本文档截取自电子工业出版社出版的图书《千金良方：MySQL 性能优化金字塔法则》的附录部分，该图书由来自沃趣科技的李春、罗小波、董红禹三位作者共同撰写。更多精彩内容可关注下方的微信公众号 "**沃趣技术**"与"**知数堂**"


附录 B show engine innodb status 详解

注：本文以 MySQL 5.5 为主，MySQL 5.6 与 MySQL 5.7 为辅进行编写，所以对于 MySQL 5.6 与 MySQL 5.7 的内容介绍并不完整，有兴趣的同学可自行研究。

- innodb 存储引擎在 show engine innodb status（老版本对应的是 show innodb status）输出中，其中包含了 innodb 引擎大量的内部状态信息，它输出就是一个单独的字符串，没有行和列，内容分为很多小段，每一段对应 innodb 存储引擎不同部分的信息，总体包含两种类型的 InnoDB Monitor，如下
    - 标准的 InnoDB Monitor 状态，包含以下类型的信息：
        * 主线程在后台完成的工作状态信息
        * 信号量等待信息
        * 有关最新外键和死锁错误的数据信息
        * 事务的锁等待信息
        * 活跃事务持有的表锁、记录锁信息
        * 待处理(挂起)的 I/O 操作和相关统计信息
        * 插入缓冲区和自适应哈希索引统计信息
        * 重做日志信息

        * 缓冲池统计信息
        * 行操作统计信息
    - InnoDB 锁状态，这些锁信息也会一并打印到标准 InnoDB Monitor 状态输出中
- 注：
    - 输出内容中包含了一些平均值的统计信息，这些平均值是自上次输出结果生成以来的统计数，因此，如果你正在检查这些值，那就要确保已经等待了至少 30s 的时间，使两次采样之间积累足够长的统计时间并多次采样，检查计数器变化从而弄清其行为，并不是所有的输出都会在一个时间点上生成，因而也不是所有的显示出来的平均值会在同一时间间隔里重新再计算。而且，innodb 有一个内部复位间隔，而它是不可预知的，各个版本也不一样。
    - 这些输出信息足够提供大多数你想要的统计信息，有一款监控工具 innotop 可以帮你计算出增量差值和平均值。下面，在你的 mysql 命令行敲下 show engine innodb status;看着输出跟着下面的步骤一步一步理解输出信息是什么含义
    - 以下使用 mysql5.5.24 版本做解读，mysql5.6.x 和 5.7.x 输出内容有部分有调整。

1、启用 InnoDB Monitor 定期输出

- 当 InnoDB Monitor 启用了定期输出时，InnoDB 每隔 15 秒将输出信息写入 mysqld 标准错误输出中(error log)，innodb_status_output 和 innodb_status_output_locks 系统变量分别用于启用标准 InnoDB Monitor 和 InnoDB Lock Monitor。例如：

```
# 启用/禁用 标准 InnoDB Monitor，通过将 innodb_status_output 系统变量设置为 ON 来启用，设置为 OFF 来禁用
SET GLOBAL innodb_status_output = ON/OFF;
```

```
# 启用/禁用 InnoDB Lock Monitor（InnoDB Lock Monitor 使用 InnoDB Standard Monitor 输出打印，必须启用 InnoDB 标准 Monitor 和 InnoDB Lock Monitor 才能定期打印 InnoDB Lock Monitor 数据）
SET GLOBAL innodb_status_output = ON/OFF;
SET GLOBAL innodb_status_output_locks = ON/OFF;

# 注，如果要在使用 SHOW ENGINE INNODB STATUS 时输出 InnoDB Lock Monitor 数据，只需启用 innodb_status_output_locks=ON 即可（无需启用 innodb_status_output = ON）
```

- 可以将标准 InnoDB Monitor 输出定向到状态文件
  - 在启动 Server 时指定--innodb-status-file 选项，可以将标准 InnoDB Monitor 输出并将其定向到状态文件中。使用此选项时，InnoDB 会在数据目录中创建一个名为 innodb_status.*pid*(pid 为 MySQL Server 的进程号)的文件，并每隔 15 秒将输出写入其中。当 Server 正常关闭时，InnoDB 会删除状态文件。但如果发生异常关闭，则可能未成功删除该文件，需要手动删除。
  - PS：--innodb-status-file 选项用于临时使用，由于 SHOW ENGINE INNODB STATUS 输出可能会影响性能，使用完之后请及时关闭

- 使用如下语句登录到数据库中执行查询

```
root@localhost : (none):26: > show engine innodb status\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
```

```
2018-05-25 23:28:20 0x7f1e1c242700 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 49 seconds
-----------------
BACKGROUND THREAD
-----------------
srv_master_thread loops: 7716 srv_active, 0 srv_shutdown, 36436 srv_idle
srv_master_thread log flush and writes: 0
----------
SEMAPHORES
----------
......
```

- PS：
  - 除了使用系统配置参数来启用定期输出之外，也可以使用计划任务结合 SHOW ENGINE INNODB STATUS 语句来定期获取 InnoDB Monitor 输出
  - 在 Windows 上，除非单独配置(log-error 选项)，否则 stderr 将定向到默认的日志文件。如果要将输出定向到控制台窗口而不是错误日志，则需要在控制台窗口(命令提示符)中使用--console 选项启动 Server
  - 在 Unix 和类 Unix 系统上，除非单独配置(log-error 选项)，否则 stderr 通常会定向到终端中
  - 启用或禁用 InnoDB Monitor 功能需要操作用户具有 PROCESS 权限

2、innodb status 分段详解

2.1. status 头部

- 即第一段信息，是头部信息，它仅仅声明了输出的开始，其内容包括当前的日期和时间，以及自上次输出以来经过的时长

```
=====================================
160129 12:07:26 INNODB MONITOR OUTPUT   #第二行是当前日期和时间
=====================================
Per second averages calculated from the last 24 seconds #第四行显示的是计算出这一平均值的时间间隔，即自上次输出以来的时间，或者是距上次内部复位的时长
```

- PS：从 innodb1.0.x 开始，可以使用命令 show engine innodb status;来查看 master thread 的状态信息:

2.2. BACKGROUND THREAD

- 第二段信息部分，显示主线程已经完成的工作统计信息

```
----------------
BACKGROUND THREAD
----------------
srv_master_thread loops: 30977173 1_second, 30975685 sleeps, 3090359 10_second, 166112 background, 165988 flush #这行显示主循环进行了 30977173 1_second 次，每秒挂起的操作进行了 30975685 sleeps 次（说明负载不是很大），10 秒一次的活动进行了 3090359 10_second 次，1 秒循环和 10 秒循环比值符合 1：10，backgroup loop 进行了 166112 次 background，flush loop 进行了 165988 次 flush，如果在一台很大压力的 mysql 上，可能看到每秒运行次数和挂起次数比例远小于 1，这是因为 innodb 对内部进行了一些优化，在高压力下，间隔时间并不总是 1 秒，因此，不能认为每秒循环和挂起的值总是相等，在某些情况下，可以通过两者之间的差值来比较反映当前数据库的负载压力。

srv_master_thread log flush and writes: 31160103
```

