内（ 00:07:00 - 00:08:00左右）机器对应IO监控出现写入尖刺，且主从实例都有类似现象，从机器监控可以看到，问题确实存在。

不仅master，进行同步的slave上有相同的现象，业务方希望找到导致该IO尖刺问题稳定出现的原因。

## 问题分析

首先确定问题来源，上图所示监控为机器级别，机器IO写入负载是否来源于mysqld进程？如果来源于mysqld进程，是来自于mysqld进程的哪一部分写入操作引起？

为了获取IO来源，在slave机上部署mysqld实例监控，以及iotop采集监控，获取对应时间段更详细的相关信息，抓取对应时间段进行IO写入的进程（线程），同时观察对应时间段mysql实例状态。

这段时间内的较大IO写入线程号为：（截取部分记录）

时间 线程号 进程名 读取速度 写入速度

00:07:34 145378 be/4 mysql201 139.10 K/s 263111.57 K/s

00:07:35 145378 be/4 mysql201 124.11 K/s 249703.84 K/s

00:07:36 145378 be/4 mysql201 120.23 K/s 289920.70 K/s

00:07:39 145378 be/4 mysql201 5168.09 K/s 875194.69 K/s

通过slave上iotop采集到的统计信息，可以看出较大写入来源为145378这一线程，确实来自于mysql进程，该时间段内没有抓到其他大量写入的记录，同时该实例slave mysql为单机独占，可以基本确定写入来源为mysql中145378这个线程，那么这个线程是哪一个线程呢？

145378这个线程号正是mysqld slave的线程，而且为单线程回放的sql线程。

Thread 85 (Thread 0x7f68c4c4c700 (LWP 145378)):

#0 0x00007fa2badd3945 in pthread\_cond\_wait@@GLIBC\_2.3.2 () from /lib64/libpthread.so.0

#4 exec\_relay\_log\_event (rli=0x1771c43c8, thd=0x7f68b0000990)

#5 handle\_slave\_sql (arg=0x1771c3010)

通过进一步搜索监控记录，可以发现，其实sql线程引入大量写入IO不仅存在于这个时间段，在其他时间段也有较高写入的记录（超过100MB/s），在00:08:00左右持续时间相对较长。这个现象反应出该问题可能贯穿于整个执行过程，只是其他时间段没有这么明显。

通过mysql实例的监控可，可以看到mysql主要进行的操作为insert，slave mysql主要在进行单线程回放，执行这些insert操作，操作数量会有一些增长，每秒操作数不稳定，峰值可能达到4000左右。

从对于slave实例的新增监控可以得到结论，这段时间的主要写入来源确实为mysqld进程，且为mysqld的sql线程，那么问题转换为：

在该业务模型下，进行单线程回放的mysql slave sql thread， 为什么会在一些时段产生较大的写入IO？产生这个现象的时候，mysql在做什么事情？

为了更好的分析问题，同时不影响线上业务，在另外一台实验机器上单独搭建mysql 5.7的slave，连到源实例的master作为slave，问题现象也可以复现，为了获取更多信息，打开mysql 5.7的performance\_schema，在实验机器的slave机监控上，依然能看到该问题存在。

现在我们需要分析一下，SQL线程回放，可能产生哪些IO写（注意其他线程的IO不会记录到SQL线程头上，例如page cleaner flush），一条SQL语句回放过程中，可能经历的路径上有哪些操作会引发IO操作。

一个slave mysql读取relay log进行日志回放，首先想到回放执行语句，可能由此引发下列写入IO：

mysql server binlog日志记录，即回放过程中语句写入的本地binlog。

innodb redo undo日志记录。

用户态page flush（free page吃紧，需要自行flush获取可用free page）。

relay log purge，删除文件。默认每当slave回放完一个完整relay log，会将对应文件进行删除。

那么，为了找到真正引发写入的来源，需要对于上述来源逐步进行分析、排除，对可能原因进行验证。

首先在5.7 slave上配置log\_slave\_updates=OFF，使得回放过程不记录binlog日志，问题现象依旧，排除binlog写入。

分钟级监控显示，master实例在对应时间段insert数量有一定幅度增加：

为了保证free page够用，调大了buffer pool，确保free page足够使用，另外关闭relay log purge功能，使得SQL线程不会触发删除relay log文件，问题现象依旧，排除清理relay以及用户态page flush。

通过sys schema统计值发现，对于文件写入，count write统计值较大的记录主要来源于redo log（例如ib\_logfile0写入7439952次，且总量为57.47GB），但是innodb自身对于log写入的单位时间统计值显示却不大。

/data1/mysql\_root/log/20120/ib\_logfile0 5 4.00 KiB 819 bytes 7439952 57.47 GiB 8.10 KiB 57.47 GiB 100.00

