InnoDB 引擎底层事务的原理

事务应该具有 4 个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为 ACID 特性。

原子性指的是一个事务中的操作要么全部成功，要么全部失败。

一致性指的是数据库总是从一个一致性的状态转换到另外一个一致性的状态。比如 A 转账给 B100 块钱，假设中间 sql 执行过程中系统崩溃 A 也不会损失 100 块，因为事务没有提交，修改也就不会保存到数据库。

隔离性指的是一个事务的修改在最终提交前，对其他事务是不可见的。

持久性指的是一旦事务提交，所做的修改就会永久保存到数据库中。

总的来说，MySQL 中事务的原子性是通过 undo log 来实现的，事务的持久性性是通过 redo log 来实现的，事务的隔离性是通过读写锁+MVCC 来实现的。

事务的一致性通过原子性、隔离性、持久性来保证。也就是说 ACID 四大特性之中，C(一致性)是目的，A(原子性)、I(隔离性)、D(持久性)是手段，是为了保证一致性，数据库提供的手段。数据库必须要实现 AID 三大特性，才有可能实现一致性。同时一致性也需要应用程序的支持，应用程序在事务里故意写出违反约束的代码，一致性还是无法保证的，例如，转账代码里从 A 账户扣钱而不给 B 账户加钱，那一致性还是无法保证。

在事务的具体实现机制上，MySQL 采用的是 WAL（Write-ahead logging，预写式日志）机制来实现的。这也是是当今的主流方案。

在使用 WAL 的系统中，所有的修改都先被写入到日志中，然后再被应用到系统中。通常包含 redo 和 undo 两部分信息。

为什么需要使用 WAL，然后包含 redo 和 undo 信息呢？举个例子，如果一个系统直接将变更应用到系统状态中，那么在机器掉电重启之后系统需要知道操作是成功了，还是只有部分成功或者是失败了（为了恢复状态）。如果使用了 WAL，那么在重启之后系统可以通过比较日志和系统状态来决定是继续完成操作还是撤销操作。

redo log 称为重做日志，每当有操作时，在数据变更之前将操作写入 redo log，这样当发生掉电之类的情况时系统可以在重启后继续操作。

undo log 称为撤销日志，当一些变更执行到一半无法完成时，可以根据撤销日志恢复到变更之间的状态。

前面说过，MySQL 中用 redo log 来在系统 Crash 重启之类的情况时修复数据（事务的持久性），而 undo log 来保证事务的原子性。

tips：Commit Logging 和 Shadow Paging

事务的日志类型的实现除了 WAL（Write-ahead logging，预写式日志）外，还有“Commit Logging”（提交日志），这种方式只有在日志记录全部都安全落盘，数据库在日志中看到代表事务成功提交的“提交记录”（Commit Record）后，才会根据日志上的信息对真正的数据进行修改，修改完成后，再在日志中加入一

条“结束记录”（End Record）表示事务已完成持久化。两者的区别是，WAL 允许在事务提交之前，提前写入变动数据，而 Commit Logging 则不行；WAL 中有 undo 日志，Commit Logging 没有。阿里的 OceanBase 则是使用的 Commit Logging 来实现事务。

实现事务的原子性和持久性除日志外，还有另外一种称为“Shadow Paging”（有中文资料翻译为“影子分页”）的事务实现机制，常用的轻量级数据库 SQLite Version 3 采用的事务机制就是 Shadow Paging。

Shadow Paging 的大体思路是对数据的变动会写到硬盘的数据中，但并不是直接就地修改原先的数据，而是先将数据复制一份副本，保留原数据，修改副本数据。在事务过程中，被修改的数据会同时存在两份，一份是修改前的数据，一份是修改后的数据，这也是“影子”（Shadow）这个名字的由来。当事务成功提交，所有数据的修改都成功持久化之后，最后一步是去修改数据的引用指针，将引用从原数据改为新复制出来修改后的副本，最后的“修改指针”这个操作将被认为是原子操作，现代磁盘的写操作可以认为在硬件上保证了不会出现“改了半个值”的现象。所以 Shadow Paging 也可以保证原子性和持久性。Shadow Paging 实现事务要比 Commit Logging 更加简单，但涉及隔离性与并发锁时，Shadow Paging 实现的事务并发能力就相对有限，因此在高性能的数据库中应用不多。

redo 日志

redo 日志的作用

InnoDB 存储引擎是以页为单位来管理存储空间的，我们进行的增删改查操作其实本质上都是在访问页面（包括读页面、写页面、创建新页面等操作）。在 Buffer Pool 的时候说过，在真正访问页面之前，需要把在磁盘上的页缓存到内存中的 Buffer Pool 之后才可以访问。但是在事务的时候又强调过一个称之为持久性的特性，就是说对于一个已经提交的事务，在事务提交后即使系统发生了崩溃，这个事务对数据库中所做的更改也不能丢失。

如果我们只在内存的 Buffer Pool 中修改了页面，假设在事务提交后突然发生了某个故障，导致内存中的数据都失效了，那么这个已经提交了的事务对数据库中所做的更改也就跟着丢失了，这是我们所不能忍受的。那么如何保证这个持久性呢？一个很简单的做法就是在事务提交完成之前把该事务所修改的所有页面都刷新到磁盘，但是这个简单粗暴的做法有些问题：

1、刷新一个完整的数据页太浪费了

有时候我们仅仅修改了某个页面中的一个字节，但是我们知道在 InnoDB 中是以页为单位来进行磁盘 IO 的，也就是说我们在该事务提交时不得不将一个完整的页面从内存中刷新到磁盘，我们又知道一个页面默认是 16KB 大小，只修改一个字节就要刷新 16KB 的数据到磁盘上显然是太浪费了。

2、随机 I0 刷起来比较慢

一个事务可能包含很多语句，即使是一条语句也可能修改许多页面，该事务修改的这些页面可能并不相邻，这就意味着在将某个事务修改的 Buffer Pool 中的页面刷新到磁盘时，需要进行很多的随机 IO，随机 IO 比顺序 IO 要慢，尤其对于传统的机械硬盘来说。

怎么办呢？我们只是想让已经提交了的事务对数据库中数据所做的修改永久生效，即使后来系统崩溃，在重启后也能把这种修改恢复出来。所以我们其实没有必要在每次事务提交时就把该事务在内存中修改过的全部页面刷新到磁盘，只需要把修改了哪些东西记录一下就好，比方说某个事务将系统表空间中的第 100 号页面中偏移量为 1000 处的那个字节的值 1 改成 2 我们只需要记录一下：

将第 0 号表空间的 100 号页面的偏移量为 1000 处的值更新为 2。

这样我们在事务提交时，把上述内容刷新到磁盘中，即使之后系统崩溃了，重启之后只要按照上述内容所记录的步骤重新更新一下数据页，那么该事务对数据库中所做的修改又可以被恢复出来，也就意味着满足持久性的要求。因为在系统崩溃重启时需要按照上述内容所记录的步骤重新更新数据页，所以上述内容也被称之为重做日志，英文名为 redo log，也可以称之为 redo 日志。与在事务提交时将所有修改过的内存中的页面刷新到磁盘中相比，只将该事务执行过程中产生的 redo 日志刷新到磁盘的好处如下：

1、redo 日志占用的空间非常小

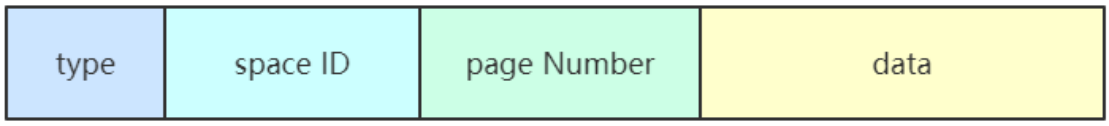
存储表空间 ID、页号、偏移量以及需要更新的值所需的存储空间是很小的。

2、redo 日志是顺序写入磁盘的

在执行事务的过程中，每执行一条语句，就可能产生若干条 redo 日志，这些日志是按照产生的顺序写入磁盘的，也就是使用顺序 IO。

redo 日志格式

通过上边的内容我们知道，redo 日志本质上只是记录了一下事务对数据库做了哪些修改。 InnoDB 们针对事务对数据库的不同修改场景定义了多种类型的 redo 日志，但是绝大部分类型的 redo 日志都有下边这种通用的结构：