2.3. SEMAPHORES

- 第三段信息部分
    - 如果有高并发的工作负载，你就要关注下接下来的段（SEMAPHORES 信号量）,它包含了两种数据：事件计数器以及

可选的当前等待线程的列表，如果有性能上的瓶颈，可以使用这些信息来找出瓶颈，不幸的是，想知道怎么使用这些信息还是有一点复杂，下面先给出一些解释：

```
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 68581015, signal count 218437328
--Thread 140653057947392 has waited at btr0pcur.c line 437 for 0.00 seconds the
semaphore:
S-lock on RW-latch at 0x7ff536c7d3c0 created in file buf0buf.c line 916
a writer (thread id 140653057947392) has reserved it in mode   exclusive
number of readers 0, waiters flag 1, lock_word: 0
Last time read locked in file row0sel.c line 3097
Last time write locked in file /usr/local/src/soft/mysql-
5.5.24/storage/innobase/buf/buf0buf.c line 3151
--Thread 140653677291264 has waited at btr0pcur.c line 437 for 0.00 seconds the
semaphore:
S-lock on RW-latch at 0x7ff53945b240 created in file buf0buf.c line 916
a writer (thread id 140653677291264) has reserved it in mode   exclusive
number of readers 0, waiters flag 1, lock_word: 0
Last time read locked in file row0sel.c line 3097
Last time write locked in file /usr/local/src/soft/mysql-
5.5.24/storage/innobase/buf/buf0buf.c line 3151
Mutex spin waits 1157217380, rounds 1783981614, OS waits 10610359
RW-shared spins 103830012, rounds 1982690277, OS waits 52051891
RW-excl spins 43730722, rounds 602114981, OS waits 3495769
Spin rounds per wait: 1.54 mutex, 19.10 RW-shared, 13.77 RW-excl

# 内容比较多，下面分段依次解释：
OS WAIT ARRAY INFO: reservation count 68581015, signal count 218437328   #这行给
出了关于操作系统等待数组的信息，它是一个插槽数组，innodb 在数组里为信号量保留了一
```

些插槽，操作系统用这些信号量给线程发送信号，使线程可以继续运行，以完成它们等着做的事情，这一行还显示出 innodb 使用了多少次操作系统的等待：保留统计（reservation count）显示了 innodb 分配插槽的频度，而信号计数（signal count）衡量的是线程通过数组得到信号的频度，操作系统的等待相对于空转等待（spin wait）要更昂贵。

--Thread 140653057947392 has waited at btr0pcur.c line 437 for 0.00 seconds the semaphore:
S-lock on RW-latch at 0x7ff536c7d3c0 created in file buf0buf.c line 916
a writer (thread id 140653057947392) has reserved it in mode exclusive
number of readers 0, waiters flag 1, lock_word: 0
Last time read locked in file row0sel.c line 3097
Last time write locked in file /usr/local/src/soft/mysql-5.5.24/storage/innobase/buf/buf0buf.c line 3151
--Thread 140653677291264 has waited at btr0pcur.c line 437 for 0.00 seconds the semaphore:
S-lock on RW-latch at 0x7ff53945b240 created in file buf0buf.c line 916
a writer (thread id 140653677291264) has reserved it in mode exclusive
number of readers 0, waiters flag 1, lock_word: 0
Last time read locked in file row0sel.c line 3097
Last time write locked in file /usr/local/src/soft/mysql-5.5.24/storage/innobase/buf/buf0buf.c line 3151

这部分显示的是当前正在等待互斥量的 innodb 线程，在这里可以看到有两个线程正在等待，每一个都是以--Thread <数字> has waited...开始，这一段内容在正常情况下应该是空的（即查看的时候没有这部分内容），除非服务器运行着高并发的工作负载，促使 innodb 采取让操作系统等待的措施，或者你对 innodb 源码熟悉，那么这里看到的最有用的信息就是发生线程等待的代码文件名 /usr/local/src/soft/mysql-5.5.24/storage/innobase/buf/buf0buf.c line 3151。

在 innodb 内部哪里才是热点？举例来说，如果看到许多线程都在一个名为 buf0buf.c 的文件上等待，那就意味着你的系统里存在着缓冲池竞争，这个输出信息还显示了这些线程等待了多少时间，其中 waiters flag 显示了有多少个等待者正在等待同一个互斥量。 如果 waiters

flag 为 0 那就表示没有线程在等待同一个互斥量（此时在 waiters flag 0 后面可能可以看到 wait is ending，表示这个互斥量已经被释放了，但操作系统还没有把线程调度过来运行）。

你可能想知道 innodb 真正等待的是什么，innodb 使用了互斥量和信号量来保护代码的临界区，如：限定每次只能有一个线程进入临界区，或者是当有活动的读时，就限制写入等。在 innodb 代码里有很多临界区，在合适的条件下，它们都可能出现在那里，常常能见到的一种情形是：获取缓冲池分页的访问权的时候。

在等待线程之后的部分信息如下，这部分显示了更多的事件计数器，在每一个情形中，都能看到 innodb 依靠操作系统等待的频度：

Mutex spin waits 1157217380, rounds 1783981614, OS waits 10610359 #这行显示的是跟互斥量相关的几个计数器

RW-shared spins 103830012, rounds 1982690277, OS waits 52051891 #这行显示读写的共享锁的计数器

RW-excl spins 43730722, rounds 602114981, OS waits 3495769 #这行显示读写的排他锁的计数器

Spin rounds per wait: 1.54 mutex, 19.10 RW-shared, 13.77 RW-excl

innodb 有着一个多阶段等待的策略，首先，它会试着对锁进行空转等待，如果经历了一个预设的空转等待周期（设置 innodb_sync_spin_loops 配置变量命令）之后还没有成功，那就会退到更昂贵更复杂的等待数组中。空转等待的成本相对较低，但是它们要不停地检查一个资源能否被锁定，这种方式会消耗 CPU 周期，但是，这没有听起来那么糟糕，因为当处理器在等待 I/O 时，一般都有一些空闲的 CPU 周期可用，即使是没有空闲的 CPU 周期，空等也要比其他方式更加廉价一些。然而，当另外一个线程能做一些事情的时候，空转等待也还会把 CPU 独占着。空转等待的替代方案就是让操作系统做上下文切换，这样，当一个线程在等待时，另外一个线程就可以被运行，然后，通过等待数组里的信号量发出信号，唤醒那个沉睡的线程，通过信号量来发送信号是比较有效的，但是上下文切换就很昂贵，这很快就会积少成多，每秒钟几千次的切换会产生大量的系统开销。

