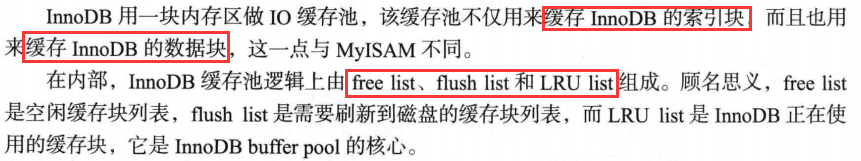
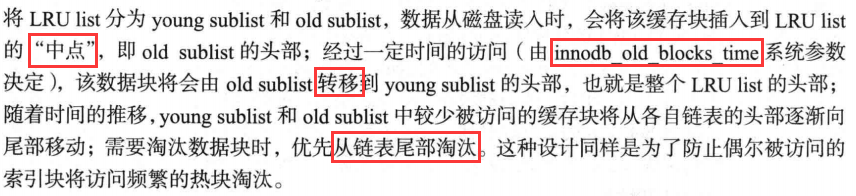
# 21.2.3 InnoDB内存优化

## Innodb缓存机制

缓存innodb索引块和数据块

缓存池由free list和flush list和LRU list组成

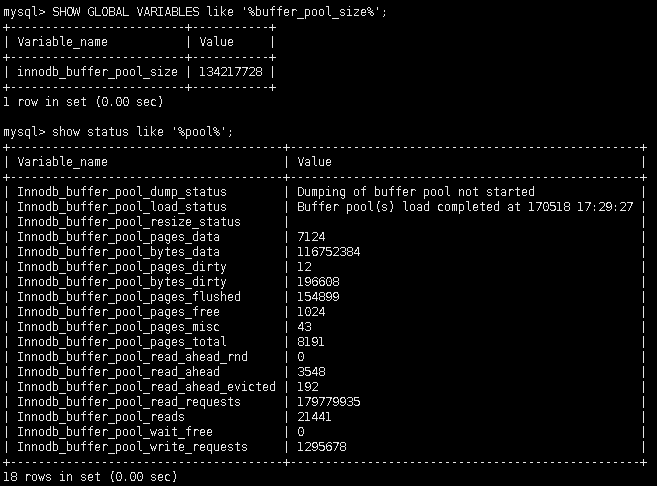




## Innodb\_buffer\_pool\_size查询

mysql> SHOW GLOBAL VARIABLES like '%buffer\_pool\_size%';

mysql> show status like '%pool%';



# 2@MySQL源码学习：简述InnoDB的BP LRU策略 - 推酷

http://www.tuicool.com/articles/e6J32ev

时间 2016-03-24 13:28:02  [云栖博客](http://www.tuicool.com/sites/r2MJJf2)

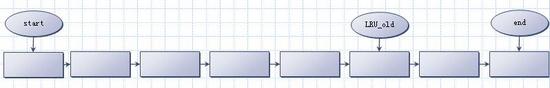
原文  [https://yq.aliyun.com/articles/8827](https://yq.aliyun.com/articles/8827?utm_source=tuicool&utm_medium=referral)

主题 [InnoDB](http://www.tuicool.com/topics/11030012)

本文简要说明 InnoDB 的 Buffer Pool(BP) 的结构、基本运行方式和策略。

**1、 LRU** **的基本形态**

由于涉及到淘汰机制， Buffer Pool (BP) 内需要一个 LRU 链。这个 LRU 链表的基本形态如下：



从图中看到， LRU 是一个链表（双向，图中没有画出反向指针）。

同时有一个 LRU\_old(buf\_pool->LRU\_old) 指针指向链表中间的一个 page 。 LRU\_old 指向的 page 及之后直到 end 的 page ，都被称为 ”old page”, 内存中 bpage->old==1 。

LRU\_old 之前到 start 的所有 page ，被称为 ”young page”, 内存中 bpage->old==0.

2、 **从头开始**

在系统初始化时，所有的 page 都是空闲的，因此全部放在 buf\_pool->free 链表中，此时 buf\_pool.LRU={count = 0, start = 0x0, end = 0x0}, 当然 buf\_pool->LRU\_old=0x0.

当有 page 请求时，从 buf\_pool->free 中取出 page ，放入 LRU 中。需要注意的是，在 LRU->count 小于 512(BUF\_LRU\_OLD\_MIN\_LEN) 时，所有的 page 都被标为 young ，插入队头。

当 LRU->count 达到 512 时候，依次作如下动作

将 buf\_pool->LRU\_old, 赋值为 LRU.start, 将 LRU 中的所有 page 都设置为 old (buf\_LRU\_old\_init)

调用 buf\_LRU\_old\_adjust\_len ，调整 buf\_pool->LRU\_old 的适当位置，成为上图的基本形态。默认配置下 old page 数目占 3/8.

有新的 page 再进入 LRU 时，先插入到 LRU\_old 的 next 位置，也就是先标为 old ，下次访问时再调整为 LRU.start ，再改为 young 。

当 BP 满了以后，即 LRU.count 为 page 总数，再需要访问新的 page 时，就只能从 LRU 末尾删除，再补入。

3、 **一点讨论**

1) 步骤 d 中所说的 ” 下次访问 ” ，实际上在放入 LRU 之后马上会发生。在 buf\_page\_get\_gen调用 buf\_page\_set\_accessed\_make\_young ，若满足条件则将此 page 调整为 LRU.start 。

需要说明一个参数 buf\_LRU\_old\_threshold\_ms 。当一个 old page 距第一次被访问的时间大于等于 buf\_LRU\_old\_threshold\_ms 时，再次被访问的时候，就会被调整为 LRU.start.

也就是说，当 buf\_LRU\_old\_threshold\_ms 为默认设置的 0 时，新插入的 page 都是先放到 LRU\_old 之后，马上被调整到 LRU.start 。

而这个“调整”，也不是简单的指针重赋，而是将这个 page 从 LRU 中移除，再插入到 LRU 头部。而从 LRU 中删除 page 的时候，若刚好碰到临界值 (<512) ，会遍历整个队列，全部设置为 young 。

虽然都是内存操作，但整个过程显得比较粗暴。大多数的系统中并不会修改 buf\_LRU\_old\_threshold\_ms 的默认值，因此这个过程则一直在被重复调用。

实际上，在第一次访问 page 需要入 LRU 队列的时候，完全可以先判断一下 buf\_LRU\_old\_threshold\_ms 的值，若为 0 ，则直接插入到 LRU 头部。对应的代码在 buf\_page\_init\_for\_read 中的两处调用 buf\_LRU\_add\_block(bpage, TRUE) 。

2) 当前的调用流程

buf\_page\_set\_accessed\_make\_young(&block->page, access\_time); --> buf\_LRU\_make\_block\_young --> buf\_LRU\_remove\_block(bpage); buf\_LRU\_add\_block\_low(bpage, FALSE);

4、 **一点声明**

本文基本上是为下周组里要来的小实习生作个介绍，因此写得又细又浅，欢迎拍砖。可以踩，但请果断留下意见。

声明：云栖社区站内文章，未经作者本人允许或特别声明，严禁转载，但欢迎分享。

# mysql之InnoDB内存管理

- 踏雪无痕SS - 博客园

http://www.cnblogs.com/chenpingzhao/p/4873258.html

InnoDB缓冲池是通过LRU算法来管理page的。频繁使用的page放在LRU列表的前端，最少使用的page在LRU列表的尾端，缓冲池满了的时候，优先淘汰尾端的page。

**InnoDB中的LRU结构**  
  
InnoDB引擎中page的默认大小为16KB，InnoDB对传统的LRU算法做了一些优化  
  
LRU列表被分成两部分，midpoint点之前的部分称为new列表，之后的部分称为old列表，new列表中的页都是最为活跌的热点数据。midpoint的位置通过参数innodb\_old\_blocks\_pct来设置。  
  
参数innodb\_old\_blocks\_pct默认值为37，表示新读取的page将被插入到LRU列表左侧的37%（差不多3/8的位置）。

[?](http://www.cnblogs.com/chenpingzhao/p/4873258.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | mysql> show variables like 'innodb\_old\_blocks%';  +------------------------+-------+  | Variable\_name          | Value |  +------------------------+-------+  | innodb\_old\_blocks\_pct  | 37    |  | innodb\_old\_blocks\_time | 0     |  +------------------------+-------+ |

**为什么不采用传统的LRU算法？**  
  
若直接将读取到的page放到LRU的首部，那么某些SQL操作可能会使缓冲池中的page被刷出。常见的这类操作为索引或数据的扫描操作。这类操作访问表中的许多页，而这些页通常只是在这次查询中需要，并不是活跃数据。如果放入到LRU首部，那么非常可能将真正的热点数据从LRU列表中移除，在下 一次需要时，InnoDB需要重新访问磁盘读取，这样性能会低下。  
  
同时，InnoDB进一步引入了另一个参数来管理LRU列表，这个参数就是innodb\_old\_blocks\_time，用于表示page放到midpoint位置后需要等待多久才会被加入到LRU列表的new端成为热点数据。

**LRU中page的变化**  
  
数据库启动时，LRU列表是空的，即没有任何page，这时page都存放在Free列表中。当需要从缓冲池中分页时，首先从Free列表中查找是 否有可用的空闲页，若有则将page从Free中删除，放入到LRU中。否则，根据LRU算法，淘汰LRU列表末尾的页分配给新的页。  
  
当页从old部分进入到new部分时，此时发生的操作为page made young。因为innodb\_old\_blocks\_time参数导致page没有从old移动到new部分称为page not made young。可以通过命令show engine innodb status来观察LRU列表及Free列表的状态。

[?](http://www.cnblogs.com/chenpingzhao/p/4873258.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | ----------------------  BUFFER POOL AND MEMORY  ----------------------  Total memory allocated 4395630592; in additional pool allocated 0  Dictionary memory allocated 28892957  Buffer pool size   262143  Free buffers       0  Database pages     258559  Old database pages 95424  Modified db pages  36012  Pending reads 0  Pending writes: LRU 0, flush list 0, single page 0  Pages made young 72342127, not young 0  8.82 youngs/s, 0.00 non-youngs/s  Pages read 72300801, created 339791, written 13639066  8.56 reads/s, 0.35 creates/s, 3.79 writes/s  Buffer pool hit rate 1000 / 1000, young-making rate 0 / 1000 not 0 / 1000  Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s  LRU len: 258559, unzip\_LRU len: 0  I/O sum[459]:cur[1], unzip sum[0]:cur[0] |

