文章已收录我的仓库：Java学习笔记与免费书籍分享

事务底层原理(INNODB)

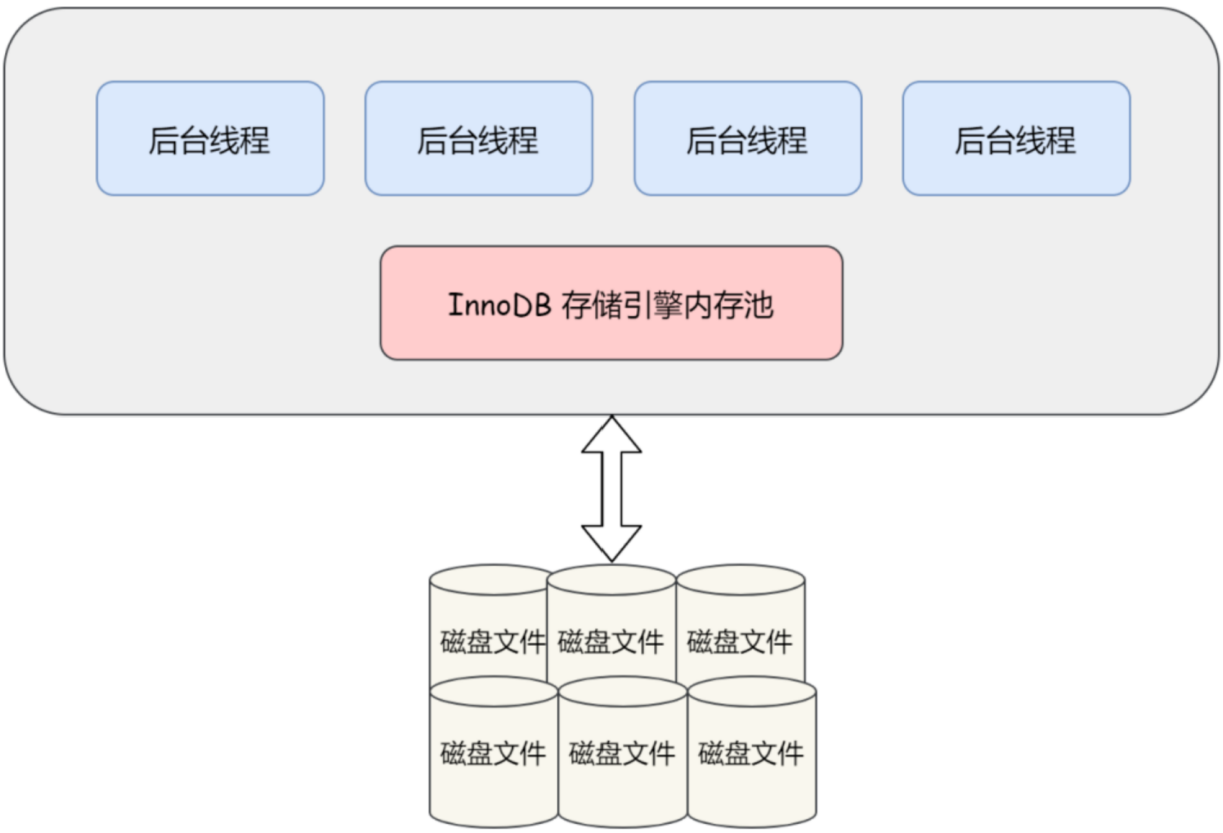
前言

事务必须满足ACID四个特性，即原子性、一致性、隔离性和持久性，隔离性由锁来保证，我们主要研究事务是如何保证原子性、一致性和持久性。

redo log

为什么需要 redo log

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，可将其视为**基于磁盘的数据库系统**（Disk-base Database），修改数据需要先将磁盘中的行记录读至内存，修改后再保存至内存持久化。



为了保证事务的持久性，每次执行一条SQL语句修改数据后就必须将修改后的值写入磁盘，而写入磁盘是非常缓慢的，在大型的程序下，频繁的写入磁盘会使得性能急速下降，由于这个原因，**在Innodb引擎中，所有的操作都被设计在内存池中进行以提高效率，如果数据在内存池中，则直接在内存中修改，否则从磁盘读入内存中操作。**

Innodb是多线程引擎，其中专门有一个后台线程`master thread`周期性的将缓冲池中的内容刷新到磁盘中，这样似乎便可以在保证持久性的同时又提高了效率，但这是不正确的，在这个周期内存在于内存池中的数据也是可能丢失的，无法保证数据的持久性，因此我们需要引入**预写日志(WAL)**策略，这里的日志就是redo log，即重做日志。

WAL是很容易理解的，即**提交事务时，先将修改过的内容提交到磁盘的重做日志文件中进行持久化，然后再修改内存页面为脏页，释放内存页面占用，一旦没有任何事务占用页面，则等待被后台进程刷新至磁盘**这样即使在等待的过程中内存数据因为数据库死机或其他原因而丢失，也可以通过重做日志中的记录来恢复该数据以保证数据的持久性。

还要注意的一点是，在未提交事务时，redo log 是存在于**内存缓冲区**中的，这样做的目的是为了提升效率，只有在特定条件如事务提交时，才将 redo log buffer 刷新到磁盘的 redo log文件中。

例如，当我们执行修改语句时，我们先将磁盘中的行数据所在的**页**读取至内存池虚拟页中，此后便在内存池中对该页进行修改，修改前，首先将要修改的值以及事务 id 的等其他信息写入 redo log 缓冲区，提交事务时，刷新redo log缓冲区到磁盘以持久化，之后再修改内存页中的数据，同时将该页标记为**脏页**，**master线程将内存中的脏页写回磁盘。**

当一个事务修改了内存脏页时，这个脏页无法被刷回（例如读写锁），直到事务提交，而事务提交的前提条件是 redo log 被持久化。不管怎样，Innodb 总能保证在脏页刷新前对应的 redo log buffer 已经被刷回磁盘。

这样即时内存中脏页丢失，数据库还是可以通过已持久化的重做日志还恢复数据。

一些问题

此时，你可能会几点疑问？

1. master线程最终也还是要将所有脏页刷新回磁盘，这相比于每修改一次数据就刷新磁盘效率高在哪呢？
2. 重做日志可能会存在于缓存中，这样也有丢失的可能。
3. 重做日志还是要写回磁盘，会不会降低效率？
4. 当脏页数据丢失时，数据库是如何通过重做日志恢复数据？

关于第一点的问题，由于每修改一次数据就写回磁盘，写回磁盘的位置可能是随机的，如果对磁盘性能稍微了解的话就会知道**随机写效率是极低的**，在等待一定时间后，此时存在大量将写回磁盘的数据，master线程可能会对这些写回进行排序，使得**顺序写回**变得可能，完全顺序读写磁盘效率是相当于读写内存的，是非常高效的，即时是不完全的顺序，其效率也要高于随机写，这就是为什么master是周期性的写回。

关于第二点的问题，要知道重做日志缓提交后就被刷新回磁盘，也就是说如果日志缓冲丢失了，通常情况下事务处于未提交的状态，在未提交时抛出异常是合理的，当程序员发现错误发生在事务提交之前时，它就应该认识到这次事务是执行失败的，下次应该重新执行该事务。

你可能还会想：如果redo log写磁盘写一半而宕机了怎么办，此时事务已提交，但脏页未被刷新？注意此时事务虽已提交。但处于**正在提交**且未提交完成的状态，只有在 redo log 写完之后才会标记为提交成功。

