[innodb的redo log以及与binary log的区别](http://blog.itpub.net/30038563/viewspace-1426310/)

**innoDB重做日志**

记录数据库页的变化，诸如space id/page no/operation code/data，大致原理等同于oracle redo log，可用于crash recovery，但是其功能有一定局限。

执行crash recovery时所操作的页必须是consistent，如果页已经损坏则无法recover，幸好及时引入了double write功能

日志头存储着最近一次checkpoint信息，日志以记录而非页的方式写入，采用512字节对齐(与磁盘扇区同等)，SSD可以设置4096字节；

Redo log不能归档，因此必须循环利用，其相应设置参数为innodb\_log\_file\_size/innodb\_log\_files\_in\_group/innodb\_log\_group\_home\_dir

选择合适的文件大小对innodb写性能至关重要，通常一个文件至少能够容纳一个小时的日志量

Mysql采用fuzzy checkpoint机制，每次读取最老的脏页而非全部脏页，将其LSN作为检查点记录到日志文件(这之前的LSN对应的日志和数据都写入文件，不需要恢复)；

恢复数据文件时，innodb扫描日志文件的检查点信息，只恢复page LSN> checkpoint LSN的内容

**redo log刷新条件**

每秒刷新一次，由master thread执行；

事务commit，由参数innodb\_flush\_log\_at\_trx\_commit决定，0表示不刷新，1同步刷新，2异步刷新至磁盘

**相关参数**

Innodb\_log\_buffer\_size：缓存redo log以待刷新至磁盘，默认8M；经常发生大事务的时候可适当调大；

Innodb\_log\_file\_size：日志文件大小，默认5M，上限为4G(InnoDB最初只为32位系统设计)；越大则检查点发生次数就越少，但crash recovery时间就越长；

Innodb\_log\_files\_in\_group：日志文件数量，默认为2，上限100；

Innodb\_flush\_log\_at\_trx\_commit：

同oracle一样，commit返回前提交所有的redo log，不同的是，innodb提供innodb\_flush\_log\_at\_trx\_commit参数定制化commit返回时日志是否刷新到磁盘，

innodb\_flush\_log\_at\_trx\_commit = 0 /\* Write log buffer to log about once a second and flush \*/

innodb\_flush\_log\_at\_trx\_commit = 1 /\* Write log buffer to log and flush to disk \*/

innodb\_flush\_log\_at\_trx\_commit = 2 /\* Write log buffer to log and flush about once per second \*/

设置为1 时数据库crash仍能保证数据一致性，而操作系统crash则可能导致部分redo丢失；

可使用有备用电池的存储与之互补，即提升速度又能保证数据一致性

**二进制日志**

头部有4字节偏移量；

二进制日志以格式描述事件开始，日志轮换时间结束；

格式描述事件包括该文件的版本号/mysql版本以及日志信息， 以及标志binlog是否正常关闭；日志轮换包含下一个binlog名称；

根据复制方式不同，二进制记录的内容也不一样：

statement记录sql本身；

row则以二进制形式呈现，每1行可能包含前镜像或后镜像，包含3中类型write\_rows\_log\_event/update\_rows\_log\_event/delete\_rows\_log\_event；

**何时刷新**

Sync\_binlog：默认为0，如果设置为N，则在N次binlog写后调用fdatasync()将其刷新至磁盘；使用autocommit时1个sql为1次写，否则1个事务为1次写；

N=1的安全度最高，在mysql crash时最多丢失一个事务，但也最慢；

**相关参数**

Max\_binlog\_size：规定binlog的最大尺寸，范围为4k – 1G

binlog\_cache\_size：每个客户端分配一个，当服务器支持事务引擎且激活bin log时使用，用于记录事务缓存以便写入binlog；

此参数有个演化史：

5.5.3 – 5.5.8：非事务语句也分配同等大小的binlog cache，实际分配的cache为此参数的2倍；

5.5.9引入binglog\_stmt\_cache\_size，专门分配非事务语句；

与之对应的还有max\_binlog\_cache\_size，假定某事务大小超过此设置，会抱错generates a Multi-statement transaction required more than 'max\_binlog\_cache\_size' bytes of storage

**与二进制日志区别**

1  二进制日志记录所有引擎的日志，而重做日志只记录InnoDB

2二进制日志记录事务的具体操作内容，而后者记录每个页更改的物理情况

3前者用于point-in-time恢复，后者用于crash recovery；如果mysql发生介质损坏，则需要从备份中恢复然后应用binary log执行point-in-time recovery

