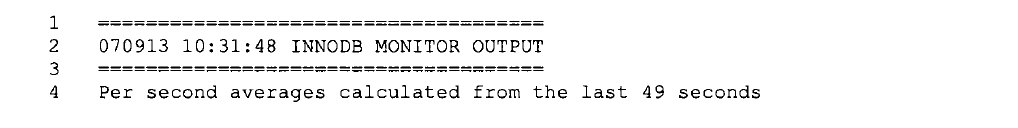
**show （engine）innodb status 的输出信息简介**

输入指令的时候最好写成 show innodb status\G, 显示的格式会更容易阅读一些

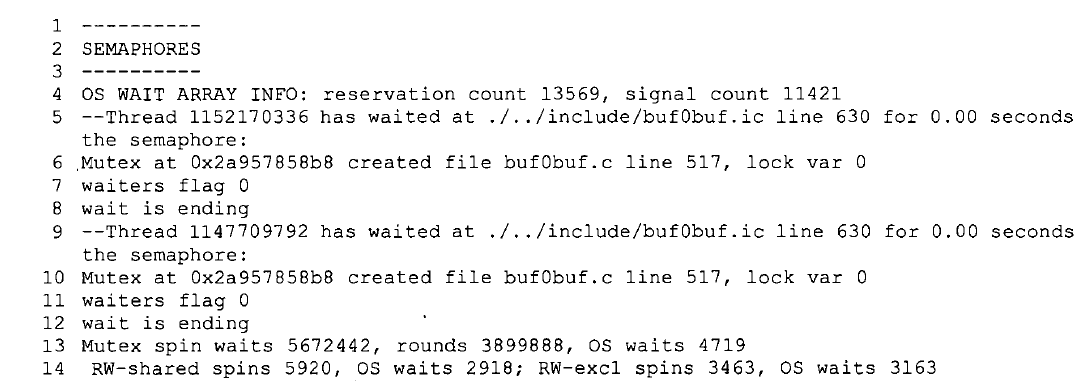
跟大多数show指令不同的是，show innodb status的输出结果是一个字符串，没有行和列。

## 【头信息】



SHOW INNODB STATUS 中的很多统计值都是每秒更新一次的，如果打算利用这些统计值的话，那么最好统计一段时间内的结果。首先要确认这是至少统计了 20-30 秒的样本数据。如果平均统计间隔是0或1秒，那么结果就没什么意义了。本例中，49seconds 是平均值计算的时间间隔，它或者是上次输出以来的时间，或者是距离上次内部复位的时长。

## 【SEMAPHORE】



信号量部分，包含了两种数据：时间计数器，以及可选的当前等待线程的列表。

**第4行：** 操作系统等待阵列的信息 （它是一个“**插槽**”式的阵列，Innodb保留了阵列里的一些插槽给信号量使用，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。 这一行还显示出了InnoDB使用了多少次操作系统的等待。 reservation count显示了InnoDB分配插槽的额度，而signal count衡量的是线程通过阵列得到信号的频度）

**第5行到第12行：**显示的是当前正在等待互斥量的InnoDB线程。在本例中有2个线程在等待，每一个都以 “thread <num> has waited…”开始。这一段应该是空的，除非你的服务器运行着高并发的工作负载，它促使InnoDB采取让操作系统等待的措施。“waiters flag”显示了有多少个等待者正在等待同一个互斥量，“wait is ending”意味着这个互斥量实际上已经被释放了，但是操作系统还没有把线程调度过来运行。"lock var" 表示当前的 mutex 对象的值(被锁住 = 1 / 释放 = 0) 值；

InnoDB使用了互斥量和信号量来保护代码的临界区，例如限定每次只有一个线程进入临界区，或者当有活动的读者时，就限制写者的加入等等。

**第13行：**跟互斥量相关的几个计数器；

**第14行：**用于显示读/写共享和排斥锁的计数器；

InnoDB有一个多阶段等待策略，首先它会对锁进行**循环等待（自旋锁）**，如果经过了一个预设的循环等待周期（innodb\_sync\_spin\_loops配置变量）之后还没成功，就退到更昂贵更复杂的等待阵列里（操作系统等待）。

**循环等待**的成本相对较低，但是它要不停地检查一个资源是否被锁定，还是消耗了CPU周期。但是这没有像它听起来那么糟糕，因为当处理器等待I/O时，一般都有一些空闲的CPU周期可用，即使没有，空等也要比其它方式更廉价一些。然后当另外一条线程能做一些事情时，循环等待也会独占处理器。

循环等待的替换方案是让操作系统做上下文切换（**操作系统等待**），这样，当这条线程等待时，另外一条线程就可以被运行，然后通过等待阵列里的信号量发送信号，唤醒那条沉睡的线程。通过信号量来发送信号是比较有效率的，但是操作系统的上下文切换代价就很大，每秒几千次的切换会产生大量的系统开销。

