**MySQL 日志：undo log、redo log、binlog 有什么用？**

**执行一条 update 语句**会和select语句走类似的流程。

更新语句的流程会涉及到 undo log（回滚日志）、redo log（重做日志） 、binlog （归档日志）这三种日志：

* **undo log（回滚日志）**：**Innodb 存储引擎层**生成的日志，实现了事务中的**原子性**，主要**用于事务回滚和 MVCC**。
* **redo log（重做日志）**：**Innodb 存储引擎层**生成的日志，实现了事务中的**持久性**，主要**用于掉电等故障恢复**；
* **binlog （归档日志）**：**Server 层**生成的日志，主要**用于数据备份和主从复制**；

## 为什么需要 undo log？

undo log 是一种用于撤销回退的日志。在事务没提交之前，MySQL 会先记录更新前的数据到 undo log 日志文件里面，当事务回滚时，可以利用 undo log 来进行回滚。

每当 InnoDB 引擎对一条记录进行操作（修改、删除、新增）时，要把回滚时需要的信息都记录到 undo log 里，比如：

* 在**插入**一条记录时，要把这条**记录的主键值**记下来，这样之后回滚时只需要把这个主键值对应的记录**删除**就好了；
* 在**删除**一条记录时，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**到表中就好了；
* 在**更新**一条记录时，要把**被更新的列的旧值**记下来，这样之后回滚时再把这些列**更新为旧值**就好了。

在发生回滚时，就读取 undo log 里的数据，然后做原先相反操作（蓝色字）

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**：

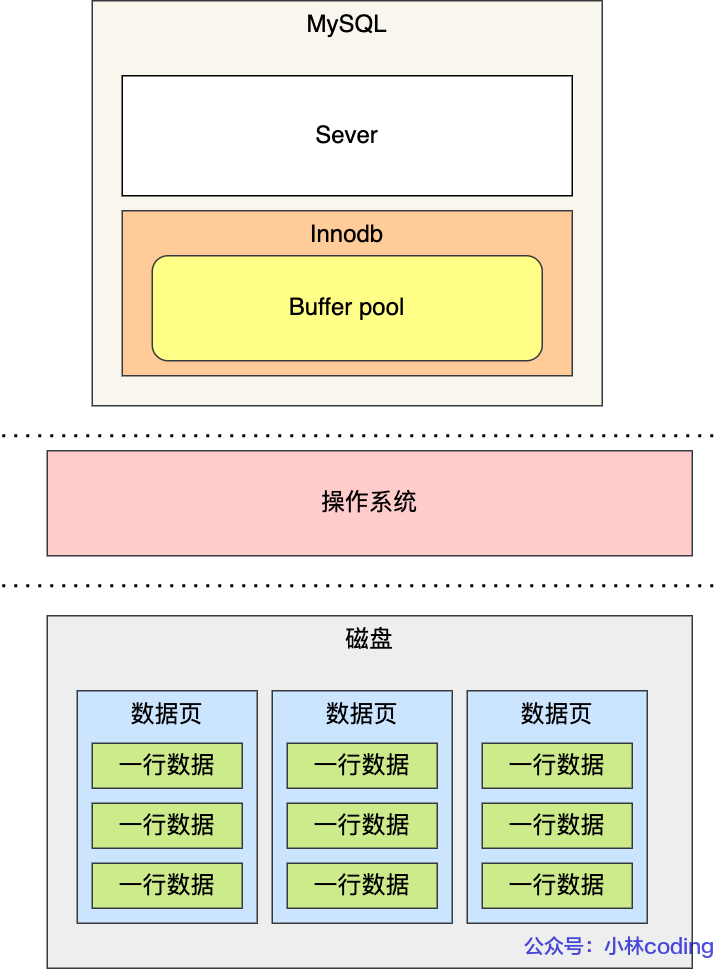
* 通过 trx\_id 可以知道该记录是被哪个事务修改的；
* 通过 roll\_pointer 指针可以将这些 undo log 串成一个链表，这个链表就被称为版本链；

undo log 两大作用：

* **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（多版本并发控制）关键因素之一**。MVCC 是通过 ReadView + undo log 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（普通 select 语句）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。

## 为什么需要 Buffer Pool？

Innodb 存储引擎设计了一个**缓冲池（Buffer Pool）**，来提高数据库的读写性能。

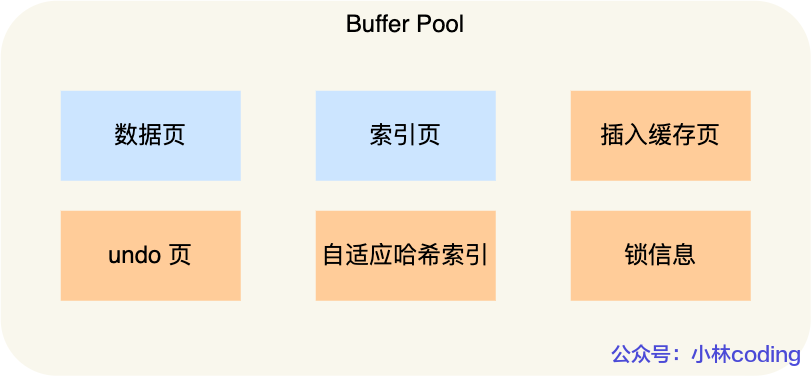


* 当**读取数据**时，如果数据存在于 Buffer Pool 中，客户端就会直接读取 Buffer Pool 中的数据，否则再去磁盘中读取。
* 当**修改数据**时，如果数据存在于 Buffer Pool 中，那直接修改 Buffer Pool 中数据所在的页，然后将其设置为**脏页**，为了减少磁盘I/O，**不会立即将脏页写入磁盘，而是由后台线程选择一个合适的时机将脏页写入到磁盘**。