    Buffer pool size表示缓冲池共有262143个page，即262143 \* 16K，约为4GB  
    Free buffers表示当前Free列表中page的数量  
    Database pages表示LRU列表中page的数量  
    Old database pages表示LRU列表中old部分的page数量  
    Modified db pages表示的是脏页(dirty page)的数量  
    Pages made young表示LRU列表中page移动到new部分的次数  
    youngs/s, non-youngs/s表示每秒这两种操作的次数  
    Buffer pool hit rate表示缓冲池的命中率，该值若小于95%，需要观察是否全表扫描引起LRU污染  
    LRU len表示LRU中总page数量  
  
可以看到Free buffers与Database pages的和不等于Buffer pool size，这是因为缓冲池中的页还会被分配给自适应哈希索引，Lock信息，Insert Buffer等页，这部分页不需要LRU算法维护。

**脏页(dirty page)**  
  
LRU列表中的page被修改后，称该页为脏页，即缓冲池中的页和磁盘上的页的数据产生了不一致。这时InnoDB通过Checkpoint机制将 脏页刷新回磁盘。而Flush列表中的页即为脏页列表。脏页既存在于LRU列表中，又存在于Flush列表中，二者互不影响。Modified db pages显示的就是脏页的数量。

**重做日志缓冲**  
  
InnoDB引擎首先将重做日志信息先放到重做日志缓冲区(redo log buffer)，然后按一定频率刷新到重做日志文件。重做日志缓冲不需要设置很大，一般每一秒都会刷新redo log buffer，配置的大小只需要保证每秒产生的事务在这个缓冲区大小之内即可。通过参数innodb\_log\_buffer\_size为设置：

[?](http://www.cnblogs.com/chenpingzhao/p/4873258.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | mysql> show variables like 'innodb\_log\_buffer%';  +------------------------+----------+  | Variable\_name          | Value    |  +------------------------+----------+  | innodb\_log\_buffer\_size | 16777216 |  +------------------------+----------+ |

在下列情况下会将重做日志缓冲中的内容刷新到磁盘重做日志文件中：  
  
    Master Thread每一秒中刷新一次  
    每个事务提交时会刷新  
    当重做日志缓冲区空间小于1/2时  
  
**额外内存池**  
  
额外的内存池用来对一些数据结构本身的内存进行分配，例如缓冲控制对象(buffer control block)记录的LRU，锁，等待等信息。额外的内存池不够时会从缓冲池中进行申请。因此，在申请了很大的InnoDB缓冲池时，额外的内存池也要适当 的调大。通过参数innodb\_additional\_mem\_pool\_size来设置大小。查看通过如下命令：

[?](http://www.cnblogs.com/chenpingzhao/p/4873258.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | mysql> show variables like '%pool\_size';  +---------------------------------+------------+  | Variable\_name                   | Value      |  +---------------------------------+------------+  | innodb\_additional\_mem\_pool\_size | 67108864   |  | innodb\_buffer\_pool\_size         | 4294967296 |  +---------------------------------+------------+ |

 原文来自：http://my.oschina.net/jockchou/blog/478082

* 作者：[踏雪无痕](http://weibo.com/chenpingzhao)
* 出处：<http://www.cnblogs.com/chenpingzhao/>
* 本文版权归作者和博客园共有，如需转载，请联系[**pingzhao1990#163.com**](mailto:pingzhao1990@163.com)

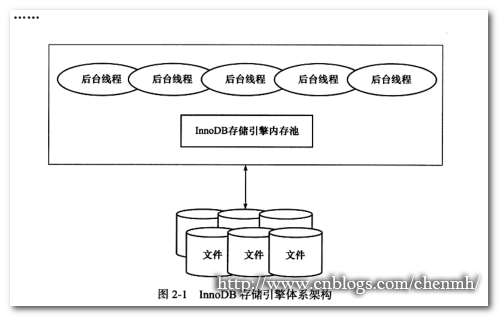
# MySQL InnoDB存储引擎

- pursuer.chen - 博客园

http://www.cnblogs.com/chenmh/p/5119593.html

**介绍**

 本篇文章是对Innodb存储引擎的概念进行一个整体的概括，innodb存储引擎的概念是mysql数据库中最关键的几个概念之一，涉及的内容非常的广；由于个人的理解能力有限如果有不对的地方还见谅。

****

**MySQL对应InnoDB版本**

MySQL 5.1》InnoDB 1.0.X

MySQL 5.5》InnoDB 1.1.X

MySQL 5.6》InnoDB 1.2.X

**后台线程**

1.Master Thread

负责将缓冲池中的数据异步刷新到磁盘，保证数据的一致性；包括刷新脏页、合并插入缓冲、undo页的回收。

2.IO Thread

innodb存储引擎中大量使用了AIO(Async IO)来处理写IO请求来提高数据库的并发性能，共有四类IO线程，分别是：insert buffer thread、log thread、read thread、write thread。其中read thread和write thread分别有四个线程，可以通过innodb\_read\_io\_threads和innodb\_write\_io\_threads来配置。

SHOW VARIABLES LIKE 'innodb\_%io\_threads'

或者

SHOW ENGINE INNODB STATUS \G;

3.Purge Thread线程

purge Thread线程用来回收事务提交后其被分配的undo页，默认是开启的，可以通过innodb\_purge\_threads=1配置多个Purge Thread线程。

show variables like 'innodb\_purge\_threads';  
  
配置2个Purge thread，只能修改配置文件配置，不能在线修改   
innodb\_purge\_threads=2

4.Page Cleaner Thread

用于多版本控制功能中回收delete和update操作产生的脏页,用来执行将脏页刷新到磁盘。

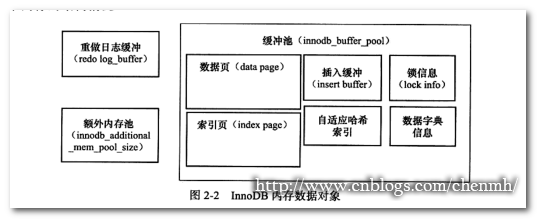
5.Binlog Dump线程

当配置了复制后，会在主服务器生成一个binlog Dump线程来读取二进制修改记录。

6.lock线程

      用于锁控制和死锁检测

**内存**

****

 不要理解以为内存中就只有innodb buffer，还包括重做日志缓冲、额外的内存池（目前还不知道比如join buffer、order buffer、key buffer、table cache buffer等是在缓冲池内部还是独立于缓冲池在内存中）

**1.缓存池**

缓存的数据主要有数据页、索引页、重做日志页（undolog）、节点信息、系统数据、插入缓冲、自适应哈希索引、数据字典、锁信息等

[复制代码](javascript:void(0);)

查看缓冲池的大小，单位字节，转化为MB需要/1024/1024

show variables like 'innodb\_buffer\_pool\_size';  
  
默认innodb有8个缓冲池，可以通过配置innodb\_buffer\_pool\_instances  
查询  
show engine innodb status \G;  
或者  
SELECT \* FROM information\_schema.innodb\_buffer\_pool\_status;

[复制代码](javascript:void(0);)

读操作：

   数据是以页为存储单位，在缓冲池中缓存了很多数据页，当第一次读取时首先将页从磁盘读取到缓存池中，当下一次再去读相同的数据页时如果该也在缓存池中就直接从缓冲池中读取而不需要再去磁盘读，最理想的方式是将所有的磁盘数据都缓存到缓冲池中但是这得内存足够大才行。

修改操作

innodb存储引擎对数据的修改也是先修改缓冲池中的数据页（如果存在），然后根据一定的频率刷新到磁盘来修改数据文件，这涉及到checkpoint机制，

插入操作（insert buffer）

因为数据是按照聚集索引的顺序排列的，所有针对聚集索引的插入一般会非常快，而非聚集索引的插入就不一定是顺序的，这个时候需要离散的访问非聚集索引页，插入的性能往往会很差，有一种情况可能例外就是非聚集索引的时间字段，而时间往往是顺序的，这种情况会比较快，针对非聚集索引的这种情况就引入了插入缓冲。

innodb中引入了插入缓冲（insert buffer），insert buffer只针不唯一的非聚集索引，对于非聚集索引的插入和更新操作不是每次直接插入到索引文件中，而是先判断插入的非聚集索引页是否存在缓冲池中，如果存在则直接插入缓冲池的非聚集索引文件中，否则先放入到一个insert buffer对象当中，但是给人的感觉它已经插入到了索引文件中，但是实际并没有，然后再以一定的频率插入到索引文件当中，在这个过程中如果存在多个相同的索引页的插入会合并插入，大大的提高了非聚集索引的插入性能，

因为每次插入是先插入到缓冲池当中不去查找索引页来判断记录的唯一性，因为去做判断需要去离散查找，所以插入缓冲不针对唯一性的非聚集索引。

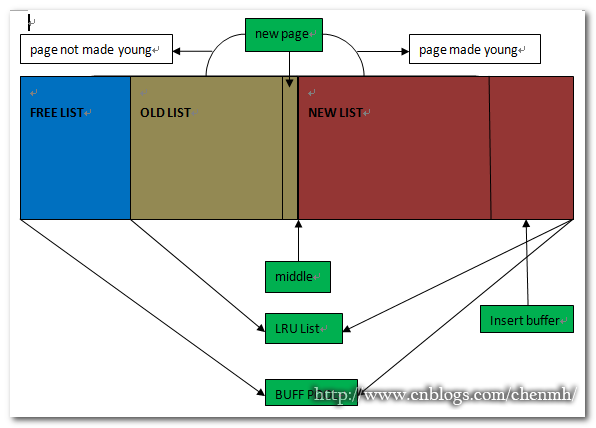
在密集写操作的情况下，插入缓冲会占用过多的缓冲池的内存，默认最大可以占到50%，源代码中的IBUF\_POOL\_SIZE\_PER\_MAX\_SIZE=2，如果将其修改为3，则最大只能使用1/3的缓冲池的内存。

**2.LRU List、Free List、Flush List**

innodb缓冲池中的页默认大小为16KB，缓冲池通过LRU(Latest Recent Used 最新少使用)算法来进行管理，将最频繁的使用的页放在LRU列表的前端，而最近少使用的页放在尾端，当缓存池中的空间不足的时会先删除尾端的页来释放空间。LRU有一个midpoint位置，默认在LRU的37%的位置，左边表示old列表，右边表示new列表（热点数据），新插入缓冲池中的页先放在midpoint的位置，如果新插入的页一来就移动到new列表的话可能会导致new列表中的某些活动也被移除到old列表中，比如表扫描操作一次性可能需要访问很多的数据页而这些数据页可能以后很少被使用，新插入的页何时才会被被放入new列表中呢，为了解决这个问题innodb引入了innodb\_old\_blocks\_time参数，该参数用来控制新插入的数据页在mid位置多久后才被加入到new列表中。

