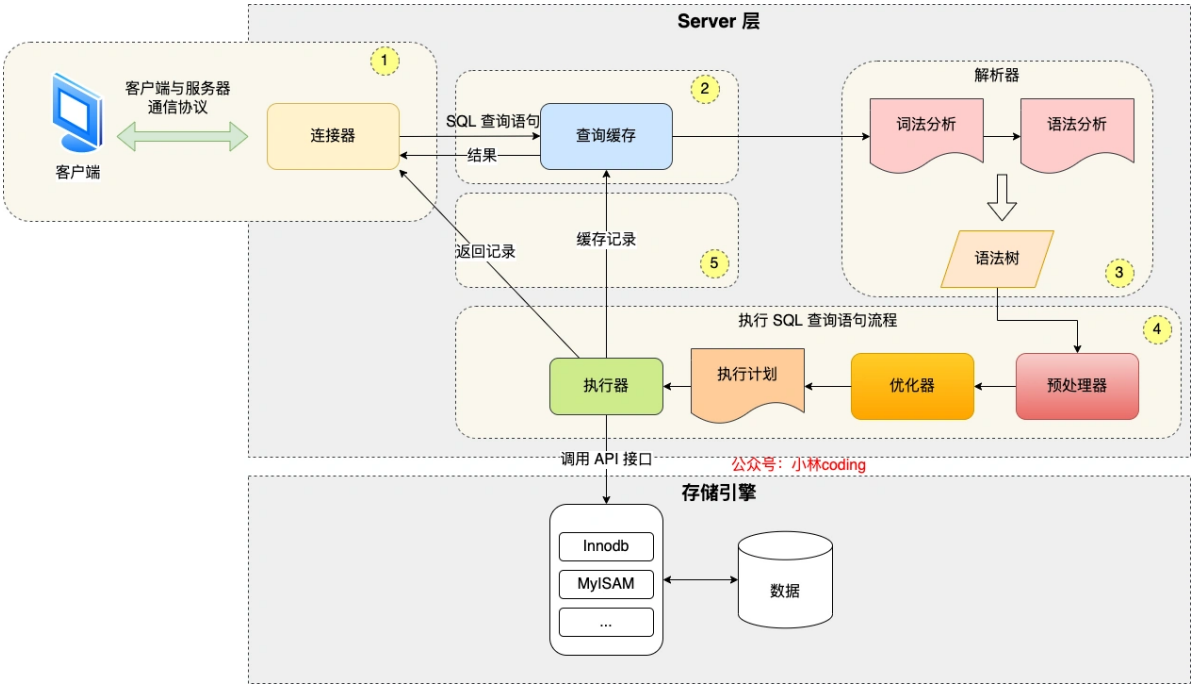
# MySQL笔记—日志篇

## 关于undo log、redo log、binlog有什么用?

引入：执行一条update语句的过程




- 客户端先通过**连接器建立连接**，连接器自会判断用户身份；
- 因为这是一条 update 语句，所以不需要经过查询缓存，但是表上有更新语句，是会把整个表的查询缓存清空的，所以说查询缓存很鸡肋，在 MySQL 8.0 就被移除这个功能了；
- **解析器**会通过词法分析识别出关键字 update，表名等等，构建出语法树，接着还会做语法分析，判断输入的语句是否符合 MySQL 语法；
- **预处理器**会判断表和字段是否存在；
- 优化器确定执行计划，因为 where 条件中的 id 是主键索引，所以决定要使用 id 这个索引；
- 执行器负责具体执行，找到这一行，然后更新。

### KeyPoint

连接器建立连接判断身份 -> 查询缓存 -> 解析器词法分析,语法分析 -> 预处理器判断表,字段是否存在 -> 优化器确定执行计划(走不走索引,走哪些索引) -> 执行器执行

在更新过程中会涉及到三个日志，分别是 undo log（回滚日志）、redo log（重做日志） 、binlog（归档日志）

- **undo log（回滚日志）**：是 **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）**：是 **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）**：是 Server 层生成的日志，主要**用于数据备份和主从复制**

# undo log具体作用?

我们在执行执行一条“增删改”语句的时候，MySQL 会**隐式开启事务**来执行“增删改”语句，执行完就自动提交事务

每次在事务执行过程中，都记录下**回滚时需要的信息到一个日志**里，那么在事务执行中途发生了**MySQL 崩溃**后，就不用担心无法回滚到事务之前的数据，我们可以**通过这个日志回滚到事务之前的数据**，而这个日志则是 undo log 日志

