1. redo日志
2. 概述

当一个事务提交后，只在内存的Buffer Pool中修改了页面，如果此时发生故障，导致内存中的数据都失效了，那么这个已提交的事务就丢失了，这是我们无法忍受的。一个简单的做法就是在事务提交完成之前，将该事务修改的所有页面都刷新到磁盘中，但是这样作非常浪费资源，因为可能修改的只是一个页中的某个字节，同时随机I/O刷新也非常地慢，因为一个事务可能对应着多个语句，一个语句也可能修改了多个页面，这会导致随机I/O的发生。

redo日志就是为了解决这样的问题，我们的目的是让已经提交了的事务对数据库中的数据所做的修改能够永久生效，即时系统崩溃，在重启后也能够将这种修改恢复过来。因此只需要在每次提交事务时将该事务在内存中修改过的页面的内容记录到redo日志中即可，满足了持久性的要求。这种日志称为重做日志，因为用于系统崩溃重启更新数据库。

1. redo日志的优势

它的优点是占用空间非常小，仅仅存储表空间ID，页号，偏移量以及想要更新的值；同时redo日志还是顺序写入磁盘的，在执行事务时，每执行一条语句，就可能产生若干条redo日志，按照产生的顺序写入磁盘，即顺序I/O。

1. redo日志格式
2. 通用格式

InnoDB中有多种类型的redo日志，绝大部分的redorizhi 都有着通用的结构，存储着日志的类型，表空间ID，页号以及这条redo日志的具体内容。

1. 简单的redo日志

简单的日志类型只需要记录在某个页面的某个偏移量处修改了几个字节的值，具体修改后的内容是什么，这种简单的redo日志称为物理日志，并根据在页面中写入数据的多少划分了多种类型。

1. 复杂的redo日志

有时，执行一条语句时会修改很多页面，包括系统数据页面和用户数据页面。如一个INSERT语句，表中有多少索引就可能更新多少棵B+树，针对某棵B+树，可能更新叶子节点页面，也可能更新内节点页面。

1. Mini-Transaction
2. 以组的形式写入redo日志

在执行语句过程中产生的redo日志，被划分为了若干个不可分割的组：如向聚簇索引对应的B+树的页面插入一条记录时产生的redo日志是一组的，不可分割；向某个二级索引对应的B+树的页面插入一条记录产生的redo日志也是不可分割的。因为在插入一条记录的时候，定位到相应的叶子节点页中，这个页可能有足够的空间供插入（乐观插入），也可能没有足够的空间（悲观插入），对于悲观插入来说，需要新申请数据页，改动统计信息等等，这些操作必须是原子性的，不可断开，因此以组的形式来记录redo日志。

如何将这些redo日志划分到一个组里面呢？在改组的最后一条redo日志后面加上一条特殊类型的redo日志，只有一个type字段代表一组日志的结尾。在系统崩溃重启解析到该类型的日志时，就知道解析到了一组完整的redo日志，否则直接放弃之前解析到的redo日志。有的只生成一条redo日志的操作，直接使用redo日志中1个字节的type字段，用第一个bit代表是否是单一的日志，用后面的7个bit代表redo日志的类型。

1. Mini-transaction

对底层页面进行一次原子访问的过程称为一个Mini-transaction（MTR）。一个事务可以包含多个语句，一个语句又包含多个MTR，每个MTR可以包含一组redo日志。

1. redo日志的写入过程
2. 存储redo日志的页

为了管理redo日志，InnoDB将通过MTR生成的redo日志放在了大小为512个字节的页中，该页的结构是：log block header , log block body, log block trailer。body存储着真正的redo日志，其他的存储的都是管理信息。

header中属性解释：

no: 每个页的编号；

data\_len: 表示页中已经使用了多少字节，初始值为12字节，填满时为512；

checkpoint\_no: checkpoint的序号；

1. redo日志缓冲区

类似于Buffer Pool，写入redo日志时也不能直接写到磁盘中，在服务器启动时就申请了一篇称为redo log buffer的连续内存空间，即log buffer, 划分为多个连续的redo日志页，默认是16MB。

1. redo日志写入log buffer

向log buffer中写入redo日志是顺序写入的，通过一个全局变量buf\_free来指明后续写入的redo日志应该写入到log buffer中的哪个位置。并不是说每生成一条redo日志就将其插入到log buffer中，而是将每个MTR运行过程中产生的日志先暂存到一个地方，MTR结束后，将过程中产生的一组redo日志全部复制到log buffer中。不同事务可能是并发执行的，因此不同事务的MTR对应的redo日志可能是交替写入log buffer中的。