/data1/mysql\_root/log/20120/ib\_logfile1 2 64.50 KiB 32.25 KiB 3025254 23.39 GiB 8.11 KiB 23.39 GiB 100.00

为了进一步排除干扰，修改mysql实现关闭redo log写入，替换mysql 5.7实验版本，统计值显示确实redo文件几乎没有写入增长，问题依旧，排除redo写入造成，再看ibdata相关记录增长，也非常有限，也可以排除undo文件写入。

到目前为止，通常能想到的用户态写入（例如sql线程回放执行一个事务），都可以排除掉，还有什么原因可以sql线程造成大量写入呢？需要重新整理一下思路。

再回到业务本身，看一下业务的库表结构模型和数据规模，表结构大体如下：

FRFrom int(10) unsigned NOT NULL DEFAULT ‘0’,

FRtBody mediumblob NOT NULL,

FPSQL varchar(20480) NOT NULL,

FBody mediumblob NOT NULL,

FResTime bigint(20) NOT NULL DEFAULT ‘0’,

FCime bigint(20) NOT NULL DEFAULT ‘0’,

FInt1 int(10) unsigned NOT NULL DEFAULT ‘0’,

FInt2 int(10) unsigned NOT NULL DEFAULT ‘0’,

FInt3 int(10) unsigned NOT NULL DEFAULT ‘0’,

FInt4 int(10) unsigned NOT NULL DEFAULT ‘0’,

FInt5 int(10) unsigned NOT NULL DEFAULT ‘0’,

FChar1 varchar(256) NOT NULL DEFAULT ‘’,

FChar2 varchar(256) NOT NULL DEFAULT ‘’,

FChar3 varchar(512) NOT NULL DEFAULT ‘’,

FChar4 varchar(1024) NOT NULL DEFAULT ‘’,

FChar5 varchar(1024) NOT NULL DEFAULT ‘’,

FExt blob,

FGrp2EvtNo bigint(20) DEFAULT NULL,

PRIMARY KEY (FEventNo)

表数量很多，超过1W张表，且单表数据量不大

表结构没有其他索引，只有主键

包含blob字段

从relay log分析插入语句，blob每次都有较大的数据量进行插入

表数量较多，插入操作，blob字段，并发插入且插入操作分散到各个表，这几点看起来有些关联。在关掉了包括binlog，relay purge，redo log等多个写入之后，再抓一下sql线程回放的堆栈，看一下写入调用的来源：

#0 os\_aio\_func (type=..., mode=mode@entry=24,

name=0x7ee4493f97b8 "./DB\_xxx/xxx\_36.ibd", file=..., buf=buf@entry=0x7ee453ed4000, offset=offset@entry=122683392, n=n@entry=1048576, read\_only=read\_only@entry=false, m1=m1@entry=0x0, m2=m2@entry=0x0)

#1 0x0000000001187922 in fil\_write\_zeros (node=0x7ee4493f5338, node=0x7ee4493f5338, read\_only\_mode=<optimized out>,

len=4194304, start=121634816, page\_size=<optimized out>)

#2 fil\_space\_extend (space=space@entry=0x7ee448d8dbd8, size=<optimized out>)

#3 0x00000000007701d1 in fsp\_try\_extend\_data\_file (space=space@entry=0x7ee448d8dbd8, header=header@entry=0x7ef4a4ff0026 "",

#4 0x000000000118f0df in fsp\_reserve\_free\_extents (n\_reserved=n\_reserved@entry=0x7f20868f4fa0, space\_id=9054, n\_ext=3,

alloc\_type=alloc\_type@entry=FSP\_NORMAL, mtr=mtr@entry=0x7f20868f5890, n\_pages=n\_pages@entry=2)

#5 0x00000000010e6394 in btr\_cur\_pessimistic\_insert (flags=flags@entry=0, cursor=cursor@entry=0x7f20868f5150,

offsets=offsets@entry=0x7f20868f50b0, heap=heap@entry=0x7f20868f50a0, entry=entry@entry=0x7ee453143488,

rec=rec@entry=0x7f20868f5570, big\_rec=big\_rec@entry=0x7f20868f5090, n\_ext=n\_ext@entry=0, thr=thr@entry=0x7ee4544623f0,

mtr=mtr@entry=0x7f20868f5890)

回顾一下基于主键索引的插入操作，对于b+树，如果插入的record的较大（例如很大的blob），可能会触发分裂操作。类似

对于innodb而言，插入的过程中，首先尝试乐观插入索引，如果空间大小不够，再尝试悲观插入，悲观插入首先保证表空间大小足够（ibd文件 innodb\_file\_per\_table=ON，每个表对应一个文件），这里用户场景，正是每个表一个ibd文件，那么16000张表总共有16000个ibd文件。如果空间不够，尝试对于ibd文件进行扩展，扩展逻辑如下：