不过在某些情况下，还是会遇到一些数据不一致的问题，我们会在文末探讨这些问题。

第三点、第四点我们下文会详细介绍。

重做日志结构

重做日志结构共有51种之多，未来还会更多，这里简单说明 insert 与 delete 的重做日志结构，对于这两种操作的重做日志结构如下图所示：

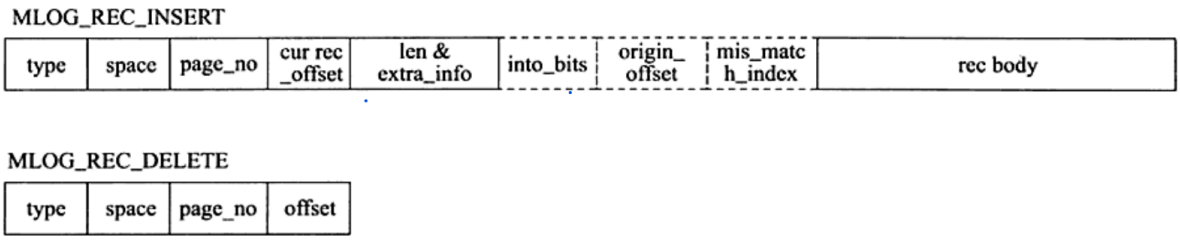


图 7-11　插入和删除的重做日志格式

他们有着通用的头部字段：

- type：重做日志的类型。
- space：对于记录所在表空间的ID。
- page_to：页偏移。

知道了 space 和 page_to 便可以确定该记录物理页以及该记录所在，后续恢复数据会利用这一点。

对于insert日志，占用内存较多，因为要存储新插入的值。

而对于delete日志，我们仅须通过 type 知道这是一条删除日志，无须数据字段。

重做日志文件结构

在理解重做日志本身结构后，我们来看看重做日志是被如何组织的。

在 InnoDB 中，重做日志以块的形式存储，这通常是512字节，与磁盘一次性读写的大小相等，因此**重做日志块的读写是一次原子操作**，不需要额外的存储。

如果重做日志大小超过了512字节，那么它将被存储在两个块中，为了消除内部碎片，一个块可能和存储多个日志文件，但代价是需要更多的标记位以表示各个日志文件。

重做日志块在缓存中被组织成类似于数组的形式，我们称这个链表为重做日志缓冲区，即 redo log buffer，日志缓冲存在于内存中。

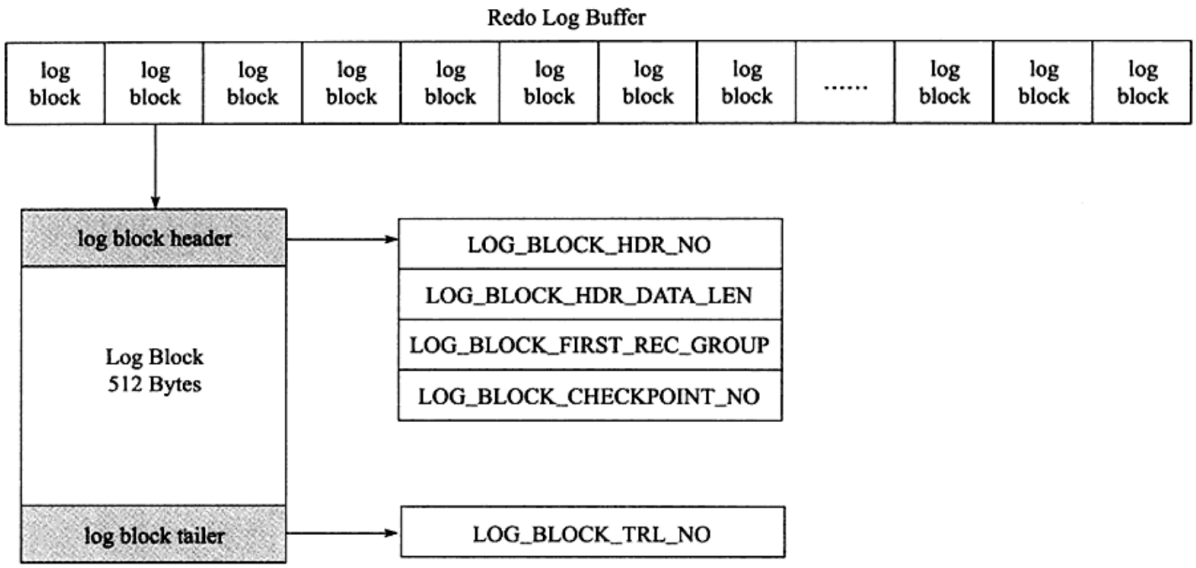


图 7-7　重做日志块缓存的结构

重做日志块允许同时存在多个重做日志，例如下图，有T1和T2两个日志，T1大小为762字节，在去掉头部和尾部20字节后，一个块仅能装下492字节，因此T1被分割装入第二块中。

头部字段可以表示该块是第几块，同时也存在字段LOG_BLOCK_HDR_DATA_LEN字段标识第一个重做日志在该块中的偏移量，例如块二中该字段被设置为270，这样数据库就知道T2的偏移量为270，而T1的尾部（或 T2 的头部）具有结束标志，数据库也知道T2（如果有的话）的偏移量是370。

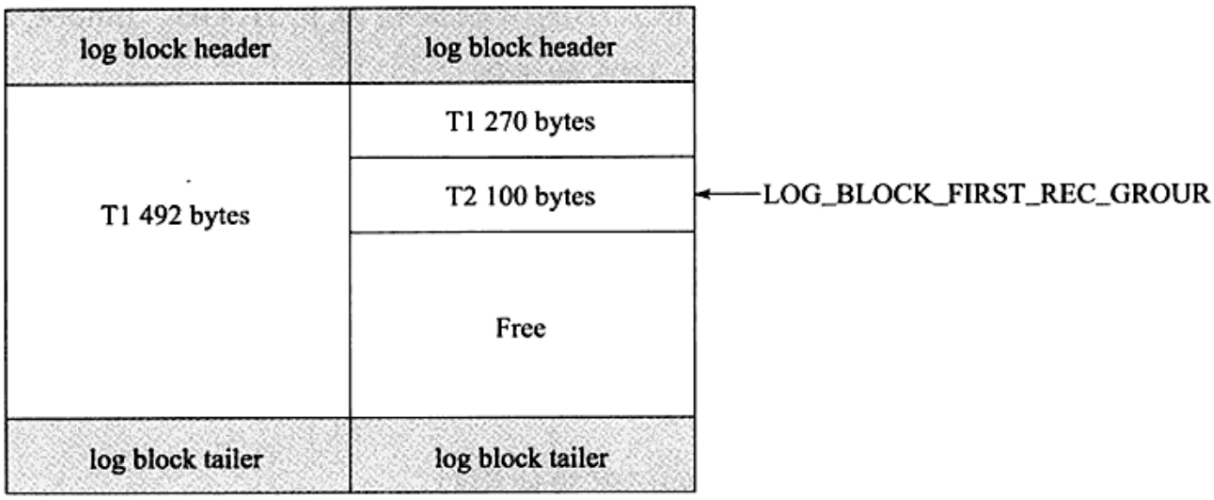
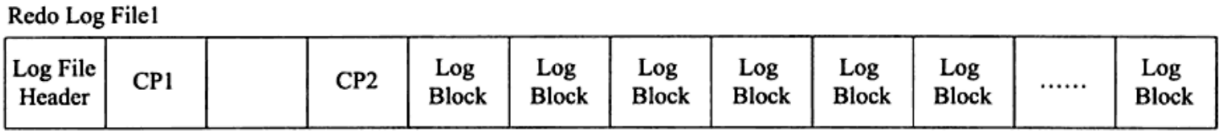