4 bin log大小达到上限或者flush log会生成一个新的文件，而redo log只能循环利用

5 bin log有索引文件，保存所有bin log文件列表

6 bin log可由每个线程写(加锁以避免混乱)，而redo log只能由master thread写

什么是double write buffer

分2部分：内存中的buffer(2M)；共享表空间中连续的128页(2M)；刷新脏页时先通过memcpy将其复制到doublewrite buffer，接着分2次将其刷新到共享表空间，然后调用fsync同步脏页到磁盘；

即先将脏数据顺序写入共享表空间，然后再调用fsync写入数据文件，如果此时OS崩溃，recovery时可以从共享表空间找到该页的副本，将其copy的数据文件然后应用redo log

由参数innodb\_doublewrite决定，默认为true

[MySQL InnoDB存储引擎undo redo解析](http://blog.csdn.net/hustyangju/article/details/40111407)

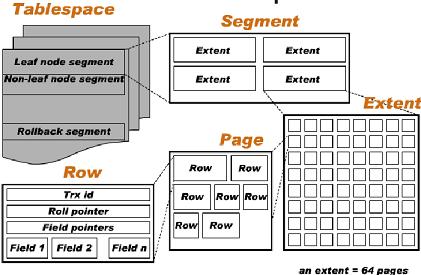
本文是介绍MySQL数据库InnoDB存储引擎重做日志漫游  
  
 **00 – Undo Log**  
Undo Log 是为了实现事务的原子性，在MySQL数据库InnoDB存储引擎中，还用Undo Log来实现多版本并发控制(简称：MVCC)。  
  
  
- 事务的原子性(Atomicity)  
  事务中的所有操作，要么全部完成，要么不做任何操作，不能只做部分操作。如果在执行的过程中发生  
  了错误，要回滚(Rollback)到事务开始前的状态，就像这个事务从来没有执行过。  
  
  
- 原理  
  Undo Log的原理很简单，为了满足事务的原子性，在操作任何数据之前，首先将数据备份到一个地方  
  （这个存储数据备份的地方称为Undo Log）。然后进行数据的修改。如果出现了错误或者用户执行了  
  ROLLBACK语句，系统可以利用Undo Log中的备份将数据恢复到事务开始之前的状态。  
  
  
除了可以保证事务的原子性，Undo Log也可以用来辅助完成事务的持久化。  
  
  
- 事务的持久性(Durability)  
  事务一旦完成，该事务对数据库所做的所有修改都会持久的保存到数据库中。为了保证持久性，数据库  
  系统会将修改后的数据完全的记录到持久的存储上。  
  
  
- 用Undo Log实现原子性和持久化的事务的简化过程  
  假设有A、B两个数据，值分别为1,2。  
  A.事务开始.  
  B.记录A=1到undo log.  
  C.修改A=3.  
  D.记录B=2到undo log.  
  E.修改B=4.  
  F.将undo log写到磁盘。  
  G.将数据写到磁盘。  
  H.事务提交  
  这里有一个隐含的前提条件：‘数据都是先读到内存中，然后修改内存中的数据，最后将数据写回磁盘’。  
  
  
  之所以能同时保证原子性和持久化，是因为以下特点：  
  A. 更新数据前记录Undo log。  
  B. 为了保证持久性，必须将数据在事务提交前写到磁盘。只要事务成功提交，数据必然已经持久化。  
  C. Undo log必须先于数据持久化到磁盘。如果在G,H之间系统崩溃，undo log是完整的，  
     可以用来回滚事务。  
  D. 如果在A-F之间系统崩溃,因为数据没有持久化到磁盘。所以磁盘上的数据还是保持在事务开始前的状态。  
  
  
缺陷：每个事务提交前将数据和Undo Log写入磁盘，这样会导致大量的磁盘IO，因此性能很低。  
  
  
如果能够将数据缓存一段时间，就能减少IO提高性能。但是这样就会丧失事务的持久性。因此引入了另外一  
种机制来实现持久化，即Redo Log.  
  