三， redo日志文件

1. redo日志刷盘时机

存在于log buffer中的redo日志在一些情况下会被刷新到磁盘中。

1. log buffer空间不足

到达容量的50%时，就会刷新到磁盘中。

1. 事务提交时

事务提交时，必须熟悉页面修改对应的redo日志到磁盘中，保持持久性。

1. 正常关闭服务器时
2. 做checkpoint时
3. redo日志文件组

MySQL的数据目录下默认有两个ib\_logfile0和ib\_logfile1文件，log buffer中的日志默认就是刷新到这两个磁盘文件中。当然，可以在启动选项中修改指定的磁盘文件。日志刷新首先刷新到第一个文件中，如果写满了，就选择后一个文件写入，如果最后一个写满了，重新回到第一个文件进行写入。这问题不就来了吗？回到第一个文件写入不就覆盖了之前的redo日志了，这里提出了checkpoint来解决这个问题。

1. redo日志文件格式

将log buffer中的redo日志刷新到磁盘的本质就是将redo日志页的镜像写入到磁盘文件中，所以redo日志磁盘文件其实也是由若干个512字节大小的页组成。在redo日志文件组中，每个文件大小都一样，格式也一样，都是由前4个页存储管理信息，后面的页存储redo日志页。

1. log sequence number(lsn)
2. 概述

InnoDB中涉及了lsn的全局变量，用来记录当前已经写入到log buffer的redo日志量，初始是8704。在统计lsn的增长量时，按照实际写入的日志量加上占用的log block header和log block trailer来计算的，即redo日志写入log buffer是按照一个MTR写入的，可能会跨用多个页，占用的header和 trailer都要作为lsn的增长量。每一组由MTR产生的redo日志都有一个唯一的lsn值对应，越小说明产生的越早。

1. flushed\_to\_disk\_lsn

同样是去全局变量，flushed\_to\_disk\_lsn用来标记当前log buffer中已经有哪些日志刷新到磁盘中了，lsn表示的是当前系统中写入到log buffer但是还没有刷新到磁盘中的redo日志，如果两者相同，说明log buffer中的所有redo日志都已经刷新到了磁盘中。

1. flush链表中的lsn

在MTR结束后，需要把在MTR执行过程中修改过的页面加入到Buffer Pool的flush链表中，当第一次修改某个已经加载到Buffer Pool中的页面时，就会把这个页面对应的控制块插入到flush链表的头部，之后再修改该页面时，由于它已经在flush链表中，就不再次进行插入了。即flush链表中存储的都是脏页，即没有更新到数据库的修改页，脏页是按照页面的第一次修改时间排序的，又由于每次插入都是头插法，因此前面的脏页修改时间晚，后面的早。

控制块中会记录两个关于页面何时修改的属性：

oldest\_modification: 第一次修改Buffer Pool中的某个缓冲页时，就将修改该页面的MTR开始时对应的lsn值写入这个属性；

newest\_modification: 每修改一次页面，都会将修改该页面的MTR结束时对应的lsn值写入这个属性中。

1. checkpoint
2. 概述

redo日志文件组的容量是有限的，我们需要循环使用redo日志文件组中的文件，这就需要判断某些redo日志占用的磁盘空间是否可以覆盖了，即它对应的脏页是否已经刷新到了磁盘中。全局变量checkpoint\_lsn，用来表示当前系统中可以被覆盖的redo日志总量是多少，初始值是8704，如果某个页被刷新到了磁盘上，它对应的redo日志就没用了，可以被覆盖，则checkpoint\_lsn+1，这个过程称为执行一次checkpoint操作。

（二）checkpoint操作的步骤

1. 计算当前系统中可以被覆盖的redo日志对应的lsn值最大是多少？

只要计算出当前系统中最早修改的脏页对应的oldest\_modification值，那么凡是系统在lsn值小于该节点的oldest\_modification值时产生的redo日志都可以被覆盖掉，将该脏页的oldest\_modification赋值给checkpoint\_lsn。即小于checkpoint\_lsn的空间都可以被覆盖掉。

1. 将checkpoint\_lsn与对应的redo日志文件组偏移量以及此次的checkpoint的编号写到日志文件的管理信息中。
2. 刷脏页

一般都是后台线程对LRU链表和flush链表进行刷脏操作，如果lsn值增长过快，就需要用户线程从flush链表中将那些最早修改的脏页同步刷新到磁盘中，这样就可以执行checkpoint了。