可以通过修改innodb\_sync\_spin\_loops的值，在循环等待和操作系统等待之间做一个平衡。

"spin waits" 和 "spin rounds" ，自旋等待和自旋轮转

* **Status**

The mutex status. The fields contains several values:

* + **count** indicates how many times the mutex was requested.
  + **spin\_waits** indicates how many times the spinlock had to run.
  + **spin\_rounds** indicates the number of spinlock rounds. (**spin\_rounds** divided by **spin\_waits** provides the average round count.)
  + **os\_waits** indicates the number of operating system waits. This occurs when the spinlock did not work (the mutex was not locked during the spinlock and it was necessary to yield to the operating system and wait).
  + **os\_yields** indicates the number of times a the thread trying to lock a mutex gave up its timeslice and yielded to the operating system (on the presumption that permitting other threads to run will free the mutex so that it can be locked).
  + **os\_wait\_times** indicates the amount of time (in ms) spent in operating system waits, if the [**timed\_mutexes**](http://dev.mysql.com/doc/refman/5.1/en/server-system-variables.html#sysvar_timed_mutexes) system variable is 1 (**ON**). If [**timed\_mutexes**](http://dev.mysql.com/doc/refman/5.1/en/server-system-variables.html#sysvar_timed_mutexes) is 0 (**OFF**), timing is disabled, so **os\_wait\_times** is 0. [**timed\_mutexes**](http://dev.mysql.com/doc/refman/5.1/en/server-system-variables.html#sysvar_timed_mutexes) is off by default.

## 【LATEST FOREIGN KEY ERROR】

当你的服务器上出现了一个**外键错误**的时候，这一段才会出现。

这一段输出对于调试InnoDB常常发生的原因模糊的外键错误很有帮助；

来看一个实例：

首先我们创建两个表，并在两者间建立一个外键，插入一些数据；

CREATE TABLE parent (

parent\_id int NOT NULL,

PRIMARY KEY(parent\_id)

）ENGINE=InnoDB;

CREATE TABLE child (

parent\_id int NOT NULL,

KEY parent\_id(parent\_id),

CONSTRAINT child\_ibfk\_1 FOREIGN KEY (parent\_id) REFERENCES parent (parent\_id)

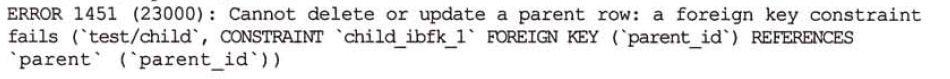
）ENGINE=InnoDB;

INSERT INTO parent(parent\_id) VALUES(1);

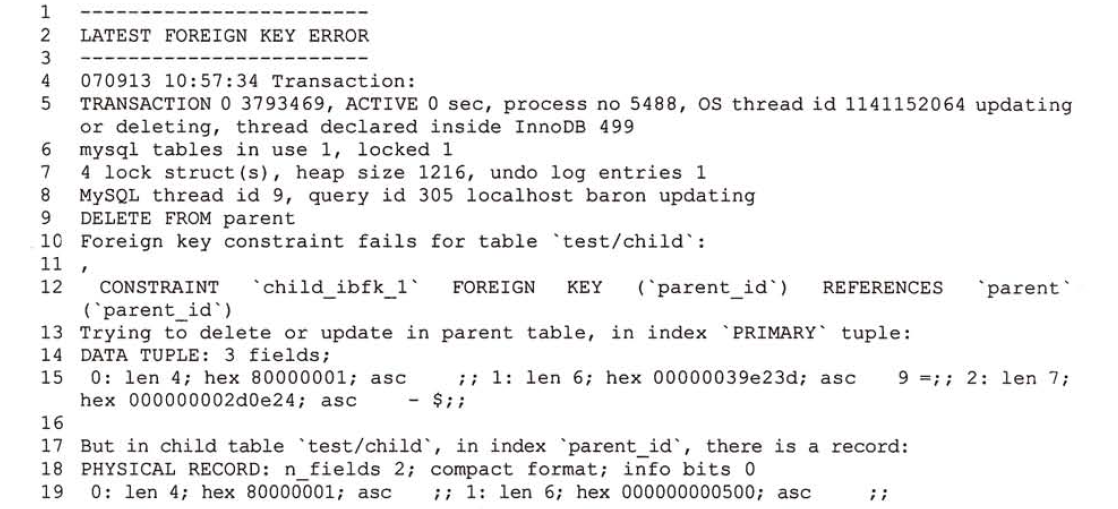
INSERT INTO child(parent\_id) VALUES(1);

基本的外键错误分两类： 1）因为插入，更新或删除数据而违反外键约束是第一类。举例来说，以下就是我们从父表中删除一行数据时发生的错误：

delet from parent;