### Buffer Pool 缓存什么？

Buffer Pool 同样需要按「页」来划分。MySQL 启动的时候，**InnoDB 会为 Buffer Pool 申请一片连续的内存空间，然后按照默认的16KB的大小划分出一个个的页， Buffer Pool 中的页就叫做缓存页**。查询一条记录时，InnoDB 是会**把整个页的数据加载到 Buffer Pool 中**，将页加载到 Buffer Pool 后，再通过页里的「页目录」去定位到某条具体的记录。

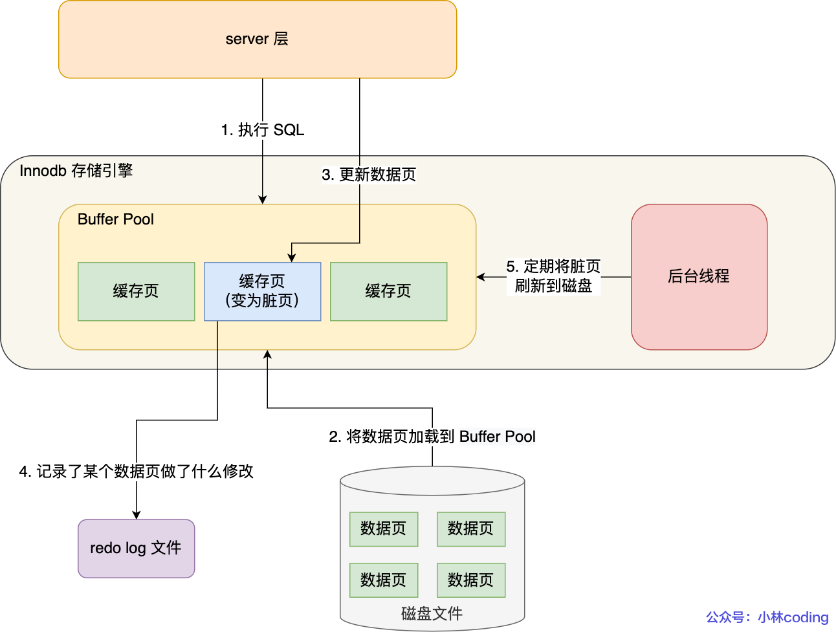
Buffer Pool的内容：



## 为什么需要 redo log ？

**WAL 技术： MySQL 的写操作并不是立刻写到磁盘上，而是先写日志，然后再在合适的时间再写到磁盘上**。

为了防止**断电**导致数据丢失的问题，当有一条记录需要更新的时候，InnoDB 引擎就会先更新内存（同时标记为脏页），然后**将本次对这个页的修改以 redo log 的形式记录下来**。后续InnoDB 引擎会在适当的时候，由后台线程将缓存在 Buffer Pool 的脏页刷新到磁盘里，这就是 **WAL （Write-Ahead Logging）技术**。



什么是 redo log？

redo log 是物理日志，**记录了某个数据页做了什么修改**，比如**对 XXX 表空间中的 YYY 数据页 ZZZ 偏移量的地方做了AAA 更新**，**每当执行一个事务**就会产生这样的一条或者多条物理日志。

在事务提交时，只要先将 redo log 持久化到磁盘即可，可以不需要等到将缓存在 Buffer Pool 里的脏页数据持久化到磁盘。

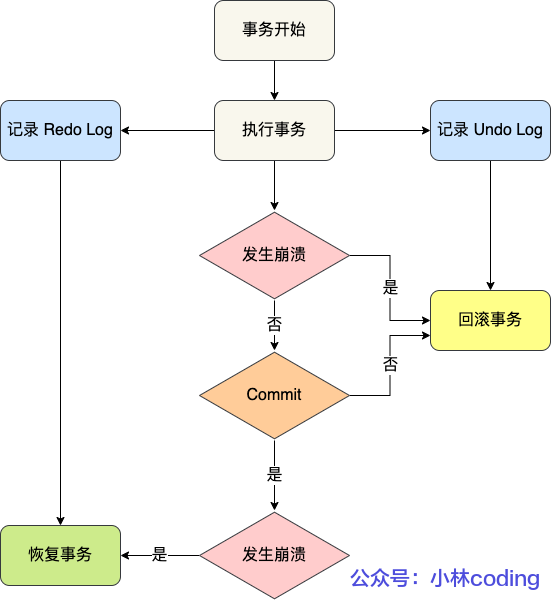
当系统崩溃时，虽然脏页数据没有持久化，但是 redo log 已经持久化，接着 MySQL 重启后，可以**根据 redo log 的内容，将所有数据恢复到最新的状态**。