你可以通过修改 innodb_sync_spin_loops 的值，试着在空转等待与操作系统等待之间达成平衡，不要担心空转等待，除非你在一秒里看到几十万个空转等待。此时，你可以考虑 performance_schema 库或者 show engine innodb mutex;查看下相关信息。

- PS：如果出现大量等待信号量的线程，需要引起足够重视，可能是由于磁盘 I/O 性能不足或 InnoDB 引起内部资源争用问题的导致（争用可能是由于并发查询量过大或操作系统线程调度问题）。通过增大系统配置变量 innodb_thread_concurrency 的值可能会有所帮助，如果需要查询互斥量信息，可以通过 SHOW ENGINE INNODB MUTEX 语句查看。

2.4. LATEST FOREIGN KEY ERROR

- 第四段信息，这一段外键错误的信息一般不会出现，除非你服务器上发生了外键错误，有时问题在于事务在插入，更新或删除一条记录时要寻找父表或子表，还有时候是当 innodb 尝试增加或删除一个外键或者修改一个已经存在的外键时，发现表之间类型不匹配，这部分输出对于调试与 innodb 不明确的外键错误发生的准确原因非常有帮助，下面搞一个示例来看看：

```
# 创建父表：
mysql> create table parent(parent_id int not null,primary key(parent_id))
engine=innodb;

# 创建子表：
mysql> create table child(child_id int not null,key child_id(child_id),constraint
i_child foreign key(child_id) references parent(parent_id)) engine=innodb;

# 插入数据：
mysql> insert into parent(parent_id) values(1);
mysql> insert into child(child_id) values(1);
```

- 有两种基本的外键错误：
  第一种：以某种可能违反外键约束关系的方法插入，更新，删除数据，将导致第一类错误，如，在父表中删除行时发生如下错误：

mysql> **delete from parent;**

ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails (`xiaoboluo`.`child`, CONSTRAINT `i_child` FOREIGN KEY (`child_id`) REFERENCES `parent` (`parent_id`)) # 错误信息相当明了，对所有由插入，删除，更新不匹配的行导致的错误都会看到相似的信息，下面是 show engine innodb status 的输出：

- 在 show engine innodb status 输出中可以看到类似如下信息

```
------------------------
LATEST FOREIGN KEY ERROR
------------------------
160128  1:17:06 Transaction:  #这行显示了最近一次外键错误的日期和时间
TRANSACTION D203D6, ACTIVE 0 sec updating or deleting
mysql tables in use 1, locked 1
4 lock struct(s), heap size 1248, 2 row lock(s), undo log entries 1
MySQL thread id 20027, OS thread handle 0x7f0a4c0f8700, query id 1813996 localhost root updating
delete from parent
Foreign key constraint fails for table `xiaoboluo`.`child`:

#上面部分显示了关于破坏外键约束的事务详情。后边部分显示了发现错误时 innodb 正尝试修改的准确数据，输出中有许多是转换成可打印格式的行数据。

CONSTRAINT `i_child` FOREIGN KEY (`child_id`) REFERENCES `parent` (`parent_id`)
Trying to delete or update in parent table, in index `PRIMARY` tuple:
DATA TUPLE: 3 fields;
0: len 4; hex 80000001; asc     ;;
1: len 6; hex 000000d203d6; asc         ;;
2: len 7; hex 1e000001ca0110; asc          ;;



But in child table `xiaoboluo`.`child`, in index `child_id`, there is a record:
PHYSICAL RECORD: n_fields 2; compact format; info bits 0
```

```
0: len 4; hex 80000001; asc       ;;
1: len 6; hex 000013a99b3e; asc        >;;
```

- 第二种：尝试修改父表的表结构时发生的错误，这种错误就没有那么清楚了，这可能会让调试更困难：

```
mysql> alter table parent modify parent_id int unsigned not null;
ERROR 1025 (HY000): Error on rename of './xiaoboluo/#sql-b695_4e3b' to
'./xiaoboluo/parent' (errno: 150)
```

- 查看 show engine innodb status 输出信息：

```
------------------------
LATEST FOREIGN KEY ERROR
------------------------
160128  1:32:33 Error in foreign key constraint of table xiaoboluo/child:
there is no index in referenced table which would contain
the columns as the first columns, or the data types in the
referenced table do not match the ones in table. Constraint:
,
CONSTRAINT "i_child" FOREIGN KEY ("child_id") REFERENCES "parent" ("parent_id")
The index in the foreign key in table is "child_id"
See http://dev.mysql.com/doc/refman/5.5/en/innodb-foreign-key-constraints.html
for correct foreign key definition.
InnoDB: Renaming table `xiaoboluo`.<result 2 when explaining filename '#sql-
b695_4e3b'> to `xiaoboluo`.`parent` failed!

# 上面的错误是数据类型不匹配，外键列必须有完全相同的数据类型，包括任何的修饰符（如这里父表多加了一个 unsigned，这也是问题所在），当看到 1025 错误并不理解为什么时，最好查看下 innodb status。在每次看到有新错误时，外键错误信息都会被重写，percona toolkit 中的 pt-fk-error-logger 工具可以用表保存这些信息以供后续分析。
```

2.5. LATEST DETECTED DEADLOCK

- 第五段信息
    - 与外键错误一样，这部分只有当服务器产生死锁时才会出现，死锁信息同样在每次有新的死锁错误时被重写，percona toolkit 中的 pt-deadlock-logger 工具可以用表保存这些信息以供后续分析死锁在等待关系图里是一个循环，就是一个锁定了行的数据结构又在等待别的锁，这个循环可以任意地大，innodb 会立即检测到死锁，因为每当有事务等待行锁的时候，它都会去检查等待关系图里是否有循环，死锁的情况可能会比较复杂，但是，这一部分只显示了最近的两个死锁的情况，它们在各自的事务里执行的最后一条语句，以及它们在等待关系图里形成环锁的信息。在这个循环里你看不到其他事务，也可能看不到在事务里早先真正获得了锁的语句，尽管如此，通常还是可以通过查看这些输出结果来确定到底是什么引起了死锁。
    - 在 innodb 里实际上有两种死锁，第一种就是常常碰到的那种，它在等待关系图里是一个真正的循环，另外一种就是在一个等待关系图里，因代价昂贵而无法检测它是不是包含了循环，如果 innodb 要在关系图里检查超过 100W 个锁，或者在检查过程中，innodb 要重做 200 个以上的事务，那它会放弃，并宣布这里有一个死锁，这些数值都是硬编码在 innodb 代码里的常量，无法配置（如果你 NB 可以修改代码然后重新编译）。第二种死锁报错你可以在输出里看到一条信息：TOO DEEP OR LONG SEARCH IN THE LOCK TABLE WAITS-FOR GRAPH
    - innodb 不仅会打印出事务和事务持有和等待的锁，而且还有记录本身，不幸的是，它至于可能超过为输出结果预留的长度（只能打印 1M 的内容且只能保留最近一次的死锁信息），如果你无法看到完整的输出，此时可以在任意库下创建 innodb_monitor 或 innodb_lock_monitor 表，这样 innodb status 信息会完整且每 15s 一次被记录到错误日志中。如：create table innodb_monitor(a int)engine=innodb;，不需要记录到错误日志中时就删掉这个表即可（注：在 MySQL 5.6.x 以上的版本中，

