redo log、undo log 和 binlog 详解

MySQL 中共有三种日志：redo log、undo log 和 binlog。

Buffer Pool

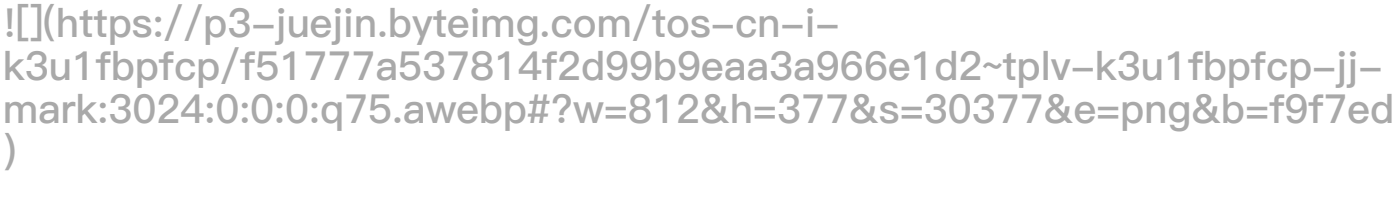
在介绍这三种日志之前，首先介绍 Buffer Pool。

Buffer Pool 在**存储引擎层**。

* 当读取数据时，如果数据存在于 Buffer Pool 中，客户端就会直接读取 Buffer Pool 中的数据，否则再去磁盘中读取。
* 当修改数据时，如果数据**存在于 Buffer Pool 中**，那**直接修改 Buffer Pool 中数据所在的页**，然后将其页**设置为脏页**（该页的内存数据和磁盘上的数据已经不一致），为了减少磁盘 I/O，**不会立即将脏页写入磁盘**，后续由后台线程选择一个合适的时机将脏页写入到磁盘。

InnoDB 会为 Buffer Pool 申请一片连续的内存空间，然后按照默认的 **16 KB** 的大小划分出一个个的页， Buffer Pool 中的页就叫做缓存页。

Buffer Pool 除了缓存**索引页**和**数据页**，还包括了 Undo 页，插入缓存、自适应哈希索引、锁信息等等。



查询一条记录时，会把**整个页**的数据都加载进 Buffer Pool 中，再通过页中的**页目录**定位某条具体的记录。

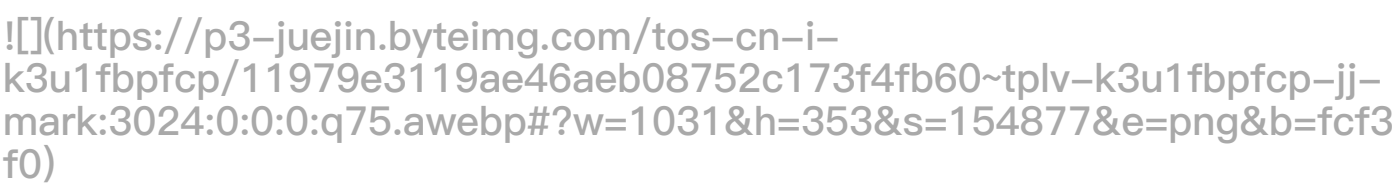
redo log

redo log 记录的是数据页的**物理变化**（xxx页xxx偏移量位置做了xxx修改，修改的值是多少），服务宕机时可用来同步数据。

redo log 保证了事务的**持久性**，**undo log** 保证了事务的**原子性**。

redo log 由两部分组成：

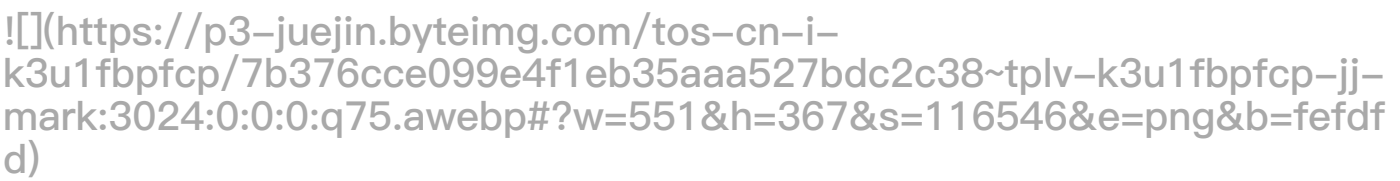
* **内存中**的**重做日志缓冲**（redo log buffer）；
* **磁盘上**的**重做日志文件**（redo log file）。



MySQL 每执行一条 DML 语句，会**先将操作写入 buffer pool**，然后就会**将记录先写入内存中的 redo log buffer** 中，后续**某个时间点**再**一次性把多个操作记录到磁盘中的 redo log file 中**。

redo log 有两个日志文件，采用**循环写**的方式。

write pos 表示 redo log **当前记录写到的位置**，**check point** 表示**当前要擦除的位置**：



如果 write pos 追上了 check point，就意味着 redo log 文件**写满**了，这时 MySQL **不能再执行新的更新操作**，也就是说 **MySQL 会被阻塞**，此时会停下来**将 Buffer Pool 中的脏页刷新到磁盘中**，然后标记 redo log 哪些记录可以被擦除，接着对旧的 redo log 记录进行**擦除**，等擦除完旧记录腾出了空间，checkpoint 就会往后移动，然后 MySQL 恢复正常运行，继续执行新的更新操作。

redo log 要写到磁盘，数据也要写磁盘，为什么要多此一举?

