**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()

####################################################

**03 – 日志的内容**

**- 数据是什么**  
  从不同的角度和层次来看，我们可以将数据库中的数据看作：  
  A. 关系数据  
  B. 元组或对象  
  C. 存在Page中的二进制序列

  因此Log中也可以记录不同的内容：  
**- 物理的日志(Physical Log)**  
  A. 记录完整的Page  
  B. 记录Page中被修改的部分(page中的偏移,内容和长度).

**优点：**因为恢复时，完全不依赖原页面上的内容，所以不要求持久化了的数据保持在一个一致的状态。  
       比如在写一个页面到磁盘上时，系统发生故障，页面上的一部数据写入了磁盘，另一部分丢失了。  
       这时仍然可以恢复出正确的数据。

**缺点：**Log记录的内容很多，占用很大的空间。如B-Tree的分裂操作，要记录约一个完整Page的内容。

**- 逻辑的日志(Logical Log)**  
  记录在关系(表)上的一个元组操作。  
  A. 插入一行记录。  
  B. 修改一行记录。  
  C. 删除一行记录。  
  逻辑日志比起物理的日志，显得简洁的多。而且占用的空间也要小的多。  
  但是逻辑日志有2个缺点：  
  A. 部分执行  
     例如：表T有2个索引，在向T插入1条记录时，需要分别向2个B-Tree中插入记录。  
     有可能第一个B-Tree插入成功了，但是第二个B-Tree没有插入成功。在恢复或  
     回滚时，需要处理这些特殊情况。  
  B. 操作的一致性问题  
     一个插入操作有一个B-Tree的分裂，页A的一半数据移到了B页，A页写入了磁盘，B页没有写入磁盘。  
     如果这时候发生了故障，需要进行恢复，逻辑日志是很难搞定的。

  逻辑的日志上的‘部分执行’的问题是比较好维护的，但是‘一致性’的问题维护起来是很复杂的。

**- 物理和逻辑结合的日志(Physiological Log)**  
  这种日志将物理和逻辑日志相结合，取其利，去其害。从而达到一个相对更好的一个状态。这种日志有2个特点:  
  A. 物理到page. 将操作细分到页级别。为每个页上的操作单独记日志。  
     比如,一个Insert分别在2个B-Tree的节点上做了插入操作，那么就分别为每一个页的操作记录一条日志。  
  B. Page内采用逻辑的日志。比如对一个B-Tree的页内插入一条记录时，物理上来说要修改Page Header的  
     内容(如,页内的记录数要加1)，要插入一行数据到某个位置，要修改相邻记录里的链表指针，要修改Slot的  
     属性等。从逻辑上来说，就是在这个页内插入了一行记录。因此Page内的逻辑日志只记录：’这是一个  
     插入操作’和’这行数据的内容‘。

  MySQL数据库InnoDB存储引擎的Redo Log 记录的就是这种物理和逻辑相结合的日志。  
  使用页内的逻辑日志，可以减少日志占用的空间。但是它毕竟还是逻辑日志，上面提到的2个问题能够避免吗？  
  A. 页面内的部分执行的情况可以认为不存在了。因为整个页面的操作是原子操作，在完成之前是不会写  
     到磁盘上的。  
  B. 操作一致性的问题仍然存在。如果在写一个Page到磁盘时发生了故障，可能导致Page Header的记  
     录数被加1了，但是数据没有刷新到磁盘上，总之页面上的数据不一致了。

  好在这个问题被缩小到了一个页面的范围内，因此比较容易解决。InnoDB存储引擎中用Double Write的方法  
  来解决这个问题。

**- Double Write**  
  Double Write的思路很简单:  
  A. 在覆盖磁盘上的数据前，先将Page的内容写入到磁盘上的其他地方(InnoDB存储引擎中的doublewrite   
     buffer，这里的buffer不是内存空间，是持久存储上的空间).  
  B. 然后再将Page的内容覆盖到磁盘上原来的数据。

  如果在A步骤时系统故障，原来的数据没有被覆盖，还是完整的。  
  如果在B步骤时系统故障，原来的数据不完整了，但是新数据已经被完整的写入了doublewrite buffer.  
  因此系统恢复时就可以用doublewrite buffer中的新Page来覆盖这个不完整的page.