  
**01 – Redo Log**  
  
  
- 原理  
  和Undo Log相反，Redo Log记录的是新数据的备份。在事务提交前，只要将Redo Log持久化即可，  
  不需要将数据持久化。当系统崩溃时，虽然数据没有持久化，但是Redo Log已经持久化。系统可以根据  
  Redo Log的内容，将所有数据恢复到最新的状态。  
  
  
- Undo + Redo事务的简化过程  
  假设有A、B两个数据，值分别为1,2.  
  A.事务开始.  
  B.记录A=1到undo log.  
  C.修改A=3.  
  D.记录A=3到redo log.  
  E.记录B=2到undo log.  
  F.修改B=4.  
  G.记录B=4到redo log.  
  H.将redo log写入磁盘。  
  I.事务提交  
  
  
- Undo + Redo事务的特点  
  A. 为了保证持久性，必须在事务提交前将Redo Log持久化。  
  B. 数据不需要在事务提交前写入磁盘，而是缓存在内存中。  
  C. Redo Log 保证事务的持久性。  
  D. Undo Log 保证事务的原子性。  
  E. 有一个隐含的特点，数据必须要晚于redo log写入持久存储。  
  
  
- IO性能  
  Undo + Redo的设计主要考虑的是提升IO性能。虽说通过缓存数据，减少了写数据的IO.  
  但是却引入了新的IO，即写Redo Log的IO。如果Redo Log的IO性能不好，就不能起到提高性能的目的。  
  为了保证Redo Log能够有比较好的IO性能，InnoDB 的 Redo Log的设计有以下几个特点：  
  
  
  A. 尽量保持Redo Log存储在一段连续的空间上。因此在系统第一次启动时就会将日志文件的空间完全分配。  
     以顺序追加的方式记录Redo Log,通过顺序IO来改善性能。  
  B. 批量写入日志。日志并不是直接写入文件，而是先写入redo log buffer.当需要将日志刷新到磁盘时  
     (如事务提交),将许多日志一起写入磁盘.  
  C. 并发的事务共享Redo Log的存储空间，它们的Redo Log按语句的执行顺序，依次交替的记录在一起，  
     以减少日志占用的空间。例如,Redo Log中的记录内容可能是这样的：  
     记录1: <trx1, insert …>  
     记录2: <trx2, update …>  
     记录3: <trx1, delete …>  
     记录4: <trx3, update …>  
     记录5: <trx2, insert …>  
  D. 因为C的原因,当一个事务将Redo Log写入磁盘时，也会将其他未提交的事务的日志写入磁盘。  
  E. Redo Log上只进行顺序追加的操作，当一个事务需要回滚时，它的Redo Log记录也不会从  
     Redo Log中删除掉。  
  
  
**02 – 恢复(Recovery)**  
  
  
- 恢复策略  
  前面说到未提交的事务和回滚了的事务也会记录Redo Log，因此在进行恢复时,这些事务要进行特殊的  
  的处理.有2中不同的恢复策略：  
  
  
  A. 进行恢复时，只重做已经提交了的事务。  
  B. 进行恢复时，重做所有事务包括未提交的事务和回滚了的事务。然后通过Undo Log回滚那些  
     未提交的事务。  
  
  
- InnoDB存储引擎的恢复机制  
  MySQL数据库InnoDB存储引擎使用了B策略, InnoDB存储引擎中的恢复机制有几个特点：  
  
  
  A. 在重做Redo Log时，并不关心事务性。 恢复时，没有BEGIN，也没有COMMIT,ROLLBACK的行为。  
     也不关心每个日志是哪个事务的。尽管事务ID等事务相关的内容会记入Redo Log，这些内容只是被当作  
     要操作的数据的一部分。  
  B. 使用B策略就必须要将Undo Log持久化，而且必须要在写Redo Log之前将对应的Undo Log写入磁盘。  
     Undo和Redo Log的这种关联，使得持久化变得复杂起来。为了降低复杂度，InnoDB将Undo Log看作  
     数据，因此记录Undo Log的操作也会记录到redo log中。这样undo log就可以象数据一样缓存起来，  
     而不用在redo log之前写入磁盘了。  
     包含Undo Log操作的Redo Log，看起来是这样的：  
     记录1: <trx1, Undo log insert <undo\_insert …>>  
     记录2: <trx1, insert …>  
     记录3: <trx2, Undo log insert <undo\_update …>>  
     记录4: <trx2, update …>  
     记录5: <trx3, Undo log insert <undo\_delete …>>  
     记录6: <trx3, delete …>  
  C. 到这里，还有一个问题没有弄清楚。既然Redo没有事务性，那岂不是会重新执行被回滚了的事务？  
     确实是这样。同时Innodb也会将事务回滚时的操作也记录到redo log中。回滚操作本质上也是  
     对数据进行修改，因此回滚时对数据的操作也会记录到Redo Log中。  
     一个回滚了的事务的Redo Log，看起来是这样的：  
     记录1: <trx1, Undo log insert <undo\_insert …>>  
     记录2: <trx1, insert A…>  
     记录3: <trx1, Undo log insert <undo\_update …>>  
     记录4: <trx1, update B…>  
     记录5: <trx1, Undo log insert <undo\_delete …>>  
     记录6: <trx1, delete C…>  
     记录7: <trx1, insert C>  
     记录8: <trx1, update B to old value>  
     记录9: <trx1, delete A>  
     一个被回滚了的事务在恢复时的操作就是先redo再undo，因此不会破坏数据的一致性.  
  