各个部分的详细释义如下：

type：该条 redo 日志的类型，redo 日志设计大约有 53 种不同的类型日志。

space ID：表空间 ID。

page number：页号。

data：该条 redo 日志的具体内容。

简单的 redo 日志类型

我们用一个简单的例子来说明最基本的 redo 日志类型。我们前边介绍 InnoDB 的记录行格式的时候说过，如果我们没有为某个表显式的定义主键，并且表中也没有定义 Unique 键，那么 InnoDB 会自动的为表添加一个称之为 row_id 的隐藏列作为主键。为这个 row_id 隐藏列赋值的方式如下：

服务器会在内存中维护一个全局变量，每当向某个包含隐藏的 row_id 列的表中插入一条记录时，就会把该变量的值当作新记录的 row_id 列的值，并且把该变量自增 1。

每当这个变量的值为 256 的倍数时，就会将该变量的值刷新到系统表空间的页号为 7 的页面中一个称之为 Max Row ID 的属性处。

当系统启动时，会将上边提到的 Max Row ID 属性加载到内存中，将该值加上 256 之后赋值给我们前边提到的全局变量。

这个 Max Row ID 属性占用的存储空间是 8 个字节，当某个事务向某个包含 row_id 隐藏列的表插入一条记录，并且为该记录分配的 row_id 值为 256 的倍数时，就会向系统表空间页号为 7 的页面的相应偏移量处写入 8 个字节的值。但是我们要知道，这个写入实际上是在 Buffer Pool 中完成的，我们需要为这个页面的修改记录一条 redo 日志，以便在系统崩溃后能将已经提交的该事务对该页面所做的修改恢复出来。这种情况下对页面的修改是极其简单的，redo 日志中只需要记录一下在某个页面的某个偏移量处修改了几个字节的值，具体被修改的内容是啥就好了，InnoDB 把这种极其简单的 redo 日志称之为物理日志，并且根据在页面中写入数据的多少划分了几种不同的 redo 日志类型：

MLOG_1BYTE（type 字段对应的十进制数字为 1）：表示在页面的某个偏移量处写入 1 个字节的 redo 日志类型。

MLOG_2BYTE（type 字段对应的十进制数字为 2）：表示在页面的某个偏移量处写入 2 个字节的 redo 日志类型。

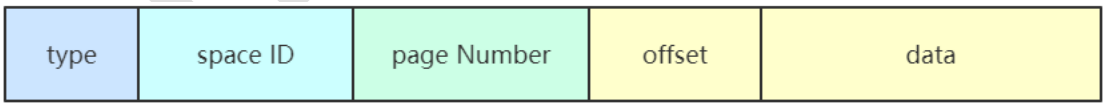
MLOG_4BYTE（type 字段对应的十进制数字为 4）：表示在页面的某个偏移量处写入 4 个字节的 redo 日志类型。

MLOG_8BYTE（type 字段对应的十进制数字为 8）：表示在页面的某个偏移量处写入 8 个字节的 redo 日志类型。

MLOG_WRITE_STRING（type 字段对应的十进制数字为 30）：表示在页面的某个偏移量处写入一串数据。

我们上边提到的 Max Row ID 属性实际占用 8 个字节的存储空间，所以在修改页面中的该属性时，会记录一条类型为 MLOG_8BYTE 的 redo 日志，MLOG_8BYTE 的 redo 日志结构如下所示：

offset 代表在页面中的偏移量。



其余 MLOG_1BYTE、MLOG_2BYTE、MLOG_4BYTE 类型的 redo 日志结构和 MLOG_8BYTE 的类似，只不过具体数据中包含对应个字节的数据罢了。MLOG_WRITE_STRING 类型的 redo 日志表示写入一串数据，但是因为不能确定写入的具体数据占用多少字节，所以需要在日志结构中还会多一个 len 字段。

复杂一些的 redo 日志类型

有时候执行一条语句会修改非常多的页面，包括系统数据页面和用户数据页面（用户数据指的就是聚簇索引和二级索引对应的 B+树）。以一条 INSERT 语句

为例，它除了要向 B+树的页面中插入数据，也可能更新系统数据 Max Row ID 的值，不过对于我们用户来说，平时更关心的是语句对 B+树所做更新：

表中包含多少个索引，一条 INSERT 语句就可能更新多少棵 B+树。

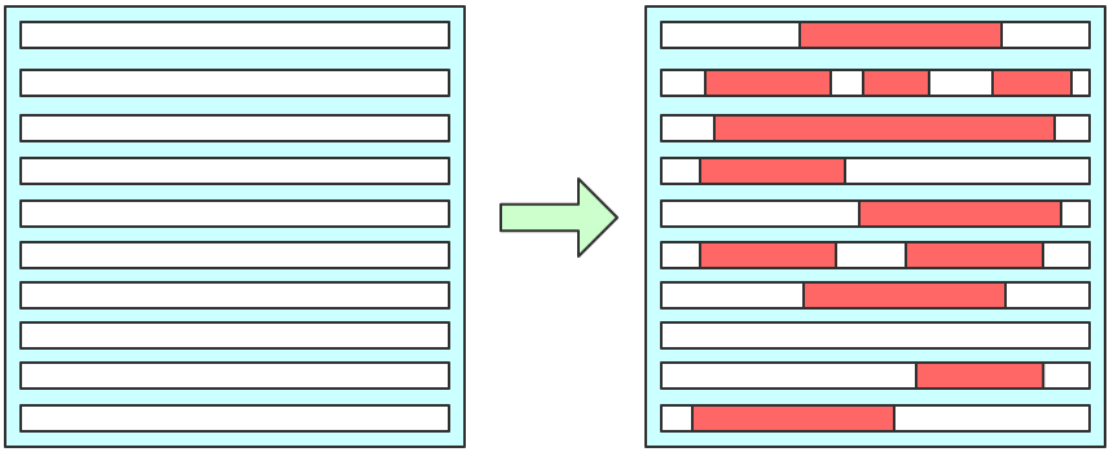
针对某一棵 B+树来说，既可能更新叶子节点页面，也可能更新非叶子节点页面，也可能创建新的页面（在该记录插入的叶子节点的剩余空间比较少，不足以存放该记录时，会进行页面的分裂，在非叶子节点页面中添加目录项记录）。

在语句执行过程中，INSERT 语句对所有页面的修改都得保存到 redo 日志中去。实现起来是非常麻烦的，比方说将记录插入到聚簇索引中时，如果定位到的叶子节点的剩余空间足够存储该记录时，那么只更新该叶子节点页面就好，那么只记录一条 MLOG_WRITE_STRING 类型的 redo 日志，表明在页面的某个偏移量处增加了哪些数据就好了么？

别忘了一个数据页中除了存储实际的记录之后，还有什么 File Header、Page Header、Page Directory 等等部分，所以每往叶子节点代表的数据页里插入一条记录时，还有其他很多地方会跟着更新，比如说：

可能更新 Page Directory 中的槽信息、Page Header 中的各种页面统计信息，比如槽数量可能会更改，还未使用的空间最小地址可能会更改，本页面中的记录数量可能会更改，各种信息都可能会被修改，同时数据页里的记录是按照索引列从小到大的顺序组成一个单向链表的，每插入一条记录，还需要更新上一条记录的记录头信息中的 next_record 属性来维护这个单向链表。

画一个简易的示意图就像是这样：



其实说到底，把一条记录插入到一个页面时需要更改的地方非常多。这时我们如果使用上边介绍的简单的物理 redo 日志来记录这些修改时，可以有两种解决方案：

方案一：在每个修改的地方都记录一条 redo 日志。

也就是如上图所示，有多少个加粗的块，就写多少条物理 redo 日志。这样子记录 redo 日志的缺点是显而易见的，因为被修改的地方是在太多了，可能记录的 redo 日志占用的空间都比整个页面占用的空间都多了。