图 7-8　LOG_BLOCK_FIRST_REC_GROUP 的例子

重做日志文件结构只需要在redo log buffer的基础上加上一些文件头信息即可，如下图所示：



这些信息的具体大小如下，Log File Header是文件头，它包含文件大小、版本号、LSN等一些有用信息，CP1和CP2名称为checkpoint，它包含检查点信息，关于检查点信息我们后面再说。

表 7-4　redo log file 前 2KB 部分的内容

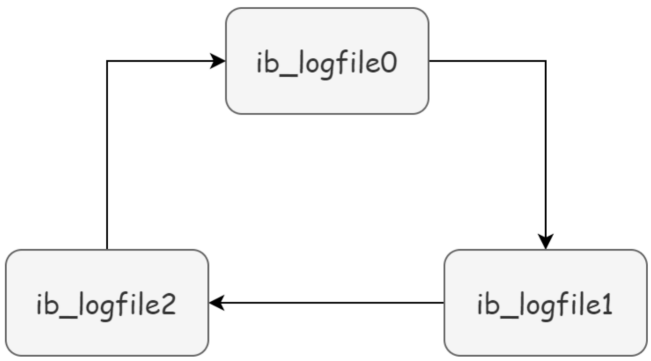
名　　称	大小（字节）
log file header	512
checkpoint1	512
空	512
checkpoint2	512

在写入文件时，从redo log buffer头部开始依次读取所有重做日志写入磁盘，重做日志在磁盘中被存储的形式与在缓冲区中存储形式相同。

log group与循环写入

一个日志组由多个日志文件组成，我们通常只使用一个日志文件组，而设置多余的日志文件组用于备份。

日志文件写回磁盘是循环写入的，什么意思呢？假设一个日志文件组有三个日志文件f1、f2和f3，我们从f1开始，顺序读取redo log buffer中的日志写入f1，当f1写满时写f2，f2写满时写f3，而f3满时则继续写f1。



log group中的多个文件在逻辑上被组织成连续的，尽管在实际的磁盘中他们不一定要连续。

写入同一个文件内的大部分记录都是被顺序组织的，因此重做日志的写入磁盘是非常快的。

日志何时写入磁盘？

这个具体的规则是：

1. 事务被提交时。
2. 当log buffer中内存不够时，通常使用超过一半便触发写入磁盘动作。
3. 写入遇到CheckPoint点时。

数据恢复：LSN标记

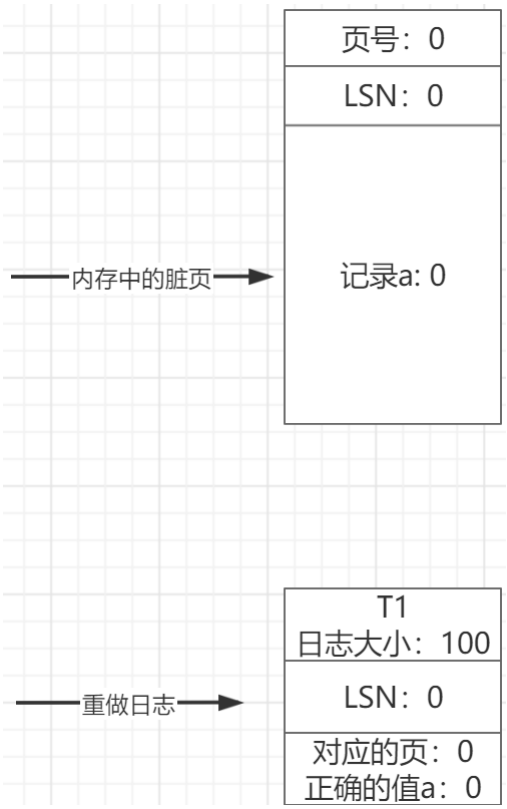
LSN称为日志的逻辑序列号(log sequence number)，在innodb存储引擎中，LSN占用8个字节。LSN的值会随着日志的写入而逐渐增大。

LSN不仅存在于redo log中，还存在于实际数据页中，LSN是用于恢复数据的关键点。

简单来说，LSN就是用于标记的版本号，在数据库中，随着事务发生并且条目被写入日志文件，LSN 会不断增大，在 FP3 之前，上限为 0xFFFF FFFF FFFF，**LSN每次增大一个重做日志的大小。**

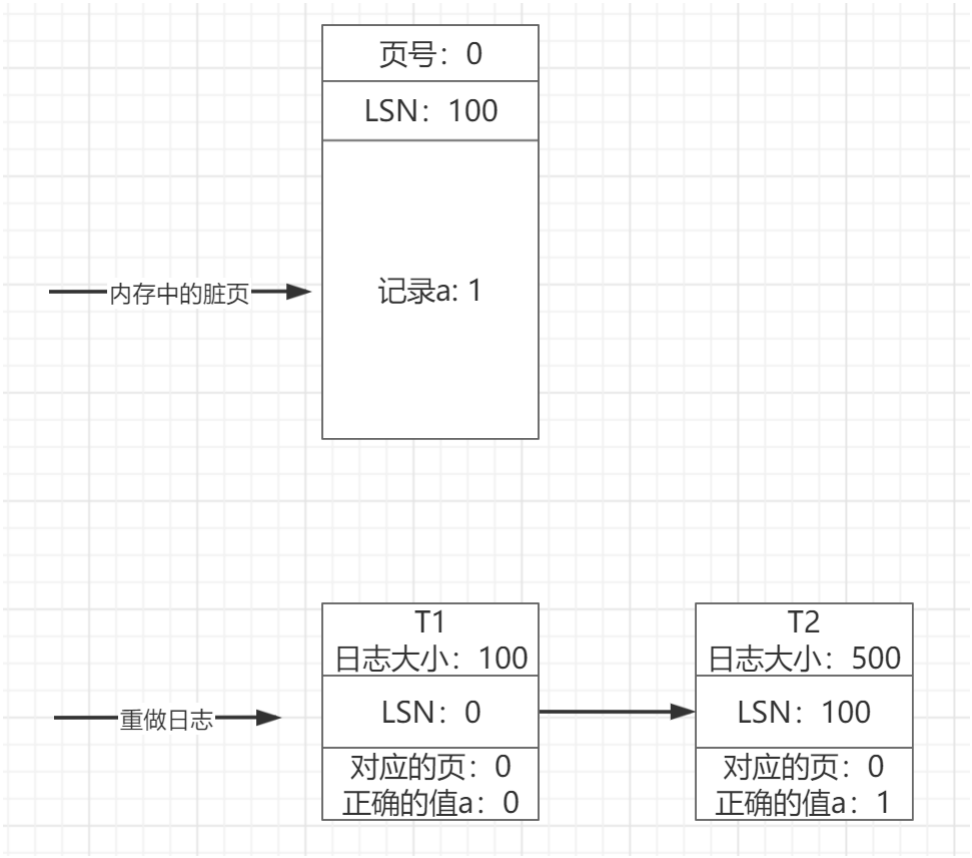
LSN在重做日志中和实际页中都会存在，这一点是很重要的。在页中的LSN表示最后一次的被刷新LSN号。

举个简单例子而言，假设初始时LSN为0，此时我们执行一条插入语句: `insert into test value(0);` ，数据库会为其生成一条重做日志，设为T1，事务提交时，重做日志被持久化，同时数据库将插入数据所在的页0设置为脏页，此时页0与对应的重做日志LSN都为0。