  
- InnoDB存储引擎中相关的函数  
  Redo: recv\_recovery\_from\_checkpoint\_start()  
  Undo: recv\_recovery\_rollback\_active()  
  Undo Log的Redo Log: trx\_undof\_page\_add\_undo\_rec\_log()

1 二进制日志记录所有引擎的日志，而重做日志只记录InnoDB；  
  
2 二进制日志记录事务的具体操作内容，而后者记录每个页更改的物理情况；  
  
3 前者在事务提交前记录，后者则在事务进行中不断写入redo entry  
  
记录数据库页的变化，诸如space id/page no/operation code/data，大致原理等同于oracle redo log，可用于crash recovery，但是其功能有一定局限。  
执行crash recovery时所操作的页必须是consistent，如果页已经损坏则无法recover，幸好及时引入了double write功能  
  
日志头存储着最近一次checkpoint信息，日志以记录而非页的方式写入，每次512字节，SSD可以设置4096字节；  
  
Redo log不能归档，因此必须循环利用，其相应设置参数为innodb\_log\_file\_size/innodb\_log\_files\_in\_group/innodb\_log\_group\_home\_dir  
选择合适的文件大小对innodb写性能至关重要，通常一个文件至少能够容纳一个小时的日志量  
  
Mysql采用fuzzy checkpoint机制，每次读取最老的脏页，将其LSN作为检查点记录到日志文件(这之前的LSN对应的日志和数据都写入文件，不需要恢复)；  
恢复数据文件时，innodb扫描日志文件的检查点信息，只恢复page LSN> checkpoint LSN的内容  
  
redo log刷新条件：  
每秒刷新一次，由master thread执行；  
事务commit，由参数innodb\_flush\_log\_at\_trx\_commit决定，0表示不刷新，1同步刷新，2异步刷新至磁盘  
  
Innodb\_flush\_log\_at\_trx\_commit  
同oracle一样，commit返回前提交所有的redo log，不同的是，innodb提供innodb\_flush\_log\_at\_trx\_commit参数定制化commit返回时日志是否刷新到磁盘，  
innodb\_flush\_log\_at\_trx\_commit = 0 /\* Write log buffer to log about once a second and flush \*/  
innodb\_flush\_log\_at\_trx\_commit = 1 /\* Write log buffer to log and flush to disk \*/  
innodb\_flush\_log\_at\_trx\_commit = 2 /\* Write log buffer to log and flush about once per second \*/  
数据库crash仍能保证数据一致性，而操作系统crash则可能导致部分记录丢失  
可使用有备用电池的存储与之互补，即提升速度又能保证数据一致性  
  
与二进制日志区别：  
1 二进制日志记录所有引擎的日志，而重做日志只记录InnoDB；  
2 二进制日志记录事务的具体操作内容，而后者记录每个页更改的物理情况；  
3 前者用于point-in-time恢复，后者用于crash recovery；如果mysql发生介质损坏，则需要从备份中恢复然后应用binary log执行point-in-time recovery

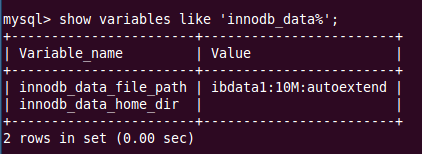
# [MySQL InnoDB的存储结构总结](http://www.cnblogs.com/benshan/archive/2013/01/08/2851714.html)

　　从物理意义上来讲，InnoDB表由共享表空间、日志文件组（redo文件组）、表结构定义文件组成。若将innodb\_file\_per\_table设置为on，则系统将为每一个表单独的生成一个table\_name.ibd的文件，在此文件中，存储与该表相关的数据、索引、表的内部数据字典信息。表结构文件则以.frm结尾，这与存储引擎无关。