方案二：将整个页面的第一个被修改的字节到最后一个修改的字节之间所有的数据当成是一条物理 redo 日志中的具体数据。

从图中也可以看出来，第一个被修改的字节到最后一个修改的字节之间仍然有许多没有修改过的数据，我们把这些没有修改的数据也加入到 redo 日志中去依然很浪费。

正因为上述两种使用物理 redo 日志的方式来记录某个页面中做了哪些修改比较浪费，InnoDB 中就有非常多的 redo 日志类型来做记录。

这些类型的 redo 日志既包含物理层面的意思，也包含逻辑层面的意思，具体指：

物理层面看，这些日志都指明了对哪个表空间的哪个页进行了修改。

逻辑层面看，在系统崩溃重启时，并不能直接根据这些日志里的记载，将页面内的某个偏移量处恢复成某个数据，而是需要调用一些事先准备好的函数，执行完这些函数后才可以将页面恢复成系统崩溃前的样子。

简单来说，一个 redo 日志类型而只是把在本页面中变动（比如插入、修改）一条记录所有必备的要素记了下来，之后系统崩溃重启时，服务器会调用相关向某个页面变动（比如插入、修改）一条记录的那个函数，而 redo 日志中的那些数据就可以被当成是调用这个函数所需的参数，在调用完该函数后，页面中的相关值也就都被恢复到系统崩溃前的样子了。这就是所谓的逻辑日志的意思。

当然，如果不是为了写一个解析 redo 日志的工具或者自己开发一套 redo 日志系统的话，那就不需要去研究 InnoDB 中的 redo 日志具体格式。

大家只要记住：redo 日志会把事务在执行过程中对数据库所做的所有修改都记录下来，在之后系统崩溃重启后可以把事务所做的任何修改都恢复出来。

redo 日志的写入过程

redo log block 和日志缓冲区

InnoDB 为了更好的进行系统崩溃恢复，把 redo 日志都放在了大小为 512 字节的块（block）中。

我们前边说过，为了解决磁盘速度过慢的问题而引入了 Buffer Pool。同理，写入 redo 日志时也不能直接直接写到磁盘上，实际上在服务器启动时就向操作系统申请了一大片称之为 redo log buffer 的连续内存空间，翻译成中文就是 redo 日志缓冲区，我们也可以简称为 log buffer。这片内存空间被划分成若干个连续的 redo log block，我们可以通过启动参数 innodb_log_buffer_size 来指定 log buffer 的大小，该启动参数的默认值为 16MB。

向 log buffer 中写入 redo 日志的过程是顺序的，也就是先往前边的 block 中写，当该 block 的空闲空间用完之后再往下一个 block 中写。

redo 日志刷盘时机

我们前边说 redo 日志在内存中有个 log buffer，可是这些日志总在内存里呆着也不是个办法，在一些情况下它们会被刷新到磁盘里，比如：

1、log buffer 空间不足时，log buffer 的大小是有限的（通过系统变量 innodb_log_buffer_size 指定），如果不停的往这个有限大小的 log buffer 里塞入

日志，很快它就会被填满。InnoDB 认为如果当前写入 log buffer 的 redo 日志量已经占满了 log buffer 总容量的大约一半左右，就需要把这些日志刷新到磁盘上。

2、事务提交时，我们前边说过之所以使用 redo 日志主要是因为它占用的空间少，还是顺序写，在事务提交时可以不把修改过的 Buffer Pool 页面刷新到磁盘，但是为了保证持久性，必须要把修改这些页面对应的 redo 日志刷新到磁盘。

3、后台有一个线程，大约每秒都会刷新一次 log buffer 中的 redo 日志到磁盘。

4、正常关闭服务器时等等。

redo 日志文件组

MySQL 的数据目录（使用 SHOW VARIABLES LIKE 'datadir'查看）下默认有两个名为 ib_logfile0 和 ib_logfile1 的文件，log buffer 中的日志默认情况下就是刷新到这两个磁盘文件中。如果我们对默认的 redo 日志文件不满意，可以通过下边几个启动参数来调节：

innodb_log_group_home_dir，该参数指定了 redo 日志文件所在的目录，默认值就是当前的数据目录。

innodb_log_file_size，

该参数指定了每个 redo 日志文件的大小，默认值为 48MB，

innodb_log_files_in_group，该参数指定 redo 日志文件的个数，默认值为 2，最大值为 100。

所以磁盘上的 redo 日志文件可以不只一个，而是以一个日志文件组的形式出现的。这些文件以 ib_logfile[数字]（数字可以是 0、1、2...）的形式进行命名。在将 redo 日志写入日志文件组时，是从 ib_logfile0 开始写，如果 ib_logfile0 写满了，就接着 ib_logfile1 写，同理，ib_logfile1 写满了就去写 ib_logfile2，依此类推。如果写到最后一个文件该咋办？那就重新转到 ib_logfile0 继续写。

既然 Redo log 文件是循环写入的，在覆盖写之前，总是要保证对应的脏页已经刷到了磁盘。在非常大的负载下，为避免错误的覆盖，InnoDB 会强制的 flush 脏页。

redo 日志文件格式

我们前边说过 log buffer 本质上是一片连续的内存空间，被划分成了若干个 512 字节大小的 block。将 log buffer 中的 redo 日志刷新到磁盘的本质就是把 block 的镜像写入日志文件中，所以 redo 日志文件其实也是由若干个 512 字节大小的 block 组成。

redo 日志文件组中的每个文件大小都一样，格式也一样，都是由两部分组成：前 2048 个字节，也就是前 4 个 block 是用来存储一些管理信息的。

从第 2048 字节往后是用来存储 log buffer 中的 block 镜像的。

Log Sequence Number

自系统开始运行，就不断的在修改页面，也就意味着会不断的生成 redo 日志。redo 日志的量在不断的递增，就像人的年龄一样，自打出生起就不断递增，永远不可能缩减了。

InnoDB 为记录已经写入的 redo 日志量，设计了一个称之为 Log Sequence Number 的全局变量，翻译过来就是：日志序列号，简称 LSN。规定初始的 lsn 值为 8704（也就是一条 redo 日志也没写入时，LSN 的值为 8704）。

redo 日志都有一个唯一的 LSN 值与其对应，LSN 值越小，说明 redo 日志产生的越早。

flushed_to_disk_lsn

redo 日志是首先写到 log buffer 中，之后才会被刷新到磁盘上的 redo 日志文件。InnoDB 中有一个称之为 buf_next_to_write 的全局变量，标记当前 log buffer 中已经有哪些日志被刷新到磁盘中了。

我们前边说 lsn 是表示当前系统中写入的 redo 日志量，这包括了写到 log buffer 而没有刷新到磁盘的日志，相应的，InnoDB 也有一个表示刷新到磁盘中的 redo 日志量的全局变量，称之为 flushed_to_disk_lsn。系统第一次启动时，该变量的值和初始的 lsn 值是相同的，都是 8704。随着系统的运行，redo 日志被不断写入 log buffer，但是并不会立即刷新到磁盘，lsn 的值就和 flushed_to_disk_lsn 的值拉开了差距。我们演示一下：

系统第一次启动后，向 log buffer 中写入了 mtr_1、mtr_2、mtr_3 这三个 redo 日志，假设这三个 mtr 开始和结束时对应的 lsn 值分别是：

mtr_1：8716 ～ 8916

mtr_2：8916 ～ 9948

mtr_3：9948 ～ 10000

此时的 lsn 已经增长到了 10000，但是由于没有刷新操作，所以此时 flushed_to_disk_lsn 的值仍为 8704。

随后进行将 log buffer 中的 block 刷新到 redo 日志文件的操作，假设将 mtr_1 和 mtr_2 的日志刷新到磁盘，那么 flushed_to_disk_lsn 就应该增长 mtr_1 和 mtr_2 写入的日志量，所以 flushed_to_disk_lsn 的值增长到了 9948。