再假设上图中的脏页已经被正确的刷新会磁盘了，那么磁盘中对应真实页的LSN即为0，这里认为磁盘中的页号也是0，这是与重做日志中的页号对应的。

我们假设T1大小为100byte，那么此时LSN号自增重做日志的大小，即 LSN = 0 + 100 = 100，现在假设我们执行一条修改语句，将 a 的值修改为 1，数据库从磁盘中读取数据到内存，为了方便，我们仍然认为读到虚拟内存中的页也为0，设此时为修改语句生成的重做日志为T2，那么此时的结构为：



这是正确的结果，此时**重做日志中页0的LSN被更新为100**，那如果内存中的脏页丢失了呢？数据库要如何恢复信息呢？

如果此时数据库宕机而丢失了此时页0的信息，那么磁盘中页0的LSN为上一次的LSN，即0，数据a的值也是0，这是完全错误的，当数据库启动时，它会扫描重做日志试图恢复错误。

首先扫描T1，发现T1中的日志指向物理页0，并且发现 $T1中的LSN <= 对应页中的LSN$，于是数据库就会认为该LSN版本的数据是一致的，未出错。

数据库继续读取T2，T2中的日志也指向物理页0，并且发现 $T2中的LSN > 对应页中的LSN$，数据库可以断定，该版本发生了错误，于是数据库读取T2中的页偏移以确定对应记录的位置，根据T2中的数据修改页0中的数据，此时设置 a = 1，并更新LSN号，于是，数据库正确的从错误中恢复了，数据的持久性得以保证。

你大概已经明白了数据恢复主要流程，数据库主要通过比较LSN版本号以得知是否出错，如果出错则根据重做日志中的信息以恢复数据。

你可以在命令行键入`mysql > SHOW ENGINE INNODB STATUS\G`，找到有关于LOG的记录，其中Log Sequence Number即为当前LSN。

在我们的描述中，一旦数据库丢失数据，其必须要扫描所有的重做日志才能从错误中恢复过来，当重做日志文件非常大时，这需要相当久的耗时，为此，引入了CheckPoint技术。

CheckPoint 技术

检查点的作用

checkpoint主要2个作用：

保证数据库的一致性，这是指将脏数据写入到硬盘，保证内存和硬盘上的数据是一样的。

缩短实例恢复的时间，实例恢复要把实例异常关闭前没有写出到硬盘的脏数据通过日志进行恢复。如果脏块过多，实例恢复的时间也会很长，检查点的发生可以减少脏块的数量，从而提高实例恢复的时间。

工作原理

前面提到过，脏页回在缓冲池中继续存在一段时间后才会被后台线程回收，如果此时我们要关闭数据库但内存中的脏页还未被刷新怎么办，难道我们还要等待其被刷新吗？但我们又如何知道内存何时被刷新呢？一旦我们直接强制性关机，岂不是无故的让数据库陷入错误，下一次启动时又需要大量的时间恢复？

在某些情况下，我们需要强制性将内存中的脏页刷新回磁盘。

一旦我们主动刷新内存，将部分脏页刷回磁盘（某些脏页可能被事务引用，无法刷新），我们令当前内存中剩余脏页的最小的 LSN 为 MLSN，由于**LSN号是递增**的，那么我们就可以肯定**所有LSN小于MLSN的版本都已经被正常刷新回磁盘持久化**，那么下一次恢复时对于这一部分我们便不用去扫描了，这样就可以极大的节省恢复时间。

通常，Innodb 并不会扫描所有的页看看能不能刷回，这太慢了，而是维护了一个 LSN 到页的映射，Innodb 会从小到大顺序遍历 LSN，直到找到一个被事务锁定或因其他原因无法刷回磁盘的 LSN 对应的页，而此 LSN 之前的页都可以被刷回，这就是 **flush 链表**，这里不再赘述。

这个最大的LSN就是检查点，顾名思义，即从这个点后开始检查，之前的点无须检查。

在特定条件下，检查点被触发，checkpoint 进程开始一个checkpoint事件，并记录下当前可顺序刷回的LSN最大的脏页，称为检查点，也叫CP RBA。

而后，checkpoint进程通知DBWn进程将所有checkpoint RBA之前的buffer cache里面的脏块写入磁盘，这时候页会被锁定，无法被事务修改。

确定脏块都被写入磁盘以后，checkpoint进程将checkpoint信息(LSN)写入/更新数据文件和控制文件中。

在此我们还要引入一个概念，称为`write_pos`(write_pos和checkpoint信息被保存在日志文件头部的CP1和CP2字段中)，即**当前写入点**，那么**数据库恢复时只需要恢复 $check$point 到 $ writepos $之间的重做日志即可。**

例如当前写入LSN为13000，上一次检查点为10000，那么只需恢复10000到13000之间的信息即可。

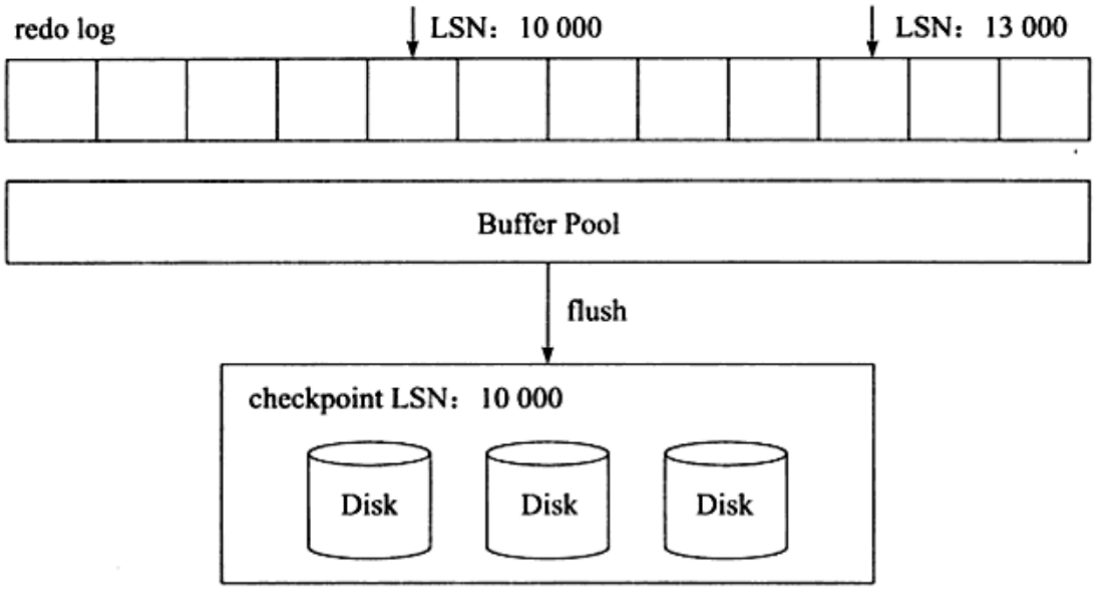


图 7-12　恢复的例子

检查点何时触发？

在日志何时写入磁盘时我们曾介绍过3种方式，事实上前两种方式其实就是触发了检查点事件，但检查点事件却不只这两点，因此我才会将第三点写上去。

现在我们来具体的探讨一些检查点事件何时被触发。

MYSQL中的检查点分为两类，即：

1. sharp checkpoint：激烈检查点，要求尽快将所有脏页都刷到磁盘上，对I/O资源的占有优先级高。
2. fuzzy checkpoint(默认的方式)：模糊检查点，会根据系统负载及脏页数量适当平衡，不要求立即将所有脏页写入磁盘，可能会刷新部分脏页。