　　以下为InnoDB的表空间结构图：



在InnoDB存储引擎中，默认表空间文件是ibdata1，初始化为10M，且可以扩展，如下图所示：



　　实际上，InnoDB的表空间文件是可以修改的，使用以下语句就可以修改：

Innodb\_data\_file\_path=ibdata1:370M；ibdata2:50M:autoextend

**使用共享表空间存储方式时，Innodb的所有数据保存在一个单独的表空间里面，而这个表空间可以由很多个文件组成，一个表可以跨多个文件存在，所以其大小限制不再是文件大小的限制，而是其自身的限制。从Innodb的官方文档中可以看到，其表空间的最大限制为64TB，也就是说，Innodb的单表限制基本上也在64TB左右了，当然这个大小是包括这个表的所有索引等其他相关数据。**

**而在使用单独表空间存储方式时，每个表的数据以一个单独的文件来存放，这个时候的单表限制，又变成文件系统的大小限制了。**

　　以下即为不同平台下，单独表空间文件最大限度。

Operating System  File-size Limit  
Win32 w/ FAT/FAT32  2GB/4GB  
Win32 w/ NTFS          2TB (possibly larger)  
Linux 2.4+          (using ext3 file system) 4TB  
Solaris 9/10          16TB  
MacOS X w/ HFS+         2TB  
NetWare w/NSS file system  8TB

※ 以下是MySQL文档中的内容：  
Windows用户请注意： FAT和VFAT (FAT32)不适合MySQL的生产使用。应使用NTFS。

共享表空间与独占表空间可以通过参数innodb\_file\_per\_table来转换，若为1，则开启独占表空间，否则，开启共享表存储。

在服务器资源有限,单表数据不是特别多的情况下, 独立表空间明显比共享方式效率更高 . 但是MySQL 默认是共享表空间 。

具体的共享表空间和独立表空间优缺点如下：

共享表空间：  
优点：  
可以放表空间分成多个文件存放到各个磁盘上（表空间文件大小不受表大小的限制，如一个表可以分布在不同步的文件上）。数据和文件放在一起方便管理。  
缺点：  
所有的数据和索引存放到一个文件中以为着将有一个很常大的文件，虽然可以把一个大文件分成多个小文件，但是多个表及索引在表空间中混合存储，这样对于一个表做了大量删除操作后表空间中将会有大量的空隙，特别是对于统计分析，日值系统这类应用最不适合用共享表空间。

独立表空间：在配置文件（my.cnf）中设置： innodb\_file\_per\_table

优点：  
1．  每个表都有自已独立的表空间。  
2．  每个表的数据和索引都会存在自已的表空间中。  
3．  可以实现单表在不同的数据库中移动。  
4．  空间可以回收（除drop table操作处，表空不能自已回收）  
a)         Drop table操作自动回收表空间，如果对于统计分析或是日值表，删除大量数据后可以通过:alter table TableName engine=innodb;回缩不用的空间。  
b)         对于使innodb-plugin的Innodb使用truncate table也会使空间收缩。  
c)         对于使用独立表空间的表，不管怎么删除，表空间的碎片不会太严重的影响性能，而且还有机会处理。  
缺点：  
单表增加过大，如超过100个G。

※ 对于启用了innodb\_file\_per\_table 的参数选项之后，在每个表对应的.idb文件内只是存放了数据、索引和插入缓冲，而撤销（undo）信息，系统事务信息，二次写缓冲等还是存放在了原来的共享表空间内。

※ 数据段即B+树的叶节点，索引段即为B+树的非索引节点。

※ InnoDB存储引擎的管理是由引擎本身完成的，表空间是由分散的页和段组成。

※ 区由64个连续的页组成，每个页大小为16K，即每个区大小为1MB，创建新表时，先有32页大小的碎片页存放数据，使用完后才是区的申请，（InnoDB最多每次申请4个区，保证数据的顺序性能）

※ 页类型有：数据页、Undo页、系统页、事务数据页、插入缓冲位图页、以及插入缓冲空闲列表页。

分类: [MySQL](http://www.cnblogs.com/benshan/category/374632.html" \t "http://www.cnblogs.com/benshan/archive/2013/01/08/_blank)