  Double write 显然会曾加磁盘的IO。直觉上IO次数增加了1倍，但是性能损失并不是很大。Peter在  
  [innodb-double-write](http://www.mysqlperformanceblog.com/2006/08/04/innodb-double-write/" \o "innodb-double-write" \t "_blank)中说性能损失不超过5-10%。应该是因为多数情况下使用了批量写入的缘故。  
  A. Double write buffer是一段连续的存储空间，可以顺序写入。  
  B. Double write有自己的写buffer.  
  C. 先将多个要做doublewrite的page写入内存的buffer，然后再一起写到磁盘上。

  代码在:buf0dblwr.cc  
  buf\_flush\_write\_block\_low()调用  
  buf\_dblwr\_write\_single\_page()或 buf\_dblwr\_add\_to\_batch()来实现doublewrite.

**- Checksum**  
  检测页面是否一致的功能是靠Checksum来完成的,每个页面修改完成后都会记算一个页面的checksum。  
  这个checksum存放在页面的尾部.每次从磁盘读一个页到内存时，都需要检测页的一致性。  
  函数buf\_page\_is\_corrupted()是用来检测page的一致性的.

**- InnoDB Redo Log的日志类型**  
  InnoDB redo log的格式可以概括为:  
  <Space ID>+<Page NO.>+<操作类型>+<数据>.

  Redo Log记录的页面操作大致可以分为以下几种类型:  
  A. 在页面上写入N个字节的内容,这些可以看作是物理的Log.  
     MLOG\_1BYTE， MLOG\_2BYTES, MLOG\_4BYTES, MLOG\_8BYTES, MLOG\_WRITE\_STRING  
     各种Page链表的指针修改，以及文件头，段页等的内容的修改都是以这种方式记录的日志。  
  B. 页面上的记录操作。  
     MLOG\_REC\_\*, MLOG\_LIST\_\*, MLOG\_COMP\_REC\_\*, MLOG\_COMP\_LIST\_\*  
     这些日志记录了对B-Tree页的INSER, DELETE, UPDATE操作和分裂合并操作。  
  C. 文件和Page操作  
     MLOG\_FILE\_CREATE, MLOG\_FILE\_RENAME, MLOG\_FILE\_DELETE,  
     MLOG\_PAGE\_CREATE, MLOG\_INIT\_FILE\_PAGE, MLOG\_PAGE\_REORGANIZE  
  D. Undo Log操作  
     MLOG\_UNDO\_\*  
     InnoDB中将undo log的操作也记入了redo log. 为什么要这样做，在前面‘恢复’已经说了.

  这里只提到了部分Redo Log的类型，完整的定义在mtr0mtr.h文件中. 通过这个类型的定义，可以  
  很容易的找到都在哪些地方使用了。

  虽说Redo Log将数据的操作细分到了页面级别。但是有些在多个页面上的操作是逻辑上不可分裂的。  
  比如B-Tree的分裂操作,对父节点和2个子节点的修改。当进行恢复时，要么全部恢复，要么全部不  
  恢复，不能只恢复其中的部分页面。InnoDB中通过mini-transaction(MTR)来保证这些不可再分  
  的操作的原子性。

**- InnoDB Undo Log的日志类型**  
  MySQL数据库InnoDB存储引擎的undo log采用了逻辑的日志。  
  InnoDB undo log的格式可以概括为:<操作类型>+<Table ID>+<数据>.

  A. 从表中删除一行记录  
     TRX\_UNDO\_DEL\_MARK\_REC(将主键记入日志)  
     在删除一条记录时，并不是真正的将数据从数据库中删除,只是标记为已删除.这样做的好处是  
     Undo Log中不用记录整行的信息.在undo时操作也变得很简单.  
  B. 向表中插入一行记录  
     TRX\_UNDO\_INSERT\_REC(将主键记入日志)  
     TRX\_UNDO\_UPD\_DEL\_REC(仅将主键记入日志) 当表中有一条被标记为删除的记录和要插入的  
     数据主键相同时， 实际的操作是更新这个被标记为删除的记录。  
  C. 更新表中的一条记录  
     TRX\_UNDO\_UPD\_EXIST\_REC(将主键和被更新了的字段内容记入日志)  
     TRX\_UNDO\_DEL\_MARK\_REC和TRX\_UNDO\_INSERT\_REC，当更新主键字段时，实际执行的过程  
     是删除旧的记录然后，再插入一条新的记录。

  因为undo log还要被MVCC和Purge使用,所以还有TRX\_ID和DATA\_ROLL\_PTR等特殊的内容记录  
  在日志中。TRX\_UNDO\_INSERT\_REC不需要记录这些内容.因为MVCC中不可内引用一个不存在的数据。  
  这也是事务将insert和update、delete的undo log分开存放的原因。事务提交后,insert的undo  
  占用的空间就可以立即释放了.