要记录 innodb status 信息使用 innodb_status_output 开关变量，要记录 innodb lock status 信息使用 innodb_status_output_locks 开关变量）。

- 下面也搞一个示例来看看：

```
# 建表：
mysql> create table test_deadlock(id int unsigned not null primary key
auto_increment,test int unsigned not null);
Query OK, 0 rows affected (0.02 sec)

# 插入测试数据：
mysql> insert into test_deadlock(test) values(1),(2),(3),(4),(5);
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

# 打开两个会话终端：
## 会话 1 执行下面的 SQL：
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test_deadlock where id=1 for update;
+----+------+
| id | test |
+----+------+
|  1 |    1 |
+----+------+

1 row in set (0.00 sec)

## 接着会话 2 执行下面的 SQL：
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from test_deadlock where id=2 for update;
+----+------+
| id | test |
+----+------+
|  2 |    2 |
+----+------+

1 row in set (0.00 sec)

## 回到会话 1 执行下面的 SQL，会发生等待：
mysql> select * from test_deadlock where id=2 for update;

## 回到会话 2 执行下面的 SQL，产生死锁，会话 2 被回滚：
mysql> select * from test_deadlock where id=1 for update;
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting
transaction
```

- 查看 innodb status 信息：

```
------------------------
LATEST DETECTED DEADLOCK
------------------------
160128  1:51:53  #这里显示了最近一次发生死锁的日期和时间
*** (1) TRANSACTION:
TRANSACTION D20847, ACTIVE 141 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 20027, OS thread handle 0x7f0a4c0f8700, query id 1818124 localhost
root statistics
select * from test_deadlock where id=2 for update
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
```

```
RECORD LOCKS space id 441 page no 3 n bits 72 index `PRIMARY` of table
`xiaoboluo`.`test_deadlock` trx id D20847 lock_mode X locks rec but not gap waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
0: len 4; hex 00000002; asc     ;;
1: len 6; hex 000000d20808; asc         ;;
2: len 7; hex ad000001ab011d; asc          ;;
3: len 4; hex 00000002; asc     ;;

*** (2) TRANSACTION:
TRANSACTION D20853, ACTIVE 119 sec starting index read
mysql tables in use 1, locked 1
3 lock struct(s), heap size 1248, 2 row lock(s)
MySQL thread id 20081, OS thread handle 0x7f0a0f020700, query id 1818204
localhost root statistics
select * from test_deadlock where id=1 for update
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 441 page no 3 n bits 72 index `PRIMARY` of table
`xiaoboluo`.`test_deadlock` trx id D20853 lock_mode X locks rec but not gap
Record lock, heap no 3 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
0: len 4; hex 00000002; asc     ;;
1: len 6; hex 000000d20808; asc         ;;
2: len 7; hex ad000001ab011d; asc          ;;
3: len 4; hex 00000002; asc     ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 441 page no 3 n bits 72 index `PRIMARY` of table
`xiaoboluo`.`test_deadlock` trx id D20853 lock_mode X locks rec but not gap waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
0: len 4; hex 00000001; asc     ;;
1: len 6; hex 000000d20808; asc         ;;
2: len 7; hex ad000001ab0110; asc          ;;
3: len 4; hex 00000001; asc     ;;
```

*** WE ROLL BACK TRANSACTION (2)

- 这部分内容比较多，下面分段逐一进行解释：

下面这部分显示的是死锁的第一个事务的信息：
*** (1) TRANSACTION:
TRANSACTION D20847, ACTIVE 141 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 20027, OS thread handle 0x7f0a4c0f8700, query id 1818124 localhost root statistics
select * from test_deadlock where id=2 for update
TRANSACTION D20847, ACTIVE 141 sec starting index read：这行表示事务 D20847，ACTIVE 141 sec 表示事务处于活跃状态 141s，starting index read 表示正在使用索引读取数据行。注："D20847"是一个十六进制的字符串，可以使用 select conv('D20847',16,10);语句将十六进制的字符串转换为十进制的数字（转换结果为 13764679）。在较新的版本中，事务 ID 可能会直接使用十进制的数字，而不是使用十六进制的字符串
mysql tables in use 1, locked 1#这行表示事务 D20847 正在使用 1 个表，且涉及锁的表有 1 个

LOCK WAIT 3 lock struct(s), heap size 376, 2 row lock(s) #这行表示在等待 3 把锁，占用内存 376 字节，涉及 2 行记录，如果事务已经锁定了几行数据，这里将会有一行信息显示出锁定结构的数目（注意，这跟行锁是两回事）和堆大小，堆的大小指的是为了持有这些行锁而占用的内存大小，Innodb 使用一种特殊的位图来表示锁具体锁住了哪几行，从理论上讲，它可将每一个锁定的行表示为一个比特，经测试证明，每个锁通常不超过 4 比特

MySQL thread id 20027, OS thread handle 0x7f0a4c0f8700, query id 1818124 localhost root statistics #这行表示该事务的线程 ID 信息，操作系统句柄信息，连接来源、用户

select * from test_deadlock where id=2 for update #这行表示事务涉及的 SQL

下面这一部分显示的是当死锁发生时，第一个事务正在等待的锁等信息：

*** (1) WAITING FOR THIS LOCK TO BE GRANTED: #这行信息表示第一个事务正在等待锁被授予

RECORD LOCKS space id 441 page no 3 n bits 72 index `PRIMARY` of table `xiaoboluo`.`test_deadlock` trx id D20847 lock_mode X locks rec but not gap waiting

Record lock, heap no 3 PHYSICAL RECORD: n_fields 4; compact format; info bits 0

0: len 4; hex 00000002; asc ;;

1: len 6; hex 000000d20808; asc ;;

2: len 7; hex ad000001ab011d; asc ;;

3: len 4; hex 00000002; asc ;;