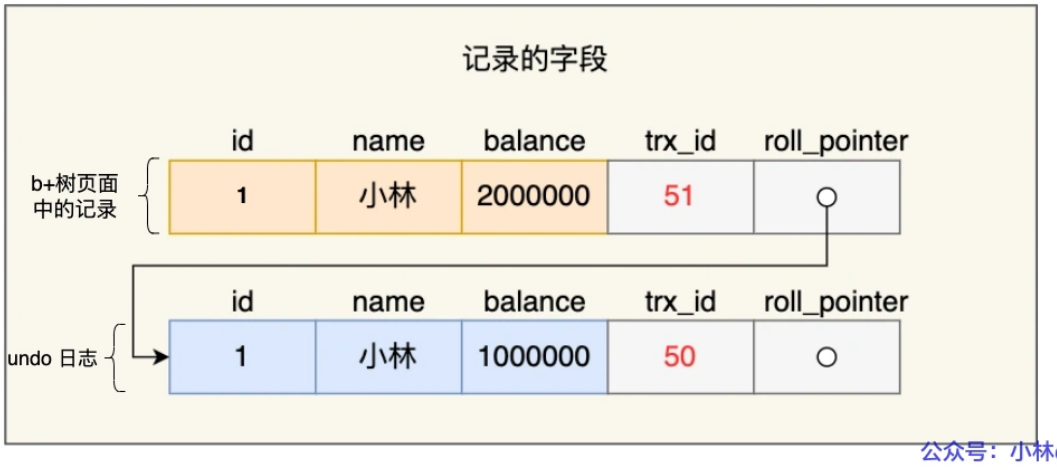
## undo log 具体写入信息

- 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**删掉**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。

## undo log 具体格式

每次写入 undo log 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 还有一个作用，通过 ReadView + undo log 实现 MVCC（多版本并发控制）**

读提交：在每次select都生成一个ReadView

可重复读：在每次事务开始前都生成一个ReadView

这两个隔离级别实现是通过「**事务的 Read View 里的字段**」和「**记录中的两个隐藏列（trx_id 和 roll_pointer）**」的比对，如果不满足可见行，就会顺着 **undo log 版本链里找到满足其可见性的记录**，从而控制并发事务访问同一个记录时的行为，这就叫 **MVCC（多版本并发控制）**