查看midpoint的位置，如果觉得热点数据空间需要更多可以将该值设小

show variables like 'innodb\_old\_blocks\_pct'  
  
查询innodb\_old\_blocks\_time值，单位毫秒，默认是1000毫秒即1秒  
show variables like 'innodb\_old\_blocks\_time'



[复制代码](javascript:void(0);)

查看缓冲池中所有页的信息,包括空闲页，所有的数据页\*16KB其实也就是缓冲池的总大小.

select \* from information\_schema.INNODB\_BUFFER\_PAGE;

查看LUR列表的信息,包括new list和old list但是不包括free list,表中的字段记录了当前的数据页的信息，包括缓冲池ID,页的类型（数据页、索引页、undo log、other）,表名,索引名,是否是old list的页,是否属于压缩页（可以将原本16K的页压缩为1K、2K、4K、8K）,压缩页的大小,是否属于脏页。

select POOL\_ID,LRU\_POSITION,SPACE,PAGE\_TYPE,FLUSH\_TYPE,NEWEST\_MODIFICATION,OLDEST\_MODIFICATION,INDEX\_NAME,DATA\_SIZE,COMPRESSED\_SIZE,COMPRESSED,IS\_OLD from information\_schema.INNODB\_BUFFER\_PAGE\_LRU;

OLDEST\_MODIFICATION>0表示脏页的数量也就是（modified db pages）  
IS\_OLD='YES'代表OLD List页  
COMPRESSED<>0代表压缩页

[复制代码](javascript:void(0);)

**flush list:**值的就是LRU中的脏页，flust list存在于New List中，即OLDEST\_MODIFICATION>0（modified db pages）

**3.日志缓冲（log buffer）：对应innodb日志文件**

查看重做日志缓冲

show variables like 'innodb\_log\_buffer\_size%';

InnoDB存储引擎首先将重做日志信息先放入到重做日志缓冲中，然后按照一定的频率将其刷新到重做日志文件当中。默认缓冲大小是8M，8M基本可以满足需求，不需要配置太大的重做日志缓冲。

刷新机制：

1.Master Thread 每一秒将重做日志缓冲刷新到重做日志文件；

2.每个事务提交时会将重做日志缓冲刷新到重做日志文件；

3.当重做日志缓冲池剩余空间小于1/2时

注意：innodb\_log\_buffer\_size的大小应该要比最大的事务大小要打，否则事务还未提交innodb\_log\_buffer\_size就已经写满就需要进行刷新操作，会造成一个事务需要多次进行磁盘日志刷新操作，导致效率低。

**4.额外的内存**

平时我们的服务器MySQL进程所使用的内存会比配置的InnoDB缓冲池的内存要大，那是因为MySQL除了缓冲池中缓存的内存额外还需要一部分内存用来控制缓冲池内部的一些资源信息，比如LRU、锁资源、等待等。

**CheckPoint机制**

为了解决CPU和磁盘直接速度的问题采用了缓冲池，所以对数据的操作都是先在缓冲池中完成，缓冲池中的数据页往往比磁盘上的数据页要新，我们将在缓冲池中已经修改但是还未应用到磁盘的数据页叫“脏页”，数据页最终还是需要更新到磁盘中，中间会涉及到CheckPoint机制。

同时为了解决因为突然服务器停机导致缓冲池中还未来得及刷新到磁盘的脏页丢失的问题，加入了重做日志文件（重做日志文件默认是配置2个，默认名称是ib\_logfile开头，重做日志文件默认大小是48M，两个重做日志文件采取循环写的方式），当事务提交时先写重做日志，当发生服务器停机后可以通过重做日志来完成恢复（服务器重启之后自己默认会恢复），所以得保证重做日志文件有剩余空间，默认机制是当重做日志文件空间达到75%-90%时就刷新一部分脏页到磁盘同时清空对应的重做日志空间。

**每次刷新多少页到磁盘：**

**Sharp Checkpoint:数据库关闭时将所有脏页都刷新回磁盘，默认方式，参数：innodb\_fast\_shutdown=1**

**Fuzzy Checkpoint:刷新部分脏页，具体分为以下四种情况**

1.Master Thread Checkpoint

Master Thread每隔几秒钟从缓冲池中将脏页刷新回磁盘

2.FLUSH\_LRU\_LIST CheckPoint

在5.6版本之后需要保证LRU默认存在1024个可用页，如果可用页不足1024页刷新部分脏页回磁盘，通过参数“innodb\_lru\_scan\_dapth”配置。

3.Async/Sync Flush Checkpoint

指的是因为重做日志文件空间不足导致的同步或异步刷新脏页回磁盘，当重做日志空间已使用的空间达到75%-90%就触发异步刷新，如果超过90%就触发同步刷新，一般不会触发同步刷新操作，除非重做日志文件太小并且进行LOAD DATA的BULK INSERT操作。

4.Dirty Page too much

保证缓冲池中脏页的比例，当缓冲池中的脏页比例达到75%时就触发刷新脏页操作，通过参数“innodb\_max\_dirty\_pages\_pct”配置。

**总结**

innodb存储引擎的概念非常的多，随便一个知识点都不止一篇文章可以写下，所以本篇只是会整体做一个描述后面会针对每一个知识点进行更细的分析。

|  |
| --- |
| 备注：      作者：[pursuer.chen](http://www.cnblogs.com/chenmh/)      博客：[http://www.cnblogs.com/chenmh](http://www.cnblogs.com/chenmh/)  本站点所有随笔都是原创，欢迎大家转载；但转载时必须注明文章来源，且在文章开头明显处给明链接。  《欢迎交流讨论》 |

# innodb - How large should be mysql innodb\_buffer\_pool\_size? - Database Administrators Stack Exchange

https://dba.stackexchange.com/questions/27328/how-large-should-be-mysql-innodb-buffer-pool-size

I have a busy database with solely InnoDB tables which is about 5GB in size. The database runs on a Debian server using SSD disks and I've set max connections = 800 which sometimes saturate and grind the server to halt. The average query per second is about 2.5K. So I need to optimize memory usage to make room for maximum possible connections.

I've seen suggestions that innodb\_buffer\_pool\_size should be up to %80 of the total memory. On the other hand I get this warning from tuning-primer script:

Max Memory Ever Allocated : 91.97 G

Configured Max Per-thread Buffers : 72.02 G

Configured Max Global Buffers : 19.86 G

Configured Max Memory Limit : 91.88 G

Physical Memory : 94.58 G

Here are my current innodb variables:

| innodb\_adaptive\_flushing | ON |

| innodb\_adaptive\_hash\_index | ON |

| innodb\_additional\_mem\_pool\_size | 20971520 |

| innodb\_autoextend\_increment | 8 |

| innodb\_autoinc\_lock\_mode | 1 |

| innodb\_buffer\_pool\_instances | 1 |

| innodb\_buffer\_pool\_size | 20971520000 |

| innodb\_change\_buffering | all |

| innodb\_checksums | ON |

| innodb\_commit\_concurrency | 0 |

| innodb\_concurrency\_tickets | 500 |

| innodb\_data\_file\_path | ibdata1:10M:autoextend |

| innodb\_data\_home\_dir | |

| innodb\_doublewrite | ON |

| innodb\_fast\_shutdown | 1 |

| innodb\_file\_format | Antelope |

| innodb\_file\_format\_check | ON |

| innodb\_file\_format\_max | Antelope |

| innodb\_file\_per\_table | ON |

| innodb\_flush\_log\_at\_trx\_commit | 2 |

| innodb\_flush\_method | O\_DIRECT |

| innodb\_force\_load\_corrupted | OFF |

| innodb\_force\_recovery | 0 |

| innodb\_io\_capacity | 200 |

| innodb\_large\_prefix | OFF |

| innodb\_lock\_wait\_timeout | 50 |

| innodb\_locks\_unsafe\_for\_binlog | OFF |

| innodb\_log\_buffer\_size | 4194304 |

| innodb\_log\_file\_size | 524288000 |

| innodb\_log\_files\_in\_group | 2 |

| innodb\_log\_group\_home\_dir | ./ |

| innodb\_max\_dirty\_pages\_pct | 75 |

| innodb\_max\_purge\_lag | 0 |

| innodb\_mirrored\_log\_groups | 1 |

| innodb\_old\_blocks\_pct | 37 |

| innodb\_old\_blocks\_time | 0 |

| innodb\_open\_files | 300 |

| innodb\_purge\_batch\_size | 20 |

| innodb\_purge\_threads | 0 |

| innodb\_random\_read\_ahead | OFF |

| innodb\_read\_ahead\_threshold | 56 |

| innodb\_read\_io\_threads | 4 |

| innodb\_replication\_delay | 0 |

| innodb\_rollback\_on\_timeout | OFF |

| innodb\_rollback\_segments | 128 |

| innodb\_spin\_wait\_delay | 6 |

| innodb\_stats\_method | nulls\_equal |

| innodb\_stats\_on\_metadata | ON |

| innodb\_stats\_sample\_pages | 8 |

| innodb\_strict\_mode | OFF |

| innodb\_support\_xa | ON |

| innodb\_sync\_spin\_loops | 30 |

| innodb\_table\_locks | ON |

| innodb\_thread\_concurrency | 4 |

| innodb\_thread\_sleep\_delay | 10000 |

| innodb\_use\_native\_aio | ON |

| innodb\_use\_sys\_malloc | ON |

| innodb\_version | 1.1.8 |

| innodb\_write\_io\_threads | 4 |

A side note that might be relevant: I see that when I try to insert a large post (say over 10KB) from Drupal (which sits on a separate web server) to database, it lasts forever and the page does not return correctly.

Regarding these, I'm wondering what should be my innodb\_buffer\_pool\_size for optimal performance. I appreciate your suggestions to set this and other parameters optimally for this scenario.

Your [**innodb\_buffer\_pool\_size**](http://dev.mysql.com/doc/refman/5.5/en/innodb-parameters.html#sysvar_innodb_buffer_pool_size) is enormous. You have it set at 20971520000. That's 19.5135 GB. If you only 5GB of InnoDB data and indexes, then you should only have about 8GB. Even this may too high.

Here is what you should do. First run this query

SELECT CEILING(Total\_InnoDB\_Bytes\*1.6/POWER(1024,3)) RIBPS FROM

(SELECT SUM(data\_length+index\_length) Total\_InnoDB\_Bytes

FROM information\_schema.tables WHERE engine='InnoDB') A;

This will give you the RIBPS, Recommended InnoDB Buffer Pool Size based on all InnoDB Data and Indexes with an additional 60%.