RECORD LOCKS space id 441 page no 3 n bits 72 index `PRIMARY` of table `xiaoboluo`.`test_deadlock` trx id D20847 lock_mode X locks rec but not gap waiting# 这行信息表示等待的锁是一个 record lock，空间 id 是 441，页编号为 3，大概位置在页的 72 位处，锁发生在表 xiaoboluo.test_deadlock 的主键上，是一个 X 锁，但是不是 gap lock。waiting 表示正在等待锁

Record lock, heap no 3 PHYSICAL RECORD: n_fields 4; compact format; info bits 0 #这行表示 record lock 的 heap no 位置

这部分剩下的内容只对调试才有用。

0: len 4; hex 00000002; asc ;;

1: len 6; hex 000000d20808; asc ;;

2: len 7; hex ad000001ab011d; asc ;;

3: len 4; hex 00000002; asc ;;

下面这部分是事务二的状态：

*** (2) TRANSACTION:

TRANSACTION D20853, ACTIVE 119 sec starting index read #事务 2 处于活跃状态 119s

mysql tables in use 1, locked 1 #正在使用 1 个表，涉及锁的表有 1 个

3 lock struct(s), heap size 1248, 2 row lock(s) #涉及 3 把锁，2 行记录

MySQL thread id 20081, OS thread handle 0x7f0a0f020700, query id 1818204 localhost root statistics

```
select * from test_deadlock where id=1 for update #第二个事务的 SQL

# 下面这部分是事务二的持有锁信息：
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 441 page no 3 n bits 72 index `PRIMARY` of table
`xiaoboluo`.`test_deadlock` trx id D20853 lock_mode X locks rec but not gap
Record lock, heap no 3 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
0: len 4; hex 00000002; asc     ;;
1: len 6; hex 000000d20808; asc      ;;
2: len 7; hex ad000001ab011d; asc       ;;
3: len 4; hex 00000002; asc     ;;

RECORD LOCKS space id 441 page no 3 n bits 72 index `PRIMARY` of table
`xiaoboluo`.`test_deadlock` trx id D20853 lock_mode X locks rec but not gap
Record lock, heap no 3 PHYSICAL RECORD: n_fields 4; compact format; info bits 0 #从
这两行持有锁信息计息后面几行调试信息上看，就是事务 1 正在等待的锁。

# 下面这部分是事务二正在等待的锁，从下面的信息上看，等待的是同一个表，同一个索引，
同一个 page 上的 record lock X 锁，但是 heap no 位置不同，即不同的行上的锁：
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 441 page no 3 n bits 72 index `PRIMARY` of table
`xiaoboluo`.`test_deadlock` trx id D20853 lock_mode X locks rec but not gap waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
0: len 4; hex 00000001; asc     ;;
1: len 6; hex 000000d20808; asc      ;;
2: len 7; hex ad000001ab0110; asc       ;;
3: len 4; hex 00000001; asc     ;;

*** WE ROLL BACK TRANSACTION (2) #这个表示事务 2 被回滚，因为两个事务的回滚开销
一样，所以选择了后提交的事务进行回滚，如果两个事务回滚的开销不同(undo 数量不同)，
那么就回滚开销最小的那个事务。当一个事务持有了其他事务需要的锁，同时又想获得其他事
务持有的锁时，等待关系图上就会产生循环，Innodb 不会显示所有持有和等待的锁，但是，
```

它显示了足够的信息来帮你确定，查询操作正在使用哪些索引，这对于你确定能否避免死锁有极大的价值。

如果能使两个查询对同一个索引朝同一个方向进行扫描，就能降低死锁的数目，因为，查询在同一个顺序上请求锁的时候不会创建循环，有时候，这是很容易做到的，如：要在一个事务里更新许多条记录，就可以在应用程序的内存里把它们按照主键进行排序，然后，再用同样的顺序更新到数据库里，这样就不会有死锁发生，但是在另一些时候，这个方法也是行不通的（如果有两个进程使用了不同的索引区间操作同一张表的时候）。

2.6. TRANSACTIONS

- 第六段信息
    - 下面这部分包含了一些关于 innodb 事务的总结信息，紧随其后的是当前活跃事务列表（如果在这里看到了锁等待细细，则表明您的应用程序可能发生了锁争用。这些锁等待输出信息可以帮助您跟踪事务死锁的原因）。如：

```
------------
TRANSACTIONS
------------
Trx id counter 4E0132AD
Purge done for trx's n:o < 4E01090B undo n:o < 0
History list length 1853

LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 4E0131D3, not started
MySQL thread id 26208218, OS thread handle 0x7fec7c582700, query id 5274800318
10.207.162.69 gdsser
---TRANSACTION 4E01323F, not started
MySQL thread id 26208217, OS thread handle 0x7fec7c1b3700, query id 5274800938
10.207.162.69 gdsser
```

```
.....

---TRANSACTION 4E0132AC, ACTIVE 0 sec preparing
2 lock struct(s), heap size 376, 1 row lock(s), undo log entries 1
MySQL thread id 26208200, OS thread handle 0x7fec567e0700, query id 5274801557
10.207.162.69 gdsser
commit
---TRANSACTION 4E0110E7, ACTIVE 188 sec
mysql tables in use 1, locked 0
MySQL thread id 26208154, OS thread handle 0x7fec7c235700, query id 5274800671
10.143.90.228 root Sending data
SELECT /*!40001 SQL_NO_CACHE */ * FROM `m_flowskillpoint`
Trx read view will not see trx with id >= 4E0110E8, sees < 4E0108EE
---TRANSACTION 4E0108EF, ACTIVE 233 sec fetching rows
mysql tables in use 1, locked 0
MySQL thread id 26208131, OS thread handle 0x7fec578e3700, query id 5274801341
10.143.90.228 root Sending data
SELECT /*!40001 SQL_NO_CACHE */ * FROM `m_flowsilver`
Trx read view will not see trx with id >= 4E0108F0, sees < 4E0108EC
---TRANSACTION 4E0108EE, ACTIVE 233 sec fetching rows
mysql tables in use 1, locked 0
MySQL thread id 26208132, OS thread handle 0x7fec7c78a700, query id 5274797797
10.143.90.228 root Sending data
SELECT /*!40001 SQL_NO_CACHE */ * FROM `m_flowmail`
Trx read view will not see trx with id >= 4E0108EF, sees < 4E0108EC
```

- 这部分内容比较多，下面分段逐一进行解释：

```
Trx id counter 4E0132AD #这行表示当前事务 ID，这是一个系统变量，每创建一个新事务都
会增加
```