## undo log 两大作用总结

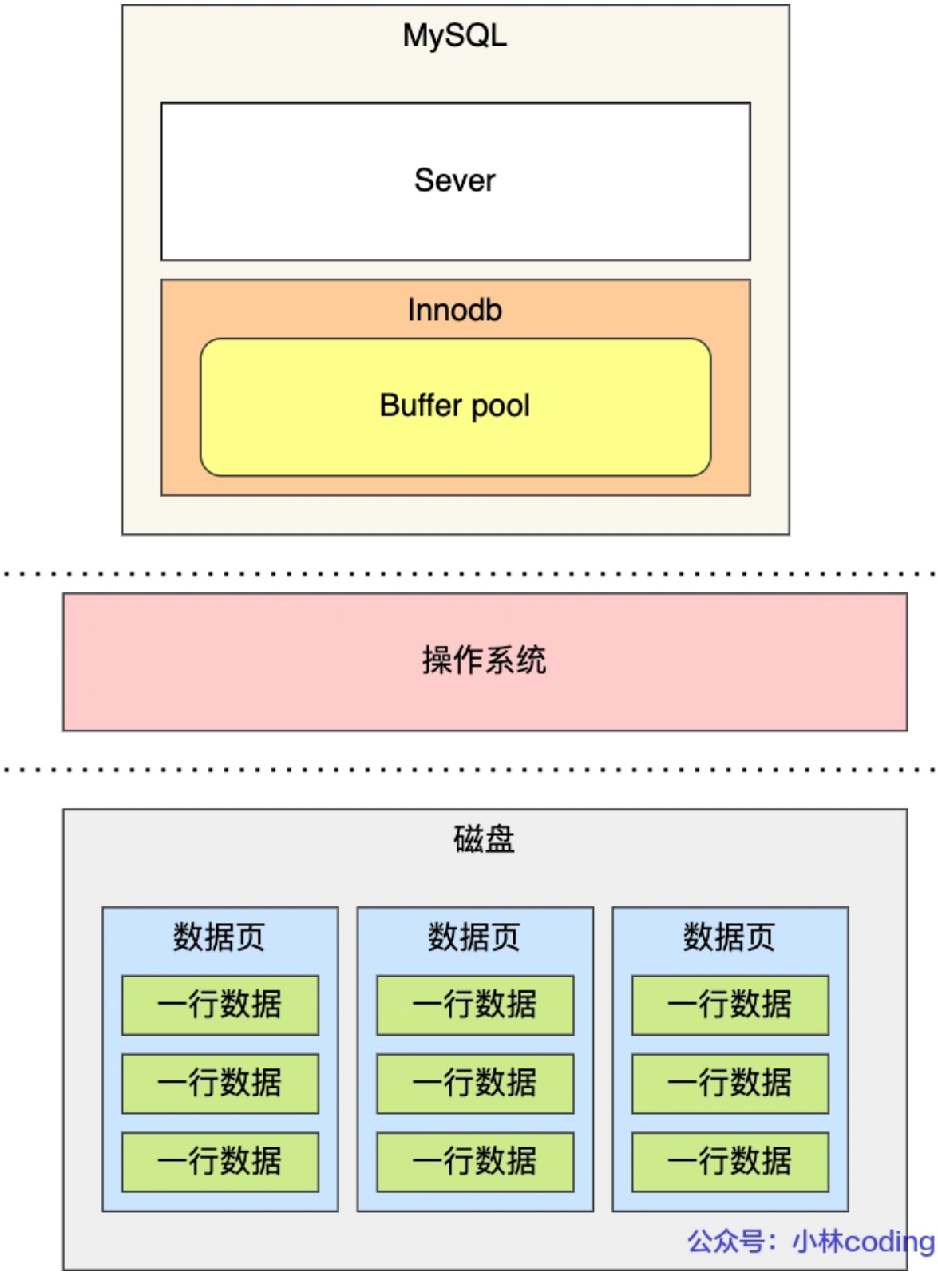
1. **实现事务回滚，保障事务的原子性**
2. **实现 MVCC（多版本并发控制）关键因素之一**

### 面试介绍：undo log

undo log 称为回滚日志，作用是在事务开始时，执行了增查改语句后，记录回滚时需要的信息到一个日志，这个信息在不同语句中有不同的内容，如果是增：主键id，删：旧记录，改：修改字段的旧记录。如果事务需要回滚，则依靠该日志进行回滚到事务开始的状态。它日志的格式为roll_point和trx_id，trx_id 记录事务id，roll_point 将这些 undo log 串成一个链表，**加上ReadView实现MVCC。**