被修改 Undo 页面，需要记录对应 redo log 吗？

undo log 会写入 Buffer Pool 中的 Undo 页面，**在内存修改该 Undo 页面后，需要记录对应的 redo log**。

redo log 和 undo log 区别在哪？

* redo log 记录了此次事务「**完成后**」的数据状态，记录的是更新**之后**的值；
* undo log 记录了此次事务「**开始前**」的数据状态，记录的是更新**之前**的值；



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**。**磁盘的「顺序写」比「随机写」高效的多**，因此 redo log 写入磁盘的开销更小。

产生的 redo log 是直接写入磁盘的吗？

执行一个事务的过程中，产生的 **redo log 也不是直接写入磁盘的**，因为这样会产生大量的 I/O 操作，而且磁盘的运行速度远慢于内存。redo log 也有自己的缓存 **redo log buffer**，每当产生一条 redo log 时，会先写入到 redo log buffer。

### redo log 什么时候刷盘？

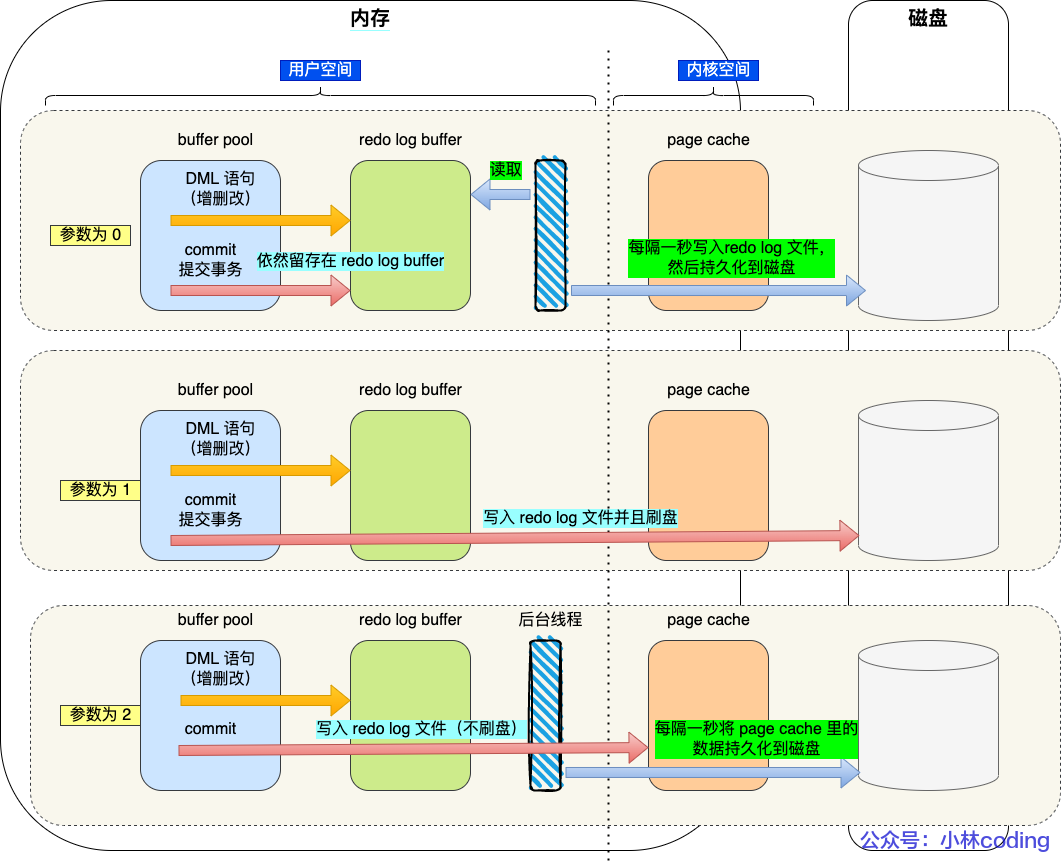
主要有下面几个时机：

* **MySQL 正常关闭**时；
* 当 redo log buffer 中记录的**写入量大于 redo log buffer 内存空间的一半**时，会触发落盘；
* InnoDB 的**后台线程每隔 1 秒**，将 redo log buffer 持久化到磁盘。
* 每次**事务提交**时都将缓存在 redo log buffer 里的 redo log 直接持久化到磁盘（这个策略可由 innodb\_flush\_log\_at\_trx\_commit 参数控制）。

innodb\_flush\_log\_at\_trx\_commit 参数控制的是什么？

**innodb\_flush\_log\_at\_trx\_commit** 参数可取的值有：0、1、2，**默认值为 1**

* 当设置该**参数为 0 时**，表示每次事务提交时 ，还是**将 redo log 留在 redo log buffer 中** ，该模式下在**事务提交时不会主动触发写入磁盘的操作**。
* 当设置该**参数为 1 时**，表示每次事务提交时，都**将缓存在 redo log buffer 里的 redo log 直接持久化到磁盘（先写到redo log文件然后马上刷盘）**，这样可以保证 MySQL 异常重启之后数据不会丢失。
* 当设置该**参数为 2 时**，表示每次事务提交时，都只是把缓存在 redo log buffer 里的 redo log **写到 redo log 文件，注意写入到「 redo log 文件」并不意味着写入到了磁盘**，而是写入到了操作系统的文件缓存（Page Cache）。