触发时机：

1. 数据库正常关闭时，即参数 innodb_fast_shutdown=0(默认地)时，需要执行sharp checkpoint。
2. redo log发生切换时(即切换日志)或者redo log快满的时候进行fuzzy checkpoint。
3. master thread每隔1秒或10秒定期进行fuzzy checkpoint。
4. innodb保证有**重做日志文件足够多的空闲页**(防止覆盖)，如果发现不足，需要移除LRU链表末尾的page，如果这些page是脏页，那么也需要fuzzy checkpoint。
5. innodb buffer pool中脏页比超过限度(通常是最大大小的一半)时也会触发fuzzy checkpoint。

关于redo log的性能

redo log日志写回

即时redo log是被顺序写回磁盘的，但只要是关于磁盘的IO，其性能仍然会下降。学到这里，你应该发现数据库的性能与磁盘性能有很大关系。

InnoDB下存在一个参数：innodb_flush_log_at_trx_commi，**可取值为: 0/1/2**。

> **innodb_flush_log_at_trx_commit=0**，表示每隔一秒把log buffer刷到文件系统中(os buffer)去，并且调用文件系统的“flush”操作将缓存刷新到磁盘上去。也就是说一秒之前的日志都保存在日志缓冲区，也就是内存上，如果机器宕掉，可能丢失1秒的事务数据。
>
> **innodb_flush_log_at_trx_commit=1**(默认)，表示在每次事务提交的时候，都把log buffer刷到文件系统中(os buffer)去，并且调用文件系统的“flush”操作将缓存刷新到磁盘上去。这样的话，数据库对IO的要求就非常高了，如果底层的硬件提供的IOPS比较差，那么MySQL数据库的并发很快就会由于硬件IO的问题而无法提升。
>
> **innodb_flush_log_at_trx_commit=2**，表示在每次事务提交的时候会把log buffer刷到文件系统中去，但并不会立即刷写到磁盘。如果只是MySQL数据库挂掉了，由于文件系统没有问题，那么对应的事务数据并没有丢失。只有在数据库所在的主机操作系统损坏或者突然掉电的情况下，数据库的事务数据可能丢失1秒之类的事务数据。这样的好处，减少了事务数据丢失的概率，而对底层硬件的IO要求也没有那么高(log buffer写到文件系统中，一般只是从log buffer的内存转移的文件系统的内存缓存中，对底层IO没有压力)。

但要知道，虽然修改参数可能带来性能的提升，但却丢失了数据的一致性与持久性。

修改语句性能

前面提到过，如果要修改信息，如果信息不再内存中，必须要先读入磁盘，再磁盘中修改，最后写回磁盘，需要两次磁盘IO和一次内存操作，效率低下。

为此INNODB引入一种称为写缓存的技术，这是针对**非唯一普通索引字段**的删除与更新语句的优化，这里我简单的介绍一下原理。

> 唯一索引总是需要检查索引冲突。

具体的，修改某个记录时(假设页不在内存中，如果在内存中不会触发写缓存)，将这个修改操作记录在缓存中，等到下次读取该数据时，再将其合并更新，并且写入缓存。

你可以发现这样减少了一次磁盘IO，因为读取、写入和写回合并了，原先需要三次磁盘IO，现在只需要两次。

但这也不是必然的，如果下一次读取即将到来，那么这个技术毫无意义，还会浪费空间，因为修改时读入内存，当下一次读取很快到来时，数据还未被刷新，读取也会读内存，此时也会节省一次磁盘读取。

更多信息可以自行查阅。

undo log

为什么需要undo log

redo log是为了保证事务的持久性和一致性，那么undo log便是为了保证事务的原子性，是为了事务回滚而设计的。

例如假如你执行插入语句，那么undo log中可能就会对应的记录一条删除语句，这样当该事务回滚时，就会执行undo log中的删除语句使得事务好像根本就没有被执行，即事务回滚成功。

undo log是一种逻辑日志，因此只是将数据库逻辑地恢复到原来的样子。所有修改都被逻辑地取消了，但是数据结构和页本身在回滚之后可能大不相同。例如已经插入的数据只是被逻辑的删除，真实数据仍然存在。这与数据区本身执行SQL语句有关，数据库正常的删除语句也是一种逻辑删除，这是因为数据库要提供MVVC多版本功能，即在并发访问下，一个事务对当前行上锁修改，另一个事务可以读取之前的版本。**undo log也是实现MVVC的底层机制。**

前置知识：了解SQL语句执行过程

我们主要说明DELETE和UPDATE语句，SELECT和INSERT并不会有什么歧异。

向我们之前所说的，现在大多数数据库都提供MVVC功能，即使一个事务执行了删除操作数据库也不会立即删除，而是判定是否有其他事务在引用该记录，然后再删除，这个过程由后台Purge线程执行。

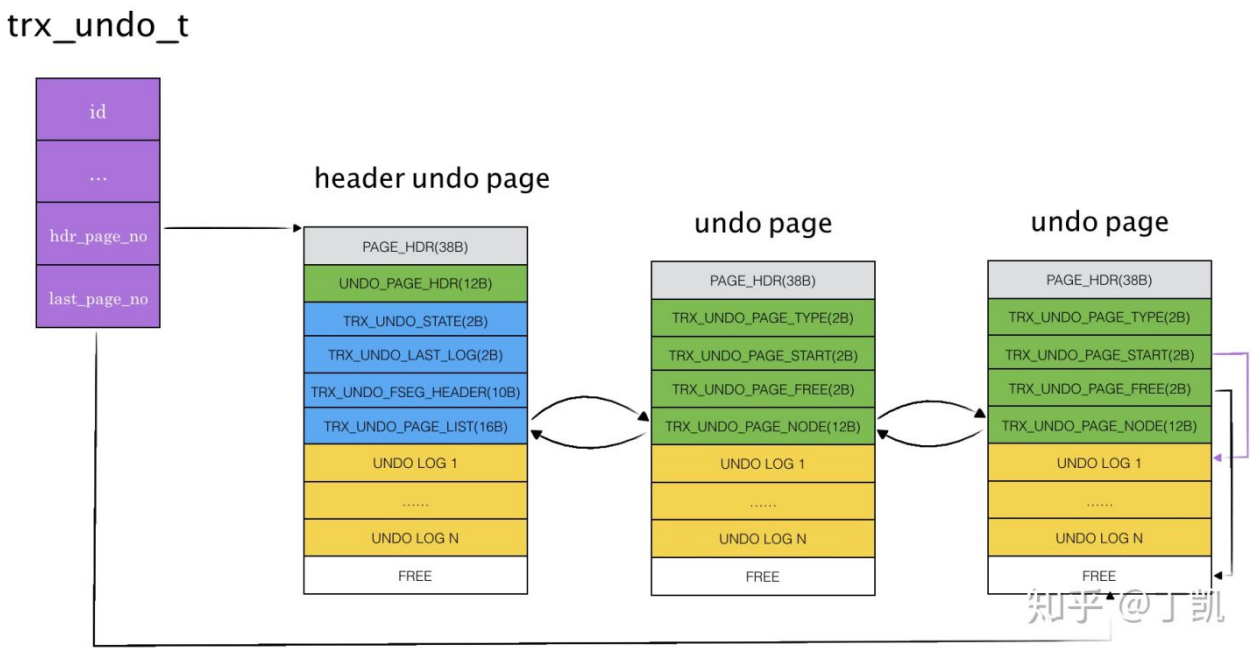
- delete操作不会直接删除，而是将delete对象打上delete flag，标记为删除，等待purge线程周期性的尝试回收。
- update分为两种情况：update的列是否是主键列。
  - 如果不是主键列，update是直接进行的。
  - 如果是主键列，update分两部执行，先删除该行(也是逻辑删除)，再插入一行目标行。