每次扩展4个extent（每个extent包含 16kb\*64 = 1MB大小供64个data page），即每次扩展形成4MB，业务场景包含16000多个表，且每次插入数据量相对固定，表结构相同，插入目标表随机分散，所以很多表大小，和后续操作，非常均衡可以看作是齐头并进。

从slave实例监控来看，insert在对应时间段超过1000个每秒，最多可以达到4000个每秒。假设相同数据量模型，其中500个表同时扩展大小，这一秒内，可能同时产生500\*4MB = 2GB左右的写入。为了印证这个写入来源，继续在innodb层添加日志跟踪，对于idb文件扩展加入以下逻辑进行日志跟踪：

每100MB扩展数据量，查看累计时间，如果累计时间在1秒以内，打印一条warning日志，且sleep 0.5s。

查看对应时间段新增日志（+8为北京时间），

对应时间扩展很频繁，除去sleep时间，大约0.1-0.2秒能够产生100MB的扩展写入，IO尖刺数据量基本吻合，与iotop抓取到的大io写入也基本吻合。至此我们基本可以得出问题结论。

## 问题结论

业务模型比较特殊，多个包含BLOB字段的小表（超过16000），业务模型为insert，写入目标表分散，在某些时间段会并发insert（平均2000个左右每秒，峰值4000个每秒），由于BLOB字段占用空间较大，各个表使用分散，导致频繁同时触发分裂，进而导致底层data page扩展，使得底层多个ibd文件同时并发extend，初始化data page产生某些时间段较大IO，反应到机器监控上为某些时间段IO尖刺

## 解决方法

业务层面，对于类似场景，考虑合并表数量，减少并发扩展带来的写入压力，可以一定程度缓解IO尖刺。

MySQL层面，考虑更加智能的数据文件扩展算法，适配上述场景。减少扩展的并发性。

MySQL层面，考虑用户指定初始化表空间大小，提前预分配和初始化，避免动态扩展。

# 数据库死锁

## 背景

说起Mysql死锁，之前写过一次有关Mysql加锁的基本介绍，对于一些基本的Mysql锁或者死锁都有一个简单的认识，可以看下这篇文章为什么开发人员需要了解数据库锁。有了上面的经验之后，本以为对于死锁都能手到擒来，没想到再一个阳光明媚的下午报出了一个死锁，但是这一次却没想象的那么简单。

## 问题初现

在某天下午，突然系统报警，抛出个异常：

仔细一看好像是事务回滚异常，写着的是因为死锁回滚，原来是个死锁问题，由于我对Mysql锁还是有一定了解的，于是开始主动排查这个问题。

首先在数据库中查找Innodb Status，在Innodb Status中会记录上一次死锁的信息,输入下面命令：

SHOW ENGINE INNODB STATUS

死锁信息如下,sql信息进行了简单处理：

------------------------

LATEST DETECTED DEADLOCK

------------------------

2019-02-22 15:10:56 0x7eec2f468700

\*\*\* (1) TRANSACTION:

TRANSACTION 2660206487, ACTIVE 0 sec starting index read

mysql tables in use 1, locked 1

LOCK WAIT 2 lock struct(s), heap size 1136, 1 row lock(s)

MySQL thread id 31261312, OS thread handle 139554322093824, query id 11624975750 10.23.134.92 erp\_crm\_\_6f73 updating

/\*id:3637ba36\*/UPDATE tenant\_config SET

open\_card\_point = 0

where tenant\_id = 123

\*\*\* (1) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 1322 page no 534 n bits 960 index uidx\_tenant of table `erp\_crm\_member\_plan`.`tenant\_config` trx id 2660206487 lock\_mode X locks rec but not gap waiting

\*\*\* (2) TRANSACTION:

TRANSACTION 2660206486, ACTIVE 0 sec starting index read

mysql tables in use 1, locked 1

3 lock struct(s), heap size 1136, 2 row lock(s)

MySQL thread id 31261311, OS thread handle 139552870532864, query id 11624975758 10.23.134.92 erp\_crm\_\_6f73 updating

/\*id:3637ba36\*/UPDATE tenant\_config SET

open\_card\_point = 0

where tenant\_id = 123

\*\*\* (2) HOLDS THE LOCK(S):

RECORD LOCKS space id 1322 page no 534 n bits 960 index uidx\_tenant of table `erp\_crm\_member\_plan`.`tenant\_config` trx id 2660206486 lock mode S

\*\*\* (2) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 1322 page no 534 n bits 960 index uidx\_tenant of table `erp\_crm\_member\_plan`.`tenant\_config` trx id 2660206486 lock\_mode X locks rec but not gap waiting