## 关于 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。**和查询缓存不同**，该缓冲池不会在执行update语句时直接失效，并且是存储引擎层面的。



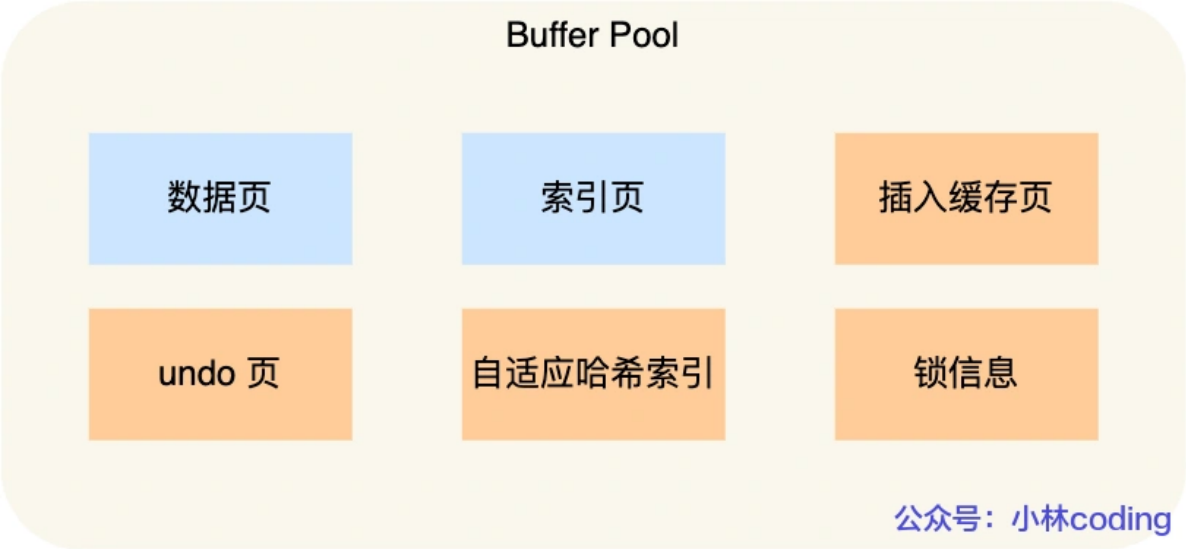
有了 Buffer Pool 后

- 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。

### Buffer Pool 缓存的内容?

**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。**此时这些缓存页都是空闲的，之后随着程序的运行，才会有磁盘上的页被缓存到 Buffer Pool 中。

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，就只需要缓冲一条记录吗?

查询一条记录，会将一个页都放入 Buffer Pool 中，再通过页里的页目录去定位具体记录

### 面试介绍：Buffer Pool

**Buffer Pool 是位于Innodb存储引擎中的缓冲池**，和mysql8.0废弃的查询缓存不是一个东西，这个缓冲池不会在执行update语句时直接失效，是为了提高查询效率才引入的这个缓冲池。读取数据时如果数据存在该缓冲池则直接返回，如果不存在则查询磁盘，**并将记录所在的页存入该缓冲池。**修改时会将该数据所在缓冲池中的页设置为脏页，由后台线程定期刷盘

## redo log 具体作用?

## 引入

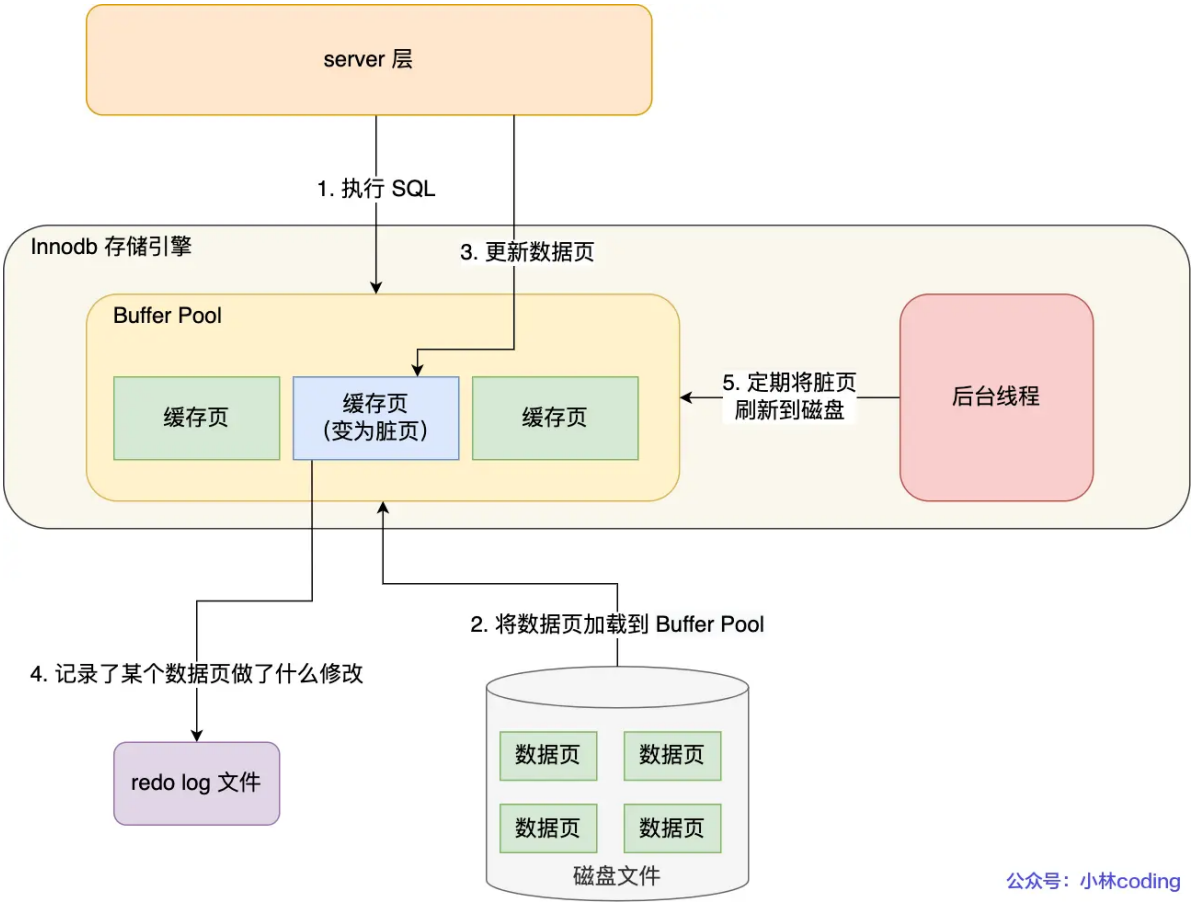
为了防止断电导致数据丢失的问题，当有一条记录需要更新的时候，InnoDB 引擎就会先更新内存（同时标记为脏页），然后将本次对这个页的修改以 redo log 的形式记录下来，**这个时候更新就算完成了。**

