三大Log+两阶段【已总结】

redo log 和 undo log 是归属于Innodb 存储引擎的

bin log 是归属于 mysql Server 这个层面生成的日志，主要为了保证可靠性

ACID

- 原子性（Atomicity）

  通过 undo log 实现

- 一致性（Consistency）

  通过 原子性、隔离性、持久性 实现

- 隔离性（Isolation）

  一般有隔离级别 通过锁实现

- 持久性（Durability）

  通过 redo log 实现

redo log 和 undo log 都是存储引擎层面上生成的日志，并且都记录了数据的修改：只不过 redo log记录的是"物理级别"上的页修改操作，undo log 记录的是逻辑操作日志，比如对某一行数据进行了INSERT语句操作，那么 undo log就记录一条与之相反的DELETE操作。

Redo Log —— 保证事务持久性，数据异常恢复和服务重启时页数据同步恢复

和大多数关系型数据库一样，InnoDB记录了对数据文件的物理更改，并保证总是日志先行，也就是所谓的WAL(log write ahead)，即在持久化数据文件前，保证之前的redo日志已经写到磁盘。

redo log作用是为了保证数据的可靠性，最终落盘，但是又为了提升写入速度引入了redo log（buffer），然后通过一些策略再落盘到对应的 redolog file 中。

1. 当发生数据修改的时候，会不断产生redolog，这些redolog 会先写入redo log buffer，同时innodb引擎会在合适的时机将记录刷新到磁盘中
2. redo log 的大小是固定的，是循环写的过程
3. 有了redo log之后，innodb就可以保证即使数据库发生异常重启，之前的记录也不会丢失，叫做crash-safe

Mysql 在写日志的时候在用户空间有一个 log Buffer区域，对应在操作系统 内核空间有一个OS Buffer，通过一定的策略，对应的log buffer中的数据刷新到os buffer中，然后再将osbuffer中的数据刷新到log文件

如果数据更新到内存中，还没有进行持久化，也就是内存中已经存在，但是没有更新到磁盘中，如果这个时候断电，重启后磁盘和内存中数据会丢失，为了保证数据尽可能完整，mysql 提供了一个配置策略 ,有三个选项，即innodb_flush_log_at_trx_commit = 0|1|2。（默认是1）

0：提交事务的时候 ，是每隔一秒把 log buffer 中的数据刷新到os buffer，并调用 fsync() 写入到log文件，也就是说一秒之前的日志都是保存在log buffer缓冲区中，如果宕机，重启可能会丢失1s 的数据。

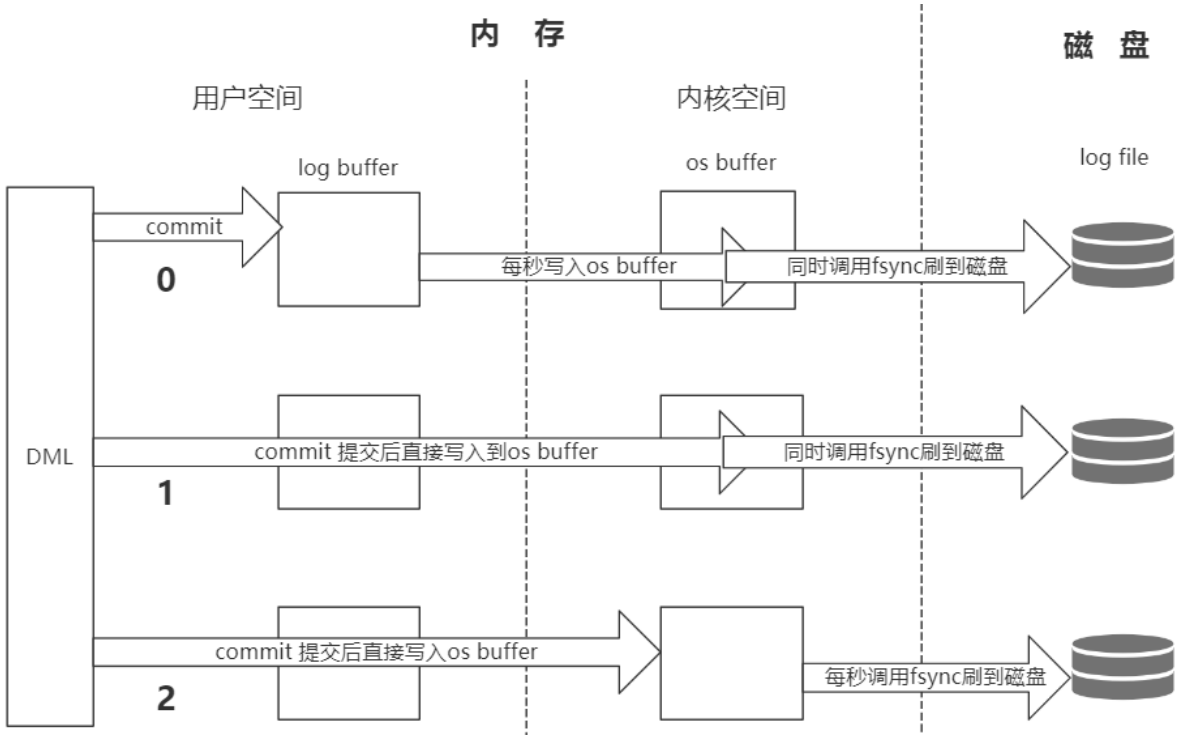
1：提交事务的时候，把logbuffer中的数据刷到os buffer，并调用文件系统的写入操作，将缓存刷新到磁盘。