写入 redo log 的方式使用了追加操作，所以磁盘操作是**顺序写**；而写入数据需要先找到写入位置，然后才写到磁盘，所以磁盘操作是**随机写**。

顺序写比随机写要快很多。

undo log
========

undo log 记录的是**逻辑日志**，当事务回滚时通过逆操作恢复原来的数据，保证了事务的**原子性**。

在事务提交之前，MySQL 会**先记录更新前的数据**到 undo log 里，当事务回滚时，读取 undo log 里面的数据，做相反的操作来进行回滚。

一条记录的每一次更新操作产生的 undo log 格式都有一个 roll_pointer 指针和一个 trx_id 事务id：

* 通过 trx_id 可以知道该记录是被哪个事务修改的；
* 通过 roll_pointer 指针可以将这些 undo log 串成一个链表，这个链表就被称为**版本链**。

undo log 主要有两个作用：

* 实现**事务回滚**，**保障事务的原子性**。事务处理过程中，如果出现了错误或者用户执行了 ROLLBACK 语句，MySQL 可以利用 undo log 中的历史数据将数据恢复到事务开始之前的状态。
* 实现 **MVCC** 的关键因素之一。MVCC 是通过 **ReadView + undo log 版本链** 实现的。undo log 为每条记录保存多份历史数据，MySQL 在执行快照读的时候，会根据事务的 Read View 里的信息，顺着 undo log 的版本链找到满足其可见性的记录。

binlog
======

归档日志，是 **Server 层**生成的日志，主要用于**数据备份**和**主从复制**。

任何存储引擎下都有 binlog，而 redo log 和 undo log 是在 InnoDB 存储引擎下才有。

binlog 文件是记录了所有**数据库表结构变更**和**表数据修改**的日志，但**不会记录查询类的操作**，比如 select 操作。

binlog 格式
---

binlog 有 **3 种格式类型**，分别是 STATEMENT（默认格式）、ROW、MIXED：

* **STATEMENT**：每一条修改数据的 **SQL 语句**都会被记录到 binlog 中。但有**动态函数**的问题，比如使用了 uuid 或者 now 这些函数，那么就会导致主库上执行的结果并不是从库执行的结果，这种随时在变的函数就会导致复制前后的**数据不一致**。
* **ROW**：记录**行数据最终被修改成什么样了**，不会出现 STATEMENT 下动态函数的问题。但缺点是如果一条语句操作多行数据，每行数据的变化结果都会被记录，会使 binlog 文件过大。
* **MIXED**：包含了 STATEMENT 和 ROW 模式，它会根据不同的情况自动使用 ROW 模式和 STATEMENT 模式。

redo log 和 binlog 区别
==========

* **redo log** 是 **InnoDB 存储引擎**实现的日志；**binlog** 是 MySQL 的 **Server 层实现的日志**，**所有存储引擎**都可以使用。
* **记录的数据格式不同**：**redo log** 记录的是**某个数据页做了什么修改**；**binlog** 有 **3 种格式类型**，分别是 STATEMENT（默认格式）、ROW、MIXED。
* **写入方式不同**：**redo log** 是**循环写**，日志空间大小固定，全部写满就从头开始，保存未被刷入磁盘的脏页日志；**binlog** 是**追加写**，写满一个文件就创建一个新的文件继续写，**不会覆盖之前的日志**，保存的是全量的日志。
* **用途不同**：**redo log** 用于**掉电等故障恢复；binlog** 用于**备份恢复、主从复制**；

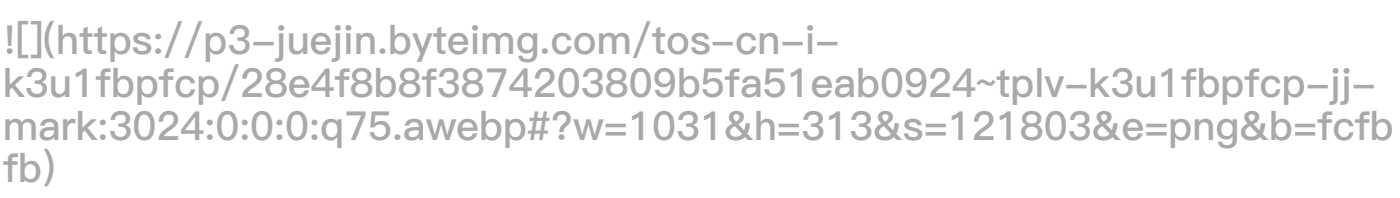
> 如果不小心整个数据库的数据被删除了，能使用 redo log 文件恢复数据吗?