Purge done for trx's n:o < 4E01090B undo n:o < 0 #这是 innodb 清除旧 MVCC 行时所用的事务 ID，将这个值和当前事务 ID 进行比较，就可以知道有多少老版本的数据未被清除。这个数字多大才是安全值没有硬性规定，如果数据没做过任何更新，那么一个巨大的数字也不意味着有未清除的数据，因为实际上所有事务在数据库里查看的都是同一个版本的数据（此时只是事务 ID 在增加，而数据没有变更），另一方面，如果有很多行被更新，那每一行就会有一个或多个版本留在内存里，减少此类开销的最好办法就是确保事务一完成就立即提交，不要让它长时间地处于打开状态，因为一个打开的事务即使不做任何操作，也会影响到 innodb 清理旧版本的行数据。 undo n:o < 0 这个是 innodb 清理进程正在使用的撤销日志编号，为 0 0 时说明清理进程处于空闲状态。

History list length 1853 #历史记录的长度，即位于 innodb 数据文件的撤销空间里的页面的数目，如果事务执行了更新并提交，这个数字就会增加，而当清理进程移除旧版本数据时，它就会减少，清理进程也会更新 Purge done for.....这行中的数值。

头部信息之后就是一个事务列表，当前版本的 mysql 还不支持嵌套事务，因此，在某个时间点上，每个客户端连接能够拥有的事务数量是有一个上限的，而且每一个事务只能属于单一连接（即一个事务只能使用单个线程执行，不能使用多个线程）。在输出信息里，每一个事务至少占有两行内容，如：

---TRANSACTION 4E0131D3, not started #每个事务的第一行以事务的 ID 和状态开始，not started 表示这个事务已经提交并且没有再发起影响事务的语句，可能刚好空闲

MySQL thread id 26208218, OS thread handle 0x7fec7c582700, query id 5274800318 10.207.162.69 gdsser #然后每个事务的第二行是一些线程等信息，MySQL thread id <数字>部分与使用 show full processlist;命令看到的 id 列相同。紧随其后的是一个内部查询 id 和一些连接信息，这些信息同样与 show full processlist 中的输出相同。

---TRANSACTION 4E01323F, not started

MySQL thread id 26208217, OS thread handle 0x7fec7c1b3700, query id 5274800938 10.207.162.69 gdsser

上面是 not started 状态的事务信息，下面来看看为 ACTIVE 状态的事务信息：

---TRANSACTION 4E0110E7, ACTIVE 188 sec #这行显示此事务处于活跃状态已经 188s，可能的所有状态有 not started，active，prepared 和 committed in memory，一旦事务日

志落盘了就会变成 not started 状态。在时间后面会显示出当前事务正在做什么（在这里为空没有显示出来），在源代码中有超过 30 个字符串常量可以显示在时间后面，如：fetching，preparing，rows，adding foreign keys 等等

mysql tables in use 1, locked 0 #该事务用到的表数和涉及表锁的表数，Innodb 一般不会锁定表，但对有些语句会锁定，如果 mysql 服务器在高于 innodb 层之上将表锁定，这里也是能够显示出来的，如果事务已经锁定了几行数据，这里将会有一行信息显示出锁定结构的数目（注意，这跟行锁是两回事）和堆大小，如：2 lock struct(s), heap size 376, 1 row lock(s), undo log entries 1，堆的大小指的是为了持有这些行锁而占用的内存大小，Innodb 是用一种特殊的位图表来实现行锁的，从理论上讲，它可将每一个锁定的行表示为一个比特，经测试显示，每个锁通常不超过 4 比特。

MySQL thread id 26208154, OS thread handle 0x7fec7c235700, query id 5274800671 10.143.90.228 root Sending data #与 show processlist 输出结果大部分相同
SELECT /*!40001 SQL_NO_CACHE */ * FROM `m_flowskillpoint` #如果事务正在运行一个查询，那么这里就会显示事务涉及的 SQL，注意：有些版本可能只显示其中一小段，而不是完整的 SQL
Trx read view will not see trx with id >= 4E0110E8, sees < 4E0108EE #这行显示了事务的读视图，它表明了因为版本关系而产生的对于事务可见和不可见两种类型的事务 ID 的范围，在这里，两个数字之间有一个事务的间隙，这个间隙里的事务可能是不可见的，innodb 在执行查询时，对于那些事务 ID 正好在这个间隙的行，还会检查其可见性。

注：如果事务正在等待一个锁，那么在查询 SQL 文本后面将可以看到这个锁的信息，在上文的死锁例子里，这样的信息看到过很多了，不幸的是，输出信息并没有说出这个锁正被其他哪个事务持有，不过可以通过 information_schema 库下的 innodb_trx，innodb_lock_waits，innodb_locks 三个表来查明这一点。如果输出信息里有很多个事务，innodb 可能会限制要打印出来的事务数目，以免输出信息增长得太大，这时就会看到...truncated...提示。

2.7. FILE I/O

- 第七段信息
  - FILE I/O 部分提供有关 InnoDB 用于执行各种类型 I/O 的线程的信息。还包括暂时挂起的 I/O 操作信息、I/O 性能的统计信息、I/O 辅助线程的状态以及性能计数器的状态信息，如下：

```
--------
FILE I/O
--------
I/O thread 0 state: waiting for i/o request (insert buffer thread)  #insert buffer thread
I/O thread 1 state: waiting for i/o request (log thread)  #log thread
I/O thread 2 state: waiting for i/o request (read thread)
I/O thread 3 state: waiting for i/o request (read thread)
I/O thread 4 state: waiting for i/o request (read thread)
I/O thread 5 state: doing file i/o (read thread) ev set  #以上为默认的 4 个 read thread
I/O thread 6 state: waiting for i/o request (write thread)
I/O thread 7 state: waiting for i/o request (write thread)
I/O thread 8 state: waiting for i/o request (write thread)
I/O thread 9 state: waiting for i/o request (write thread) #以上为默认的 4 个 write thread
Pending normal aio reads: 128 [0, 0, 0, 128] , aio writes: 0 [0, 0, 0, 0] ,  #读线程和写线程挂起操作的数目等，aio 的意思是异步 I/O
ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0  #insert buffer thread 挂起的 fsync()操作数目等
Pending flushes (fsync) log: 0; buffer pool: 0  #log thread 挂起的 fsync()操作数目等

146246831 OS file reads, 760501349 OS file writes, 247143684 OS fsyncs  #这行显示了读，写和 fsync()调用执行的数目，在你的机器环境负载下这些绝对值可能会有所不同，因此更重要的是监控它们过去一段时间内是如何改变的。
1 pending preads, 0 pending pwrites  #这行显示了当前被挂起的读和写操作数
145.49 reads/s, 783677 avg bytes/read, 28.75 writes/s, 10.67 fsyncs/s  #这行显示了在头部显示的时间（指的是第 1 部分的时间）段内的每秒平均值。
```

```
# 注：三行挂起读写线程、缓冲池线程、日志线程的统计信息的值是检测 I/O 受限的应用的一个好方法，如果这些 I/O 大部分有挂起操作，那么负载可能 I/O 受限。在 linux 系统下使用参数：innodb_read_io_threads 和 innodb_write_io_threads 两个变量来配置读写线程的数量，默认为各 4 个线程。
## insert buffer thread：负责插入缓冲合并，如：记录被从插入缓冲合并到表空间中
## log thread：负责异步刷事务日志
## read thread：执行预读操作以尝试预先读取 innodb 预感需要的数据
## write thread：刷新脏页缓冲
```