  这些类型定义在:trx0rec.h.  
  记录日志的过程在:trx\_undo\_page\_report\_insert()和trx\_undo\_page\_report\_modify()中。  
  Undo操作在row0undo.c, row0uins.c和row0umod.c中, 入口函数是row\_undo().

**- 逻辑日志的一致性问题**  
  前面说了逻辑日志的一致性问题是很复杂的，为什么undo log要用逻辑日志呢？  
  因为redo log使用了physiological日志和MTR，就可以保证在恢复时重做完redo log后，  
  数据是一致。在执行undo时，就不必考虑这个问题了。

##############################################

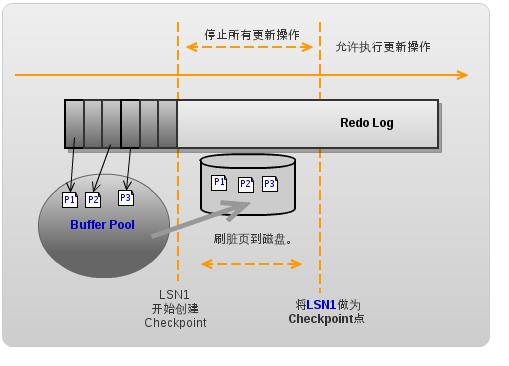
**04 – Checkpoint**

理论上来说,如果MySQL数据库InnoDB存储引擎的buffer足够大，就不需要将数据本身持久化。将全部的redo log重新执行一遍  
就可以恢复所有的数据。但是随着时间的积累，Redo Log会变的很大很大。如果每次都从第一条记  
录开始恢复，恢复的过程就会很慢，从而无法被容忍。为了减少恢复的时间，就引入了Checkpoint机制。

**- 脏页(dirty page)**  
  如果一个数据页在内存中修改了，但是还没有刷新到磁盘。这个数据页就称作脏页。

**- 日志顺序号(Log Sequence Number)**  
  LSN是日志空间中每条日志的结束点，用字节偏移量来表示。在Checkpoint和恢复时使用。

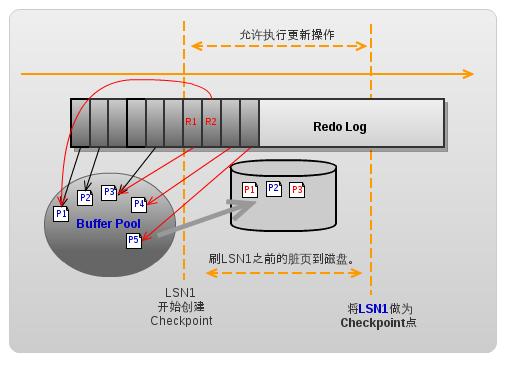
**- 原理**  
  假设在某个时间点，所有的脏页都被刷新到了磁盘上.这个时间点之前的所有Redo Log就不需要重  
  做了。系统记录下这个时间点时redo log的结尾位置作为checkpoint. 在进行恢复时，从这个  
  checkpoint的位置开始即可。Checkpoint点之前的日志也就不再需要了，可以被删除掉。为了  
  更好的利用日志空间，InnoDB以**环形缓存(**[**circular buffer**](https://en.wikipedia.org/wiki/Circular_buffer)**)**的方式来使用日志空间。



**—**Sharp Checkpoint

**- Sharp Checkpoint**  
  对于繁忙的系统来说，很少会出现这样的的一个时间点。为了能创造出这样一个时间点，最简单的办  
  法就是,在某个时间开始停止一切更新操作，直到所有的脏页被刷新到磁盘，Checkpoint被记录。   
  显然对于繁忙的系统, 这种方法是不合适的。能不能在checkpoint时不停止用户的操作呢？

**- Fuzzy Checkpoint**  
  如下图所示,如果刷脏页的同时用户还在更新数据，LSN1前的某个脏页在刷到持久存储之前就有可能被  
  LSN1之后的某个操作给修改了。当checkpoint完成时，LSN1后的部分操作(R1,R2对应的操作)也被  
  持久化了。当Sharp checkpoint完成时，持久存储中存储的数据是某个确切时间点的内存数据的快照。  
  Fuzzy checkpoint完成时，持久存储中存储的数据**不是某个确切时间点**的内存数据的快照。从某种  
  程度上，可以说持久存储中的数据丧失了一致性。在恢复时，必须要解决这个问题。