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**（在undo log，redo log都会用到这种技术）

**WAL 技术指的是， MySQL 的写操作并不是立刻写到磁盘上，而是先写日志，然后在合适的时间再写到磁盘上**

**WAL技术优点**

1. 读和写可以完全地并发执行，不会互相阻塞（但是写之间仍然不能并发）。
2. WAL 在大多数情况下，拥有更好的性能（因为无需每次写入时都要写两个文件）。
3. 磁盘 I/O 行为更容易被预测。（从数据随机读写变为顺序读写）
4. 使用更少的 fsync()操作，减少系统脆弱的问题。




**KeyPoint**

执行sql -> 将磁盘中数据页加载到 Buffer Pool 中 -> 更新该页并设置为脏页 -> redo log记录修改 -> 后台进程定时将脏页刷到磁盘

## 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

## 修改 Undo 页面，需要记录对应 redo log 吗?

需要, 修改undo 页面也要记录redo log

## redo log 和 undo log 区别?

这两种日志是属于 InnoDB 存储引擎的日志，它们的区别在于：

- 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；

所以有了 redo log，再通过 WAL 技术，InnoDB 就可以保证即使数据库发生异常重启，之前已提交的记录都不会丢失，这个能力称为 **crash-safe**（崩溃恢复）。可以看出来， **redo log 保证了事务四大特性中的持久性。**