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

------------

给大家简单的分析解释一下这段死锁日志，事务1执行Update语句的时候需要获取uidx\_tenant这个索引再where条件上的X锁(行锁)，事务2执行同样的Update语句，也在uidx\_tenant上面想要获取X锁(行锁)，然后就出现了死锁，回滚了事务1。当时我就很懵逼，回想了一下死锁产生的必要条件:

1、互斥。

2、请求与保持条件。

3、不剥夺条件。

4、循环等待。

从日志上来看事务1和事务2都是取争夺同一行的行锁，和以往的互相循环争夺锁有点不同，怎么看都无法满足循环等待条件。经过同事提醒，既然从死锁日志中不能进行排查，那么就只能从业务代码和业务日志从排查。这段代码的逻辑如下：

public int saveTenantConfig(PoiContext poiContext, TenantConfigDO tenantConfig) {

try {

return tenantConfigMapper.saveTenantConfig(poiContext.getTenantId(), poiContext.getPoiId(), tenantConfig);

} catch (DuplicateKeyException e) {

LOGGER.warn("[saveTenantConfig] 主键冲突，更新该记录。context:{}, config:{}", poiContext, tenantConfig);

return tenantConfigMapper.updateTenantConfig(poiContext.getTenantId(), tenantConfig);

}

}

这段代码的意思是保存一个配置文件，如果发生了唯一索引冲突那么就会进行更新，当然这里可能写得不是很规范，其实可以用

insert into ...

on duplicate key update

也可以达到同样的效果，但是就算用这个其实也会发生死锁。看了代码之后同事又给我发了当时业务日志,

可以看见这里有三条同时发生的日志，说明都发生了唯一索引冲突进入了更新的语句，然后发生的死锁。到这里答案终于稍微有点眉目了。

这个时候再看我们的表结构如下(做了简化处理)：

CREATE TABLE `tenant\_config` (

`id` bigint(21) NOT NULL AUTO\_INCREMENT,

`tenant\_id` int(11) NOT NULL,

`open\_card\_point` int(11) DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `uidx\_tenant` (`tenant\_id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 ROW\_FORMAT=COMPACT

我们的tenant\_id是用来做唯一索引，我们的插入和更新的where条件都是基于唯一索引来操作的。

UPDATE tenant\_config SET

open\_card\_point = 0

where tenant\_id = 123

到了这里感觉插入的时候对唯一索引加锁有关系，接下来我们进行下一步的深入剖析。

## 深入剖析

上面我们说有三个事务进入update语句，为了简化说明这里我们只需要两个事务同时进入update语句即可，下面的表格展示了我们整个的发生过程：

小提示:S锁是共享锁，X锁是互斥锁。一般来说X锁和S，X锁都互斥，S锁和S锁不互斥。

我们从上面的流程中看见发生这个死锁的关键需要获取S锁，为什么我们再插入的时候需要获取S锁呢？因为我们需要检测唯一索引？在RR隔离级别下如果要读取那么就是当前读,那么其实就需要加上S锁。这里发现唯一键已经存在，这个时候执行update就会被两个事务的S锁互相阻塞，从而形成上面的循环等待条件。

小提示: 在MVCC中，当前读和快照读的区别:当前读每次需要加锁（可以使共享锁或者互斥锁）获取到最新的数据，而快照读是读取的是这个事务开始的时候那个快照，这个是通过undo log去进行实现的。

这个就是整个死锁的原因，能出现这种死锁的还有一个情况，就是同一时间来三个插入操作，其中先插入的那个事务如果最后回滚了，其余两个事务也会出现这种死锁。

## 解决方案

这里的核心问题是需要把S锁给干掉，这里有三个可供参考的解决方案:

将RR隔离级别，降低成RC隔离级别。这里RC隔离级别会用快照读，从而不会加S锁。

再插入的时候使用select \* for update,加X锁，从而不会加S锁。

可以提前加上分布式锁，可以利用Redis,或者ZK等等，分布式锁可以参考我的这篇文章。聊聊分布式锁

第一种方法不太现实，毕竟隔离级别不能轻易的修改。第三种方法又比较麻烦。所以第二种方法是我们最后确定的。

## 总结

说了这么多，最后做一个小小的总结吧。排查死锁这种问题的时候有时候光看死锁日志有时候会解决不了问题，需要结合整个的业务日志，代码以及表结构来进行分析，才能得到正确的结果。当然上面有一些数据库锁的基本知识如果不了解可以查看我的另一篇文章为什么开发人员需要了解数据库锁。

最后这篇文章被我收录于JGrowing-CaseStudy篇，一个全面，优秀，由社区一起共建的Java学习路线，如果您想参与开源项目的维护，可以一起共建，github地址为：https://github.com/javagrowing/JGrowing