<http://mysql.taobao.org/monthly/2016/02/01/>

一．mysql体系结构和存储引擎

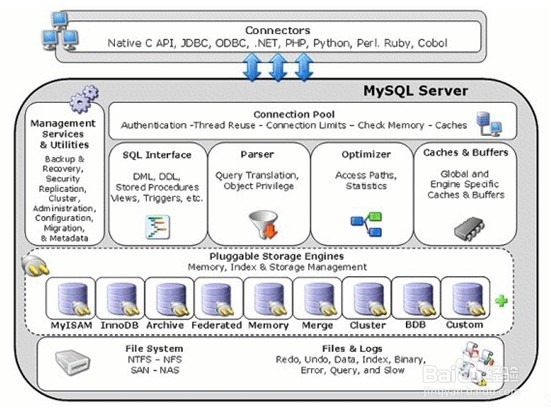
1.1、数据库和实例的区别

数据库：物理操作系统或其他形式文件类型的集合。在mysql下数据库文件可以是frm，myd，myi，ibd结尾的文件。

         数据库实例：由数据库后台进程/线程以及一个共享内存区组成。数据库实例才是真正用来操作数据库文件的。

         mysql数据库是单进程多线程的程序，与sql server比较类似。也就是说，Mysql数据库实例在系统上的表现就是一个进程。

1.2、mysql的体系结构



mysql由连接池组件、管理服务和工具组件、sql接口组建、查询分析器组件、优化器组件、缓存组件、插件是存储引擎、物理文件。

1.3、mysql存储引擎

      1.3.1、innodb存储引擎，特点支持外键、行锁、非锁定读(默认情况下读取不会产生锁)、mysql-4.1开始支持每个innodb引擎的表单独放到一个表空间里。innodb通过使用MVCC来获取高并发性，并且实现sql标准的4种隔离级别，同时使用一种被称成next-key locking的策略来避免换读(phantom)现象。除此之外innodb引擎还提供了插入缓存(insert buffer)、二次写(double write)、自适应哈西索引(adaptive hash index)、预读(read ahead)等高性能技术。

      1.3.2、myisam存储引擎，myisam特点是不支持事物，适合olap应用，myisam表由MYD和MYI组成。mysql-5.0版本之前，myisam默认支持的表大小为4G，从mysql-5.0以后，myisam默认支持256T的表单数据。myisam只缓存索引数据。

      1.3.3、NDB存储引擎，特点是数据放在内存中，mysql-5.1版本开始可以将非索引数据放到磁盘上。NDB之前的缺陷是join查询是mysql数据库层完成的，而不是存储引擎完成的，复杂的join查询需要巨大的网络开销，速度很慢。当前mysql cluster7.2版本中已经解决此问题，join查询效率提高了70倍。

      1.3.4、memeory存储引擎，将数据放到内存中，默认使用hash索引，不支持text和blob类型，varchara是按照char的方式来存储的。mysql数据库使用memory存储引擎作为临时表还存储中间结果集(intermediate result)，如果中间集结果大于memorg表的容量设置，又或者中间结果集包含text和blog列类型字段，则mysql会把他们转换到myisam存储引擎表而放到磁盘上，会对查询产生性能影响。

      1.3.5、archive存储引擎，压缩能力较强，主要用于归档存储。

      1.3.6、federated存储引擎，不存储数据，他指向一台远程mysql数据库上的表。

      1.3.7、maria存储引擎，myisam的后续版本,支持缓存数据和索引，行锁设计，支持mvcc，支持事务和非事务安全的选项，以及更好的BLOG字符类型的处理性能。

      1.3.8、其他存储引擎，sphinx用于全文索引，infobright用于数据仓库。

1.4连接Mysql

1.4.1、TCP/IP：基于网络的连接，连接进行权限检查。

         1.4.2、命名管道和共享内存：Windows系统上同一服务器上的两进程可通过命名管道连接，需在配置文件中启用--enable-named-pipe选项。

         1.4.3、Unix套接字：客户端与服务端位于同一服务器时才可使用，可以在my.cnf中指定-socket=/tmp/mysql.sock，连接时指定./mysql -S/tmp/mysql.sock。

二．InnoDB存储引擎

2.2、innodb引擎架构

       InnoDB的多个内存块组成了内存池，负责如下工作：

1).维护所有进程/线程需要访问的多个内部数据结构。

      2).缓存磁盘上的数据，方便快速的读取，并且在对磁盘文件的数据进行修改之前在这里缓存。

      3).重做日志缓存。

      后台线程的主要作用是负责刷新内存池中的数据，保证缓冲池中的内存缓存是最近的数据，此外、将已经修改的数据文件刷新到磁盘文件

      2.2.1、后台线程

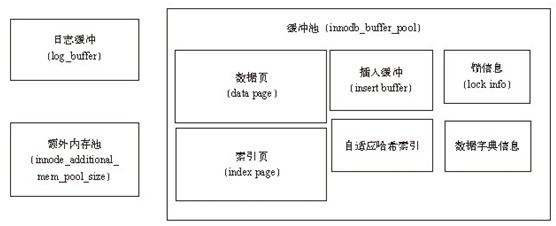
      innodb存储引擎后台有7个线程，—–4个IO线程(insert buffer thread,log thread,read thread,write thread)，1个master thread，一个lock监控线程，一个错误监控线程。