For Example

mysql> SELECT CEILING(Total\_InnoDB\_Bytes\*1.6/POWER(1024,3)) RIBPS FROM

-> (SELECT SUM(data\_length+index\_length) Total\_InnoDB\_Bytes

-> FROM information\_schema.tables WHERE engine='InnoDB') A;

+-------+

| RIBPS |

+-------+

| 8 |

+-------+

1 row in set (4.31 sec)

mysql>

With this output, you would set the following in /etc/my.cnf

[mysqld]

innodb\_buffer\_pool\_size=8G

Next, service mysql restart

After the restart, run mysql for a week or two. Then, run this query:

SELECT (PagesData\*PageSize)/POWER(1024,3) DataGB FROM

(SELECT variable\_value PagesData

FROM information\_schema.global\_status

WHERE variable\_name='Innodb\_buffer\_pool\_pages\_data') A,

(SELECT variable\_value PageSize

FROM information\_schema.global\_status

WHERE variable\_name='Innodb\_page\_size') B;

This will give you how many actual GB of memory is in use by InnoDB Data in the InnoDB Buffer Pool at this moment.

I have written about this before : [What to set innodb\_buffer\_pool and why..?](https://dba.stackexchange.com/questions/19164/what-to-set-innodb-buffer-pool-and-why/19181#19181)

You could just run this DataGB query right now rather than reconfiguring, restarting and waiting a week.

This value DataGB more closely resembles how big the InnoDB Buffer Pool should be + (percentage specified in innodb\_change\_buffer\_max\_size). I am sure this will be far less than the 20000M you have reserved right now. The savings in RAM can be used for tuning other things like

* [join\_buffer\_size](http://dev.mysql.com/doc/refman/5.5/en/server-system-variables.html#sysvar_join_buffer_size)
* [sort\_buffer\_size](http://dev.mysql.com/doc/refman/5.5/en/server-system-variables.html#sysvar_sort_buffer_size)
* [read\_buffer\_size](http://dev.mysql.com/doc/refman/5.5/en/server-system-variables.html#sysvar_read_buffer_size)
* [read\_rnd\_buffer\_size](http://dev.mysql.com/doc/refman/5.5/en/server-system-variables.html#sysvar_read_rnd_buffer_size)
* [max\_connection](http://dev.mysql.com/doc/refman/5.5/en/server-system-variables.html#sysvar_max_connections)

CAVEAT #1

This is very important to note : At times, InnoDB may require an additional 10% over the value for the[**innodb\_buffer\_pool\_size**](http://dev.mysql.com/doc/refman/5.5/en/innodb-parameters.html#sysvar_innodb_buffer_pool_size). Here is what the [MySQL Documentation](http://dev.mysql.com/doc/refman/5.5/en/innodb-parameters.html#sysvar_innodb_buffer_pool_size) says on this:

The larger you set this value, the less disk I/O is needed to access data in tables. On a dedicated database server, you may set this to up to 80% of the machine physical memory size. Be prepared to scale back this value if these other issues occur:

Competition for physical memory might cause paging in the operating system.

InnoDB reserves additional memory for buffers and control structures, so that the total allocated space is approximately 10% greater than the specified size.

The address space must be contiguous, which can be an issue on Windows systems with DLLs that load at specific addresses.

The time to initialize the buffer pool is roughly proportional to its size. On large installations, this initialization time may be significant. For example, on a modern Linux x86\_64 server, initialization of a 10GB buffer pool takes approximately 6 seconds. See Section 8.9.1, [“The InnoDB Buffer Pool”](http://dev.mysql.com/doc/refman/5.5/en/innodb-buffer-pool.html).

CAVEAT #2

I See the following values in your my.cnf

| innodb\_io\_capacity | 200 |

| innodb\_read\_io\_threads | 4 |

| innodb\_thread\_concurrency | 4 |

| innodb\_write\_io\_threads | 4 |

These number will impede InnoDB from accessing multiple cores

Please set the following:

[mysqld]

innodb\_io\_capacity = 2000

innodb\_read\_io\_threads = 64

innodb\_thread\_concurrency = 0

innodb\_write\_io\_threads = 64

I have written about this before in the DBA StackExchange

* May 26, 2011 : [About single threaded versus multithreaded databases performance](https://dba.stackexchange.com/questions/2918/about-single-threaded-versus-multithreaded-databases-performance/2948#2948)
* Sep 12, 2011 : [Possible to make MySQL use more than one core?](https://dba.stackexchange.com/questions/5666/possible-to-make-mysql-use-more-then-one-core/5670#5670)
* Sep 20, 2011 : [Multi cores and MySQL Performance](https://dba.stackexchange.com/questions/5926/multi-cores-and-mysql-performance/5968#5968)

I just [answered a question like this in ServerFault using a more concise formula](https://serverfault.com/a/441097/69271):

SELECT CONCAT(CEILING(RIBPS/POWER(1024,pw)),SUBSTR(' KMGT',pw+1,1))

Recommended\_InnoDB\_Buffer\_Pool\_Size FROM

(

SELECT RIBPS,FLOOR(LOG(RIBPS)/LOG(1024)) pw

FROM

(

SELECT SUM(data\_length+index\_length)\*1.1\*growth RIBPS

FROM information\_schema.tables AAA,

(SELECT 1.25 growth) BBB

WHERE ENGINE='InnoDB'

) AA

) A;

# InnoDB variables and status explained | MySQL, Galera Cluster and MariaDB support and services

http://www.fromdual.com/innodb-variables-and-status-explained

**INNODB VARIABLES AND STATUS EXPLAINED**

**INNODB BUFFER POOL**

The InnoDB Buffer Pool is the memory area where the InnoDB Storage Engine caches its data and index blocks. Each InnoDB data and index block has a size of [Innodb\_page\_size](http://dev.mysql.com/doc/refman/5.5/en/server-status-variables.html#statvar_Innodb_page_size) (16384 byte = 16 kbyte). The InnoDB Buffer Pool is configured in bytes with the [innodb\_buffer\_pool\_size](http://dev.mysql.com/doc/refman/5.5/en/innodb-parameters.html#sysvar_innodb_buffer_pool_size) variable. On a dedicated system the InnoDB Buffer Pool can be configured up to 80% of the systems physical RAM (free).

The [innodb\_buffer\_pool\_awe\_mem\_mb](http://dev.mysql.com/doc/refman/5.1/en/innodb-parameters.html#sysvar_innodb_buffer_pool_awe_mem_mb) variable is relevant only on 32-bit Windows systems with more than 4 Gbyte of RAM using the so-called *Address Windowing Extensions* ([AWE](http://en.wikipedia.org/wiki/Address_Windowing_Extensions)).

The usage of the InnoDB Buffer Pool can be measured with the SHOW GLOBAL STATUS LIKE 'Innodb\_buffer\_pool\_pages\_%' command. The sum of *data*, *misc* and *free* pages is equivalent to total pages. And the number of *total* pages multiplied by [Innodb\_page\_size](http://dev.mysql.com/doc/refman/5.5/en/server-status-variables.html#statvar_Innodb_page_size) corresponds to your[innodb\_buffer\_pool\_size](http://dev.mysql.com/doc/refman/5.5/en/innodb-parameters.html#sysvar_innodb_buffer_pool_size).

Innodb\_buffer\_pool\_pages\_data 1757

Innodb\_buffer\_pool\_pages\_misc + 10

Innodb\_buffer\_pool\_pages\_free + 2072

Innodb\_buffer\_pool\_pages\_total = 3839

Innodb\_buffer\_pool\_pages\_total x Innodb\_page\_size = innodb\_buffer\_pool\_size

3839 x 16384 = 62898176 (= 60 Mbyte)

[Innodb\_buffer\_pool\_pages\_data](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_pages_data) shows the number of dirty and clean data and index pages.[Innodb\_buffer\_pool\_pages\_misc](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_pages_misc) shows the number of pages that are busy because they have been allocated for administrative overhead such as row locks or the adaptive hash index.

A small number of [Innodb\_buffer\_pool\_pages\_free](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_pages_free) pages does not necessarily indicate that you InnoDB Buffer Pool is too small. Where instead a large number of free pages over a longer period is a strong indicator that your InnoDB Buffer Pool is too big and can easily be decreased.

[Innodb\_buffer\_pool\_pages\_dirty](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_pages_dirty) indicates the number of InnoDB buffer pool data pages that have been changed in memory, but the changes are not yet written (flushed) to the InnoDB data files. The opposite of a dirty page is a clean page.

The InnoDB main thread tries to write pages from the InnoDB Buffer Pool so that the percentage of dirty (not yet written) pages will not exceed the value of [Innodb\_buffer\_pool\_pages\_dirty](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_pages_dirty).

Innodb\_buffer\_pool\_pages\_data \* innodb\_max\_dirty\_pages\_pct / 100 > Innodb\_buffer\_pool\_pages\_dirty

1757 \* 90 / 100 > 5

[Innodb\_buffer\_pool\_pages\_flushed](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html" \l "statvar_Innodb_buffer_pool_pages_flushed" \t "_blank) indicates the number of requests to flush pages from the InnoDB buffer pool to the data file.

Similiar information about the InnoDB Buffer Pool constitution can be retrieved with the command: SHOW ENGINE INNODB STATUS\G:

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

BUFFER POOL AND MEMORY

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

Buffer pool size 512

Free buffers 490

Database pages 22

Modified db pages 0

Normally, writes to the InnoDB Buffer Pool happen in the background. However, if it is necessary to read or create a page and no clean pages are available, it is also necessary to wait for pages to be flushed first. The[Innodb\_buffer\_pool\_wait\_free](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_wait_free) counter counts how many times this has happened.[Innodb\_buffer\_pool\_wait\_free](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_wait_free) greater than 0 is a strong indicator that the InnoDB Buffer Pool is too small.

**INNODB BUFFER POOL HIT RATIO**

[Innodb\_buffer\_pool\_read\_requests](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_read_requests) indicates the the number of logical read requests (read from memory) InnoDB has done.

[Innodb\_buffer\_pool\_reads](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_reads) indicates the number of logical reads that InnoDB could not satisfy from the buffer pool, and had to read directly from the disk (physical reads).

The InnoDB Buffer Pool hit ratio is a indicator how often your pages are retrieved from memory instead of disk:

Innodb\_buffer\_pool\_read\_requests / (Innodb\_buffer\_pool\_read\_requests + Innodb\_buffer\_pool\_reads) \* 100 = InnoDB Buffer Pool hit ratio

1600770 ( 1600770 + 1715) \* 100 = 99.9%

The same ratio can be calculated over the last n seconds with the SHOW ENGING INNODB STATUS command:

Per second averages calculated from the last 58 seconds

...