## 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**

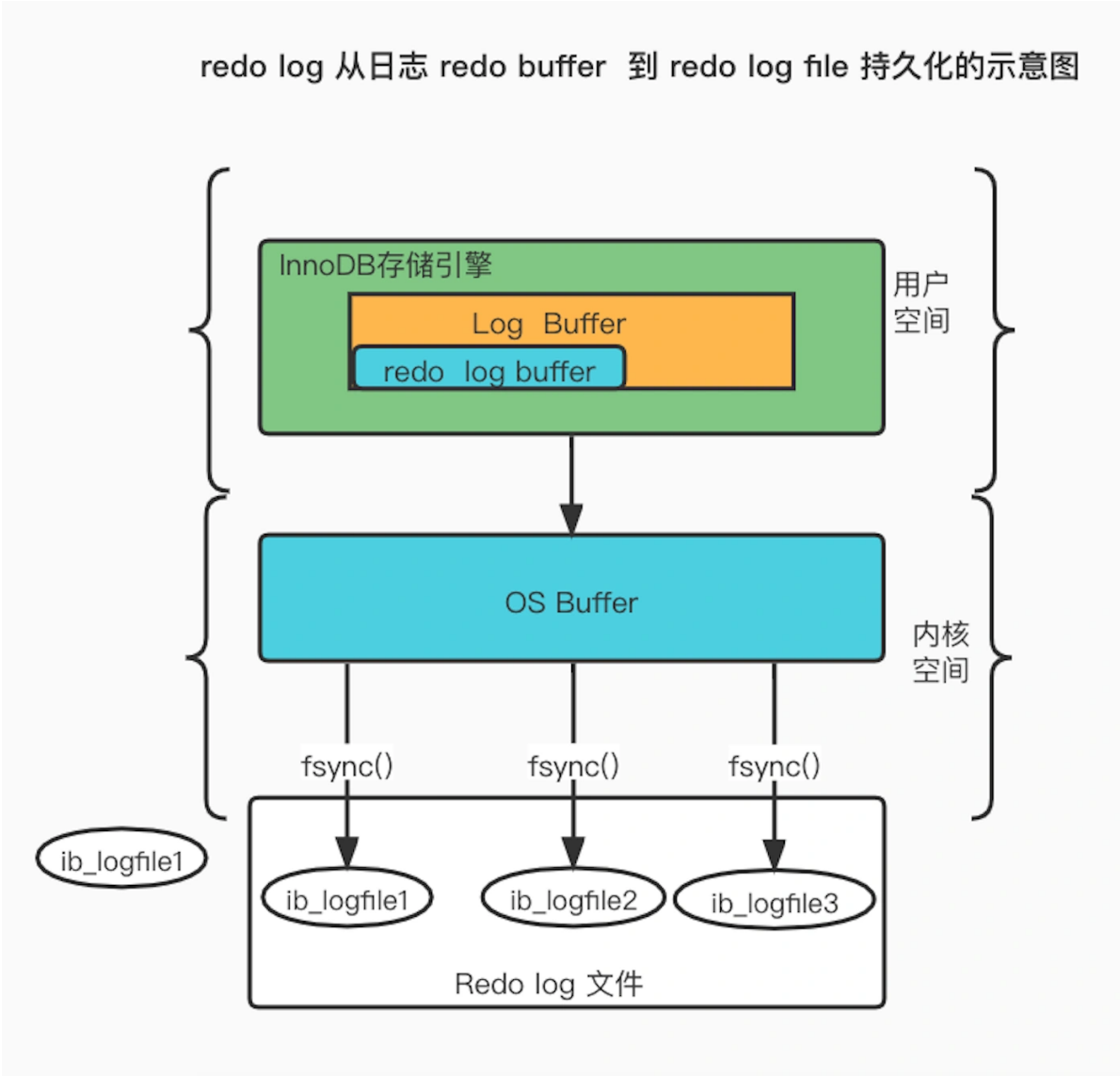
## redo log 两大作用总结

1. **实现事务的持久性，让 MySQL 有 crash-safe (故障恢复)的能力**，能够保证 MySQL 在任何时间段突然崩溃，重启后之前已提交的记录都不会丢失；
2. **将写操作从「随机写」变成了「顺序写」**，提升 MySQL 写入磁盘的性能。

## 产生的 redo log 是直接写入磁盘的吗?

不是

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**，后续在持久化到磁盘如下图




redo log buffer **默认大小 16 MB**，可以通过 `innodb_log_Buffer_size` 参数动态的调整大小,使**大事务也可以不必直接写入磁盘**

## redo log 刷盘时机?

- MySQL正常关闭
- redo log buffer 内存使用达到一半时
- InnoDB 的后台线程每隔 1 秒, 将 redo log buffer 持久化到磁盘。
- 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?**

**默认 :** 一个事务提交后便将 redo log buffer 中的 redo log **顺序写**到磁盘

- **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**，在事务提交时不会主动触发写入磁盘
- **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**，这样可以保证 MySQL 异常重启之后数据不会丢失。
- **参数为 2 时**，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，注意写入到「 redo log 文件」并不意味着写入到了磁盘**，因为操作系统的文件系统中有个 Page Cache，Page Cache 是专门用来缓存文件数据的，所以写入「 redo log文件」意味着**写入到了操作系统的文件缓存**(由**fync()函数**控制写入磁盘时机)。

即: **0不变, 1 直接到磁盘, 2到操作系统的缓存中**

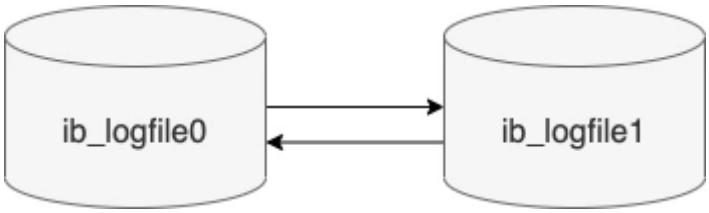
**innodb_flush_log_at_trx_commit 为 0 和 2 的时候，什么时候才将 redo log 写入磁盘?**