      2.2.2、内存

      innodb存储引擎内存由以下三个部分组成：缓冲池(buffer pool),重做日志缓存(redo log buffer),额外的内存池(additional memory pool)。可以使用 show engine innodb status来查看innodb\_buffer\_pool的使用情况。

      innodb\_buffer\_pool\_size：具体看，缓冲池中的数据库类型有：索引页、数据库页、undo页、插入缓存页(insert buffer)、自适应hash(adaptive hashindex)、innodb存储的锁信息(lock info)、数据字典信息(data dictionary)。

         InnoDB工作方式：将数据文件按页(每页16K)读入InnoDBbuffer pool，然后按最近最少使用算法(LRU)保留缓存数据，最后通过一定频率将脏页刷新到文件。



2.3、master thread

      2.3.1、master thread源码分析



      2.3.2、master thread的潜在问题

      1、由于硬件的发展，现在的硬件性能已经提高了很多，如果innodb每秒最大刷新100个脏页，那么效率会很低，为了解决这个问题，innodb plugin提供了一个参数innodb\_io\_capacity，用来表示磁盘IO的吞吐量，默认值是200，规则如下：在合并插入缓存时，合并插入缓存的数量为innodb\_io\_capacity的5%；在从缓冲区刷新脏页时，啥新脏页的数量为innodb\_io\_capacity。

      2、关于innodb\_max\_dirty\_pages\_pct值的争议，如果值过大，内存也很大或者服务器压力很大，那么效率很降低，如果设置的值过小，那么硬盘的压力会增加，建议是在75-80.并且innodb plugin引进了innodb\_adaptive\_flushng(自适应的刷新)，该值影响每秒刷新脏页的数量。

2.4、关键特性,为innodb提高性能的技术

      2.4.1、插入缓存

      当一个表有非聚集索引时，对于非聚集索引的叶子节点的插入不是顺序的，这时候需要离散的访问非聚集索引页，性能就在这里降低了，这是由于b+树的原理导致的。插入缓存就是用来解决这个问题的。

      对于非聚集索引的插入和更新操作，不是每一次都直接插入索引页，而是先判断插入的非聚集索引页是否在缓存中，如果在就直接插入，如果不在就放入到一个插入缓冲区中，好似欺骗数据库这个非聚集索引已经插入到叶子节点了。然后再以一定的频率插入缓存和非聚集索引页字节点的合并操作。

      插入缓存的使用需要满足以下两个条件(也就是非唯一的辅助索引)：索引是辅助索引；索引不是唯一的。

      2.4.2、两次写

      两次写给innodb带来的是可靠性，主要用来解决部分写失败(partial page write)。在应用重做日之前，我们需要一个页的副本，当写入失效发生时，先通过页的副本来还原该页，再进行重做，这就是doublewrite。

      doublewrite有两部分组成，一部分是内存中的doublewrite buffer，大小为2M，另外一部分就是物理磁盘上的共享表空间中联系的128个页，即两个区，大小同样为2M。当缓冲池的张也刷新时，并不直接写硬盘，而是回通过memcpy函数将脏页先拷贝到内存中的doublewrite buffer，之后通过doublewrite buffer再分两次写，每次写入1M到共享表空间的物理磁盘上，然后马上调用fsync函数，同步磁盘。

      2.4.3、自适应哈西索引

      由于innodb不支持hash索引，但是在某些情况下hash索引的效率很高，于是出现了 adaptive hash index功能，innodb存储引擎会监控对表上索引的查找，如果观察到建立hash索引可以提高性能的时候，则自动建立hash索引。

2.5、启动、关闭、恢复

innodb\_fast\_shutdown影响InnoDB表关闭。该参数有0、1、2三个参数。

      0 MySQL关闭时  完成所有的full purge和merge insertbuffer操作

         1默认值 只将缓冲池内的一些脏页刷新至磁盘

         2将日志都写入日志文件不会有任何事务丢失但下次启动时会进行recovery

      innodb\_force\_recovery影响整个innodb存储引擎的恢复状况，该值默认为0，表示当需要恢复时，需要执行所有的恢复操作，当不能进行有效恢复时，如数据页发生了corruption，mysql数据库可能宕机，并把错误写入错误日志中。