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

BUFFER POOL AND MEMORY

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

...

Buffer pool hit rate 1000 / 1000

A InnoDB Buffer Pool hit ratio below 99.9% is a weak indicator that your InnoDB Buffer Pool could be increased.

**INNODB ADDITIONAL MEMORY POOL**

The [innodb\_additional\_mem\_pool\_size](http://dev.mysql.com/doc/refman/5.1/en/innodb-parameters.html#sysvar_innodb_additional_mem_pool_size) variables configures the size in bytes of a memory pool InnoDB uses to store data dictionary information and other internal data structures. The more tables you have in your application, the more memory you need to allocate here. If InnoDB runs out of memory in this pool, it starts to allocate memory from the operating system and writes warning messages to the MySQL error log.  
How much memory InnoDB has allocated for this additional memory pool can be found with:

pager grep 'additional pool'

SHOW ENGINE INNODB STATUS\G

Total memory allocated 20618000; in additional pool allocated 676608

**QUESTIONS TO ANSWER**

* What happens if innodb reaches innodb\_max\_dirty\_pages\_pct?
* What is stored in innodb buffer pool beside data and index (=misc, row logs, undo?)
* How is flushed related to write requests?

buffer pool lru buffer pool instances

[Innodb\_buffer\_pool\_read\_ahead\_rnd](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_read_ahead_rnd) indicates the number of *random* read-aheads initiated by InnoDB. This happens when a query scans a large portion of a table but in random order. This variable was removed in newer MySQL releases.

[Innodb\_buffer\_pool\_read\_ahead\_seq](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_read_ahead_seq) indicates the number of *sequential* read-aheads initiated by InnoDB. This happens when InnoDB does a sequential full table scan. This variable was removed in newer MySQL releases.

[Innodb\_buffer\_pool\_write\_requests](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Innodb_buffer_pool_write_requests) indicates the number writes done to the InnoDB buffer pool.

The ratio of write requests to pages flushed should be an indicator of how many rows are changed in a block before it is flushed to disk:

Innodb\_buffer\_pool\_write\_requests / Innodb\_buffer\_pool\_pages\_flushed = row changes per flush

8367 / 8160 = 1.02 row changes per flush

A value much higher that 1 is an indicator of a good locality of data

State of 5.0.92

# MySQL系列：innodb引擎分析之内存管理 - 推酷

http://www.tuicool.com/articles/2MNrmq

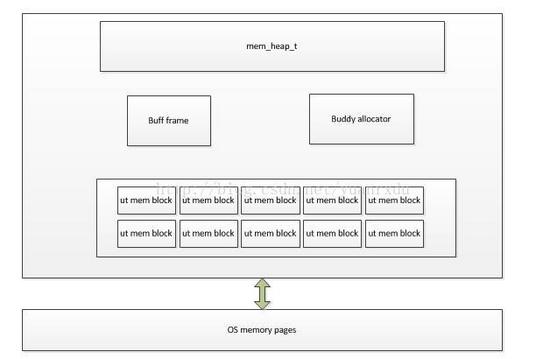
时间 2014-11-10 20:05:05  [CSDN博客](http://www.tuicool.com/sites/vQZRB3)

原文  [http://blog.csdn.net/yuanrxdu/article/details/40985363](http://blog.csdn.net/yuanrxdu/article/details/40985363?utm_source=tuicool&utm_medium=referral)

主题 [InnoDB](http://www.tuicool.com/topics/11030012)

在innodb中实现了自己的内存池系统和内存堆分配系统，在innodb的内存管理系统中，大致分为三个部分：基础的内存块分配管理、内存伙伴分配器和内存堆分配器。innodb定义和实现内存池的主要目的是提供内存的使用率和效率，防止内存碎片和内存分配跟踪和调试。我们先来看看他们的关系和结构。

以下是它的关系结构图：



ut\_mem\_block块是基础内存管理

Buddy allocator是内存伙伴分配器

mem\_heap是内存堆分配器

1.基础内存管理

innodb中的内存分配和内存释放是通过统一的结构进行管理，具体的实现在ut0mem.h和ut0mem.c当中，其中最重要的就是对malloc和free的封装。通过一个链表结构体来管理已经分配的内存，结构体如下：

**typedef** ut\_mem\_block\_struct

{

ulint size; /\*这个被分配block的内存大小\*/

ulint magic\_n; /\*节点魔法字，用于校验所用\*/

UT\_LIST\_NODE\_T(**ut\_mem\_block\_t**) mem\_block\_list; /\*block list node,指定prev node和next node\*/

};

关于block的list定义是个全局的变量，UT\_LIST\_BASE\_NODE\_T(ut\_mem\_block\_t) ut\_mem\_block\_list;所有分配的block都会加入到这个list当中。在ut\_malloc\_low函数分配内存的时候会将分配的block加入到list当中，在ut\_free的时候会所释放的内存所在的block从list当中删除。除了这两个函数以外，innodb还提供ut\_free\_all\_mem函数来释放所有分配的block和统计分配内存的总数ut\_total\_allocated\_memory功能。

基础内存管理的方法如下：

ut\_malloc\_low                    分配一个n长度的内存块，并将分配的块记录到ut\_mem\_block\_list当中.

ut\_malloc                            与ut\_malloc\_low功能相同，但是会用0初始化所分配的内存。

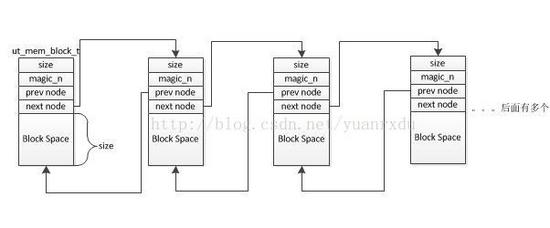
ut\_free                                释放一个分配的内存块，并将其从ut\_mem\_block\_list当中删除。

ut\_free\_all\_mem                 释放ut\_mem\_block\_list所有的内存块并清空ut\_mem\_block\_list

以上函数是支持多线程并发操作的，也就是说是线程安全的。

innodb这样做的目的是保证所有malloc出去的内存都在  ut\_mem\_block\_list当中，以便管理。

基础内存管理的结构如下：



2.伙伴分配器

 innodb的伙伴分配器是基于ut\_malloc\_low函数之上的内存管理器，在创建伙伴分配器时，innodb会一下用ut\_malloc\_low开辟一个很大的内存块，然后用伙伴分配来分配这个块的内存使用。innodb的伙伴分配器是基于2的基数为基础的管理方式，其buddy alloc pool的定义如下：

**struct** **mem\_pool\_struct**

{

byte\* buf; /\*整体内存的句柄\*/

ulint size; /\*整体内存大小\*/

ulint reserved; /\*当前分配出去的总内存大小\*/

mutex mutex; /\*多线程互斥量\*/

UT\_LIST\_BASE\_NODE\_T(mem\_area\_t) free\_list[64]; /\*area\_t链表数组,每个数组单元能管理2的i次方内存块列表，i是数组的下标\*/

}；

**struct** **mem\_area\_struct**

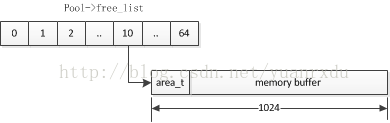
{

ulint size\_and\_free; /\*area的内存大小（一定是2的次方），最后一个bit表示是否已经释放\*/

UT\_LIST\_NODE\_T(mem\_area\_t) free\_list; /\*area链表的上下area,因为buddy area是会分裂的，有可能多个\*/

};

mem\_area\_t是一个buddy的内存区域，也就是mem\_area\_struct。以下是 一个32位机器管理1024字节内存块的buddy list分布：



每一个area是有mem\_area\_t头和可分配的内存（memory\_buffer）确定的，memory\_buffer的长度不小于 mem\_area\_t头的长度，在32位机器上mem\_area\_t的头应该是16个字节（8字节对齐）。

2.1mem\_area\_t的分裂

在内存分配的过程中，有可能会造成mem\_area\_t的分裂，还是以上面的例子来说，加入我们要分配一个200字节的内存，这时候伙伴分配器的分配流程是这样的：

1.找到一个离200+sizeof(mem\_area\_t)最近的2的i次方的数（256），确定i = 8，

2.在 free\_list[i]的列表中查找是否有空闲的node,如果有，将node职位no free.如果没有，对i + 1层执行查找是否有可用的内存，

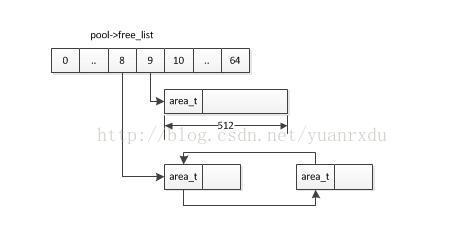
3.在上面的例子中，i+1=9,free\_list是空的，继续在i+2层找，一次类推，直到找到有node的层，也就是i = 10;

4.首先对10层进行分裂，分裂成两512大小的第9层节点，并从10删除area，在第9层加入2个512的node.

5.然后在对第9层的第一个节点进行分裂，分裂两个大小为256的第8层节点，并从第九层删除，在第8层加入2个节点。

6.将第一个256大小的area分配给对应的操作者，并置为no free标识。

以下是分配了一个200字节的内存池结构:



如果分配出去后的area\_t会从free\_list[i]链表中删除，也就是说在buddy上将是记录的。

2.2mem\_area\_t的合并

如果200字节分配出去后，使用完毕会归还给buddy allocator,还是拿上面的例子来说，就会发生area合并的情况,步骤如下：

1.使用者归还伙伴分配的内存，首先会根据area\_t的信息去找到自己的buddy,也就是第8层另外一个没有被分配的area.

2.找到buddy area后，判断buddy area是否是释放状态，如果是，触发合并，将自己和buddy area从第8层删除，合并成一个512大小的第9层area,

3.在重复1 ~ 2步，又会将自己和第九层另外一个buddy area合并成一个1024大小的第10层area.