综上所述，当有新的 redo 日志写入到 log buffer 时，首先 lsn 的值会增长，但 flushed_to_disk_lsn 不变，随后随着不断有 log buffer 中的日志被刷新到磁盘上，flushed_to_disk_lsn 的值也跟着增长。如果两者的值相同时，说明 log buffer 中的所有 redo 日志都已经刷新到磁盘中了。

Tips：应用程序向磁盘写入文件时其实是先写到操作系统的缓冲区中去，如果某个写入操作要等到操作系统确认已经写到磁盘时才返回，那需要调用一下操作系统提供的 fsync 函数。其实只有当系统执行了 fsync 函数后，flushed_to_disk_lsn 的值才会跟着增长，当仅仅把 log buffer 中的日志写入到操作系统缓冲区却没有显式的刷新到磁盘时，另外的一个称之为 write_lsn 的值跟着增长。

当然系统的 LSN 值远不止我们前面描述的 lsn，还有很多。

查看系统中的各种 LSN 值

我们可以使用 SHOW ENGINE INNODB STATUS 命令查看当前 InnoDB 存储引擎中的各种 LSN 值的情况，比如：

SHOW ENGINE INNODB STATUS\G

```
---
LOG
---
Log sequence number 12708714
Log flushed up to   12708714
Pages flushed up to 12708714
Last checkpoint at  12708705
0 pending log flushes, 0 pending chkp writes
10 log i/o's done, 0.07 log i/o's/second
```

其中：

Log sequence number：代表系统中的 lsn 值，也就是当前系统已经写入的 redo 日志量，包括写入 log buffer 中的日志。

Log flushed up to：代表 flushed_to_disk_lsn 的值，也就是当前系统已经写入磁盘的 redo 日志量。

Pages flushed up to：代表 flush 链表中被最早修改的那个页面对应的 oldest_modification 属性值。

Last checkpoint at：当前系统的 checkpoint_lsn 值。

innodb_flush_log_at_trx_commit 的用法

我们前边说为了保证事务的持久性，用户线程在事务提交时需要将该事务执行过程中产生的所有 redo 日志都刷新到磁盘上。会很明显的降低数据库性能。如果对事务的持久性要求不是那么强烈的话，可以选择修改一个称为 innodb_flush_log_at_trx_commit 的系统变量的值，该变量有 3 个可选的值：

0：当该系统变量值为 0 时，表示在事务提交时不立即向磁盘中同步 redo 日志，这个任务是交给后台线程做的。

这样很明显会加快请求处理速度，但是如果事务提交后服务器挂了，后台线程没有及时将 redo 日志刷新到磁盘，那么该事务对页面的修改会丢失。

1：当该系统变量值为 1 时，表示在事务提交时需要将 redo 日志同步到磁盘，可以保证事务的持久性。1 也是 innodb_flush_log_at_trx_commit 的默认值。

```
mysql> show variables like 'innodb_flush_log_at_trx_commit';
+--------------------------------+-------+
| Variable_name                  | Value |
+--------------------------------+-------+
| innodb_flush_log_at_trx_commit | 1     |
+--------------------------------+-------+
1 row in set (0.00 sec)
```

2：当该系统变量值为 2 时，表示在事务提交时需要将 redo 日志写到操作系统的缓冲区中，但并不需要保证将日志真正的刷新到磁盘。

这种情况下如果数据库挂了，操作系统没挂的话，事务的持久性还是可以保证的，但是操作系统也挂了的话，那就不能保证持久性了。

undo 日志

事务回滚的需求

我们说过事务需要保证原子性，也就是事务中的操作要么全部完成，要么什么也不做。但是偏偏有时候事务执行到一半会出现一些情况，比如：

情况一：事务执行过程中可能遇到各种错误，比如服务器本身的错误，操作系统错误，甚至是突然断电导致的错误。

情况二：程序员可以在事务执行过程中手动输入 ROLLBACK 语句结束当前的事务的执行。

这两种情况都会导致事务执行到一半就结束，但是事务执行过程中可能已经修改了很多东西，为了保证事务的原子性，我们需要把东西改回原先的样子，这个过程就称之为回滚（英文名：rollback），这样就可以造成这个事务看起来什么都没做，所以符合原子性要求。

每当我们要对一条记录做改动时（这里的改动可以指 INSERT、DELETE、UPDATE），都需要把回滚时所需的东西都给记下来。比方说：

你插入一条记录时，至少要把这条记录的主键值记下来，之后回滚的时候只需要把这个主键值对应的记录删掉。

你删除了一条记录，至少要把这条记录中的内容都记下来，这样之后回滚时再把由这些内容组成的记录插入到表中。

你修改了一条记录，至少要把修改这条记录前的旧值都记录下来，这样之后回滚时再把这条记录更新为旧值。

这些为了回滚而记录的这些东西称之为撤销日志，英文名为 undo log/undo 日志。这里需要注意的一点是，由于查询操作（SELECT）并不会修改任何用户记录，所以在查询操作执行时，并不需要记录相应的 undo 日志。

当然，在真实的 InnoDB 中，undo 日志其实并不像我们上边所说的那么简单，不同类型的操作产生的 undo 日志的格式也是不同的。

事务 id

给事务分配 id 的时机

一个事务可以是一个只读事务，或者是一个读写事务：

我们可以通过 START TRANSACTION READ ONLY 语句开启一个只读事务。

在只读事务中不可以对普通的表（其他事务也能访问到的表）进行增、删、改操作，但可以对用户临时表做增、删、改操作。

我们可以通过 START TRANSACTION READ WRITE 语句开启一个读写事务，或者使用 BEGIN、START TRANSACTION 语句开启的事务默认也算是读写事务。

在读写事务中可以对表执行增删改查操作。

如果某个事务执行过程中对某个表执行了增、删、改操作，那么 InnoDB 存储引擎就会给它分配一个独一无二的事务 id，分配方式如下：

对于只读事务来说，只有在它第一次对某个用户创建的临时表执行增、删、改操作时才会为这个事务分配一个事务 id，否则的话是不分配事务 id 的。

我们前边说过对某个查询语句执行 EXPLAIN 分析它的查询计划时，有时候在 Extra 列会看到 Using temporary 的提示，这个表明在执行该查询语句时会用到内部临时表。这个所谓的内部临时表和我们手动用 CREATE TEMPORARY TABLE 创建的用户临时表并不一样，在事务回滚时并不需要把执行 SELECT 语句过程中用到的内部临时表也回滚，在执行 SELECT 语句用到内部临时表时并不会为它分配事务 id。

对于读写事务来说，只有在它第一次对某个表（包括用户创建的临时表）执行增、删、改操作时才会为这个事务分配一个事务 id，否则的话也是不分配事务 id 的。

有的时候虽然我们开启了一个读写事务，但是在这个事务中全是查询语句，并没有执行增、删、改的语句，那也就意味着这个事务并不会被分配一个事务 id。

上边描述的事务 id 分配策略是针对 MySQL 5.7 来说的，前边的版本的分配方式可能不同。

事务 id 生成机制

这个事务 id 本质上就是一个数字，它的分配策略和我们前边提到的对隐藏列 row_id（当用户没有为表创建主键和 UNIQUE 键时 InnoDB 自动创建的列）的分配策略大抵相同，具体策略如下：

服务器会在内存中维护一个全局变量，每当需要为某个事务分配一个事务 id 时，就会把该变量的值当作事务 id 分配给该事务，并且把该变量自增 1。

每当这个变量的值为 256 的倍数时，就会将该变量的值刷新到系统表空间的页号为 5 的页面中一个称之为 Max Trx ID 的属性处，这个属性占用 8 个字节的存储空间。

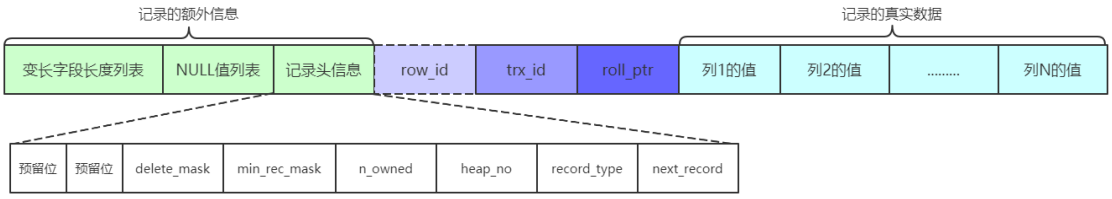
当系统下一次重新启动时，会将上边提到的 Max Trx ID 属性加载到内存中，将该值加上 256 之后赋值给我们前边提到的全局变量（因为在上次关机时该全局变量的值可能大于 Max Trx ID 属性值）。