三．文件

3.1参数文件

      Mysql实例可以不需要参数文件，这是所有的参数值取决于编译Mysql时指定的默认值和源代码中指定参数的默认值。其参数文件是Mysql.cnf。

3.1.1、什么是参数

      参数是一个键/值对。可以使用show variables like命令查看，也可以通过information\_schema的GLOBAL\_VARIABLES视图来查找。

3.1.2、参数类型

      参数文件分为两类：动态参数和静态参数。动态参数意味着你可以在Mysql实例运行中进行更改；静态参数说明在整个实例生命周期内都不得进行更改，好像是只读的。对于动态参数，又可以分为global和session关键字，表明该参数的修改是基于当前会话还是真格实例的生命周期。有些动态参数只能在会话中进行修改，如autocommit；有些参数修改完后，在整个实例生命周期中都会生效，如binlog\_cache\_size；而有些参数既可以在会话又可以在整个实例的生命周期内生效，如read\_buffer\_size。

3.2、日志文件

3.2.1、错误日志

      错误日志对Mysql的启动、运行、关闭过程进行了记录。出现Mysql不能正常启动时，第一个必须查找的文件应该就是错误日志文件。使用show variables like ‘log\_error’来定位文件。

3.2.2、慢查询日志

      慢查询能为SQL语句的优化带来很好的帮助。设定一个阀值，将运行时间超过该值的所有SQL语句都记录到慢查询日志文件中。用参数long\_query\_time来设置。另一个参数log\_queries\_not\_using\_indexes，若运行的SQL语句没有使用索引，则这条SQL语句会被记录下来。

3.2.3、查询日志

      查询日志记录了所有对Mysql请求的信息，不论这些请求是否得到正确的执行。默认文件名为：主机名.log。

3.2.4、二进制日志

      二进制记录了对数据库执行更改的所有操作，但是不包括SELECT和SHOW操作，还包括了执行时间和更改操作时间。可用来恢复某些数据，同时也可以用来复制同步远程数据库。将binlog\_format设置成row，可以支持事务隔离级别为READ COMMITTED，以获得更好的并发性。在使用MIXED格式下，mysql采用STATEMENT格式进行二进制日志文件的记录，但是有一些情况下会使用ROW格式，可能的情况如下：

1、表的存储引擎为NDB，这个时候DML操作都会以ROW格式记录。

2、使用了uuid()、user(),current\_user(),found\_rows(),row\_count(),等不确定函数。

3、使用了insert delay语句

4、使用了用户定于的函数(UDF)

5、使用了临时表(temporary table)

注意：针对系统库mysql里面的表发生变化的处理规则如下：

1、 如果采用insert，update，delete直接操作表，则日志根据binlog\_format设定的格式记录。

2、 如果使用grant，revoke，set password等DCL语句，那么无论如何都会使用SBR模式记录。

3、 blockhole引擎不支持row格式，ndb引擎不支持statement格式。

3.3、套件字文件

 Unix系统下本地连接Mysql可以采用Unix套接字方法，需要一个套接字文件，可以使用show variableslike ‘socket’查询。

3.4、pid文件和表结构定义文件

         pid文件是实例启动是记录自己进程ID号的文件，表结构定义文件是以frm为后缀名的文件，还可以用来存放视图的定义。

3.5、innodb引擎文件

3.5.1、表空间文件

         默认表空间文件为ibdata1文件innodb\_data\_file\_path存储数据，innodb\_file\_per\_table可以按表分别产生一个表空间.db文件，但仅存该表的数据索引和插入缓冲等信息，其他信息如undo信息，系统事务信息，double write buffer等还是存放在默认表空间(ibdata1或表空间组)里。

3.5.2、重做日志文件

      redo log是在实例或者介质失败的时候，用来保证数据完整性。每个innodb存储引擎至少有一个重做日志组,每个重做日志文件组下至少又2个重做日志文件，如默认的ib\_logfile0、ib\_logfile1.为了得到更高的可靠性，你可以设置多个重做镜像日志组。

      因为重做日志条目先被写到日志缓冲中，然后根据一定条件刷新到磁盘重做日志文件中。与redo log相关的就是innodb\_flush\_log\_at\_trx\_commit的值，对innodb的性能影响很大。他有0，1，2三个值，0代表提交事务时，并不同步写redo log，而是等master threas每秒写。1代表commit的时候就将redo log缓存写入磁盘，2代表commit的时候将redo log缓存异步的写入磁盘。