InnoDB 的后台线程每隔 1 秒：

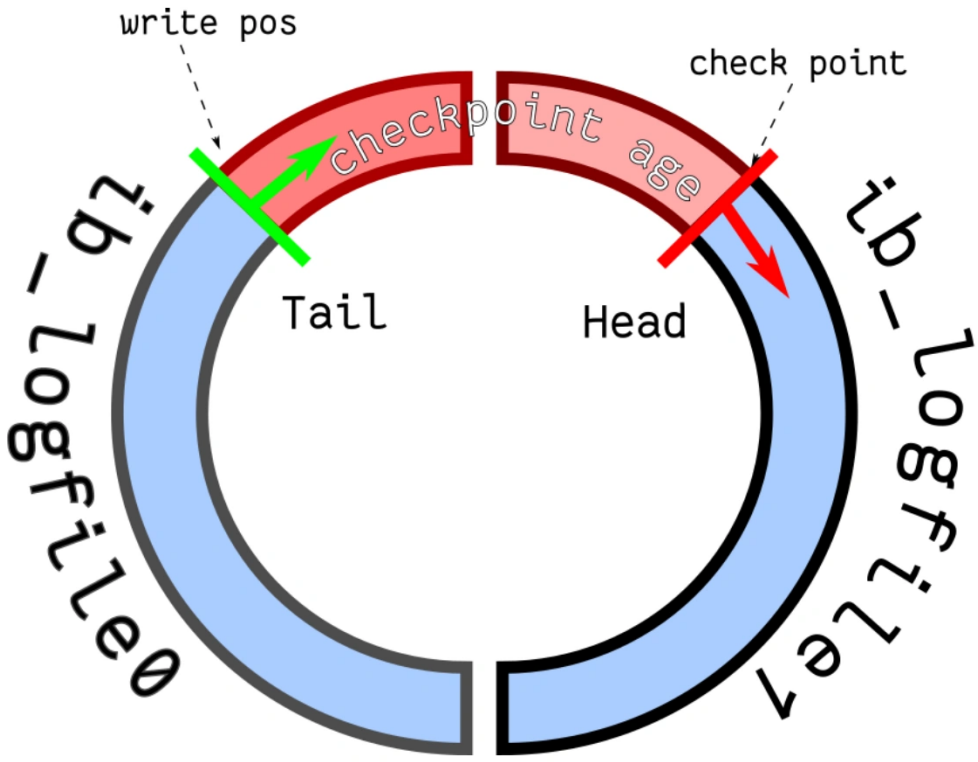
- 针对参数 0 ： 会把缓存在 redo log buffer 中的 redo log ， 通过调用 `write()` 写到操作系统的 Page Cache，然后调用 `fsync()` 持久化到磁盘。**所以参数为 0 的策略，MySQL 进程的崩溃会导致上一秒钟所有事务数据的丢失**;
- 针对参数 2 ： 调用 fsync，将缓存在操作系统中 Page Cache 里的 redo log 持久化到磁盘。**所以参数为 2 的策略，较取值为 0 情况下更安全，因为 MySQL 进程的崩溃并不会丢失数据，只有在操作系统崩溃或者系统断电的情况下，上一秒钟所有事务数据才可能丢失。**

## 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` ，**redo log 采用循环写的方式**，从头开始写，写到末尾就又回到开头，相当于一个环形




我们知道 redo log 是为了防止 Buffer Pool 中的脏页丢失而设计的，那么如果随着系统运行，**Buffer Pool 的脏页刷新到了磁盘**中，那么 redo log 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




- write pos 和 checkpoint 的移动都是顺时针方向;
- write pos ~ checkpoint 之间的部分（图中的红色部分），用来**记录新的更新操作**;
- check point ~ write pos 之间的部分（图中蓝色部分）：**待落盘的脏数据页记录**;

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**，所以在并发量大时设置好 redo log 文件大小非常重要

### 面试介绍：redo log

**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**，然后将本次对这个页的修改以 redo log 的形式记录下来，后来会在适当时候将缓冲池中的数据刷到磁盘，这种**先写到日志后写入磁盘的行为称为WAL**，而其中实现记录数据的日志就是redo log。

- redolog能改善磁盘随机读写的问题，因为它是顺序写入磁盘的，比数据的随机写入高效的多
- 实现事务的持久性，让 MySQL 有 crash-safe (故障恢复)的能力

刷盘时机：

- MySQL退出时
- 每隔一秒时
- redo log buffer内存使用一半时
- innodb_flush_log_at_trx_commit参数控制
    - 0：事务提交还是在redo log buffer中, 每隔一秒会被写入操作系统缓存中,后续同2
    - 1：事务提交直接持久化
    - 2：事务提交写到redo log文件，即操作系统缓存中，有fsync() 调用时机决定
-