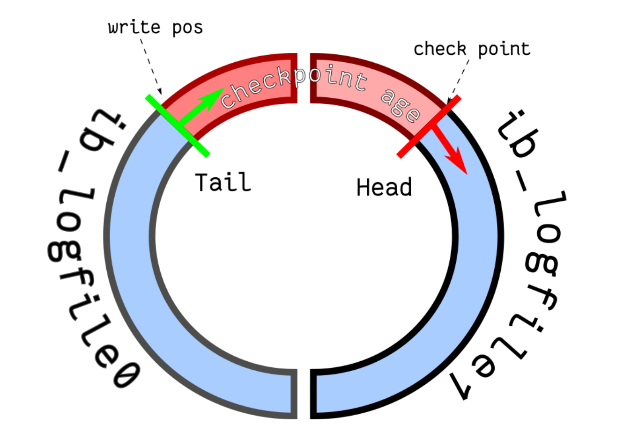


* 数据**安全性**：参数 1 > 参数 2 > 参数 0
* 写入**性能**：参数 0 > 参数 2> 参数 1

### redo log 文件写满了怎么办？

默认情况下， InnoDB 存储引擎有 1 个重做日志文件组( redo log Group），「重做日志文件组」由有 2 个redo log 文件组成，这两个 redo 日志的文件名叫 ：ib\_logfile0 和 ib\_logfile1 。

重做日志文件组是以**循环写**的方式工作的，**一个写满了，从另一个开头开始写**。随着系统运行，**Buffer Pool 的脏页刷新到了磁盘中，那么 redo log 中对应的记录也就没用了**，这时候我们擦除这些旧记录，以腾出空间记录新的更新操作。InnoDB 用 **write pos 表示 redo log 当前记录写到的位置**，用 **checkpoint 表示当前要擦除的位置**



**注意写入到「 redo log 文件」并不意味着写入到了磁盘**

如果 write pos 追上了 checkpoint，就意味着 **redo log 文件满了，这时 MySQL 不能再执行新的更新操作，也就是说 MySQL 会被阻塞**（*因此所以针对并发量大的系统，适当设置 redo log 的文件大小非常重要*），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。

**一次 checkpoint 的过程就是脏页刷新到磁盘中变成干净页，然后标记 redo log 哪些记录可以被覆盖的过程。**

## 为什么需要 binlog ？

MySQL 在完成一条**更新操作后，Server 层还会生成一条 binlog**，等之后事务**提交**的时候，会将该事物执行过程中产生的**所有 binlog 统一写入 binlog 文件**。

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，**不会记录查询类**的操作，比如 **SELECT** 和 **SHOW** 操作。

### redo log 和 binlog 的区别？

1、适用对象不同：

* binlog 是 MySQL 的 **Server 层实现**的日志，**所有存储引擎**都可以使用；
* redo log 是 **Innodb 存储引擎实现**的日志；

2、文件格式不同：

* 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** 都会被记录到 binlog 中（**相当于记录了逻辑操作**），主从复制中 slave 端再根据 SQL 语句重现。但 STATEMENT 有动态函数的问题，比如uuid 或者 now 这些函数，在主库上执行的结果并不是在从库执行的结果，这种随时在变的函数会**导致复制的数据不一致**；
  + **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。缺点是每行数据的变化结果都会被记录，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 是物理日志，**记录的是在某个数据页做了什么修改**，比如对 XXX 表空间中的 YYY 数据页 ZZZ 偏移量的地方做了AAA 更新；

3、写入方式不同：

* 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**以前的日志，保存的是全量的日志。
* redo log 是**循环写**，日志空间大小是固定，**全部写满就从头开始**，保存未被刷入磁盘的脏页日志。