不可以使用 redo log 文件恢复，只能使用 binlog 文件恢复。

因为 **redo log** 文件是循环写，是会**边写边擦除日志**的，只记录未被刷入磁盘的数据的物理日志，已经刷入磁盘的数据都会从 redo log 文件里擦除。

binlog 文件保存的是**全量日志**，也就是保存了所有数据变更的情况，理论上只要记录在 binlog 上的数据，都可以恢复，所以如果不小心整个数据库的数据被删除了，得用 binlog 文件恢复数据。

MySQL 主从复制
==========



主从同步的核心就是 binlog 日志，binlog 日志是 Server 层的日志，记录了所有 DML 和 DDL 语句。主从同步具体过程分为三步：

* 主库在**事务提交时**，把**数据变更记录在 binlog 中**；
* **从库读取主库的 binlog**，写入到从库的**中继日志** relay log中；
* 从库**重做中继日志中的事件**，这样数据就保持同步了。

两阶段提交
=====

为什么需要两阶段提交?

事务提交后，redo log 和 binlog 都要持久化到磁盘，但是这两个是独立的逻辑，可能出现半成功的状态，这样就造成**两份日志之间的逻辑不一致**。
redo log 影响主库的数据，binlog 影响从库的数据。

如果在**将 redo log 刷入到磁盘之后**，MySQL 突然宕机了，而 **binlog 还没有来得及写入**。

MySQL 重启后，通过 redo log 能将旧数据恢复，但是 binlog 里面没有记录这条更新语句，在主从架构中，binlog 会被复制到从库，由于 binlog 丢失了这条更新语句，从库的这一行数据是旧值，与主库的值**不一致**。

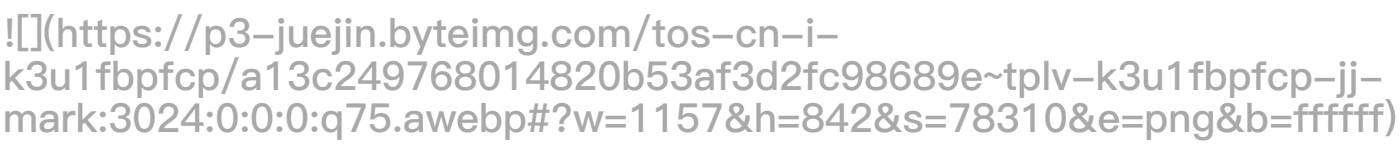
如果在**将 binlog 刷入到磁盘之后**，MySQL 突然宕机了，而 **redo log 还没有来得及写入**。

由于 redo log 还没写，崩溃恢复以后这个事务无效，所以该行数据还是旧值，而 binlog 里面记录了这条更新语句，在主从架构中，binlog 会被复制到从库，从库执行了这条更新语句，所以从库中这行数据是新值，与主库的值不一致。

MySQL 使用**两阶段提交**解决两份日志之间的逻辑不一致问题。

两阶段提交

两阶段提交把单个事务的提交拆分成了2个阶段，分别是**准备阶段**和**提交阶段**。



当提交事务时，MySQL 内部会开启一个 **XA 事务**，分两阶段来完成事务的提交，**将 redo log 的写入拆成了两个步骤：prepare 和 commit，中间再穿插写入 binlog**，具体如下：

* **prepare 阶段**：将 **XID**（内部 XA 事务的 ID） 写入到 **redo log**，同时将 redo log 对应的事务状态设置为 **prepare**，然后**将 redo log 持久化到磁盘**；
* **commit 阶段**：把 **XID** 写入到 **binlog**，然后将 binlog 持久化到磁盘，接着**调用引擎的提交事务接口**，将 **redo log 状态设置为 commit**，此时该状态并不需要持久化到磁盘，只需要 write 到文件系统的 page cache 中就够了；如果 **binlog 没有写入成功**，那么在**磁盘中就找不到对应的 XID**，说明 binlog 刷盘失败，这时 **redo log 会回滚**，从而

保证了两个日志的逻辑一致性。

可以看到，**对于处于 prepare 阶段的 redo log，即可以提交事务，也可以回滚事务，这取决于是否能在 binlog 中查找到与 redo log 相同的 XID**，如果有就提交事务，如果没有就回滚事务。

这样通过相同的 XID 就可以保证 redo log 和 binlog 这两份日志的一致性了。

所以说，**两阶段提交是以 binlog 写成功为事务提交成功的标识**，因为 binlog 写成功了，就意味着能在 binlog 中查找到与 redo log 相同的 XID。

binlog 组提交
==========

为了对两阶段提交进行优化，MySQL 引入 binlog 组提交，当有多个事务提交的时候，会将多个 binlog 刷盘操作合并成一个，从而减少磁盘 I/O 的次数。

引入了组提交机制后，prepare 阶段不变，只针对 commit 阶段，将 commit 阶段拆分为三个过程：

* **flush 阶段**：多个事务按进入的顺序将 binlog 从 cache 写入文件（不刷盘）；
* **sync 阶段**：对 binlog 文件做 fsync 操作（多个事务的 binlog 合并一次刷盘）；
* **commit 阶段**：各个事务按顺序做 InnoDB commit 操作；
 原文链接: https://juejin.cn/post/7383086531042951209