2.8. INSERT BUFFER AND ADAPTIVE HASH INDEX

- 第八段信息
  - 这部分显示了 insert buffer 和 adaptive hash index 两个部分的结构的状态

```
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------

Ibuf: size 12, free list len 27559, seg size 27572, 18074934 merges  #这行显示了关于 size（size 12 代表了已经合并记录页的数量）、free list（代表了插入缓冲中空闲列表长度）和 seg size 大小（seg size 27572 显示了当前 insert buffer 的长度，大小为 27572*16K=440M 左右）的信息。18074934 merges 代表合并插入的次数
merged operations: #这个标签下的一行信息 insert，delete mark，delete 分别表示 merge 操作合并了多少个 insert buffer，delete buffer，purge buffer
insert 81340470, delete mark 8893610, delete 818579
discarded operations: #这个标签下的一行信息表示当 change buffer 发生 merge 时表已经被删除了，就不需要再将记录合并到辅助索引中了
insert 0, delete mark 0, delete 0
```

```
Hash table size 87709057, node heap has 10228 buffer(s) #这行显示了自使用哈希索引的状态，其中，Hash table size 87709057 表示 AHI 的大小，node heap has 10228 buffer(s) 表示 AHI 的使用情况
1741.05 hash searches/s, 539.48 non-hash searches/s #这行显示了在头部第 1 部分提及的时间内 Innodb 每秒完成了多少哈希索引操作，1741.05 hash searches/s 表示每秒使用 AHI 搜索的情况，539.48 non-hash searches/s 表示每秒没有使用 AHI 搜索的情况(因为哈希索引只能用于等值查询，而范围查询，模糊查询是不能使用哈希索引的。)，通过 hash searches: non-hash searches 的比例大概可以了解使用哈希索引后的效率，哈希索引查找与非哈希索引查找的比例仅供参考，自适应哈希索引无法配置，但是可以通过 innodb_adaptive_hash_index=ON|OFF 参数来选择是否启用这个特性。

# innodb 从 1.0.x 开始引入 change buffer，可以视为 insert buffer 的升级，从这个版本开始，innodb 可以对 DML 操作(insert,delete,update)都进行缓冲，他们分别是 insert buffer,delete buffer,purge buffer，当然和之前 insert buffer 一样，change buffer 适用对象仍然是非唯一索引的辅助索引，因为没有 update buffer，所以对一条记录进行 update 操作可以分为两个过程：
# A：将记录标记为删除
# B：真正将记录删除
#因此，delete buffer 对应 update  操作的第一个过程，即将记录标记为删除，purge buffer 对应 update 的第二个过程，即将记录真正地删除
```

2.9. LOG

- 第九段信息
    - 这部分显示了关于 innodb 事务日志（重做日志）子系统的统计信息。内容包括当前日志序列号(LSN)、日志刷新到磁盘的位置(LSN)以及 InnoDB 上次检查点的位置(LSN)。此外还包括被挂起的写操作统计信息和写性能统计信息，如下：

```
---
LOG
```

```
---

Log sequence number 1351392990515   #这行显示了当前最新数据产生的日志序列号
Log flushed up to    1351392989504 #这行显示了日志已经刷新到哪个位置了（已经落盘到事务日志中的日志序列号）
Last checkpoint at   1351373900020 #这行显示了上一次检查点的位置（一个检查点表示一个数据和日志文件都处于一致状态的时刻，并且能用于恢复数据），如果上一次检查点落后与上一行太多，并且差异接近于事务日志文件的大小，Innodb 会触发“疯狂刷”，这对性能而言非常糟糕。
0 pending log writes, 0 pending chkp writes   #这行显示了当前挂起的日志读写操作，可以将这行的值与第 7 部分 FILE I/O 对应的值做比较，以了解你的 I/O 有多少是由于日志系统引起的。
286879989 log i/o's done, 15.92 log i/o's/second #这行显示了日志操作的统计和每秒日志 I/O 数，可以将这行的值与第 7 部分 FILE I/O 对应的值做比较，以了解你的 I/O 有多少是由于日志系统引起的。
```

2.10. BUFFER POOL AND MEMORY

- 第十段信息
    - 这部分显示了关于 innodb 缓冲池中的页读和写的统计数据及其如何使用内存的统计，从这些数字中可以计算查询当前正在执行的数据文件 I/O 操作。

```
----------------------
BUFFER POOL AND MEMORY
----------------------

Total memory allocated 45357793280; in additional pool allocated 0   #这行显示了由 innodb 分配的总内存，以及其中多少是额外内存池分配，额外内存池仅分配了其中很小一部分内存，由内部内存分配器分配，现在的 innodb 版本一般使用操作系统的内存分配器，但老版本使用自己的，这是由于在那个时代有些操作系统并未提供一个非常好的内存分配实现。
```

Dictionary memory allocated 12681573

Buffer pool size 2705015 #从这行开始的下面 4 行显示缓冲池度量值，以页为单位，度量值有总的缓冲池大小，空闲页数，分配用来存储数据库页的页数，以及脏页数。这行显示了缓冲池总共有多少个页，即即 2705015*16K，共有 43G 的缓冲池

Free buffers 5 #这行显示了缓冲池空闲页数

Database pages 2694782 #这行显示了分配用来存储数据库页的页数，即，表示 LRU 列表中页的数量,包含 young sublist 和 old sublist

Old database pages 994651 #这行显示了 LRU 中的 old sublist 部分页的数量

Modified db pages 10610 #这行显示脏页数

Pending reads 128 #这行显示了挂起读的数量

Pending writes: LRU 0, flush list 0, single page 0 #这行显示了挂起写的数量

#注意，这里挂起的读和写操作并不与 FILE I/O 部分的值匹配，因为 Innodb 可能合并许多的逻辑读写操作到一个物理 I/O 操作中，LRU 代表最近使用到的被挂起数量，它是通过冲刷缓冲中不经常使用的页来释放空间以供给经常使用的页的一种方法，冲刷列表 flush list 存放着检查点处理需要冲刷的旧页被挂起的数量，单页 single page 被挂起的数量（single page 写是独立的页面写，不会被合并)。