你可能会奇怪为什么为什么更新操作会有两种情况，道理是很简单的，非主键列直接修改是可以做到的，但主键列是在B+树上的，直接修改会使得B+树数据结构不符合，直接修改难以操作，逻辑复杂，因此采用先删除再插入的方式。

undo log存储结构

innodb存储引擎对undo log的管理采用段的方式，其存在于rollback segment之中，rollback segment称为回滚段，每个回滚段中有1024个undo log segment，在每个undo log segment中进行undo log页的申请，所有undo log页被组织成链表的形式。

同redo log类似，undo log页也是可被重用的，即一个页可能存在两个事务对应的undo log页。



undo log日志结构

与redo log不同，redo log是基于物理页信息的，而undo log是基于SQL语句的，具体的，我们将undo log分为两种，即 insert undo log 和 update undo log。

insert undo log主要针对于insert操作，而update undo log针对于delete和update两种操作，其具体结构如下图所示：

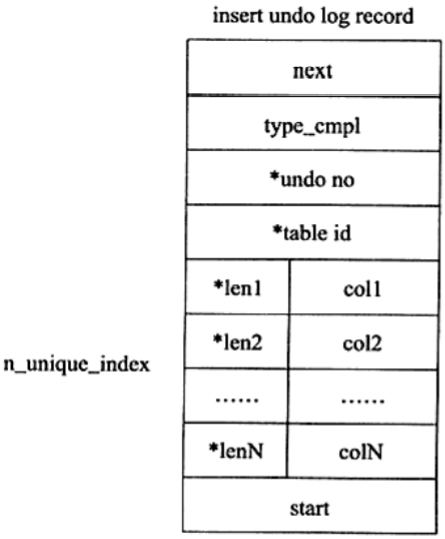


图 7-14　insert undo log 的格式

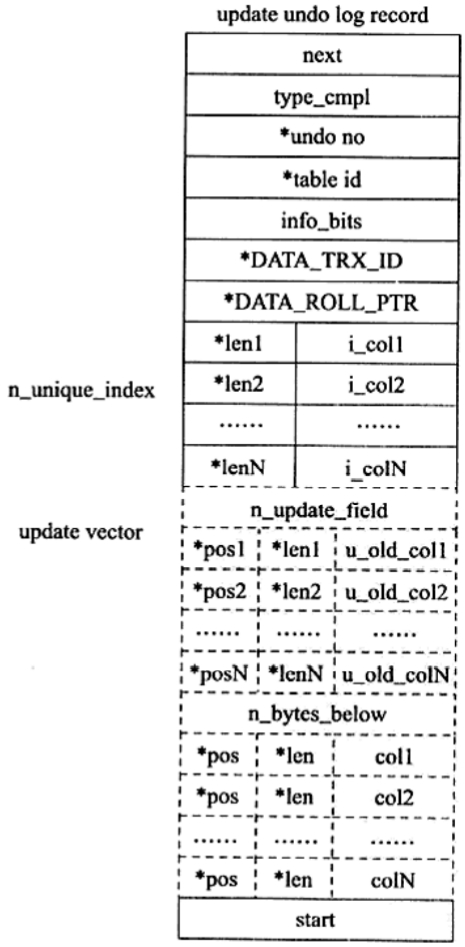
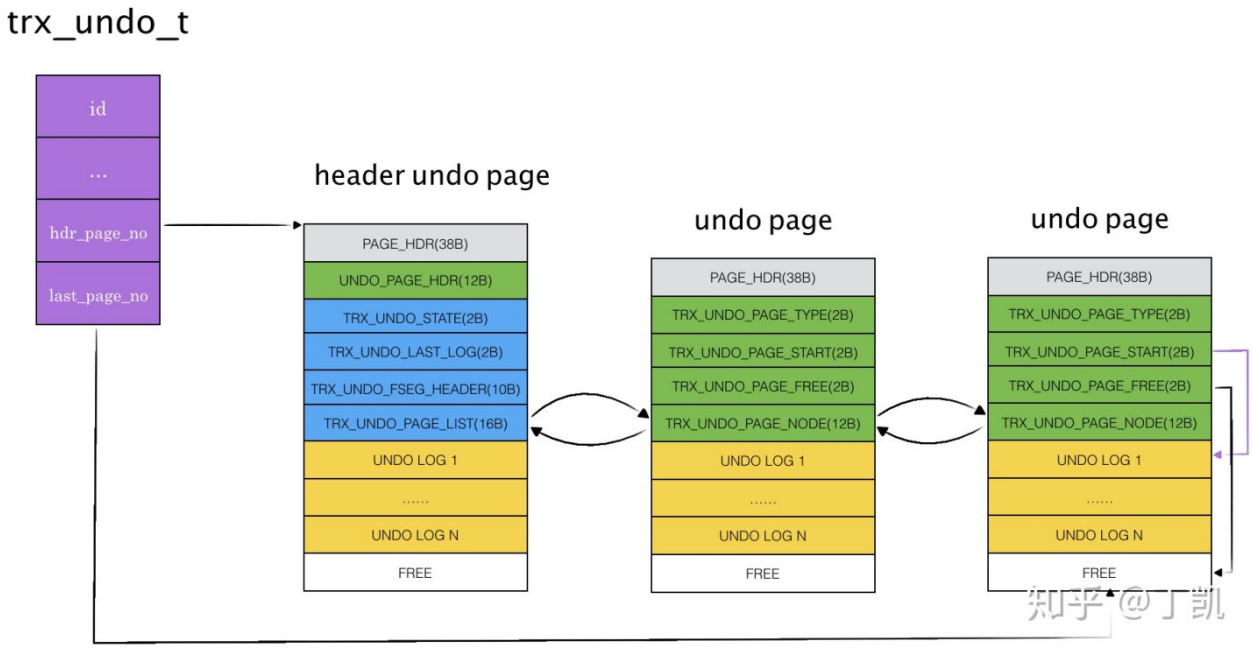


图 7-15　update undo log 格式

这里只需要知晓undo log表达的是一种反语句，即执行插入则undo log代表删除，执行主键上更新，undo log代表删除新值与插入旧值。

每个undo log以链表的形式被组织在不同undo page中，next字段即代表下一个undo log，而undo page又以链表的形式被组织。每个undo log记录了对应的事务id、记录表空间、偏移等基础信息，例如我们知道事务id，从前往后遍历链表找到所属事务 id 的 log (链表从头部插入，后插入的在前，回滚时从前往后查找即可)，即可回滚事务，当然这样查找效率比较低，更好的办法是每个事务记录一个回滚指针。



何时刷新回磁盘？

如果你仔细看了redo log的内容，你至少应该清楚修改后的数据页会被标记为脏页，等待后台线程回收，在这之前，该数据页会先被redo log所记录。

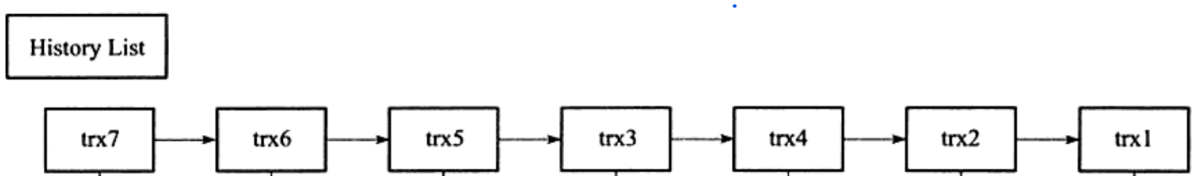
回到问题，undo log何时被刷新会磁盘？答案是随后台线程。每一条语句都会产生 undo log，undo log page 被当作脏页，就像普通脏页那样被后台线程定期刷新回磁盘。