2：提交事务的时候，将数据写入 os buffer，然后每隔1s 调用文件系统的flush操作，将 os buffer中的数据刷新到磁盘上去。同样 ，如果mysql挂了文件系统正常，重启服务后可能会丢失1s的数据

在mysql 5.6.6 以后， 上面提到的每隔1s的刷新频率可以通过 innodb_flush_log_at_timeout 进行配置（默认是1s），由于当 innodb_flush_log_at_trx_commit 设置成1的时，刷新os buffer 和调用文件系统的flush操作是即时发生的，所以 这种情况下innodb_flush_log_at_timeout 这个配置的对应的值不管是多少都是不起作用的

关于如上三种方式， 设置为1的时候是最安全的，设置成0、1的时候可能会造成1s数据丢失的问题，但是这种情况下IO次数较少，效率是较高的。0 和 2 的区别是2的情况多了

一次数据在内存的拷贝过程。



Undo Log —— 未commit的数据通过undo log回滚

- 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（MVCC）。
- 在操作任何数据之前，首先会将数据备份到一undo log日志文件，然后进行数据的修改，如果出现了错误或者用户进行了rollback操作，系统可以利用undo log中备份的数据恢复到数据之前的状态
- 注：undo log 是逻辑日志，可以理解为：

  delete 记录时，undolog 中会记录一跳对应的insert 记录

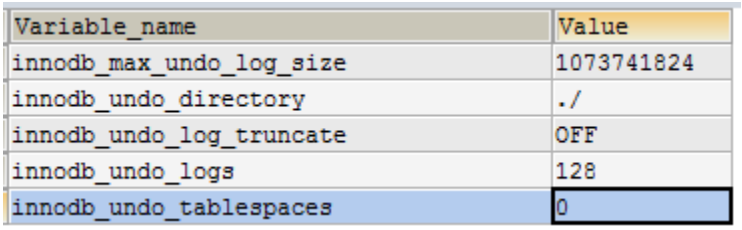
  insert 记录时，undo log中会记录一条对应的delete 记录

  update 纪录时，它记录一跳相反的update记录

undo在完成事务回滚和MVCC之后，就可以清除掉了，因为在大量事务操作的过程中，会需要大量的undo，所以Mysql（5.7）提供了一些配置项用来方便及时的回收undo，具

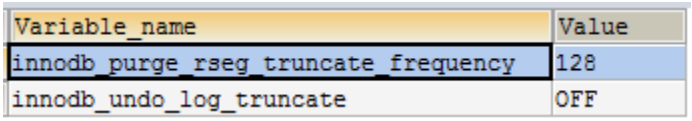
体如下：

SHOW GLOBAL VARIABLES LIKE '%undo%'