Pages made young 3014373561, not young 0 #这行显示了 LRU 列表中页移动到 LRU 首部的次数，因为该服务器在运行阶段改变没有达到 innodb_old_blocks_time 阀值的值，因此 not young 为 0

6960.42 youngs/s, 0.00 non-youngs/s #表示每秒 young 和 non-youngs 这两类操作的次数

Pages read 2946570833, created 43450158, written 574214278 #这行显示了 innodb 被读取，创建，写入了多少页，读/写页的值是指的从磁盘读到缓冲池的数据，或者从缓冲池写到磁盘中的数据，创建页指的是 innodb 在缓冲池中分配但没有从数据文件中读取内容的页，因为它并不关心内容是什么（如，它们可能属于一个已经被删除的表）

6960.54 reads/s, 4.42 creates/s, 9.33 writes/s #这行显示了对应上面一行的每秒 read，create，write 的页数

Buffer pool hit rate 955 / 1000, young-making rate 45 / 1000 not 0 / 1000 #这行显示了缓冲池的命中率，它用来衡量 innodb 在缓冲池中查找到所需页的比例，它度量自上次 Innodb 状态输出后到本次输出这段时间内的命中率，因此，如果服务器自那以后一直很安静，你将会看到 No buffer pool page gets since the last printout。它对于度量缓存池的大小并没有用处。

Pages read ahead 6928.54/s, evicted without access 8.21/s, Random read ahead 0.00/s #这行显示了页面预读，随机预读的每秒页数

LRU len: 2694782, unzip_LRU len: 0 #innodb1.0.x 开始支持压缩页的功能，将原来 16K 的页压缩为 1K，2K，4K，8K，而由于页的大小发生了变化，LRU 列表也有了些改变，对于非 16K 的页，是通过 unzip_LRU 列表进行管理的，可以看到 unzip_LRU len 为 0 表示没有使用压缩页.

I/O sum[60790]:cur[30], unzip sum[0]:cur[0]

对于压缩页的表，每个表的压缩比例可能不同，可能存在有的表页大小为 8K，有的表页大小为 2K 的情况，unzip_LRUs 怎样从缓存池中分配内存的呢？首先，在 unzip_LRU 列表中对不同压缩页大小的页进行分别管理，其次，通过伙伴算法进行内存的分配，例如：需要从缓存池中申请页为 4K 的大小，其过程如下：

a：检查 4K 的 unzip_LRU 列表，检查是否有可用的空闲页

b：若有，则直接使用

c：若没有，检查 8K 的 unzip_LRU 列表

d：若能够得到空闲页，将页分成 2 个 4K 的页，存放到 4K 的 unzip_LRU 列表

e：若不能得到空闲页，从 LRU 列表中申请一个 16K 的页，将页分成 1 个 8K，2 个 4K 的页，分别存放到各自大小对应的 unzip_LRU 列表中。

注：可能出现 Free buffers 和 Database pages 之和不等于 Buffer pool size，因为缓冲池中的页肯会被分配给自适应哈希索引，lock 信息，insert buffer 等，而这部分页不需要 LRU 算法进行维护，因此不在 LRU 列表中。

- 如果 innodb buffer pool 使用参数 innodb

innodb_buffer_pool_instances=num 设置了大于 1 个缓冲池实例，那么就会按照这个参数把 innodb_buffer_pool_size=xxx 平分为 num 份。每份的信息显示类似如下，这部分的内容和 9.1 小节内容类似，就不再多说。

```
----------------------
INDIVIDUAL BUFFER POOL INFO
----------------------

---BUFFER POOL 0
Buffer pool size   541003
Free buffers       1
Database pages     538965
Old database pages 198933
Modified db pages  2190
Pending reads 128
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 603372180, not young 0
1441.81 youngs/s, 0.00 non-youngs/s
Pages read 589705199, created 8703138, written 116954697
1441.61 reads/s, 0.75 creates/s, 1.83 writes/s
Buffer pool hit rate 955 / 1000, young-making rate 45 / 1000 not 0 / 1000
Pages read ahead 1436.98/s, evicted without access 0.87/s, Random read ahead 0.00/s
LRU len: 538965, unzip_LRU len: 0
I/O sum[12158]:cur[6], unzip sum[0]:cur[0]

---BUFFER POOL 1
Buffer pool size   541003
Free buffers       1
Database pages     538959
Old database pages 198931
Modified db pages  2025
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
```

```
Pages made young 602366394, not young 0
1481.35 youngs/s, 0.00 non-youngs/s
Pages read 588738997, created 8708171, written 113209540
1480.56 reads/s, 0.83 creates/s, 1.92 writes/s
Buffer pool hit rate 958 / 1000, young-making rate 42 / 1000 not 0 / 1000
Pages read ahead 1473.73/s, evicted without access 1.96/s, Random read ahead 0.00/s
LRU len: 538959, unzip_LRU len: 0
I/O sum[12158]:cur[6], unzip sum[0]:cur[0]
```

2.11. ROW OPERATIONS

- 最后一段信息
  - 这部分显示了主线程正在做什么，包括每种类型的行操作的数量和性能统计数据，如下：

```
--------------
ROW OPERATIONS
--------------

0 queries inside InnoDB, 0 queries in queue  #这行显示了 innodb 内核内有多少个线程，队列中有多少个线程，队列中的查询是 innodb 为限制并发执行的线程数量而不运行进入内核的线程。查询在进入队列之前会休眠等待。
5 read views open inside InnoDB  #这行显示了有多少打开的 innodb 读视图，读视图是包含事务开始点的数据库内容的 MVCC 快照，你可以看看某特定事务在第 6 部分 TRANSACTIONS 是否有读视图
Main thread process no. 4368, id 140653691242240, state: sleeping  #这行显示了内核的主线程状态
Number of rows inserted 3429012215, updated 153529675, deleted 112310240, read 3739562987410  #这行显示了多少行被插入，更新和删除，读取
428.52 inserts/s, 7.21 updates/s, 0.46 deletes/s, 1047933.92 reads/s #这行显示了对应上面一行的每秒平均值，如果想查看 innodb 有多少工作量在进行，那么这两行是很好的参考值
```

```
---------------------------
END OF INNODB MONITOR OUTPUT   #要注意了，如果看不到这行输出，可能是有大量事务或者是有一个大的死锁截断了输出信息
============================

# 注：内核的主线程状态可能的状态值有如下一些：
## A：doing background drop tables
## B：doing insert buffer merge
## C：flushing buffer pool pages
## D：making checkpoint
## E：purging
## F：reserving kernel mutex
## G：sleeping
## H：suspending
## I：waiting for buffer pool flush to end
## J：waiting for server activity
```