4、用途不同：

* binlog 用于**备份恢复、主从复制**；
* redo log 用于**掉电等故障恢复**。

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？

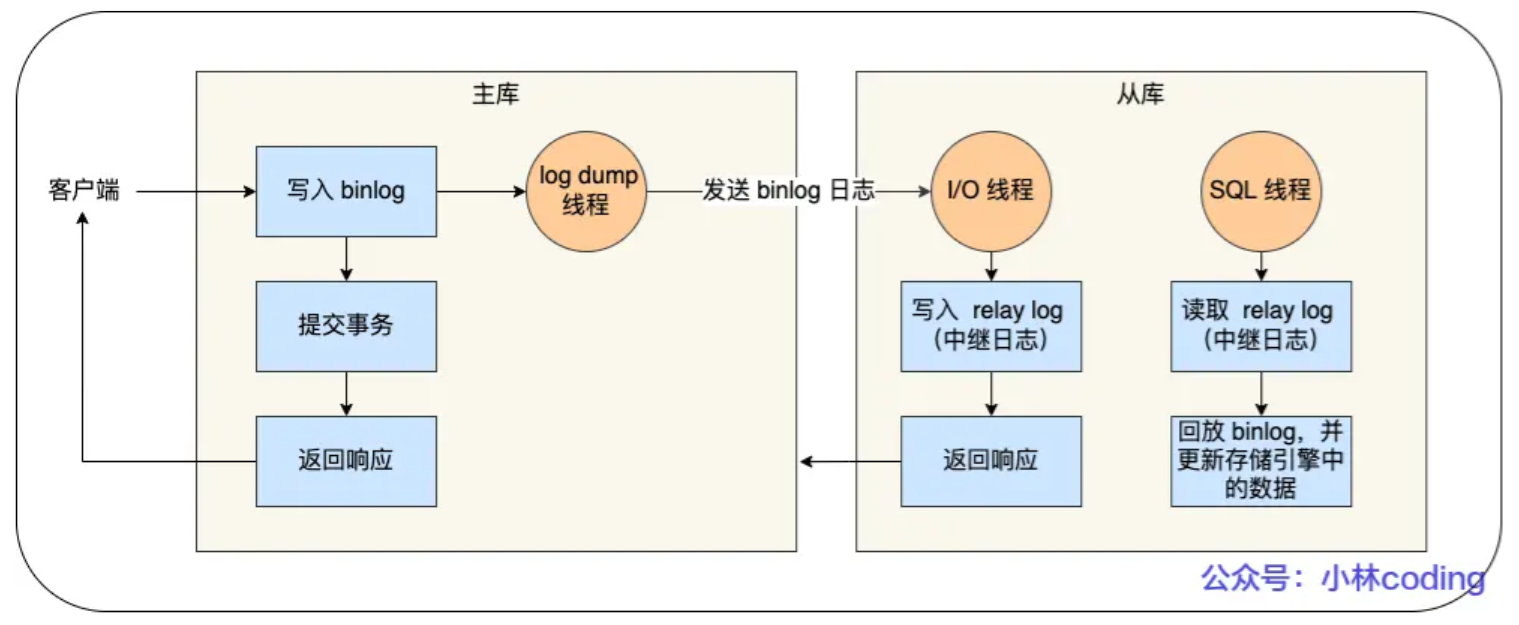
**不可以使用 redo log 文件恢复，只能使用 binlog 文件恢复。**

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**全量的日志**，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。

### 主从复制是怎么实现？

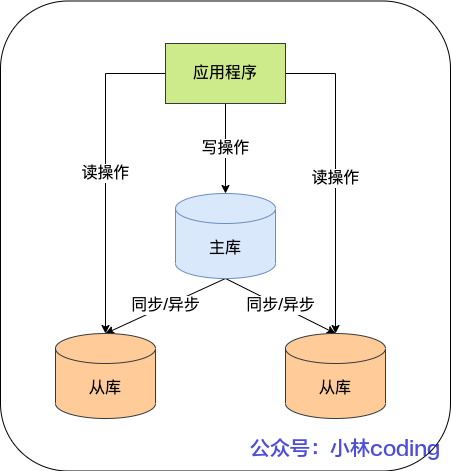
MySQL 的主从复制依赖于 binlog ，也就是记录 MySQL 上的所有变化并以**二进制**形式保存在磁盘上。**复制的过程就是将 binlog 中的数据从主库传输到从库上**。这个过程一般是**异步**的。



MySQL 集群的主从复制过程分为 3 个阶段：

* **写入 Binlog**：MySQL 主库在收到客户端提交事务的请求之后，会先写入 binlog，再提交事务，更新存储引擎中的数据，事务提交完成后，返回给客户端“操作成功”的响应。
* **同步 Binlog**：从库会创建一个专门的 I/O 线程，连接主库的 **log dump 线程**，来接收主库的 binlog 日志，再把 binlog 信息写入 **relay log 中继日志**里，再返回给主库“复制成功”的响应。
* **回放 Binlog**：从库会创建一个用于回放 binlog 的线程，去读 relay log 中继日志，然后回放 binlog 更新存储引擎中的数据，最终**实现主从的数据一致性**。

**在完成主从复制之后，可以在写数据时只写主库，在读数据时只读从库，这样即使写请求会锁表或者锁记录，也不会影响读请求的执行。**



从库是不是越多越好？

从库数量增加，**主库也要创建同样多的 log dump 线程来处理复制的请求，对主库资源消耗比较高，同时还受限于主库的网络带宽**。在实际使用中，一个主库一般跟 2～3 个从库（**1 套数据库，1 主 2 从 1 备主**），这就是一主多从的 MySQL 集群结构。