2.3buddy allocator的接口函数:

mem\_pool\_create                构建一个buddy allocator

mem\_area\_alloc                   用buddy allocator分配一块内存

mem\_area\_free                    将一块内存归还给buddy allocator

mem\_pool\_get\_reserved      获得buddy allocator已经使用的内存大小

3内存分配堆(memory heap)

innodb中的内存管理最终的体现形式是mem\_heap\_t内存分配与管理，所有关于内存分配的操作都会调用mem\_heap的API方法，mem\_heap\_t的结构定义如下：

**struct** mem\_block\_info\_struct

{

ulint magic\_n; /\*魔法字\*/

**char** file\_name[8]; /\*分配内存的文件\*/

ulint line; /\*分配内存的文件所在行\*/

ulint len; /\*block的长度\*/

ulint type; /\*依赖的底层分配类型，有DYNAMIC、BUFFER、BTR\_SEARCH三种类型\*/

ibool init\_block; /\*是否是外部分配的内存块\*/

ulint free; /\*被占用的空间大小\*/

ulint start; /\*可分配内存的起始位置\*/

byte\* free\_block; /\*备用block,仅仅在BTR\_SEARCH方式可用\*/

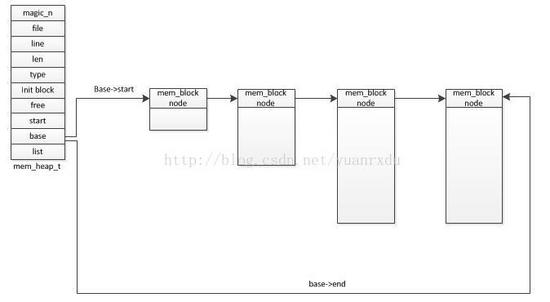
UT\_LIST\_BASE\_NODE\_T(**mem\_block\_t**) base;

UT\_LIST\_NODE\_T(**mem\_block\_t**) list;

};

**备注 ： mem\_block\_info\_struct/mem\_block\_info\_t/mem\_block\_t/mem\_heap\_t是等价**

mem\_heap\_t的内存结构如下：



关于mem\_heap\_t的几个要点：

1.一个mem\_block\_t最小空间不小于64字节，标准的大小是8KB,在非MEM\_HEAP\_BUFFER模式下分配的空间不大于page size - 200（page size一般为16KB）

2.mem\_heap\_t有三种类型，分别是DYNAMIC、BUFFER、BTR\_SEARCH，在DYNAMIC模式下都是基于buddy allocator进行mem\_block\_t分配的，在BTR\_SEARCH模式下，使用free\_block来作为内存分配，在BUFFER模式下比较复杂，如果分配的内存大小< page size的一半时，使用buddy alloc,否则使用buf\_frame的内存分配方式（这个是属于buf0buf.XX里面的方式，还未开始分析）。

3.mem\_heap\_t在分配新的mem\_block\_t的时候一定是分配一个heap最后节点大小的两倍，如果分配的大小超过MEM\_MAX\_ALLOC\_IN\_BUF（相当于一个page size）的时候，heap 类型判断，在不是DYNAMIC模式下，最大就是一个MEM\_MAX\_ALLOC\_IN\_BUF大小。如果其他模式下就是设置成MEM\_BLOCK\_STANDARD\_SIZE标准大小，在这些限制外，如果需要分配的内存大于这些限制，以分配内存大小为准进行mem\_block\_t分配。分配好的mem\_block\_t总是加入到heap base list的最后，也就是heap堆栈的顶端。

4.mem\_heap\_t在释放mem\_block\_t时候总是从顶端开始释放，直到不能释放为止（mem\_block\_t没有被使用者归还）。在mem\_block\_t释放的时候也是需要参考DYNAMIC、BUFFER、BTR\_SEARCH类型进行相对于的归还规则（和2要点是相对应的）。

**mem\_heap\_t函数方法说明:**

mem\_heap\_create                                        用DYNAMIC模式创建一个mem\_heap\_t

mem\_heap\_create\_in\_buffer                        用BUFFER模式创建一个mem\_heap\_t

mem\_heap\_create\_in\_btr\_search                 用BTR\_SEARCH模式创建一个mem\_heap\_t

mem\_heap\_free                                            释放mem\_heap\_t对象

mem\_alloc                                                    创建在MEM\_HEAP\_DYNAMIC模式下，并分配一块指定大小的内存（在这种方式下mem\_heap\_t只会有一个mem\_block\_t）

mem\_free                                                      归还mem\_heap\_t分配的内存，并释放mem\_heap\_t

mem\_heap\_alloc                                           在指定的mem\_heap\_t上分配一块内存

mem\_heap\_get\_heap\_top                            获得heap顶端块可使用内存的地址

mem\_heap\_empty                                        清空指定的mem\_heap\_t

mem\_heap\_get\_top                                     获得heap顶部的指定n大小的mem\_block\_t指针

mem\_heap\_free\_top                                    释放heap顶部N大小的mem\_block\_t块

4总结

innodb提供内存池和heap分配方式来统一管理内存，最主要的目的是提高内存的率。在MySQL-5.6的版本中，innodb提供两种选择，一种是使用innodb提供的内存池管理内存，还有一种是提供系统的malloc和free来作为内存管理。MySQL默认的是系统管理内存方式，一些有经验的DBA会使用系统的管理内存方式+TMalloc来做内存优化，借助TMalloc高效的内存管理方式实现MySQL的性能提升。

# 2@Mysql - chenlvzhou的专栏

- 博客频道 - CSDN.NET

http://blog.csdn.net/chenlvzhou/article/category/2648393

# 2@MySQL InnoDB体系架构之内存 - 推酷

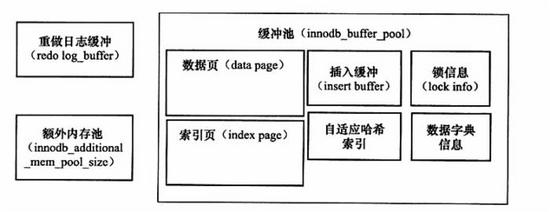
http://www.tuicool.com/articles/Mz6fEnI

MySQL InnoDB体系架构之内存

- chenlvzhou的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/chenlvzhou/article/details/41545839?utm\_source=tuicool&utm\_medium=referral

先上图



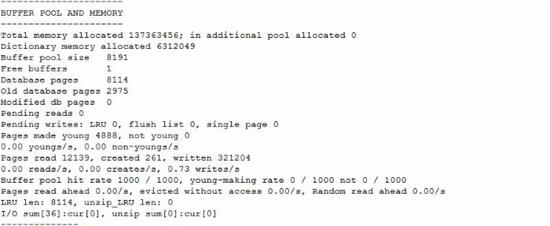
## 1.缓冲池

InnoDB存储引擎是基于磁盘存储的，并将其中的记录按照页的方式进行管理。通常使用缓冲池来提高数据库的整体性能。缓冲池简单说就是一块内存，通过内存的速度弥补磁盘速度较慢对数据库性能的影响。在数据库中进行读操作时，首先将从磁盘读到的页存放在缓冲池中，下一次读取相同的页时，首先判定是否存在缓冲池中，如果有就是被命中直接读取，没有的话就从磁盘中读取。在数据库进行改操作时，首先缓冲池中的页，然后在以一定的频率刷新到磁盘上。这里的刷新机制不是每页在发生变更时触发。而是通过一种checkpoint机制刷新到磁盘的。可以通过innodb\_buffer\_pool\_size(单位块)参数来设置缓冲池的大小。缓冲池中的数据库类型有：索引页、数据页、undo页、插入缓存页(insert buffer)、自适应hash(adaptive hash index)、innodb存储的锁信息(lock info)、数据字典信息(data dictionary)。通过参数innodb\_buffer\_pool\_instances设置允许有多个缓冲池实例，每个页根据哈希值平均分配到不同的实例中，以减少数据库内部资源的竞争，增加数据库的并发处理能力。

## 2.LRU List、Free List和Flush List

innoDB存储引擎的缓冲池是通过LRU算法来进行管理的，即最频繁使用的页在LRU列表(管理已经读取到的页)的前段，最少使用的页在LRU列表的尾端。缓冲池中每页大小16K。但是它对LRU算法做了一些优化，加入了midpoint位置。新获取的数据放入midpoint，而不是放入列表头部。通常midpoint在LRU列表的5/8处。可以通过innodb\_old\_blocks\_pct参数来设置。把midpoint之后的列表称为old列表，之前的称为new列表。为什么在innoDB存储引擎采用midpoint呢？这是因为如直接将读取到的也放入LRU的首端，某些SQL操作导致缓冲池中的页被刷出，从而影响缓冲池的效率。可以通过参数innodb\_old\_blocks\_time设置页读取到midpoint位置后需要等待多久才会加入LRU的new列表。

当数据库刚启动时，LRU列表是空的，这时所有的页都放在Free列表中的。当需要从缓冲池中分配页时，首先从Free列表中查看是否有空闲的空闲页，若有则从Free列表中删除然后加入到LRU列表中。否则根据LRU算法，淘汰LRU列表末尾的页，将该内存空间分配给新的页。当页从LRU的old部分加入到new部分，此时发生的操作是page made young，而因为innodb\_old\_blocks\_time参数的设置导致页没有从old加入到new的操作称为page not made young。从InnoDB存储引擎1.0.X版本开始支持压缩页的功能。即将原来16K的页压缩为1K、2K、4K和8K,由于页的大小发生变化，LRU列表页有相应的变化，对于非16K的页时通过unzip\_LRU列表进行维护。在LRU列表中的页被修改后，该页称为脏页(dirry page)，这时数据库通过checkpoint机制将脏页刷回磁盘。而Flush列表中的页就是脏页列表。脏页即存在于LRU列表页存在于Flush列表。LUR列表用来管理缓冲池中页的可用性，Flush列表用来管理将也刷回磁盘。可以通过show engine innodb status命令来查看LRU列表和Free列表的使用情况和运行状态。



由图可以看出:

Buffer pool size总共有8191个页(8191\*16k);

Free buffers表示当前Free列表中的页的数量;

Datebase pages表示LRU列表中页的数量。可能Free buffers和Database pages之和不等于Buffer pool size。这是因为缓冲池中的也还可能会被分配给自适应哈希索引、lock信息、insert buffer等页。而这部分是不需要LRU算法维护的，因此不在LRU列表中。

page made young表示LRU列表中页移动到前段的次数。

Buffer pool hit rate表示缓冲池的命中率，该值不能小于95%。

LRU len表示LRU列表中有多少页。

unzip\_LRU len表示压缩的页数；其中LRU len包含unzip\_LRU len。Modified db pages表示当前脏页的数量。

还可以通过  information\_schema.INNODB\_BUFFER\_PAGE\_LRU表来观察LRU列表中每个页的具体信息。

## 3.重做日志缓冲(redo log buffer)

innoDB存储引擎首先将重做日志信息放入这个缓冲区，然后按照一定的频率将其刷入重做日志文件中。重做日志缓冲区一般不需要很多，只要保证每秒产生的事务量在这个缓冲大小之内即可。可以通过innodb\_log\_buffer\_size参数设置大小。