Variable_name	Value
innodb_max_undo_log_size	1073741824
innodb_undo_directory	./
innodb_undo_log_truncate	OFF
innodb_undo_logs	128
innodb_undo_tablespaces	0

SHOW GLOBAL VARIABLES LIKE '%truncate%'



Variable_name	Value
innodb_purge_rseg_truncate_frequency	128
innodb_undo_log_truncate	OFF

- innodb_max_undo_log_size

  控制最大undo tablespace文件的大小，当超过这个阀值（默认是1G），会触发truncate回收（收缩）动作，truncate后空间缩小到10M

- innodb_undo_directory

  undo文件存放的位置

- innodb_undo_log_truncate

  innodb_undo_log_truncate设置成ON的时候，innodb_max_undo_log_size如果达到阀值，就会被truncate到初始大小。

- innodb_undo_logs

  undo回滚段的数量，默认128， 至少大于等于35。5.7之前该参数叫innodb_rollback_segments。

- innodb_undo_tablespaces

  独立表空间个数， 默认为0，最大128。0表示不开启独立表空间，undo依然存放在ibdata文件。>0时，会在innodb_undo_directory指定的目录下创建对应个数undo文件（undo001、undo002...）,每个文件的默认大小为10M。但是参数必须大于或等于2，即回收（收缩）一个undo log日志文件时，要保证另一个undo log是可用的

- innodb_purge_rseg_truncate_frequency

控制回收（收缩）undo log的频率，想要增加释放回滚区间的频率，就得降低innodb_purge_rseg_truncate_frequency设定值

Bin Log—— 记录语句的原始逻辑，便于数据恢复

- binlog是server层的日志，主要做mysql层面的事情
- binlog 是记录所有数据结构的变更（如create、alter table等）以及数据的怎删改的二进制日志
- binlog的主要目的是复制和恢复数据，复制主要用于主从复制，Master端开启binlog，将二进制日志传给slaves达到数据一致的目的。恢复主要是数据异常或丢失后的数据恢复
- binlog的启用：my.cnf 配置文件中配置 log-bin 选项即可 （如： log-bin=xxxxxxx）
- binlog 中会记录所有的逻辑，并且采用追加写的方式
- 与redolog的区别
  1. redo 是innodb独有的，binlog是所有引擎都可以使用的
  2. redo是物理日志，记录的是在某个数据页上进行了什么修改，binlog是逻辑日志，记录的是这个语句的原始逻辑
  3. redo 是循环写的，空间会用完，binlog是追加写的，不会覆盖之前的日志信息
- binlog 恢复、复制可参考：https://zhuanlan.zhihu.com/p/33504555

binlog的写入

对于支持事务的数据库来说，必须是提交了事务之后才会记录binlog日志，而binlog刷新到磁盘的时机和sync_binlog有关。所以sync_binlog 这个参数对于Mysql是很重要的，它的设置可以直接印象到mysql的性能和数据安全问题，但是既要保证性能，也好兼顾数据安全本身就是一个伪命题。所以在实际业务中需要根据自己本身也业务需求对该参数进行设置。sync_binlog 的设置参数如下：