这个错误信息简单易懂，以下是通过show innodb status看到的结果：



第4行：外键错误发生的时间和日期；

第5行到第9行：引发外键错误的事务的细节；

第10行到第19行：错误发生时，InnoDB正在更改的数据，

下面看另外一种外键错误（试着修改父表结构），

ALTER TABLE parent MODIFY parent\_id INT UNSIGNED NOT NULL;



这个错误信息看上去就不是那么好理解了，我们来看一下show innodb status的输出：



很明显，show innodb status的输出信息更详尽和易于理解一些。

## 【LATEST DETECTED DEAD LOCK】

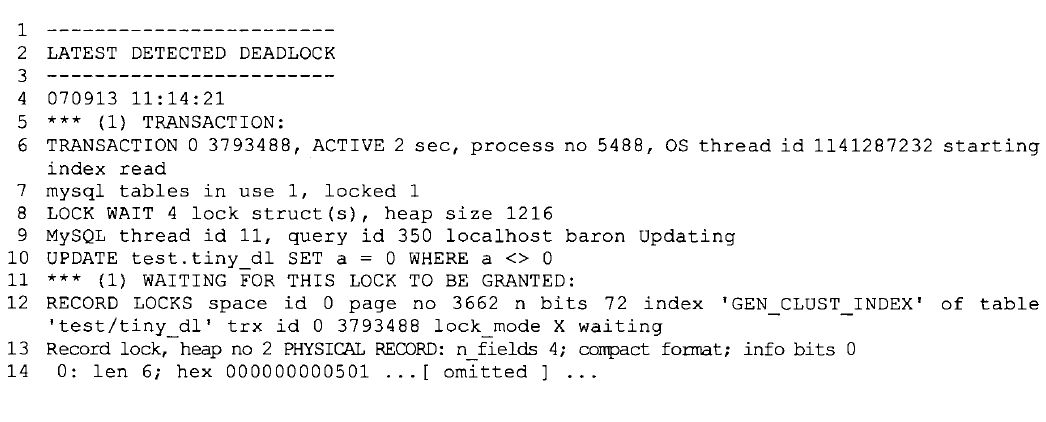
只有在出现死锁时，本段内容才会出现；、

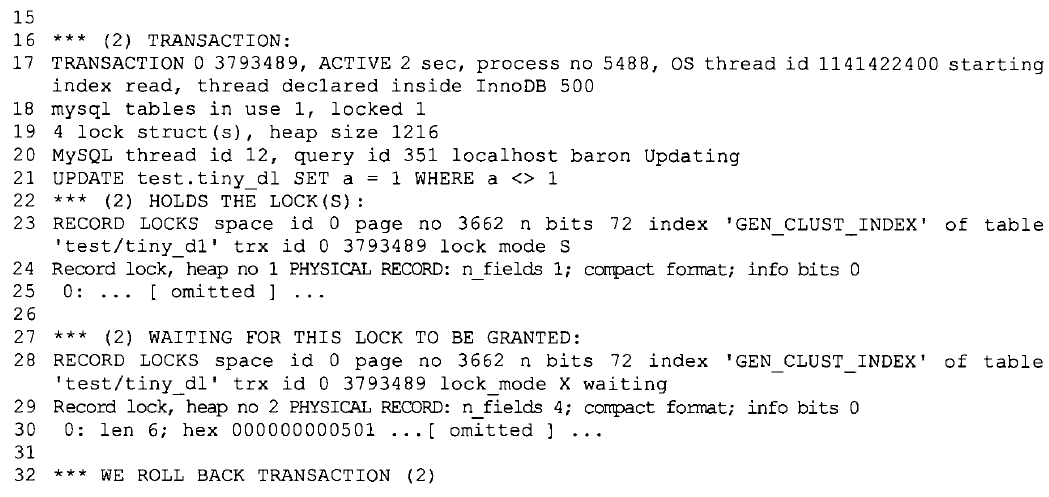
死锁在等待关系图（Waits-for Graph）中是一个循环，InnoDB会立刻检测到死锁，因为每当有事务等待行锁的时候，它都会去等待关系图里检查是否有循环。死锁情况可能会比较复杂，但是在LATEST DETECTED DEAD LOCK的输出中只显示了最近两个死锁的情况，以及它们在各自的事务里执行的最后一条语句，它们在图里形成循环的锁的信息。你看不到循环里的其它事务，也看不到在事务里早先真正获得了锁的语句，尽管如此，通常你还是可以通过查看输出结果来确定到底是什么引起了死锁。