## 4.额外的内存池

在innoDB存储引擎中，对内存的管理是通过一种称为内存堆(heap)的方式进行。在对一些数据结构本身的内存进行分配时，需要从额外的内存池中进行申请，当该区域的内存不够时，需要从缓冲池中申请。例如L:分配了缓冲池，但是每个缓冲池中的帧缓冲(frame buffer)还有对应的缓冲控制对象(buffer control block)，这些记录了一些诸如LRU、锁、等待等信息，而这个对象的内存就需要从额外内存池中申请。因此在申请了很大的缓冲池是也要考虑相应增加这个值。

# show engine innodb status 详解

\_没钱的烦恼\_新浪博客

http://blog.sina.com.cn/s/blog\_73a36be80101cse2.html

这个比较详细！不错

很多人让我来阐述一下 **SHOW INNODB STATUS**的输出信息, 了解 SHOW INNODB STATUS 都输出了些什么信息，并且我们能从这些信息中获取什么资讯，得以提高 MySQL 性能。

首先，让我们来了解一下 SHOW INNODB STATUS 输出的基础，它打印了很多关于 InnoDB 内部性能相关的计数器、统计、事务处理信息等。在 MySQL 5 中，InnoDB 的性能统计结果也在 **SHOW STATUS** 结果中显示了。大部分和 SHOW INNODB STATUS 的其他信息相同，在旧版本中还没有这个功能。

SHOW INNODB STATUS 中的很多统计值都是每秒更新一次的，如果你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。InnoDB 首先输出以下信息：

1.=====================================  
2.060717 3:07:56 INNODB MONITOR OUTPUT  
3.=====================================  
4.Per second averages calculated from the last 44 seconds

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。  
说实在的我不喜欢InnoDB提供的平均值，因为很难取得合理的平均间隔统计值，如果你是写脚本来取得 SHOW INNODB STATUS 结果的话，那么最好取得全局的统计结果，然后取得平均值。当然了，直接查看输出的结果信息也是很有用的。

下一部分显示了信号(Semaphores)相关信息：

1.----------  
2.SEMAPHORES  
3.----------  
4.OS WAIT ARRAY INFO: reservation count 13569, signal count 11421  
5.--Thread 1152170336 has waited at ./../include/buf0buf.ic line 630 for 0.00 seconds the semaphore:  
6.Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0  
7.waiters flag 0  
8.wait is ending  
9.--Thread 1147709792 has waited at ./../include/buf0buf.ic line 630 for 0.00 seconds the semaphore:  
10.Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0  
11.waiters flag 0  
12.wait is ending  
13.Mutex spin waits 5672442, rounds 3899888, OS waits 4719  
14.RW-shared spins 5920, OS waits 2918; RW-excl spins 3463, OS waits 3163

这段可以分成2个部分。一部分是当前的等待，这部分只是包含了在高并发环境下的全部记录，因此 InnoDB 会频繁回退到系统等待。如果等待是通过自旋锁来解决的话，那么这些信息就就不会显示了。

通过这部分信息，你就会知道系统负载的热点在哪了。不过这需要了解一下源码相关的知识 - 从上面的信息中就可以看出来是哪个源码文件中的哪行(不同的版本结果可能不同)，只是从这里却看不出来任何信息。尽管如此，还是可以从文件名中猜到一些东 西 - 比如本例中，文件名 "buf0buf.ic" 预示着和一些缓冲池争夺有关系。如果想了解更多，就去看源码吧。

还有一些关于等待的更多细节。"lock var" 表示当前的 mutex 对象的值(被锁住 = 1 / 释放 = 0) 值，"waiters flag" 表示当前的等待个数。另外，本例中还可以看到等待状态信息 "wait is ending"，这表示 mutex 已经释放，但是系统调度线程还正在处理。

第二块是事件统计 - "reservation count" 和 "signal count" 显示了 innodb 使用内部同步阵列的活跃程度 - 时间片(slot)分配以及线程信号使用同步阵列的频繁程度。这些统计信息可以用于表示 innodb 回退到系统等待的频率。还有关于系统等待的直接相关信息，可以看到"OS Waits"的互斥信号灯(mutexes)，以及读写锁。这些信息中显示了互斥锁和共享锁。系统等待和 "保留(reservation)" 不完全一样，在回退到用 sync\_array 的复杂等待模式前，innodb 会尝试 "输出(yield)" 到系统，希望下一次调度时间对象里命名线程已经释放了。系统等待相对较慢，如果每秒发生了上万次系统等待，则可能会有问题。另一个观察方法是查看系统状态 中的上下文(context)交换频率。

另一块重要的信息是 "spin waits" 和 "spin rounds" 的数量。相较于系统等待，自旋锁是低成本的等待；不过它是一个活跃的等待，会浪费一些cpu资源。因此如果看到大量的自旋等待和自旋轮转，则很显然它浪费 了很多cpu资源。浪费cpu时间和无谓的上下文切换之间可以用 innodb\_sync\_spin\_loops 来平衡。

接下来的这段显示死锁状况:

1.------------------------  
2.LATEST DETECTED DEADLOCK  
3.------------------------  
4.060717 4:16:48  
5.\*\*\* (1) TRANSACTION:  
6.TRANSACTION 0 42313619, ACTIVE 49 sec, process no 10099, OS thread id 3771312 starting index read  
7.mysql tables in use 1, locked 1  
8.LOCK WAIT 3 lock struct(s), heap size 320  
9.MySQL thread id 30898, query id 100626 localhost root Updating  
10.update iz set pad='a' where i=2  
11.\*\*\* (1) WAITING FOR THIS LOCK TO BE GRANTED:  
12.RECORD LOCKS space id 0 page no 16403 n bits 72 index `PRIMARY` of table `test/iz` trx id 0 42313619 lock\_mode X locks rec but not gap waiting  
13.Record lock, heap no 5 PHYSICAL RECORD: n\_fields 4; compact format; info bits 0  
14. 0: len 4; hex 80000002; asc ;; 1: len 6; hex 00000285a78f; asc ;; 2: len 7; hex 00000040150110; asc @ ;; 3: len 10; hex 61202020202020202020; asc a ;;  
15.   
16.\*\*\* (2) TRANSACTION:  
17.TRANSACTION 0 42313620, ACTIVE 24 sec, process no 10099, OS thread id 4078512 starting index read, thread declared inside InnoDB 500  
18.mysql tables in use 1, locked 1  
19.3 lock struct(s), heap size 320  
20.MySQL thread id 30899, query id 100627 localhost root Updating  
21.update iz set pad='a' where i=1  
22.\*\*\* (2) HOLDS THE LOCK(S):  
23.RECORD LOCKS space id 0 page no 16403 n bits 72 index `PRIMARY` of table `test/iz` trx id 0 42313620 lock\_mode X locks rec but not gap  
24.Record lock, heap no 5 PHYSICAL RECORD: n\_fields 4; compact format; info bits 0  
25. 0: len 4; hex 80000002; asc ;; 1: len 6; hex 00000285a78f; asc ;; 2: len 7; hex 00000040150110; asc @ ;; 3: len 10; hex 61202020202020202020; asc a ;;  
26.   
27.\*\*\* (2) WAITING FOR THIS LOCK TO BE GRANTED:  
28.RECORD LOCKS space id 0 page no 16403 n bits 72 index `PRIMARY` of table `test/iz` trx id 0 42313620 lock\_mode X locks rec but not gap waiting  
29.Record lock, heap no 4 PHYSICAL RECORD: n\_fields 4; compact format; info bits 0  
30. 0: len 4; hex 80000001; asc ;; 1: len 6; hex 00000285a78e; asc ;; 2: len 7; hex 000000003411d9; asc 4 ;; 3: len 10; hex 61202020202020202020; asc a ;;  
31.   
32.\*\*\* WE ROLL BACK TRANSACTION (2)

这里显示了 Innodb 最后检测到事务引发的死锁，包括发生死锁时的状态，加了什么锁，在等待什么锁释放，以及 Innodb 决定哪个事务会被回滚。注意，innodb只显示了事务持有锁的相关简单信息。并且只显示了每个事务最后执行的语句，发生死锁的记录就是由于这些语句引起 的。查看复杂的死锁信息还需要查看日志文件，才能找到真正引发冲突的语句。大部分情况下，SHOW INNODB STATUS 显示的信息基本足够了。

下面是关于外键约束引发的死锁信息：

1.------------------------  
2.LATEST FOREIGN KEY ERROR  
3.------------------------  
4.060717 4:29:00 Transaction:  
5.TRANSACTION 0 336342767, ACTIVE 0 sec, process no 3946, OS thread id 1151088992 inserting, thread declared inside InnoDB 500  
6.mysql tables in use 1, locked 1  
7.3 lock struct(s), heap size 368, undo log entries 1  
8.MySQL thread id 9697561, query id 188161264 localhost root update  
9.insert into child values(2,2)  
10.Foreign key constraint fails for table `test/child`:  
11.,  
12. CONSTRAINT `child\_ibfk\_1` FOREIGN KEY (`parent\_id`) REFERENCES `parent` (`id`) ON DELETE CASCADE  
13.Trying to add in child table, in index `par\_ind` tuple:  
14.DATA TUPLE: 2 fields;  
15. 0: len 4; hex 80000002; asc ;; 1: len 6; hex 000000000401; asc ;;  
16.   
17.But in parent table `test/parent`, in index `PRIMARY`,  
18.the closest match we can find is record:  
19.PHYSICAL RECORD: n\_fields 3; 1-byte offs TRUE; info bits 0  
20. 0: len 4; hex 80000001; asc ;; 1: len 6; hex 0000140c2d8f; asc - ;; 2: len 7; hex 80009c40050084; asc @ ;;

Innodb会显示引发错误的语句。外键约束定义失败，以及定义关系最密切的父表。有很多嵌接信息都是用16进制表示，不过对于问题诊断并不是太重 要，它们主要用于给 Innodb 的开发者来查看或者用于调试目的。