1. sync_binlog=0

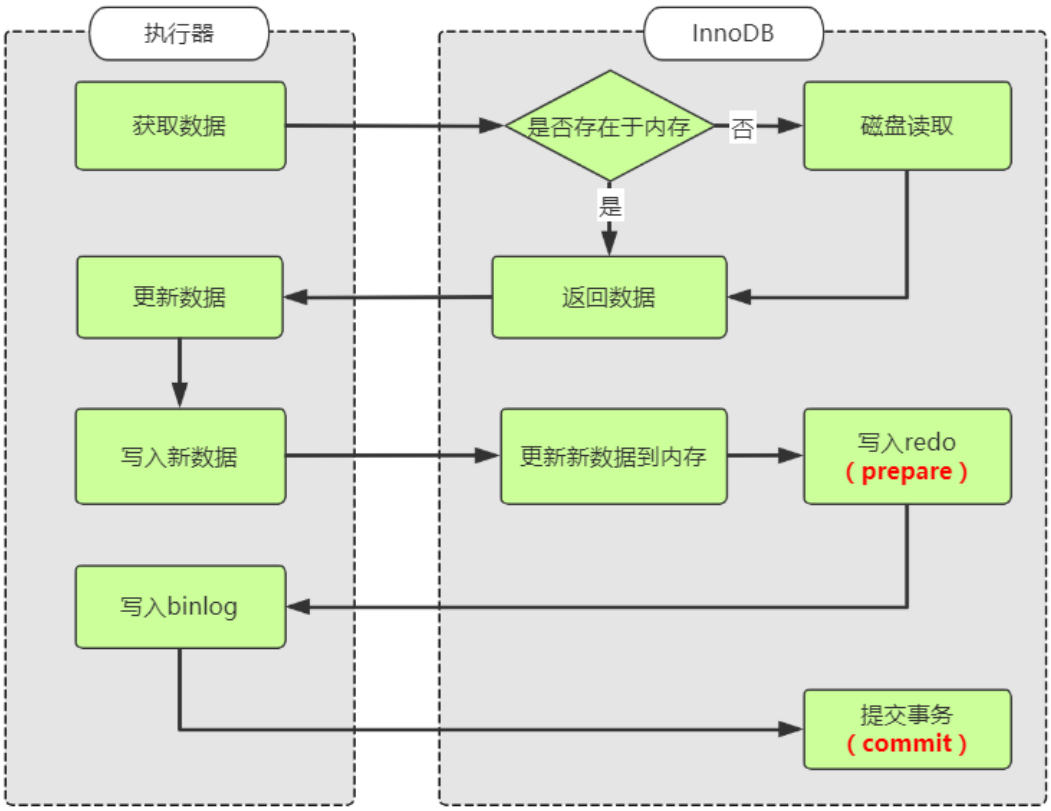
事务提交之后，不会立即将binlog cache中的日志信息刷新到磁盘，而是采用操作系统机制将缓存进行同步。这个设置可以保证性能，但是风险也最大，因为当系统一旦故障，就会丢失所有binlog cache中的数据

2. sync_binlog=n

   n 表示进行 n 次缓存操作后，再将binlog cache 日志信息统一持久化到磁盘，所以当sync_binlog=1时，即表示每次操作都会刷新到磁盘，这种情况即使在操作过程中系统故障也会丢失一个事物的数据，但是这种情况性能损耗最大。

两阶段提交

数据更新过程：



1. 执行器先从引擎中获取数据，如果内存中有，直接返回，如果内存中不存在，从磁盘copy到内存再返回。
2. 执行器获取到数据后会先修改数据，然后调用引擎接口重新写入数据，将数据更新到内存。
3. 将数据跟新到内存的同时，会将数据写入到redo log，这个时候事务阶段处于prepare阶段。prepare后会通知执行器可以进行接下来的操作
4. 前面操作完成后执行器会生成这个操作的binlog
5. 执行器调用引擎的事务接口，引擎把刚刚写完的redo改成commit状态，完成更新。

注：每个事务binlog的末尾，会记录一个XID event，标志事务是否提交成功，这样在进行recovery的过程中，binlog最后一个XID event之后的内容都应该被清楚

关于两阶段提交的具体原理可以参考：https://www.cnblogs.com/hustcat/p/3577584.html

两阶段的目的：

保证数据一致性，prepare阶段意味着已经放进内存了，然后写binlog，这样两个日志文件是一致的，最后在commit，这样就就可以保证数据的一致性。

没有两阶段提交会怎样？

1. 先写redolog，后写binlog

   插入一条数据，redolog写完，binlog还没有写完的时候宕机，这样redolog中会存在这条数据，binlog会不存在，在使用过程中可以通过redolog对该记录进行恢复，但是如果通过binlog同步，则找不到这条记录。

2. 先写binlog，后写redolog

   同理，插入一条记录，先写binlog，写redolog的时候宕机，重启以后通过redo进行事务恢复的时候，因为redolog没有写入，所以不会恢复这一条数据，但是如果通过binlog进行数据同步的时候会多出一条数据。

https://www.cnblogs.com/f-ck-need-u/archive/2018/05/08/9010872.html