在InnoDB里实际上有两种死锁，一种是人们常常碰到的，它在等待关系图里是一个真正的循环；另一种是在一个等待关系图里，因代价昂贵而无法判定它到底是不是包含了循环。如果InnoDB要在图里检查超过100万个锁，或者在检查过程中，InnoDB要重做200个以上的事务，它就会放弃，并宣布这里有一个死锁。这时你在输出里就会看到一条信息“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的锁和等待的锁，还会打出记录本身。不幸的是有时它会很大，甚至超出你为输出结果预留的长度，以至于你都没法看到其它的输出信息了，，，对此唯一的补救办法是制造一个小死锁取代大死锁，或者使用补丁（在<http://lists.mysql.com/internals/35174>可以得到）。

下面看一个死锁的样例：





第4行：死锁发生的时间；

第5行到第10行：死锁里第一个事务的信息；

第11行到第15行：当死锁发生时，第一个事务正在等待的锁。第14行的信息被忽略了，因为它只对调试有用，第12行的信息尤为重要，它告诉我们这个事务正在等待的是test.tiny\_dl表里的GEN-CLUST\_INDEX的排它锁。（GEN-CLUST\_INDEX索引是当你没有指定主键的时候，InnoDB在内部自动创建的）

第16行到第21行：第二个事务的状态；

第22行到第26行：第二个事务持有的锁；为了简洁，第25行中的几条记录被删除了，这些记录中的一条就是第一个事务正在等待的是哪一条记录；

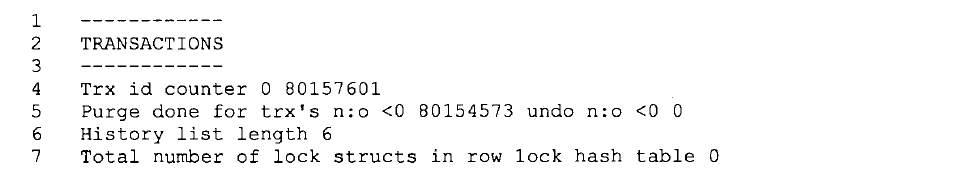
第27行到第31行: 第二个事务正在等待的锁；

第32行：哪个事务被选中成为死锁的牺牲品；InnoDB会把看上去最容易回滚（也就是更新的记录数最少的）的事务作为牺牲品；

如果能使两个查询在同一个索引，同一个方向上进行扫描，就能减低死锁的数目。因为查询在同一个顺序上请求锁的时候不会创建循环。有时这很容易做到，比如你要在一个事务里更新很多条记录，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-----这样就不会发生死锁。但是有时候这个方法也是行不通的，比如两个进程使用了不同的索引区操作同一张表的时候。

## 【TRANSACTIONS】

这一段包含InnoDB事务的一些摘要信息，它们跟随在当前活动事务之后，以下是前几行（头信息）：



**第4行：**当前事务ID，它是一个系统变量，每创建一个新事务就会累加；

**第5行：**InnoDB清除旧版本的MVCC行时所用的事务ID。通过这个值和当前的事务进行比较，你会发现有多少老版本的数据已经被清除掉了。如果同时有很多行被更新，那每一行都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当事务完成时立即将它提交，不要让它长时间处于打开的状态。因为即使一个打开的事务部做任何操作，它也会影响InnoDB清除旧版本的行数据。

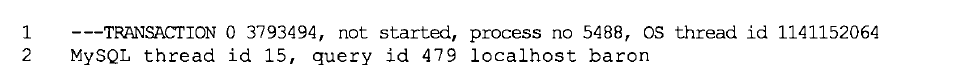
事务ID是当前事务的标识，事务的id每次都会增加。*Purge done for trx's n:o* 是指净化(purge)线程已经完成的事务数。Innodb仅清除那些被当前事务认为不再需要的旧版本数据。那些未提交的旧事务可能会阻塞净化线程并且消耗资源。通过查看2次清除事务数之差，就可以知道是否发生了这种情况。少数情况下，净化线程可能难以跟上更新的速度，2次查看值之差可能会越来越大；那 么，*innodb\_max\_purge\_lag* 就派得上用场了。 *"undo n:o"* 显示了净化线程当前正在处理的回滚日志号，如果当前不处于活跃状态，则它的值是0