**—**Fuzzy Checkpoint

**- 幂等(Idempotence)规则**  
  如上图所示,checkpoint 在LSN1位置，当checkpoint完成时R1,R2对应的修改也被刷到了持久存储。  
  恢复时要从LSN1位置开始，包括R1, R2在内。重新执行后，数据还能正确吗？  
  幂等规则要求无论redo log被执行了多少次，数据始终正确。  
  InnoDB的redo log, 物理到Page，Page内是逻辑日志。  
  物理日志，天然支持幂等规则. 但是逻辑日志 需要特殊处理，才能支持满足幂等规则。

**- 数据页的最新(最大)LSN**  
  为了满足幂等规则，InnoDB中每个数据页上都记录有一个LSN。每次更新数据页时，将LSN修改为  
  当前操作的redo log的LSN。在恢复时，如果数据页的LSN大于等于当前redo log的LSN，则跳过此  
  日志。

**- 异步Checkpoint**  
  实现了幂等规则后，脏页就可以在任何时间，以任何顺序写入持久存储了。InnoDB的buffer pool有  
  一套单独的机制来刷脏页。因此很多情况下checkpoint时，并不写脏页到存储。只是将所有脏页的  
  最小的LSN记做checkpoint.  
  checkpoint的实现在log0log.c.  
  log\_checkpoint()实现异步checkpoint.

**- 同步Checkpoint**  
  InnoDB的buffer pool通过LRU的算法来决定哪些脏页应该被写入持久存储。如果包含最小LSN的  
  页面频繁的被更新，它就不会被刷到存储上。这样就可能导致checkpoint点很长一段时间无法前进,  
  甚至导致日志空间被占满。这时就要按照LSN由最小到大的顺序写一部分脏页到持久存储。  
  log\_checkpoint\_margin().  
  log\_calc\_max\_ages()用来计算，‘判断是否要执行同步checkpoint’用到的参数.

**05 – 缓存池(Buffer Pool)**  
学习到这里，我更倾向于说这是一个”Redo+Undo+Buffer”的模式。为了提搞IO性能，脏页缓存在buffer中，  
Redo log也要先缓存在内存中，doublewrite也有内存buffer. Buffer pool在这个模式中是至关重要的。

**- 页分类**  
  Buffer pool内的页分为三种:  
  A. 未被使用的页(空白的buffer),没有映射到一个数据文件中页。  
  B. 净页，映射到了一个数据文件页，而且没有被修改过。内容和数据文件的页一样。  
  C. 脏页，映射到了一个数据文件页，并且数据被修改过。内容和数据文件的页不一样。

**- LRU**  
  InnoDB维护了两个LRU列表。当空间不足时，用来决定哪些脏页应该被首先写入磁盘，哪些净页应该被释放掉。  
  A. buffer\_pool->LRU，普通LRU链表，记录所有数据缓冲页。  
  B. buffer\_pool->unzip\_LRU，是压缩页（row\_format=compressed）解压后数据缓冲页LRU链表。  
  LUR链表中的页面按最近一次的访问的时间顺序排列,头部是最后一次被访问的页面，尾部是最早一次被  
  访问的页面。无论是读还是写一个页面上的数据，都要先获取这个页面。因此可以在获取页面时，维护  
  LRU链表.当获取一个页面后，将其放到LRU链表的头部即可。  
  buf\_page\_get\_gen()和buf\_page\_get\_zip()用来获取一个页面，他们调用  
  buf\_unzip\_LRU\_add\_block()和buf\_page\_set\_accessed\_make\_young()来维护LRU链表。

**- flush\_list**  
  同步checkpoint时，需要根据数据页修改的先后顺序来将脏页写入持久存储。因此除了LRU链表，  
  buffer pool中还有一个按脏页修改先后顺序排列的链表,叫flush\_list.当需要同步checkpoint时，  
  根据flush\_list中页的顺序刷数据到持久存储。  
  A. 一个页只在flush\_list中出现1次，因为一个页面只需要写一次。  
  B. 按页面最早一次被修改的顺序排列。

**06 – Mini-Transaction(MTR)**  
前面提到Redo Log将数据的操作细分到了页面级别。但是有些在多个页面上的操作是逻辑上不可分裂的。  
InnoDB中用Mini-Transaction来表示这些不可再细分的逻辑操作。