MySQL 主从复制有哪些模型？

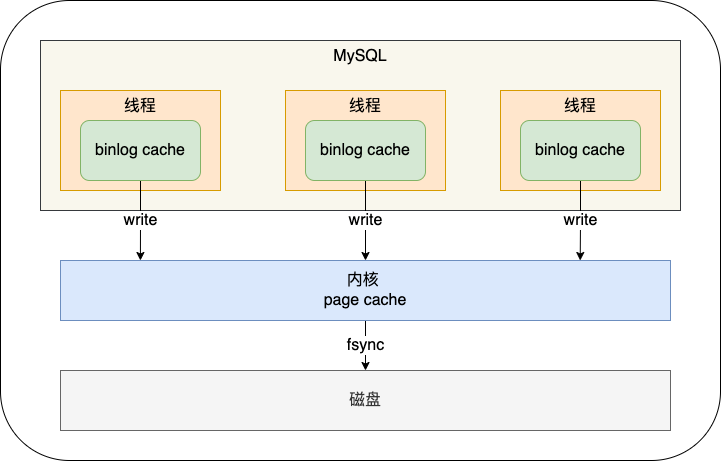
* **异步复制**（**默认**模型）：MySQL 主库提交事务的线程并**不会等待 binlog 同步到各从库，就返回客户端结果**。这种模式一旦主库宕机，数据就会发生丢失。
* **同步复制**：MySQL 主库提交事务的线程要**等待所有从库的复制成功响应，才返回客户端结果**。这种方式在实际项目中基本上没法用，原因有两个：一是**性能很差**，因为要复制到所有节点才返回响应；二是**可用性也很差**，主库和所有从库任何一个数据库出问题，都会影响业务。
* **半同步复制**：MySQL 5.7 版本之后增加的一种复制方式，介于两者之间，事务线程不用等待所有的从库复制成功响应，只要**一部分复制成功响应回来就行**，比如一主二从的集群，只要数据成功复制到任意一个从库上，主库的事务线程就可以返回给客户端。这种**半同步复制的方式，兼顾了异步复制和同步复制的优点，即使出现主库宕机，至少还有一个从库有最新的数据，不存在数据丢失的风险**。

### binlog 什么时候刷盘？

事务执行过程中，先把日志写到 binlog cache（Server 层的 cache），事务提交的时候，再把 binlog cache 写到 binlog 文件中（只是写入page cache而不是写入磁盘）。

什么时候 binlog cache 会写到 binlog 文件？

**事务提交时**



**每个线程有自己的 binlog cache**，但是最终都**写到同一个 binlog 文件**：

* write指把日志写入到 binlog 文件，但是并没有把数据持久化到磁盘。
* fsync将数据持久化到磁盘，涉及磁盘 I/O，所以频繁的 fsync 会导致磁盘的 I/O 升高。

MySQL提供一个 sync\_binlog 参数来控制数据库的 binlog 刷到磁盘上的频率：

* sync\_binlog = 0 的时候，表示每次提交事务都**只 write，不 fsync，后续交由操作系统决定何时将数据持久化到磁盘**；
* sync\_binlog = 1 的时候，表示每次提交事务都会 **write，然后马上执行 fsync**；
* sync\_binlog =N(N>1) 的时候，表示每次提交事务都 **write，但累积 N 个事务后才 fsync**

在MySQL中系统**默认的设置是 sync\_binlog = 0**，也就是不做任何强制性的磁盘刷新指令，这时候的**性能是最好的，但是风险也是最大**的。因为一旦主机发生异常重启，还没持久化到磁盘的数据就会丢失。

而当 sync\_binlog 设置为 1 的时候，是**最安全但是性能损耗最大**的设置。因为当设置为 1 的时候，即使主机发生异常重启，最多丢失一个事务的 binlog，而已经持久化到磁盘的数据就不会有影响，不过就是对写入性能影响太大。

如果能容少量事务的 binlog 日志丢失的风险，为了提高写入的性能，一般会 sync\_binlog 设置为 100~1000 中的某个数值。

## update语句的执行过程

当**优化器分析出成本最小的执行计划**后，执行器就按照执行计划开始进行更新操作。

具体更新一条记录 UPDATE t\_user SET name = 'xiaolin' WHERE id = 1; 的流程如下:

1. 执行器负责具体执行，会调用存储引擎的接口，通过主键索引树搜索获取 id = 1 这一行记录：
   * 如果 id=1 这一行所在的数据页本来就在 buffer pool 中，就直接返回给执行器更新；
   * 如果记录不在 buffer pool，将数据页从磁盘**读入到 buffer pool**，返回记录给执行器。
2. 执行器得到聚簇索引记录后，会看一下更新前的记录和更新后的记录是否一样：
   * 如果**一样的话就不进行后续更新流程**；
   * 如果**不一样**的话就把**更新前的记录和更新后的记录都当作参数**传给 InnoDB 层，让 InnoDB 真正的执行更新记录的操作；
3. 开启事务， InnoDB 层**更新记录前**，首先要记录相应的 undo log，因为这是更新操作，需要**把被更新的列的旧值记下来**，也就是要生成一条 **undo log**，undo log 会写入 Buffer Pool 中的 Undo 页面，在内存**修改该 Undo 页面后**，需要记录对应的 **redo log**。
4. InnoDB 层开始更新记录，会先更新内存（同时标记为脏页），然后将记录写到 redo log 里面，这个时候更新就算完成了。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。（**WAL 技术**）
5. 至此，一条记录更新完了。
6. 在一条更新语句执行完成后，然后开始记录该语句对应的 **binlog**，此时记录的 binlog 会被保存到 binlog cache，并没有刷新到硬盘上的 binlog 文件，在事务提交时才会统一将该事务运行过程中的所有 binlog 刷新到硬盘。
7. 事务提交，剩下的就是「两阶段提交」的事情。