**第6行：**历史记录的长度，它就是位于InnoDB数据文件的撤销空间的未清除事务的数目。

当一个事务执行了更新并提交后，这个数字就会增加；当清除进程移除一个旧版本的数据时，这个数字就会减少。

**第7行：**锁结构的数目；每一个锁结构经常持有许多个行锁，所以它不等同于被锁定的行数。

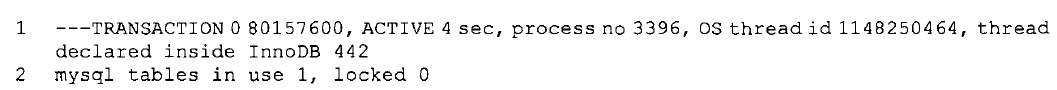
头部信息之后就是一个事务列表，每一个事务至少占有两行内容：

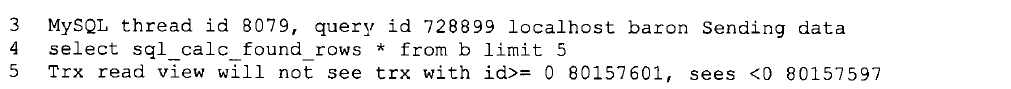


**第1行：**事务的ID和状态；当前事务处于“not started”，说明它已经被提交，但是还没有发出会影响到其它事务的执行语句，它可能就是空闲着。接下来就是一些进程和线程的信息。

**第2行：**MySQL的线程ID，它和SHOW FULL PROCESSLIST里显示的ID是一致的，紧跟它之后是一个内部查询号码和一些连接信息。

来看一个更复杂些的例子：





**第1行：**显示出这个事务活动了4秒钟，它可能的状态是“not started”,”active”,”prepared”,”commited in memory”(一旦被提交到磁盘上状态就变位 not started)。虽然在本示例中没有显示出来，但是在其它条件下你会看到关于事务当前在做什么的信息，例如“fetching rows”,”adding foreign keys”等等。 “thread declared inside InnoDB 442” 的意思是该线程正在InnoDB内核里做一些操作，并且还有442张“票子”可以使用。也就是说同样的SQL查询可以重入InnoDB内核442次。这“票子”是系统用来限制内核中线程并发操作的手段，防止它在某些平台上运行失常。即使线程的状态是“inside InnoDB”, 它也不必把所有的工作都放到InnoDB里面来做。查询大概就是在服务器级上做一些操作，然后通过某个途径和InnoDB互动一下就可以了。你会看到这些事务的状态都是“sleeping before joining InnoDB queue”或者“waiting in InnoDB queue”。

**第2行：** 当前语句使用到的和锁定的表有多少；如果事务已经锁定了任意几行数据，这里将有一行信息显示出锁定结构的数目（注意，锁定结构和行锁是两回事）和堆的大小（为了持有这些行锁而占用的内存大小）。在MySQL更新的版本，这一行显示当前事务持有的行锁的实际数目。

**第4行：**如果当前在执行一个查询，则显示出查询文本；

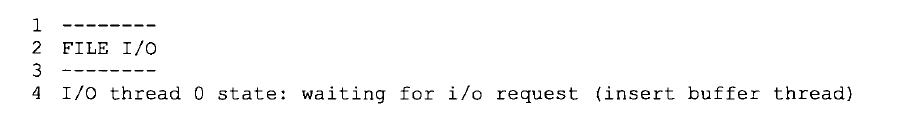
**第5行：**事务的读视图；它表明了事务的ID范围，包括了可见的和不可见的两种类型。本例中，有四个事务在“缺口”中，这四个事务可以是可见的，也可以是不可见的。

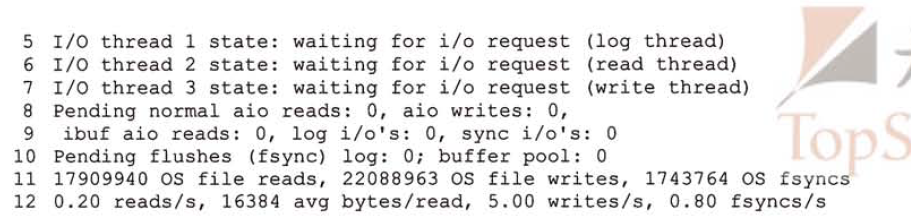
如果输出信息里有多个事务，InnoDB就限制了打印出来的事务数目，你会看到“truncated”字样。

MySQL中，每个连接如果没有活动的事务，则它的状态是 *not started*，如果有活动的事务，则是 *ACTIVE*。注意，尽管事务是活动的，但是其连接的状态却可能是 "睡眠(sleep)" - 如果是在一个有多条语句的事务里的话。Innodb 会同时显示系统的线程号以及进程号，这有助于利用gdb来调试或者其他类似用途。另外，事务的状态也会根据当前实际状态来显示，例如 *"读取记录(fetching rows)"*，em>"更新(updating)"等等。Innodb 会根据 *innodb\_thread\_concurrency* 的值来限制同时并发的线程数。如果线程当前不在 Innodb 的内核中运行，则它的状态可能是 *"waiting in InnoDB queue"* 或 *"sleeping before joining InnoDB queue"*。后面这个状态有点意思 - Innodb 为了避免有太多的线程同时抢着要进入运行队列，那么就会尝试让这些线程进入等待状态(如果没有足够的空闲插槽(slot)的话)。这就可能会导致 Innodb 内核中当前活跃的线程数可能比 *innodb\_thread\_concurrency* 的值还小。某种负载环境下，这可能有助于减小线程进入队列的时间。可以通过调整 *innodb\_thread\_sleep\_delay* 来实现，它的单位是微妙。