这样就可以保证整个系统中分配的事务 id 值是一个递增的数字。先被分配 id 的事务得到的是较小的事务 id，后被分配 id 的事务得到的是较大的事务 id。

trx_id 隐藏列

我们在学习 InnoDB 记录行格式的时候重点强调过：聚簇索引的记录除了会保存完整的用户数据以外，而且还会自动添加名为 trx_id、roll_pointer 的隐藏列，

如果用户没有在表中定义主键以及 UNIQUE 键，还会自动添加一个名为 row_id 的隐藏列。



其中的 trx_id 列就是某个对这个聚簇索引记录做改动的语句所在的事务对应的事务 id 而已（此处的改动可以是 INSERT、DELETE、UPDATE 操作）。至于 roll_pointer 隐藏列我们后边分析。

undo 日志的格式

为了实现事务的原子性，InnoDB 存储引擎在实际进行增、删、改一条记录时，都需要先把对应的 undo 日志记下来。一般每对一条记录做一次改动，就对应着一条 undo 日志，但在某些更新记录的操作中，也可能会对应着 2 条 undo 日志。

一个事务在执行过程中可能新增、删除、更新若干条记录，也就是说需要记录很多条对应的 undo 日志，这些 undo 日志会被从 0 开始编号，也就是说根据生成的顺序分别被称为第 0 号 undo 日志、第 1 号 undo 日志、...、第 n 号 undo 日志等，这个编号也被称之为 undo NO。

我们前边说明表空间的时候说过，表空间其实是由许许多多的页面构成的，页面默认大小为 16KB。这些页面有不同的类型，其中有一种称之为 FIL_PAGE_UNDO_LOG 类型的页面是专门用来存储 undo 日志的。也就是说 Undo page 跟储存的数据和索引的页等是类似的。

FIL_PAGE_UNDO_LOG 页面可以从系统表空间中分配，也可以从一种专门存放 undo 日志的表空间，也就是所谓的 undo tablespace 中分配。先来看看不同操作都会产生什么样子的 undo 日志。

INSERT 操作对应的 undo 日志

当我们向表中插入一条记录时最终导致的结果就是这条记录被放到了一个数据页中。如果希望回滚这个插入操作，那么把这条记录删除就好了，也就是说在写对应的 undo 日志时，主要是把这条记录的主键信息记上。InnoDB 的设计了一个类型为 TRX_UNDO_INSERT_REC 的 undo 日志。

当我们向某个表中插入一条记录时，实际上需要向聚簇索引和所有的二级索引都插入一条记录。不过记录 undo 日志时，我们只需要考虑向聚簇索引插入记录时的情况就好了，因为其实聚簇索引记录和二级索引记录是一一对应的，我们在回滚插入操作时，只需要知道这条记录的主键信息，然后根据主键信息做对应的删除操作，做删除操作时就会顺带着把所有二级索引中相应的记录也删除掉。后边说到的 DELETE 操作和 UPDATE 操作对应的 undo 日志也都是针对聚簇索引记录而言的。

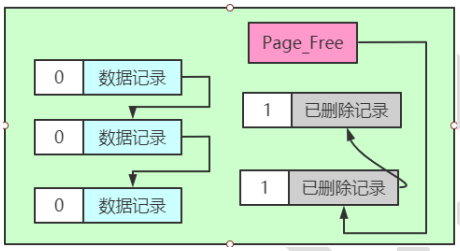
roll_pointer 的作用

roll_pointer 本质上就是一个指向记录对应的 undo 日志的一个指针。比方说我们向表里插入了 2 条记录，每条记录都有与其对应的一条 undo 日志。记录被存储到了类型为 FIL_PAGE_INDEX 的页面中（就是我们前边一直所说的数据页），undo 日志被存放到了类型为 FIL_PAGE_UNDO_LOG 的页面中。roll_pointer 本质就是一个指针，指向记录对应的 undo 日志。

DELETE 操作对应的 undo 日志

我们知道插入到页面中的记录会根据记录头信息中的 next_record 属性组成一个单向链表，我们把这个链表称之为正常记录链表；被删除的记录其实也会根据记录头信息中的 next_record 属性组成一个链表，只不过这个链表中的记录占用的存储空间可以被重新利用，所以也称这个链表为垃圾链表。Page Header 部分有一个称之为 PAGE_FREE 的属性，它指向由被删除记录组成的垃圾链表中的头节点。

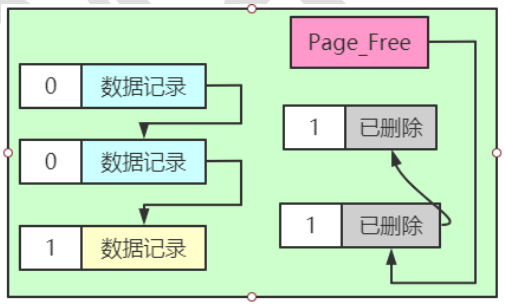
假设此刻某个页面中的记录分布情况是这样的



我们只把记录的 delete_mask 标志位展示了出来。从图中可以看出，正常记录链表中包含了 3 条正常记录，垃圾链表里包含了 2 条已删除记录。页面的 Page Header 部分的 PAGE_FREE 属性的值代表指向垃圾链表头节点的指针。

假设现在我们准备使用 DELETE 语句把正常记录链表中的最后一条记录给删除掉，其实这个删除的过程需要经历两个阶段：

阶段一：将记录的 delete_mask 标识位设置为 1，这个阶段称之为 delete mark。



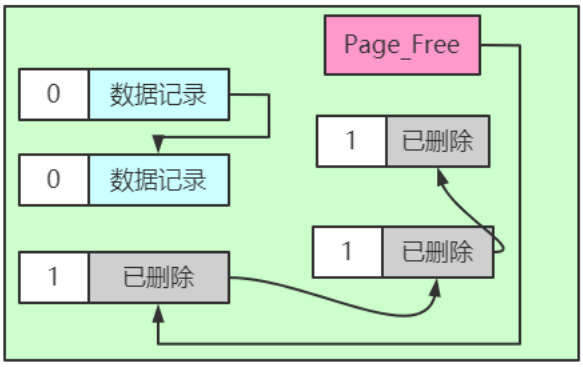
可以看到，正常记录链表中的最后一条记录的 delete_mask 值被设置为 1，但是并没有被加入到垃圾链表。也就是此时记录处于一个中间状态。在删除语句所在的事务提交之前，被删除的记录一直都处于这种所谓的中间状态。

为啥会有这种奇怪的中间状态呢？其实主要是为了实现 MVCC 的功能。

阶段二：当该删除语句所在的事务提交之后，会有专门的线程后来真正的把记录删除掉。所谓真正的删除就是把该记录从正常记录链表中移除，并且加入到

垃圾链表中，然后还要调整一些页面的其他信息，比如页面中的用户记录数量 PAGE_N_RECS、上次插入记录的位置 PAGE_LAST_INSERT、垃圾链表头节点的指针 PAGE_FREE、页面中可重用的字节数量 PAGE_GARBAGE、还有页目录的一些信息等等。这个阶段称之为 purge。