## 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**。

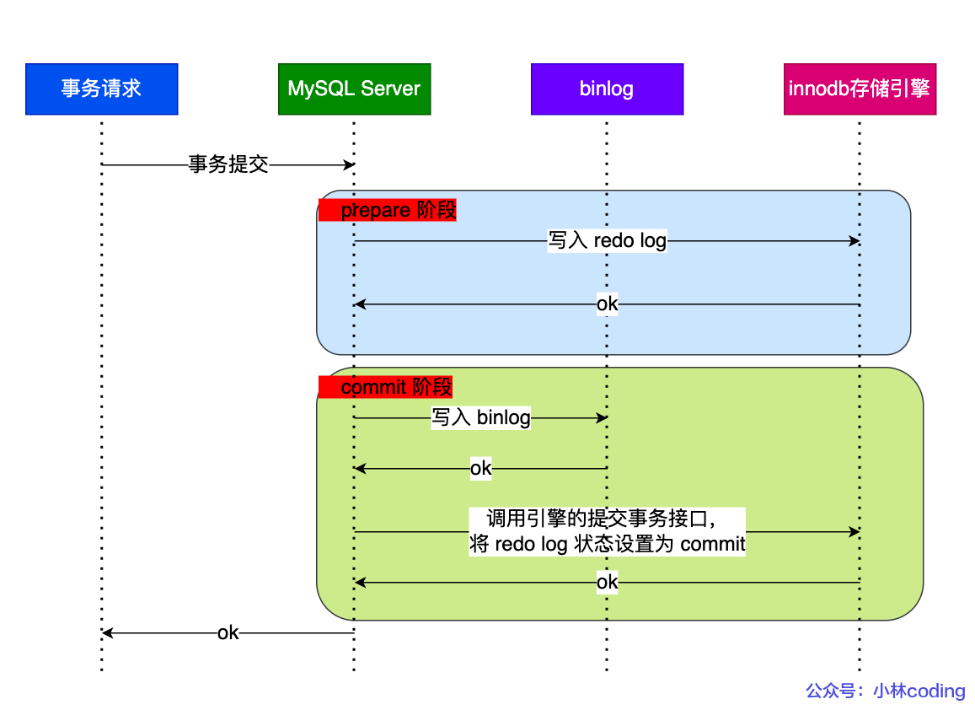
redo log 影响主库的数据，binlog 影响从库的数据。redo log 和 binlog 必须保持一致才能保证**主从数据一致**。

**MySQL 为了避免出现两份日志之间的逻辑不一致的问题，使用了「两阶段提交」来解决**，两阶段提交其实是分布式事务一致性协议，它可以**保证多个逻辑操作要不全部成功，要不全部失败**，不会出现半成功的状态。

**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（Prepare）阶段」和「提交（Commit）阶段」**，每个阶段都由**协调者**（Coordinator）和**参与者**（Participant）共同完成