## 【FILE IO】

显示I/O help 线程的状态，用性能计数器的方式来表示：





第4行到第7行：I/O helper线程当前状态；

第8行到第10行：每个helper线程还没完成的操作的数目，以及日志和缓冲池线程还没完成的fsync（）的数目，“aio”的意思是“asynchronous I/O”； 第8行，第9行里显示的未决数值是检测I/O密集应用的好办法，如果这些类型的I/O里大多有一些未决的操作，说明当前的工作负载可能就是I/O密集的。

第11行：读，写以及fsync（）调用的执行次数。

第12行：在头信息指明的那段时间里，平均一秒钟执行各项操作的次数；

你至少会看到以下四条线程：

插入缓冲区线程：

负责插入缓冲区的合并（例如将记录从插入缓冲区合并到表空间里）；

日志线程：

负责异步的日志刷新；

读线程：

负责读前置（read-ahead）的操作，预测InnoDB将要使用的数据，将它们读取进来；

写线程：

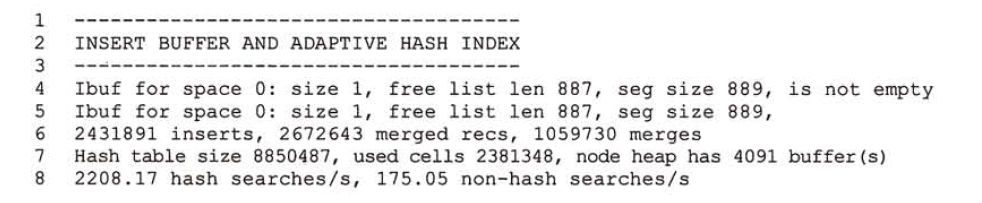
刷新“脏”缓冲区

每个线程都会显示正在进行的操作数量。另外，正在执行的 fsync 操作数量也会显示出来。有写数据时，Innodb需要确保数据最终被写到磁盘上，只是把它们放在系统缓存里是不够的。通常是调用 fsync() 来完成的。如果它的值一直很高，那意味这Innodb可能是处于I/O负载较高状态。注意，由线程执行请求引发的I/O请求是不计算在内的，因此尽管系统的 I/O负载较高，但是它们的值却可能为 0。

接下来显示的是I/O操作的平均统计值，它们对于图形显示或者监控很有用。  
"16384 avg bytes/read" 是读请求的平均值。随机I/O的话，每个页的大小是16K，全表扫描或索引扫描时的预读会导致这个值明显的增加。因此，它体现了预读的效率。

## 【INSERT BUFFER AND ADAPTIVE HASH INDEX】

这一节显示的是INSERT BUFFER AND ADAPTIVE HASH INDEX的状态；



**第4行：**插入缓冲区的大小，它的“自由列表”的长度，以及它的分段大小。for space 0似乎是说明了存在多个缓冲区的可能，但是这从未被实现过，插入缓冲区只有一个。

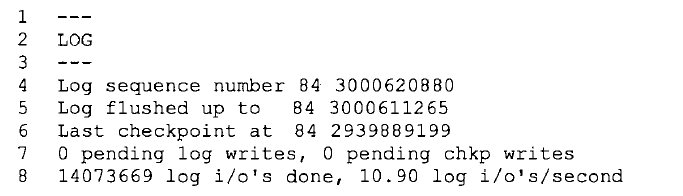
**第5行：**冗余信息；

**第6行：**显示InnoDB已经做了多少次缓冲区操作，查看其中合并到插入缓冲区的比例是评判缓冲区效率的好方法；

**第7行：** 显示自适应散列索引（adaptive hash index）的状态；

**第8行：** 显示了根据头部信息划定的期限内InnoDB做过的散列索引的次数，散列索引的查找次数和非散列索引查找次数的比例是另一项效率度量值，因为散列查找快过非散列查找；

## 【LOG】



这一段显示了InnoDB事务LOG子系统的统计信息；

**第4行：**当前日志的顺序号；

**第5行：**日志已经刷新到的点位，日志顺序号就是已经写入日志文件的字节总数，因此你能用它来计算日志缓冲区里还有多少数据没被刷新到日志文件中。本例中是9615个字节（13000620880-13000611265）

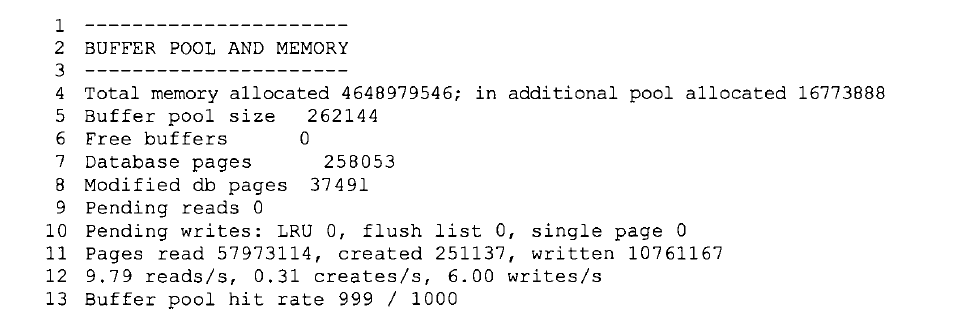
**第6行：**最近的一个检查点（一个检查点代表了一个瞬间，在那一刻里数据和日志文件都处于可知的状态，它可以被用来做还原）