这里有个点就是**undo log也需要redo log以保证其正确性。**

事务提交时

事务提交后，undo log不能立即被删除，这是因为数据库要提供MVVC多版本控制，一些事务线程会通过undo log来推断出上一个版本，InnoDB会将undo log放入删除链表(也叫历史链表)中(先提交的undo log总是在尾端)，等待purge线程真正的删除它们。

例如下图删除链表，trx1表示事务1产生的undo log，它是最先提交的，因此在尾端。



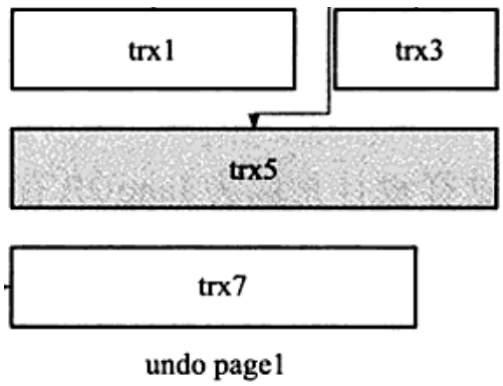
Purge操作

Purge周期的清理记录以及undo log，清理记录只需要检查删除标记位即可，清理undo log要稍微麻烦一点。

Purge线程必须保证一个undo log没有任何事务引用它才能删除这个日志，引用关系在undo page中对应的undo log中存在记录，因此可以通过查找undo page中的undo log

查看对应是否存在引用。历史链表中并不包含此信息，仅仅包含指向undo page中对应log的指针，这样做是为了节省内存。

如图所示，在undo page1中，其中trx5表示正在被其他事务引用：



完整的图如下图所示：

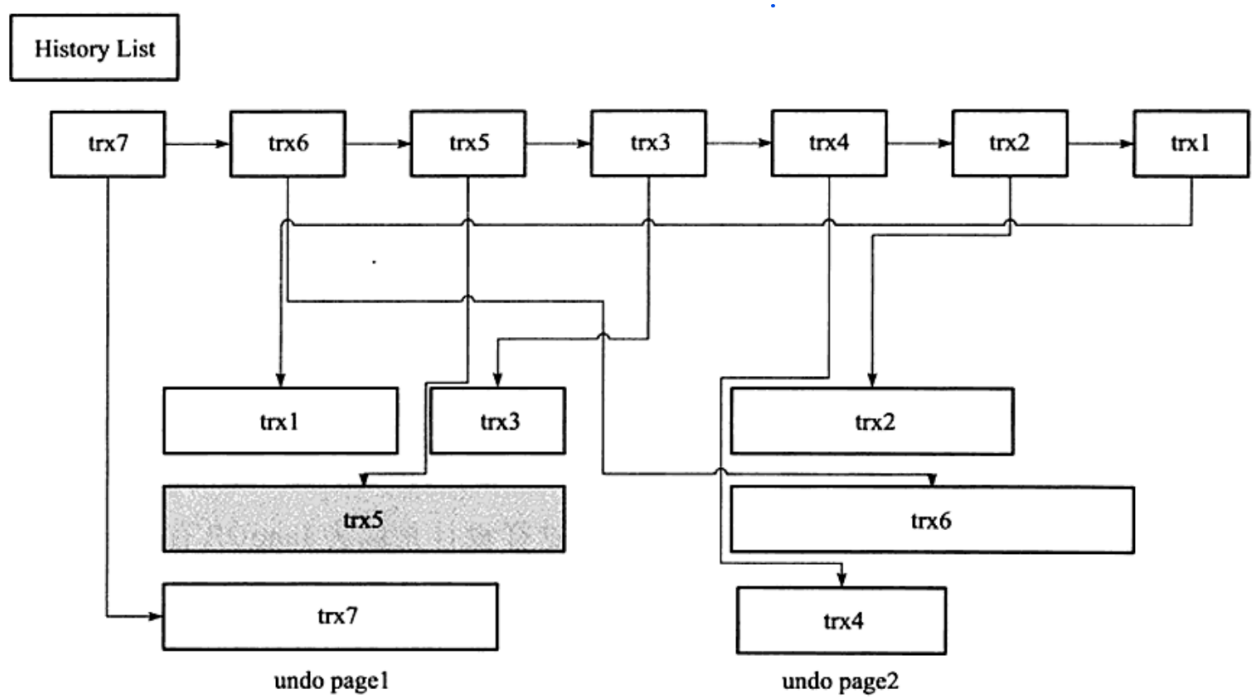


图 7-17　undo log 与 history 列表的关系

你可能会想：从尾到头遍历删除链表，查找对应undo page中的信息，没有引用则删除。

这样做没有任何问题，但效率是非常低下的，要明白删除链表中的顺序其实并不是实际事务log存储的顺序，真正的undo log在事务未提交前就被写入undo page，而在提交后才被写入链表，这就导致了链表中的顺序实际上并不是undo page中log存放的数据，由于undo log可能被刷新回磁盘，如果根据链表遍历可能会产生大量**随机读写**，效率低效，InnoDB采用先读链表，在顺序读page的方式实现。

具体的，先读链表尾部trx1，在undo page1中查找，发现没有引用则删除trx1；此后继续在undo page1中顺序查找，下一个是trx3，删除trx3；继续顺序查找，接下来是trx5，发现存在引用，则放弃查找；回到链表尾部，此时尾部为trx2，则到undo page2中顺序查找...

这样做会产生大量顺序读写，提高效率。

何时回滚？如何回滚？

当程序员手动调用回滚时此时会回滚；或当事务未提交而发生错误时，此时也会回滚以保证原子性。

回滚时，根据事务 id 和回滚指针查找 undo log page 中的 undo log lists，顺序的执行 undo log中的反语句以回滚信息。

MVCC

MVCC，即多版本并发控制，简单地说就是读取时如果该行被写锁锁定，则可以读取之前的版本，即快照读，我们的讨论基于 RR 隔离级别。

MYSQL 的 MVCC 也是依赖于 undolog 实现的，具体的，每个行记录都有一些隐式的字段，例如：

- db_trx_id，当前操作该记录的事务ID。
- db_roll_pointer，指向当前记录的 undolog page。
- 删除字段、隐式自增键等。

当开启快照读时，对于每个 SELECT 语句，INNODB 会生成一个 Read View（读视图），**这个视图用以判断其他事务对当前事务的可见性**，它通常有如下几个属性：

- trx_ids: 当前系统活跃 (即已开始但未提交) 事务版本号集合。
- low_limit_id: 创建当前 read view 时 “当前系统最大事务 ID + 1”，注意是系统最大事务而不是活跃事务中最大事务。
- up_limit_id: 创建当前 read view 时 “系统正处于活跃事务最小 ID。
- creator_trx_id: 创建当前read view的事务版本号。

举个例子我开启的事务 ID 是 8，则 Read View 可能是：`trx_ids: [4, 6, 8]`，`low_limit_id：25`，`up_limit_id：4`，`creator_trx_id：8`，这表示当前正在进行的事务 ID 有 [4, 6, 8]，整个系统中最大的事务 ID 是 25 - 1 = 24，这说明这个事务已经完成了，活跃事务的最小 ID 是 4。