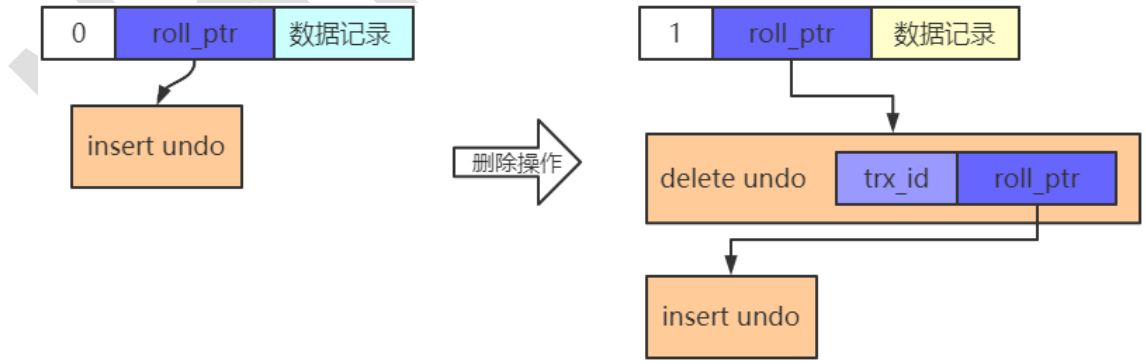
把阶段二执行完了，这条记录就算是真正的被删除掉了。这条已删除记录占用的存储空间也可以被重新利用了。



从上边的描述中我们也可以看出来，在删除语句所在的事务提交之前，只会经历阶段一，也就是 delete mark 阶段（提交之后我们就不用回滚了，所以只需考虑对删除操作的阶段一做的影响进行回滚）。InnoDB 中就会产生一种称之为 TRX_UNDO_DEL_MARK_REC 类型的 undo 日志。

版本链

同时，在对一条记录进行 delete mark 操作前，需要把该记录的旧的 trx_id 和 roll_pointer 隐藏列的值都给记到对应的 undo 日志中来，就是我们图中显示的 old trx_id 和 old roll_pointer 属性。这样有一个好处，那就是可以通过 undo 日志的 old roll_pointer 找到记录在修改之前对应的 undo 日志。比方说在一个事务中，我们先插入了一条记录，然后又执行对该记录的删除操作，这个过程的示意图就是这样：



从图中可以看出来，执行完 delete mark 操作后，它对应的 undo 日志和 INSERT 操作对应的 undo 日志就串成了一个链表。这个链表就称之为版本链。版本链在事务的隔离性中有很重要的作用，具体请参考第四期 MySQL《07-VIP-深入理解 MVCC 与 BufferPool 缓存机制》

UPDATE 操作对应的 undo 日志

在执行 UPDATE 语句时，InnoDB 对更新主键和不更新主键这两种情况有截然不同的处理方案。

不更新主键的情况

在不更新主键的情况下，又可以细分为被更新的列占用的存储空间不发生变化和发生变化的情况。

就地更新（in-place update）

更新记录时，对于被更新的每个列来说，如果更新后的列和更新前的列占用的存储空间都一样大，那么就可以进行就地更新，也就是直接在原记录的基础上修改对应列的值。再次强调一边，是每个列在更新前后占用的存储空间一样大，有任何一个被更新的列更新前比更新后占用的存储空间大，或者更新前比更新后占用的存储空间小都不能进行就地更新。

先删除掉旧记录，再插入新记录

在不更新主键的情况下，如果有任何一个被更新的列更新前和更新后占用的存储空间大小不一致，那么就需要先把这条旧的记录从聚簇索引页面中删除掉，然后再根据更新后列的值创建一条新的记录插入到页面中。

请注意一下，我们这里所说的删除并不是 delete mark 操作，而是真正的删除掉，也就是把这条记录从正常记录链表中移除并加入到垃圾链表中，并且修改页面中相应的统计信息（比如 PAGE_FREE、PAGE_GARBAGE 等这些信息）。由用户线程同步执行真正的删除操作，真正删除之后紧接着就要根据各个列更新后的值创建的新记录插入。

这里如果新创建的记录占用的存储空间大小不超过旧记录占用的空间，那么可以直接重用被加入到垃圾链表中的旧记录所占用的存储空间，否则的话需要在页面中新申请一段空间以供新记录使用，如果本页面内已经没有可用的空间的话，那就需要进行页面分裂操作，然后再插入新记录。

针对 UPDATE 不更新主键的情况（包括上边所说的就地更新和先删除旧记录再插入新记录），InnoDB 设计了一种类型为 TRX_UNDO_UPD_EXIST_REC 的 undo 日志。

更新主键的情况

在聚簇索引中，记录是按照主键值的大小连成了一个单向链表的，如果我们更新了某条记录的主键值，意味着这条记录在聚簇索引中的位置将会发生改变，比如你将记录的主键值从 1 更新为 10000，如果还有非常多的记录的主键值分布在 1 ~ 10000 之间的话，那么这两条记录在聚簇索引中就有可能离得非常远，甚至中间隔了好多个页面。针对 UPDATE 语句中更新了记录主键值的这种情况，InnoDB 在聚簇索引中分了两步处理：

将旧记录进行 delete mark 操作

也就是说在 UPDATE 语句所在的事务提交前，对旧记录只做一个 delete mark 操作，在事务提交后才由专门的线程做 purge 操作，把它加入到垃圾链表中。这

里一定要和我们上边所说的在不更新记录主键值时，先真正删除旧记录，再插入新记录的方式区分开！

之所以只对旧记录做 delete mark 操作，是因为别的事务同时也可能访问这条记录，如果把它真正的删除加入到垃圾链表后，别的事务就访问不到了。这个功能就是所谓的 MVCC。

创建一条新记录

根据更新后各列的值创建一条新记录，并将其插入到聚簇索引中（需重新定位插入的位置）。

由于更新后的记录主键值发生了改变，所以需要重新从聚簇索引中定位这条记录所在的位置，然后把它插进去。

针对 UPDATE 语句更新记录主键值的这种情况，在对该记录进行 delete mark 操作前，会记录一条类型为 TRX_UNDO_DEL_MARK_REC 的 undo 日志；之后插入新记录时，会记录一条类型为 TRX_UNDO_INSERT_REC 的 undo 日志，也就是说每对一条记录的主键值做改动时，会记录 2 条 undo 日志。