**第7行到第8行：**还未完成的日志操作和统计信息，拿这些跟 FILE I/O段的信息比较，你会发现I/O里有多少是跟日志子系统相关的操作触发的；

这一段显示的是Innodb的日志子系统相关信息。可以看到当前的日志序列号 - 相当于Innodb自从表空间开始创建直到现在已经写入日志文件的总字节数。还可以看到日志已经刷新到哪个点，同样也可以根据最后检查点计算出还有多少日志没有刷新到文件中去。Innodb采用模糊检查点，因此这行显示的是已经从缓冲池中刷新到文件的日志序列号。由于更高的日志序列号可能不会被立刻刷新到日志文件中去，因此日志序列号不能被覆盖掉。通过监控刷新到哪个日志的日志序列，可以判定 innodb\_log\_buffer\_size 的设置是否合理，如果看到超过 30% 的日志还没有刷新到日志文件中，则需要考虑增加它的值了。

另外，还能看到日志写入以及检查点的数目。根据日志I/O操作的数目可以区分开表空间相关的I/O请求和日志I/O请求数量，进而可以确定到底需要几个日志文件。注意，innodb\_flush\_log\_at\_trx\_commit 的值可以影响到日志写操作的代价高或低。如果 innodb\_flush\_logs\_at\_trx\_commit=2，则日志是写到系统缓存，然后再顺序写到日志文件中，因此相对会快很多。

## 【BUFFER POOL AND MEMORY】



**第4行：** InnoDB申请的内存总数，以及其中有多少是在附加内存池中申请到的；

**第5行到第8行：**缓冲池的度量值，以页为单位，包括缓冲池的总大小，空闲页的数量，用于存储数据库页的页的数量，和“脏”数据库页的数量。 InnoDB在缓冲池里把一些页面用作锁的索引，自适应索引散列以及其它系统结果，因此，在池里的数据库页的数量永远不会等于池的总大小；

**第9行到第10行：**未完成的读和写的数目（例如InnoDB要为缓冲池做的逻辑读和写的数目）。这些值会跟FILE I/O段里的数据不匹配，因为InnoDB可能会将许多逻辑操作合并为一个单独的物理I/O操作，LRU意思是“最近最少使用”，它在缓冲池里把不常用的页刷新到次磁盘上，为常用的页腾出空间。刷新列表里保存的一些旧页，它们须由检查点进程来刷新，单页写属于独立的页写操作，它们不会被合并掉；

第8行显示了这个缓冲池里容纳了37491个脏页，在某个点上需要被刷新到磁盘上去，（它们已经在内存里更新，但是磁盘上的还没更新），然而，第10行也显示出在那一刻并没有刷新操作在进行，这不成问题，因为InnoDB会在需要的时候刷新它们；

**第11行：**显示了InnoDB已经读，创建和写了多少页，页读，写的值来自于从磁盘读入到缓冲池的数据，反之亦然。页创建的值来自于InnoDB在缓冲池里申请的页，不包括从数据文件读取出来的内容，因为它不关心内容是什么；（例如它们可能属于一个已经被删除的表）

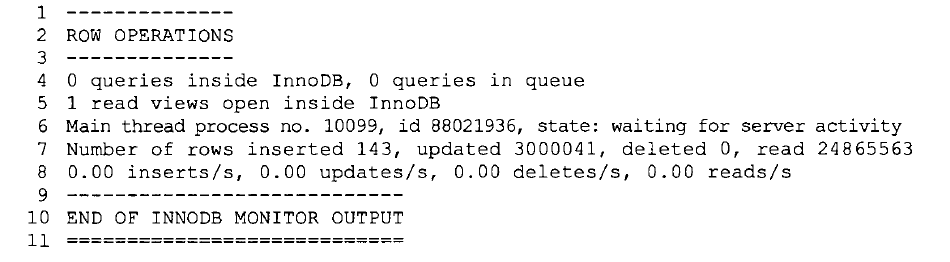
**第13行：**报告了缓冲池的命中率，它可以衡量InnoDB在缓冲池里找到所需要的页的比例，是缓存效率的度量值，它的衡量范围是自最近一次InnoDB状态打印以来的命中率，因此如此自那时以来服务器一直很安静，你就会看到“no buffer pool page gets since the last printout”