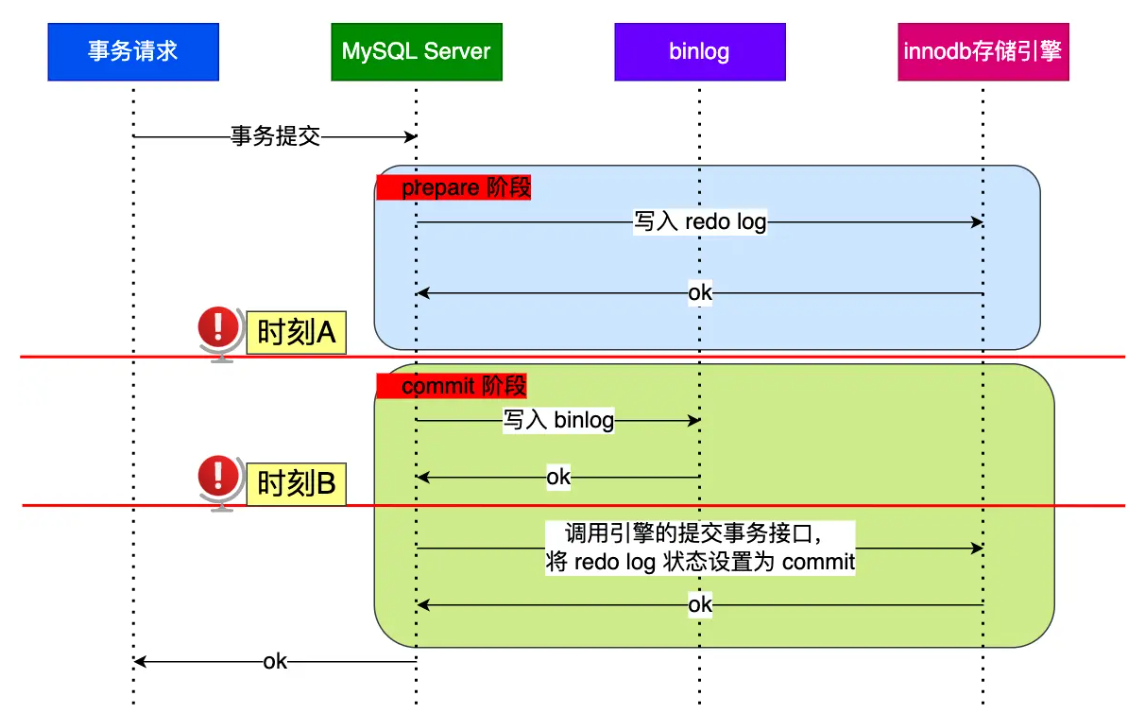
### 两阶段提交的过程

当客户端执行 commit 语句或者在自动提交的情况下，MySQL 内部开启一个 **XA 事务**，**分两阶段来完成 XA 事务的提交**，如下图：



* **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（innodb\_flush\_log\_at\_trx\_commit = 1 的作用）；
* **commit 阶段**：把 XID 写入到 binlog，然后将 binlog 持久化到磁盘（sync\_binlog = 1 的作用），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，因为只要 binlog 写磁盘成功，就算 redo log 的状态还是 prepare 也没有关系，一样会被认为事务已经执行成功；

### 异常重启会出现什么现象？



在 MySQL 重启后会按顺序扫描 redo log 文件，**碰到处于 prepare 状态的 redo log，就拿着 redo log 中的 XID 去 binlog 查看是否存在此 XID**：

* **如果 binlog 中没有当前内部 XA 事务的 XID，说明 redo log 完成刷盘，但是 binlog 还没有刷盘，则回滚事务**。对应**时刻 A** 崩溃恢复的情况。
* **如果 binlog 中有当前内部 XA 事务的 XID，说明 redo log 和 binlog 都已经完成了刷盘，则提交事务**。对应**时刻 B** 崩溃恢复的情况。

**两阶段提交是以 binlog 写成功为事务提交成功的标识**

事务没提交的时候，redo log 会被持久化到磁盘吗？

会的。见[#redo log 什么时候刷盘？](#_redo_log_什么时候刷盘？)

### 两阶段提交有什么问题？

* **磁盘 I/O 次数高**：对于“双1”配置，每个事务提交都会进行两次 fsync（刷盘），一次是 **redo log 刷盘**，另一次是 **binlog 刷盘**。
* **锁竞争激烈**：两阶段提交虽然能够保证「单事务」两个日志的内容一致，但在「多事务」的情况下，却不能保证两者的提交顺序一致，因此，在两阶段提交的流程基础上，还需要加一个**锁**来保证提交的原子性，从而保证多事务的情况下，两个日志的提交顺序一致。

#### 组提交

**MySQL 引入了 binlog 组提交（group commit）机制，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；

**每个阶段都有一个队列**，每个队列有锁进行保护，而不再锁住提交事务的整个过程。**锁粒度减小了，这样就使得多个阶段可以并发执行，从而提升效率**。

MySQL 5.7开始有 **redo log 组提交。**在此之前每个事务各自执行 prepare 阶段，也就是各自将 redo log 刷盘，不能进行组提交。MySQL 5.7 版本中将 redo log 刷盘操作推迟到组提交的 flush 阶段，也就是说 **prepare 阶段融合在了 flush 阶段**。

针对“双 1” 配置，每个阶段的过程：

flush 阶段

**用于支撑 redo log 的组提交。**

第一个事务会成为 flush 阶段的 Leader，负责将同组事务的 redolog 刷盘（相当于prepare阶段）。完成了 prepare 阶段后，将这一组事务执行过程中产生的 binlog 写入 binlog 文件（调用 write，不会调用 fsync，**不会刷盘**）。

sync 阶段

**用于支持 binlog 的组提交**。

第一组事务的 binlog 写入到 binlog 文件后，**会等待一段时间**，等待的时长由 Binlog\_group\_commit\_sync\_delay 参数控制，**目的是为了组合更多事务的 binlog，然后再一起刷盘**

但是，在等待的过程中，如果事务的数量**提前达到**了 Binlog\_group\_commit\_sync\_no\_delay\_count 参数设置的值，就不用继续等待了，就**马上将 binlog 刷盘**

commit 阶段

调用引擎的提交事务接口，将 redo log 状态设置为 commit。

## MySQL 磁盘 I/O 很高，有什么优化的方法？

* **设置组提交的两个参数**： binlog\_group\_commit\_sync\_delay 和 binlog\_group\_commit\_sync\_no\_delay\_count 参数，延迟 binlog 刷盘的时机，从而减少 binlog 的刷盘次数。**没有丢失数据的风险**，因为 binlog 早被写入到 page cache
* 将 **sync\_binlog** 设置为大于 1 的值（比较常见是 100~1000），表示每次提交事务都 write，但**累积 N 个事务后才 fsync**，相当于延迟了 **binlog 刷盘**的时机。但是这样做的风险是，**主机掉电时会丢 N 个事务的 binlog 日志**。
* 将 **innodb\_flush\_log\_at\_trx\_commit** 设置为 2。表示每次事务提交时，都只是缓存在 redo log buffer 里的 redo log 写到 **redo log** 文件但不刷盘，而是让操作系统控制持久化到磁盘的时机。但是这样做的风险是，**主机掉电的时候会丢数据**。