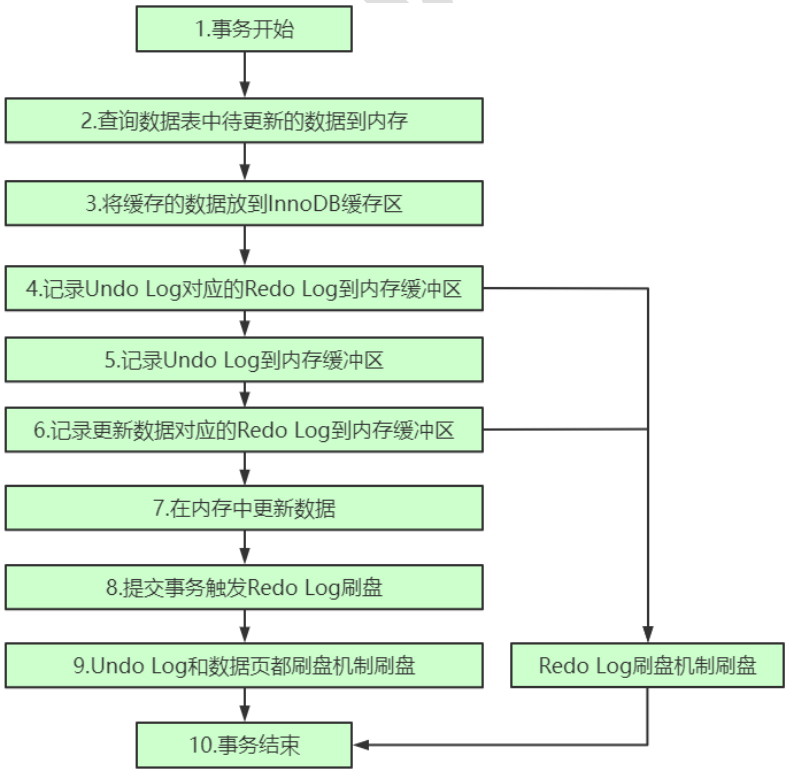
总结事务的流程

总的来说，事务流程分为事务的执行流程和事务恢复流程。

事务执行

我们已经知道了 MySQL 的事务主要主要是通过 Redo Log 和 Undo Log 实现的。

MySQL 事务执行流程如下图



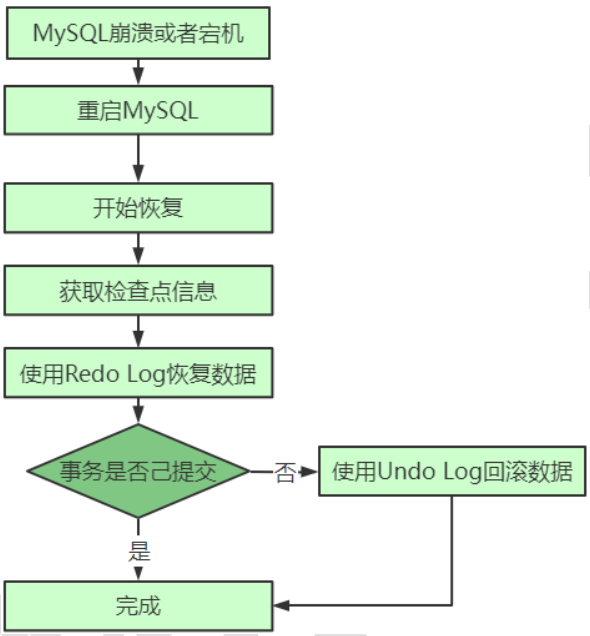
可以看出，MySQL 在事务执行的过程中，会记录相应 SQL 语句的 UndoLog 和 Redo Log，然后在内存中更新数据并形成数据脏页。接下来 RedoLog 会根据一定规则触发刷盘操作，Undo Log 和数据脏页则通过刷盘机制刷盘。事务提交时，会将当前事务相关的所有 Redo Log 刷盘，只有当前事务相关的所有 Redo Log 刷盘成功，事务才算提交成功。

事务恢复

如果一切正常，则 MySQL 事务会按照上图中的顺序执行。如果 MySQL 由于某种原因崩溃或者宕机，当然进行数据的恢复或者回滚操作。

如果事务在执行第 8 步,即事务提交之前，MySQL 崩溃或者宕机,此时会先使用 Redo Log 恢复数据,然后使用 Undo Log 回滚数据。

如果在执行第 8 步之后 MySQL 崩溃或者宕机，此时会使用 Redo Log 恢复数据，大体流程如下图所示。



很明显，MySQL 崩溃恢复后，首先会获取日志检查点信息，随后根据日志检查点信息使用 Redo Log 进行恢复。MySQL 崩溃或者宕机时事务未提交，则接下来使用 Undo Log 回滚数据。如果在 MySQL 崩溃或者宕机时事务已经提交，则用 Redo Log 恢复数据即可。

恢复机制

在服务器不挂的情况下，redo 日志简直就是个大累赘，不仅没用，反而让性能变得更差。但是万一数据库挂了，就可以在重启时根据 redo 日志中的记录就可以将页面恢复到系统崩溃前的状态。

MySQL 可以根据 redo 日志中的各种 LSN 值，来确定恢复的起点和终点。然后将 redo 日志中的数据，以哈希表的形式，将一个页面下的放到哈希表的一个槽中。之后就可以遍历哈希表，因为对同一个页面进行修改的 redo 日志都放在了一个槽里，所以可以一次性将一个页面修复好（避免了很多读取页面的随机 IO）。

并且通过各种机制，避免无谓的页面修复，比如已经刷新的页面，进而提升崩溃恢复的速度。

崩溃后的恢复为什么不用 binlog？

什么是 BinLog？请参考第四期 MySQL《03-VIP-Mysql 内部 SQL 执行原理》一章。

1、这两者使用方式不一样

binlog 会记录表所有更改操作，包括更新删除数据，更改表结构等等，主要用于人工恢复数据，而 redo log 对于我们是不可见的，它是 InnoDB 用于保证 crash-safe 能力的，也就是在事务提交后 MySQL 崩溃的话，可以保证事务的持久性，即事务提交后其更改是永久性的。

一句话概括：binlog 是用作人工恢复数据，redo log 是 MySQL 自己使用，用于保证在数据库崩溃时的事务持久性。

2、redo log 是 InnoDB 引擎特有的，binlog 是 MySQL 的 Server 层实现的，所有引擎都可以使用。

3、redo log 是物理日志，记录的是“在某个数据页上做了什么修改”，恢复的速度更快；binlog 是逻辑日志，记录的是这个语句的原始逻辑，比如“给 ID=2 这的 c 字段加 1 ”；

4、redo log 是“循环写”的日志文件，redo log 只会记录未刷盘的日志，已经刷入磁盘的数据都会从 redo log 这个有限大小的日志文件里删除。binlog 是追加日志，保存的是全量的日志。

5、最重要的是，当数据库 crash 后，想要恢复未刷盘但已经写入 redo log 和 binlog 的数据到内存时，binlog 是无法恢复的。虽然 binlog 拥有全量的日志，但没有一个标志让 innoDB 判断哪些数据已经入表(写入磁盘)，哪些数据还没有。

比如，binlog 记录了两条日志：

给 ID=2 这一行的 c 字段加 1

给 ID=2 这一行的 c 字段加 1

在记录 1 入表后，记录 2 未入表时，数据库 crash。重启后，只通过 binlog 数据库无法判断这两条记录哪条已经写入磁盘，哪条没有写入磁盘，不管是两条都恢复至内存，还是都不恢复，对 ID=2 这行数据来说，都不对。

但 redo log 不一样，只要刷入磁盘的数据，都会从 redo log 中抹掉，数据库重启后，直接把 redo log 中的数据都恢复至内存就可以了。

Redo 日志和 Undo 日志的关系

数据库崩溃重启后，需要先从 redo log 中把未落盘的脏页数据恢复回来，重新写入磁盘，保证用户的数据不丢失。当然，在崩溃恢复中还需要把未提交的事务进行回滚操作。由于回滚操作需要 undo log 日志支持，undo log 日志的完整性和可靠性需要 redo log 日志来保证，所以数据库崩溃需要先做 redo log 数据恢复，然后做 undo log 回滚。

在事务执行过程中，除了记录 redo 一些记录，还会记录 undo log 日志。Undo log 记录了数据每个操作前的状态，如果事务执行过程中需要回滚，就可以根据 undo log 进行回滚操作。

因为 redo log 是物理日志，记录的是数据库页的物理修改操作。所以 undo log（可以看成数据库的数据）的写入也会伴随着 redo log 的产生，这是因为 undo log 也需要持久化的保护。

事务进行过程中，每次 sql 语句执行，都会记录 undo log 和 redo log，然后更新数据形成脏页。事务执行 COMMIT 操作时，会将本事务相关的所有 redo log 进行落盘，只有所有的 redo log 落盘成功，才算 COMMIT 成功。然后内存中的 undo log 和脏页按照同样的规则进行落盘。如果此时发生崩溃，则只使用 redo log 恢复数据。

同时写 Redo 和 Binlog 怎么保持一致?