接下来是显示 Innodb 当前活跃的事务：

1.------------  
2.TRANSACTIONS  
3.------------  
4.Trx id counter 0 80157601  
5.Purge done for trx's n:o <0 80154573 undo n:o <0 0  
6.History list length 6  
7.Total number of lock structs in row lock hash table 0  
8.LIST OF TRANSACTIONS FOR EACH SESSION:  
9.---TRANSACTION 0 0, not started, process no 3396, OS thread id 1152440672  
10.MySQL thread id 8080, query id 728900 localhost root  
11.show innodb status  
12.---TRANSACTION 0 80157600, ACTIVE 4 sec, process no 3396, OS thread id 1148250464, thread declared inside InnoDB 442  
13.mysql tables in use 1, locked 0  
14.MySQL thread id 8079, query id 728899 localhost root Sending data  
15.select sql\_calc\_found\_rows \* from b limit 5  
16.Trx read view will not see trx with id>= 0 80157601, sees <0 80157597  
17.---TRANSACTION 0 80157599, ACTIVE 5 sec, process no 3396, OS thread id 1150142816 fetching rows, thread declared inside InnoDB 166  
18.mysql tables in use 1, locked 0  
19.MySQL thread id 8078, query id 728898 localhost root Sending data  
20.select sql\_calc\_found\_rows \* from b limit 5  
21.Trx read view will not see trx with id>= 0 80157600, sees <0 80157596  
22.---TRANSACTION 0 80157598, ACTIVE 7 sec, process no 3396, OS thread id 1147980128 fetching rows, thread declared inside InnoDB 114  
23.mysql tables in use 1, locked 0  
24.MySQL thread id 8077, query id 728897 localhost root Sending data  
25.select sql\_calc\_found\_rows \* from b limit 5  
26.Trx read view will not see trx with id>= 0 80157599, sees <0 80157595  
27.---TRANSACTION 0 80157597, ACTIVE 7 sec, process no 3396, OS thread id 1152305504 fetching rows, thread declared inside InnoDB 400  
28.mysql tables in use 1, locked 0  
29.MySQL thread id 8076, query id 728896 localhost root Sending data  
30.select sql\_calc\_found\_rows \* from b limit 5  
31.Trx read view will not see trx with id>= 0 80157598, sees <0 80157594

如果当前连接不是很多，则会显示全部事务列表；如果有大量连接，则 Innodb 只会显示他们的数量，减少输出的列表信息，使得输出结果不会太多。

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。

History list length 6 是指在回滚空间中的未清除事务数。随着事务的提交，它的值会增加；随着清除线程的运行，它的值会减小。

Total number of lock structs in row lock hash table 是指事务分配过的行锁结构总数。它和曾经被锁住过的行总数不一定相等，通常是一个锁结构对应多行记录。

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。"Thread declared inside InnoDB 400" 的意思是 Innodb 内核正在运行该线程，并且还需要400个票。Innodb 会根据 innodb\_thread\_concurrency 的值来限制同时并发的线程数不超过它。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。

mysql tables in use 1, locked 0 是指事务中已经用过的数据表个数(已经访问过了的)，以及被锁的个数。Innodb 一般情况不会锁表，因此锁表数一般是0，除非是 ALTER TABLE 或者其他类似 LOCK TABLES 的语句。

除了Innodb相关的特定信息外，一些基本信息可以通过 来查看，例如正在执行什么语句，查询ID号，查询状态等。

下面这部分显示的是跟IO相关的具体信息：

1.--------  
2.FILE I/O  
3.--------  
4.I/O thread 0 state: waiting for i/o request (insert buffer thread)  
5.I/O thread 1 state: waiting for i/o request (log thread)  
6.I/O thread 2 state: waiting for i/o request (read thread)  
7.I/O thread 3 state: waiting for i/o request (write thread)  
8.Pending normal aio reads: 0, aio writes: 0,  
9. ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0  
10.Pending flushes (fsync) log: 0; buffer pool: 0  
11.17909940 OS file reads, 22088963 OS file writes, 1743764 OS fsyncs  
12.0.20 reads/s, 16384 avg bytes/read, 5.00 writes/s, 0.80 fsyncs/s

本部分显示了IO助手线程状态 - 插入缓冲线程，日志线程，读、写线程。它们分别对应插入缓冲合并，异步日志刷新，预读以及刷新脏数据。源自查询的正常读取是由正在运行的查询执行的。在 Unix/Linux平台下，总能看见4个线程，在Windows上可以通过 innodb\_file\_io\_threads 来调整。每个线程准备好之后都能看到其状态：waiting for i/o request 或者正在执行特定的操作。

每个线程都会显示正在进行的操作数量 - 同时正要执行或者正在执行的操作数量。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IO负载较高状态。注意，由线程执行请求引发的IO请求是不计算在内的，因此尽管系统的 IO负载较高，但是它们的值却可能为 0。

接下来显示的是IO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IO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。

1.-------------------------------------  
2.INSERT BUFFER AND ADAPTIVE HASH INDEX  
3.-------------------------------------  
4.Ibuf for space 0: size 1, free list len 887, seg size 889, is not empty  
5.Ibuf for space 0: size 1, free list len 887, seg size 889,  
6.2431891 inserts, 2672643 merged recs, 1059730 merges  
7.Hash table size 8850487, used cells 2381348, node heap has 4091 buffer(s)  
8.2208.17 hash searches/s, 175.05 non-hash searches/s

本部分显示了插入缓冲以及自适应哈希索引的状态。第一行显示了插入缓冲的状态 - 段的大小以及空闲列表，以及缓冲中有多少记录。接下来显示了缓冲中已经完成了多少次插入，有多少记录已经合并，有多少次合并已经完成。合并次数除以插入次 数得到的比率可以反映出插入缓冲的效率如何。

Innodb采用哈希索引建立内存页索引形成自适应哈希索引而不是采 B-tree 索引，得以加速行记录到内存页的检索。这里显示了哈希表的大小，以及自适应哈希索引使用了多少单元和缓冲。可以通过计算利用哈希索引检索的次数以及没利用 它检索的次数来了解哈希索引的效率。

当前对自适应哈希索引基本没有什么办法可以调整它，主要还是用于查看。

1.---  
2.LOG  
3.---  
4.Log sequence number 84 3000620880  
5.Log flushed up to 84 3000611265  
6.Last checkpoint at 84 2939889199  
7.0 pending log writes, 0 pending chkp writes  
8.14073669 log i/o's done, 10.90 log i/o's/second

接下来显示的是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IO请求和日志IO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。

1.----------------------  
2.BUFFER POOL AND MEMORY  
3.----------------------  
4.Total memory allocated 4648979546; in additional pool allocated 16773888  
5.Buffer pool size 262144  
6.Free buffers 0  
7.Database pages 258053  
8.Modified db pages 37491  
9.Pending reads 0  
10.Pending writes: LRU 0, flush list 0, single page 0  
11.Pages read 57973114, created 251137, written 10761167  
12.9.79 reads/s, 0.31 creates/s, 6.00 writes/s  
13.Buffer pool hit rate 999 / 1000

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IO，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。

最后我们可以看到缓冲池的命中率，它预示着缓冲池的效率。1000/1000 相当于 100% 的命中率。不过这样也很难说明缓冲池的命中率就足够高了，这要需要根据不同的负载环境而定。通常情况下，950/1000 就够了，有些时候在IO负载较高的环境下，命中率可能为 995/1000。

1.--------------  
2.ROW OPERATIONS  
3.--------------  
4.0 queries inside InnoDB, 0 queries in queue  
5.1 read views open inside InnoDB  
6.Main thread process no. 10099, id 88021936, state: waiting for server activity  
7.Number of rows inserted 143, updated 3000041, deleted 0, read 24865563  
8.0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s

最后一部分，显示了数据行操作以及一些系统信息相关情况。

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。

接下来可以看到自从系统启动以来，所有的数据行操作数量及其平均值。它们可以很方便地用于监控以及画出系统状态图，数据行操作次数可以很好的衡量 Innodb的负载。不是所有的数据行操作带来的负载都是一样的，存取10字节的行比10Mb的行相比会小了很多，不过相对于查询的总次数来说这个信息可 是有用的多了，差别也很大。

还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。[http://www.dbaunion.com](http://www.dbaunion.com/)

# mysql 优化技巧心得一(key\_buffer\_size设置).

- rainysia的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/rainysia/article/details/8767946

1: 对于[**MySQL**](http://lib.csdn.net/base/mysql)来说,登录的时候,加一个参数

**[sql]** [view plain](http://blog.csdn.net/rainysia/article/details/8767946) [copy](http://blog.csdn.net/rainysia/article/details/8767946)

1. #mysql -u root -p --prompt="\\u@\\d \\r:\\m:\\s>"

--prompt 是加一些登录后的参数,上面参数 u 是加入登录的用户名, d 是登录后选择后(use dbname)的[**数据库**](http://lib.csdn.net/base/mysql)表, r m s 是时间

2: key\_buffer\_size 对MyISAM表性能影响很大.

**[sql]** [view plain](http://blog.csdn.net/rainysia/article/details/8767946) [copy](http://blog.csdn.net/rainysia/article/details/8767946)

1. mysql> show variables like 'key\_buffer\_size';
2. +-----------------+------------+
3. | Variable\_name | Value |
4. +-----------------+------------+
5. | key\_buffer\_size | 536870912 |
6. +-----------------+------------+

分配了512MB内存给mysql key\_buffer\_size，我们再看一下key\_buffer\_size的使用情况：

**[sql]** [view plain](http://blog.csdn.net/rainysia/article/details/8767946) [copy](http://blog.csdn.net/rainysia/article/details/8767946)

1. mysql> show **global** status like 'key\_read%';
2. +------------------------+-------------+
3. | Variable\_name | Value |
4. +------------------------+-------------+
5. | Key\_read\_requests | 27813678764 |
6. | Key\_reads | 6798830 |
7. +------------------------+-------------+

一共有27813678764个索引读取请求，有6798830个请求在内存中没有找到直接从硬盘读取索引，计算索引未命中缓存的概率：  
  
key\_cache\_miss\_rate = Key\_reads / Key\_read\_requests \* 100%  
  
比如上面的数据，key\_cache\_miss\_rate为0.0244%，4000个索引读取请求才有一个直接读硬盘，已经很BT了，key\_cache\_miss\_rate在0.1%以下都很好(每1000个请求有一个直接读硬盘)，如果key\_cache\_miss\_rate在0.01%以下的话，key\_buffer\_size分配的过多，可以适当减少。  
MySQL服务器还提供了key\_blocks\_\*参数：

**[sql]** [view plain](http://blog.csdn.net/rainysia/article/details/8767946) [copy](http://blog.csdn.net/rainysia/article/details/8767946)

1. mysql> show **global** status like 'key\_blocks\_u%';
2. +------------------------+-------------+
3. | Variable\_name | Value |
4. +------------------------+-------------+
5. | Key\_blocks\_unused | 0 |
6. | Key\_blocks\_used | 413543 |
7. +------------------------+-------------+

Key\_blocks\_unused表示未使用的缓存簇(blocks)数，Key\_blocks\_used表示曾经用到的最大的blocks数，比如这台服务器，所有的缓存都用到了，要么增加key\_buffer\_size，要么就是过渡索引了，把缓存占满了。比较理想的设置：

Key\_blocks\_used / (Key\_blocks\_unused + Key\_blocks\_used) \* 100% ≈ 80%