**- MTR的一致性**  
  为了满足MTR的一致性，MTR做了如下的设计：  
  A. MTR的所有日志被封装在一起，当MTR提交时一起写入redo log buffer.  
     这样做有2个好处：  
     \* 减少并发MTR对redo log buffer 的竞争。  
     \* 连续的存储在一起，恢复时的处理过程更简单。  
  B. InnoDB在redo log的层面，将一个MTR中的所有日志作为Redo log的最小单元。在恢复时，一个MTR  
     中的所有日志必须是完整的才能进行恢复。

**- MTR日志的封装**  
  为了在日志文件中区分不同的MTR，MTR将MLOG\_SINGLE\_REC\_FLAG或MLOG\_MULTI\_REC\_END写入  
  redo log(mtr\_log\_reserve\_and\_write()).  
  A. 如果MTR的日志中只有一行记录，在日志的开始处添加MLOG\_SINGLE\_REC\_FLAG，表示MTR中只有  
     一条记录。  
  B. 如果MTR的日志中有多行记录，在日志的结尾处添加一个类型为MLOG\_MULTI\_REC\_END的日志，  
     代表MTR的日志到此结束.

**- MTR的LSN**  
  A. 因为在将日志写入redo log buffer时，才能获得LSN。所以修改数据时，并没有修改页上的LSN。  
     需要在MTR获得LSN后统一修改。  
  B. 一个MTR只有一个LSN. 一个MTR内修改的所有页的LSN相同。这样checkpoint就不会出现在MTR的中间。  
  C. 在获得LSN后，如果被MTR修改的脏页不在buffer pool的flush\_list里，就会被添加进去。

  看mtr\_memo\_slot\_note\_modification()和buf\_flush\_note\_modification().

**- 页级锁**  
  提交时才写日志到redo log的做法，决定了MTR要使用页级锁。  
  A. 一个页面不能同时被多个活动的MTR修改。  
  B. MTR中数据页的锁，直到MTR提交时(日志写入redo log buffer)后才释放。

  锁对象存储在mtr的memo中。调用mtr\_s\_lock和mtr\_x\_lock来加锁时，锁对象被保存到memo中。  
  解锁在mtr\_memo\_slot\_release()中完成。

**- MTR的ROLLBACK**  
  看完MTR的代码发现mtr没有记录undo日志，也不能rollback. MTR都是很小的操作单元，而且每个MTR  
  都有明确的操作目标，因此比较容易保证其正确性。  
  A. 因为页面操作是在内存中完成，并且页面有固定的格式，因此很多的页面操作是不会失败的。  
     InnoDB存储引擎中的很多写页面的函数都没有返回值.  
  B. 在对任何页面操作前，先要检查是否可能发生错误。如果可能发生错误就不能往下执行。  
     如,当插入一行记录到B-Tree的节点时，首先检查页面有足够的空间。  
  C. 使用更大粒度的锁(如B-Tree的锁),并且按照一定的顺序加锁。这样才能不导致死锁问题。

  以上是自己看代码后的大概印象，不一定说到了正点上。MTR模块的代码虽简单，但是MTR在其他模块大量的  
  使用。要透彻的理解MTR，估计还得要看其他模块的代码，整理出来大部分MTR操作过程才行.

**06 – 参考**  
  A. [Database Systems: The Complete Book (2nd Edition)](http://www.amazon.com/Database-Systems-Complete-Book-2nd/dp/0131873253/ref=sr_1_1?ie=UTF8&qid=1307950354&sr=8-1)  
  B. [Transaction Processing: Concepts and Techniques](http://www.amazon.com/Transaction-Processing-Concepts-Techniques-Management/dp/1558601902/ref=sr_1_1?ie=UTF8&qid=1307950283&sr=8-1)  
  C. [how-innodb-performs-a-checkpoint](http://www.xaprb.com/blog/2011/01/29/how-innodb-performs-a-checkpoint/)  
  D. [InnoDB fuzzy checkpoints](https://www.facebook.com/note.php?note_id=408059000932" \t "_blank)  
  E. [Heikki Tuuri Innodb answers – Part I](http://www.mysqlperformanceblog.com/2007/10/26/heikki-tuuri-innodb-answers-part-i/" \t "_blank)  
  F. [Heikki Tuuri Innodb answers – Part II](http://www.mysqlperformanceblog.com/2007/11/04/heikki-tuuri-answers-to-innodb-questions-part-ii/" \t "_blank)