当我们开启了 MySQL 的 BinLog 日志，很明显需要保证 BinLog 和事务日志的一致性，为了保证二者的一致性，使用了两阶段事务 2PC（所谓的两个阶段是指：第一阶段：准备阶段和第二阶段：提交阶段，具体的内容请参考分布式事务的相关的课程内容）。步骤如下：

1）当事务提交时 InnoDB 存储引擎进行 prepare 操作。

2）MySQL 上层会将数据库、数据表和数据表中的数据的更新操作写入 BinLog 文件。

3）InnoDB 存储引擎将事务日志写入 Redo Log 文件中。

MySQL8 中的新增特性

MySQL8 在功能上的我们需要关注增强主要有：账户与安全方面；系统方面；索引方面；InnoDB 方面。

账户与安全

用户的创建与授权

用户的创建与授权需要两条单独的 SQL 语句执行

在 MySQL5.7 的版本：

> grant all privileges on *.* to '用户名'@'主机' identified by '密码';

在 MySQL8.0 需要分开执行：

>create user '用户名'@'主机' identified by '密码';

>grant all privileges on *.* to '用户名'@'主机';

用以前的一条命令在 8.0 里面创建用户，会出现 sql 语法错误

密码管理

MySQL8.0 的密码管理策略提供了 3 个变量：

1、password_history 修改密码不允许与最近几次使用或的密码重复，默认是 0，即不限制；

2、password_reuse_interval 修改密码不允许与最近多少天的使用过的密码重复，默认是 0,即不限制；

3、password_require_current 修改密码是否需要提供当前的登录密码，默认是 OFF,即不需要；如果需要，则设置成 ON。

还可针对某一个用户单独设置密码管理策略

alter user '用户名'@'主机' password history 5;

其他

认证插件更新；角色管理也有调整。

索引

隐藏索引，被隐藏的索引不会被优化器使用，但依然真实存在，主要用于软删除，可以根据需要后续真正删除或者重新可视化。我们可以隐藏一个索引，然后观察对数据库的影响。如果数据库性能有所下降，就说明这个索引是有用的，于是将其“恢复显示”即可；如果数据库性能看不出变化，说明这个索引是多余的，可以考虑去除了。当索引被隐藏时，它的内容仍然是和正常索引一样实时更新的，这个特性本身是专门为优化调试使用。如果你长期隐藏一个索引，那还不如干脆删掉，因为毕竟索引的存在会影响插入、更新和删除的性能。

开始真正支持降序索引，以往的 MySQL 虽然支持降序索引，但是写盘的时候依然是升序保存。MySQL8.0 中则是真正的按降序保存。

不再对 group by 操作进行隐式排序。

索引中可以使用函数表达式，创建表时创建一个函数索引，查询的时候使用同样的函数就可以利用索引了。

系统方面

原子 ddl 操作

MySQL5.7 执行 drop 命令 drop table t1,t2; 如果 t1 存在，t2 不存在，会提示 t2 表不存在，但是 t1 表仍然会被删除，MySQL8.0 执行同样的 drop 命令，会提示 t2 表不存在，而且 t1 表不会被删除，保证了原子性。

自增列持久化

解决了之前的版本，主键重复的问题。MySQL5.7 及其以前的版本，MySQL 服务器重启，会重新扫描表的主键最大值，如果之前已经删除过 id=100 的数据，但是表中当前记录的最大值如果是 99，那么经过扫描，下一条记录的 id 是 100，

而不是 101。MySQL8.0 则是每次在变化的时候，都会将自增计数器的最大值写入 redo log，同时在每次检查点将其写入引擎私有的系统表。则不会出现自增主键重复的问题。

重构 SQL 分析器

改进 SQL 分析器。旧的分析器由于其语法复杂性和自顶向下的分析方式从而有严重的限制，导致难以维护和扩展。

新的系统字典表

整合了存储有关数据库对象信息的事务数据字典，所有的元数据都用 InnoDB 引擎进行存储

支持资源管理

支持创建和管理资源组，并允许将服务器运行的线程分配给特定的组，以便线程根据资源组可用的资源执行

更好支持文档型数据库和 JSON

聚合函数

JSON_ARRAYAGG(),将多行数据组合成 json 数组

JSON_OBJECTAGG()，用于生成 json 对象

JSON 实用函数

JSON_PRETTY() 输出 json 数据的时候，格式化。

JSON_STORAGE_SIZE() json 数据所占用的存储空间(单位：字节)

JSON_STORAGE_FREE() json 数据更新后所释放的空间(单位：字节)

JSON 合并函数

MySQL8.0 废弃了 JSON_MERGE()函数，推荐使用以下两个函数合并 JSON 数据

JSON_MERGE_PATCH()

JSON_MERGE_PRESERV()

JSON 表函数

MySQL8.0 新增了 JSON_TABLE()函数，将 JSON 数据转换成关系表，可以将该函数的返回结果当做一个普通的临时表进行 sql 查询。

支持 RANK(), LAG()、NTILE()等函数

rank()

按照某字段的排序结果添加排名，但它是跳跃的、间断的排名，例如两个并列第一名后，下一个是第三名，1、1、3、4。

其他函数请自行查询 MySQL 官方手册。

正则表达式增强

提供了 REGEXP_LIKE()， EGEXP_INSTR(), REGEXP_REPLACE(), REGEXP_SUBSTR()等函数

新增备份锁

允许在线备份期间的 DML，同时防止可能导致快照不一致的操作。 备份锁由 LOCK INSTANCE FOR BACKUP 和 UNLOCK INSTANCE 语法支持

默认字符集

默认字符集由 latin1 变为 utf8mb4

配置参数

全局参数持久化

MySQL 8.0 版本支持在线修改全局参数持久化，通过加上 PERSIST 关键字，可以将调整持久化到新的配置文件中，再次重启 db 还可以应用到最新的参数。

支持会话级别动态调整部分参数

有利于提升语句性能。

```
select /*+ SET_VAR(sort_buffer_size = 16M) */ id    from test order id ;
insert    /*+ SET_VAR(foreign_key_checks=OFF) */ into test(name) values(1);
```

默认参数的调整

调整 back_log 的默认值，保持和 max_connections 一致，增强突发流量带来的连接处理容量。

修改 event_scheduler 默认为 ON，之前默认是关闭的。

调整 max_allowed_packet 的默认值，从 4M 增加到 64M。

调整 bin_log,log_slave_updates 默认值为 on。

调整 expire_logs_days 的过期时间为 30 天，老版本是 7 天，生产环境时，检查该参数，防止 binlog 过多造成空间紧张。

调整 innodb_undo_log_truncate 默认为 ON

调整 innodb_undo_tablespaces 默认值为 2

调整 innodb_max_dirty_pages_pct_lwm 默认值 10

调整 innodb_max_dirty_pages_pct 默认值为 90

新增 innodb_autoinc_lock_mode 默认值为 2

InnoDB

innodb 增强

新增 INFORMATION_SCHEMA.INNODB_CACHED_INDEXES，查看每个索引缓存在 InnoDB 缓冲池中的索引页数。

InnoDB 临时表都将在共享临时表空间 ibtmp1 中创建。

对于 SELECT ... FOR SHARE 和 SELECT ... FOR UPDATE 语句，InnoDB 支持 NOWAIT 和 SKIP LOCKED。

innodb_undo_tablespaces 的最小值为 2，并且不再允许将 innodb_undo_tablespaces 设置为 0。 最小值 2 确保回滚段始终在撤消表空间中创建，而不是在系统表空间中创建。

支持 ALTER TABLESPACE ... RENAME TO 语法。

新增 innodb_dedicated_server，让 InnoDB 根据服务器上检测到的内存量自动配置 innodb_buffer_pool_size，innodb_log_file_size，innodb_flush_method。

新增 INFORMATION_SCHEMA.INNODB_TABLESPACES_BRIEF 视图。

新增了动态配置项 innodb_deadlock_detect，用来禁用死锁检查，因为在高并发系统中，当大量线程等待同一个锁时，死锁检查会大大拖慢数据库。

支持使用 innodb_directories 选项在服务器脱机时将表空间文件移动或恢复到新位置。

InnoDB 性能提升

并发

废除 buffer pool mutex,将原来一个 mutex 拆分成多个，提高并发

拆分 LOCK_thd_list 和 LOCK_thd_remove 这两个 mutex，大约可提高线程链接效率 5%。

行缓存

MySQL8.0 的优化器可以估算将要读取的行数，因此可以提供给存储引擎一个合适大小的 row buffer 来存储需要的数据。大批量的连续数据扫描的性能将受益于更大的 record buffer

改进扫描性能

改进 InnoDB 范围查询的性能，可提升全表查询和范围查询 5-20%的性能。

改进成本模型

InnoDB 缓冲区可以估算缓存区中的有多少表和索引，这可以让优化器选择访问方式时知道数据是否可以存储在内存中还是必须存储到磁盘上。

废弃特性

移除了一些功能，例如 query cache

在线文档：

http://note.youdao.com/noteshare?id=b4bb951d2d5da6551e6b962f1f35d04c&sub=8634788ECCA24FEE8A698223CDC1BC20