这部分显示了缓冲池和内存的利用率相关信息。可以看到Innodb分配的所有内存(有些时候可能比你设置的还要多点)，以及额外的内存池分配情况(可以检查它的大小是否正好)，缓冲池总共有多少个内存页，有多少空闲内存页，数据库分配了多少个内存页以及有多少个脏内存页。从这些信息中，就可以判断内存缓冲池是否设定合理，如果总是有大量空闲内存页，则不需要设置那么多内存，可以适当减小一点。如果空闲内存页为 0，这种情况下数据库内存页就不一定会和缓冲池的总数一致，因为缓冲池还需要保存锁信息，自适应哈希索引以及其他系统结构等信息。

等待中的读写是指内存缓冲池级别的请求。Innodb可能会把多个文件级别的请求合并到一个上，因此各不相同。我们还可以看到Innodb提交的各种不同类型的I/O，LRU内存页中需要刷新的页 - 脏内存页，它们不会被长时间存取；刷新列表 -  
检查点进程处理完之后需要刷新的旧内存页；独立内存页 - 独立的写内存页。

我们还可以看到内存页总共读写了多少次。已经创建的内存页是当前一个内存页中的内容没有读取到内存缓冲池中时，专门为新数据创建的空内存页。

## 【ROW OPERATIONS】



**第4行：**显示了InnoDB内核里有多少条线程；位于队列里的查询就是InnoDB出于限制并发执行的线程数的目的，而还未将其置入内核的线程。这些查询在进入队列前也可以进入睡眠状态；

**第5行：**显示了InnoDB已经打开了多少个读视图。一个读视图就是在事务开始之时取得的一张源自数据库内容的一致的MVCC“快照”，在TRANSACTIONS节里，你查看到一个特定的事务是否拥有一个读视图，

**第6行：**显示了内核主线程的状态可能的状态值包括：

archiving log (如果日志归档功能已经开启的话)

doing background drop tables

doing insert buffer merge

flushing buffer pool pages

flushing log

making checkpoint

purging

reserving kernel mutex

sleeping

suspending

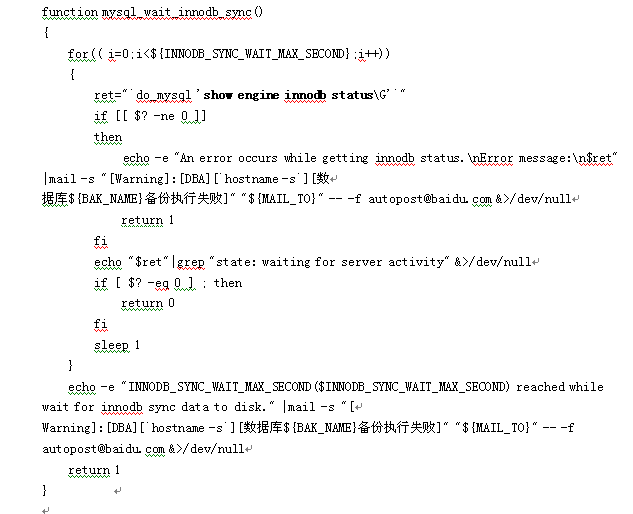
waiting for buffer pool flush to end

waiting for server activity

**第7行到第8行：**显示了行插入，更新，删除和读取次数的统计信息，以及这些数值在每一秒里的平均值。如果你想知道InnoDB正在完成多少工作，查看这些数值就是个不错的办法

show innodb status 的输出以第9到第13行为结束，如果你没看到这几行文字，说明你的系统里可能有一个非常大的死锁，它截断了这些信息的输出；

在备份脚本bak\_space\_mis\_mysql.sh 的一个函数中，用到了，show innodb status ,如下：



可以看出函数mysql\_wait\_innodb\_sync在INNODB\_SYNC\_WAIT\_MAX\_SECOND秒内等待innodb刷盘，并每秒执行一次“show innodb status”来检查row operations 中的状态值，如果是“waiting for server activity”说明刷盘完毕，则正常退出，否则sleep一秒，然后继续判断，直到超时；

最后一部分，一开始显示了Innodb线程队列状态 - 有多少线程处于等待或活跃的。Innodb内部打开了多少读视图 -这是在事务开始后，但是当前还没有活跃语句的情况，Innodb主线程的状态控制了系统操作调度的数量 - 刷新脏内存页、检查点、净化线程、刷新日志、合并插入缓冲等。 "state" 的值则表示了主线程当前的状态。 有一点需要注意的是，SHOW INNODB STATUS 不是一成不变的，有些时间点上可能会不相符。SHOW INNODB STATUS结果中，不同时间可能会显示不同结果，因此有些时候可能会看到冲突的信息。这是由于设计时需要由全局锁提供一致性信息，导致了大量的开销。