知道这个有什么用呢？我们可以用来分析其他事务对当前事务是否可见，设当前事务 ID 为 MyId，正打算**读取的行上的记录** ID 是 OtherId，如果：

1. OtherId < up_limit_id。如果其他事务 ID 小于 up_limit_id，说明这个事务肯定提交了，提交的事务对所有事务可见。
2. OtherId == creator_trx_id。否则，如果这个事务 ID 就是自己 ID，则自己做的肯定可见，例如在一个事务中先更新再读取。
3. OtherId >= low_limit_id。如果其他事务 ID 大于等于系统最大事务 ID，说明这个事务是在快照创建之后生产的，即这个事务应该晚于当前事务，这种情况下肯定不可见。

4. 否则，遍历 trx_ids，判断这个事务是否存在于 trx_ids 中，如果存在说明事务未提交，未提交的事务肯定不可见，否则会产生脏读；如果不存在 trx_ids 中，说明这个事务已经 commit 了，并且肯定是先于当前事务的(事务产生快照时活跃列表没有它)，这种情况下可以认为可见，

知道了可见与不可见有啥用？

如果当前行标识的事务 ID 对当前事务不可见，则必须要通过 db_roll_pointer 找到 undolog page 进行回滚，找到上一个版本的记录，然后继续判断可不可见，这就是所谓的 MVVC。

举个例子，例如 A 先开启事务，ID 为 11，准备读取行记录 X，但还没有执行。

此时 B 开启事务，ID 为 12，修改行记录 X，B 顺利修改，这个时候 X 上的行记录隐式字段的事务ID被设置为 12，注意 B 事务未提交，仍在活跃。

A 终于开始准备读取 X 了，A 生成一个 ReadView，假设为：`trx_ids: [11, 12]`，`low_limit_id：13`，`up_limit_id：11`，`creator_trx_id：8`。

此时 A 发现 X 的事务 ID 是 12，11 <= 12 < 13，无法判断，则继续遍历 trx_ids，最终找到了 12，则 A 认为有其他事务正在锁定当前记录 X，则通过 X 回滚指针回滚读取上一个版本，递归做相同判断。

读已提交和可重复读的区别在于：**读以提交每次读取时总是会生成新的 ReadView，而可重复读只会在第一次读取时生成 ReadView。**

bing log

为什么需要bing log？

binlog是一个二进制格式的文件，用于记录用户对数据库**更新的SQL语句信息**，更改数据库表和更改内容的SQL语句都会记录到binlog里。

bing log是一种逻辑日志，记录着SQL操作。例如执行插入语句，bing log可能会记录`插入一条 XXX 记录;` bing log不会记录查询操作，事实上undo和redo也不会记录。

那么bing log有什么用呢？不是有了 redo log 和 undo log吗？

undo log记录的是反语句，主要为了实现回滚操作，事务提交后undo log就可能会被删除。redo log的作用是为了保证内存中的数据的持久性，当内存中的脏页被刷新，redo log也就失效，会被覆盖写。redo 和 undo 都有一个问题，即它们都是短暂存在的。

而bing log记录着所有执行操作，并持久性的存在于磁盘中。bing log 主要用于查看数据库的历史变更记录、增量备份(可以比较相对于上一次备份增加那些内容，仅备份这些增加的内容)、时间回溯(恢复到指定时间点的状态)或数据库复制。

bing log在站在MYSQL服务层上的，其负责整个MYSQL安全备份与恢复，针对于所有的引擎。而redo和undo仅仅知识InnoDB提供的、针对于事务的日志，是完全不同的概念。

事务二段式提交(内部)

MYSQL默认开启bing log，也建议开启bing log，不过当开启bing log后，在INNODB引擎下就会存在一些问题，即**bing log和redo log不一致的问题**。

举个简单的例子，在原来的状态中，事务提交后redo log被刷新回磁盘，此时bing log还未写入磁盘，数据库突然宕机，恢复时，由于redo log写入，数据不会有任何问题，但原来的bing log还未被写入！此时数据库状态与bing log不一致，我们违反了数据库的一致性！

必须在bing log也被正确写回后才算事务提交成功。bing log写回失败必须要做某些处理！

二段式提交就是为了解决redo log和bing log写回不一致的问题。

在二段式提交中，分为两个阶段：

1. prepare过程
2. commit过程

当事务提交时，会立即陷入第一个阶段，即prepare阶段。

prepare过程：

- 设置该事务对应undo state=TRX_UNDO_PREPARED，表示正式进入prepare过程。
- 将redo日志刷新回磁盘，修改内存数据页。

而后会陷入第二阶段。

commit过程：

- 将事务产生的binlog写回磁盘，当这个动作被完成时，事务被确认已提交，即使后面的动作未完成。
- 将undo写入删除链表中，等待purge删除。

考虑问题，当redo log写回磁盘失败时，此时数据仍未变脏，事务ACID特性未被破坏，该事务执行失败，回到最初的样子；当bing log写回磁盘失败时，数据页已变脏，必须借助undo log回滚事务，当undo log丢失时，根据redo log重建undo log。

这样，我们便解决了一系列的问题，事务的ACD特性(本文章为设计隔离性)得到保证。

深入数据一致性：Double Write 技术

如果你学过文件系统，你就会知道上述三大 log 其实并不足以保证一致性，在文件系统中，一次文件修改不仅需要修改文件内容，还需要修改 inode 中的文件元数据，而修改元数据和修改文件内容也必须要是原子的。

在 Mysql 中也会存在写入不完整的问题，由于内存页为 16k，而磁盘页通常是 4k，磁盘能保证的原子写入只有一个扇区大小 512b，这就会导致一些数据不一致问题，并且是 redo log 无法解决的问题。

我们提到，一个内存页存在许多元数据，例如 LSN 就是其中之一，我们以 LSN 举个例子，例如当我们将内存页刷回的时候，一个内存页会被分成 4 部分刷回物理磁盘，假设此时带有 LSN 那部分页成功刷回，而其他页因为因为突然宕机而丢失了，这个时候会发生什么情况呢？

Innodb 开始进行数据恢复，但是他发现这个页的 LSN 和 redo log 是一致的，于是确信这个页是正常的...

当然你可能会想，那我在页的 tail 设置个合法位，检测一下合不合法不就好了？

的确可以，但是页不合法该怎么办呢？重做是不行的，因为重做日志并没有包含页的元数据，如果页的元数据丢失了，重做日志是无法解决问题的。

> Mysql 中页的尾部有个 File Trialer 字段，用以检测是否合法

所以针对内存页的写入，需要引入双写技术，简单来说就是先搞个备份。

在刷新脏页时，需要经历如下步骤：

1. 将脏页 memcpy 追加到 Double Write Buffer 中，Double Write Buffer 总大小通常是 2MB，即 16 个内存页。
2. Double Write Buffer 会以 1MB 为单位的顺序写入到物理磁盘共享表空间中，一旦写入完成，立即调用 fsnyc 离散的同步这 1MB 数据到磁盘中。

这样一旦数据库恢复时检测到物理磁盘中某个页不合法时，可以在共享表空间中查看页的副本，如果页的副本是合法的，直接覆盖即可。当然 Double Write 的写入也可能会出现问题，如果页的副本是不合法的，则会丢弃页的副本，**但是这里磁盘页肯定是合法的，因为 DoubleWrite 数据出问题，则数据肯定未能同步磁盘，磁盘中的页未被修改，仍然是之前的旧页**，这个时候运用 redo log 恢复即可。

前文也说到，重做日志块被设计成 512 kb，从而保证重做日志块的写入肯定是原子的，所以无需担心重做日志的合法性。