# 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

### 为什么有了 binlog， 还要有 redo log?

最开始 MySQL 里并没有 InnoDB 引擎，MySQL 自带的引擎是 MyISAM，但是 **MyISAM 没有 crash-safe 的能力**，binlog 日志只能用于归档。

redo log 和 bin log 的一个很大的区别就是，**一个是循环写，一个是追加写。**也就是说 redo log **只会记录未刷入磁盘的日志**，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。

bin log 是追加日志，保存的是全量的日志。这就会导致一个问题，那就是没有标志能让 InnoDB 从 bin log 中判断哪些数据已经刷入磁盘了，哪些数据还没有。

举个例子，bin log 记录了两条日志：

```
记录 1：给 id = 1 这一行的 age 字段加 1
记录 2：给 id = 1 这一行的 age 字段加 1
```

假设在记录 1 刷盘后，记录 2 未刷盘时，数据库崩溃。重启后，只通过 bin log 数据库是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 = 1 这行数据来说，都是不对的。

## *redo log 和 binlog 有什么区别?

*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**，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**，比如你用了 **uuid 或者 now** 这些函数，你在主库上执行的结果并不是你在从库执行的结果，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**（这种格式的日志，就不能称为逻辑日志了），不会出现 STATEMENT 下动态函数的问题。但 ROW 的缺点是每行数据的变化结果都会被记录，比如**执行批量 update 语句，更新多少行数据就会产生多少条记录**，使 binlog 文件过大，而在 **STATEMENT 格式下只会记录一个 update 语句**而已；
    - 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；

  即：**statement时会记录语句，但会出现动态函数的问题；row时记录的是记录情况，占用内存量大**
- 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：

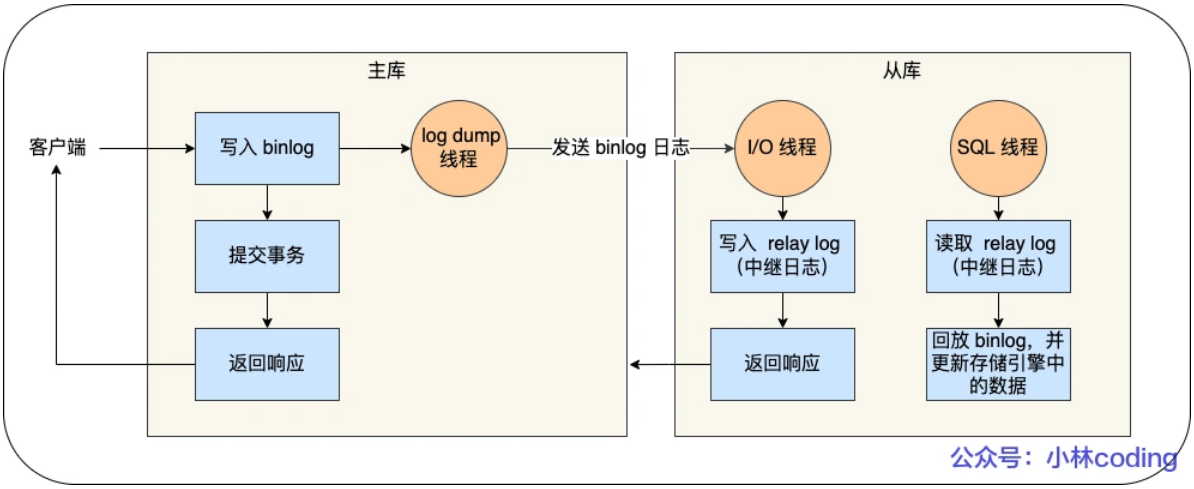
- binlog 用于备份恢复、主从复制；
- redo log 用于掉电等故障恢复。

## 为什么 binlog 用于备份恢复?

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 文件保存的是全量的日志，也就是保存了所有数据变更的情况。

## binlog 用于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，这个过程一般是异步的




- 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。
- 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 的中继日志里**，再返回给主库“复制成功”的响应。
- 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。

PS：在主从复制时，从库会创建一个专门的 I/O 线程，**连接主库的 log dump 线程，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**

**主从复制模型**

- **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。基本不使用
- **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。
- **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**

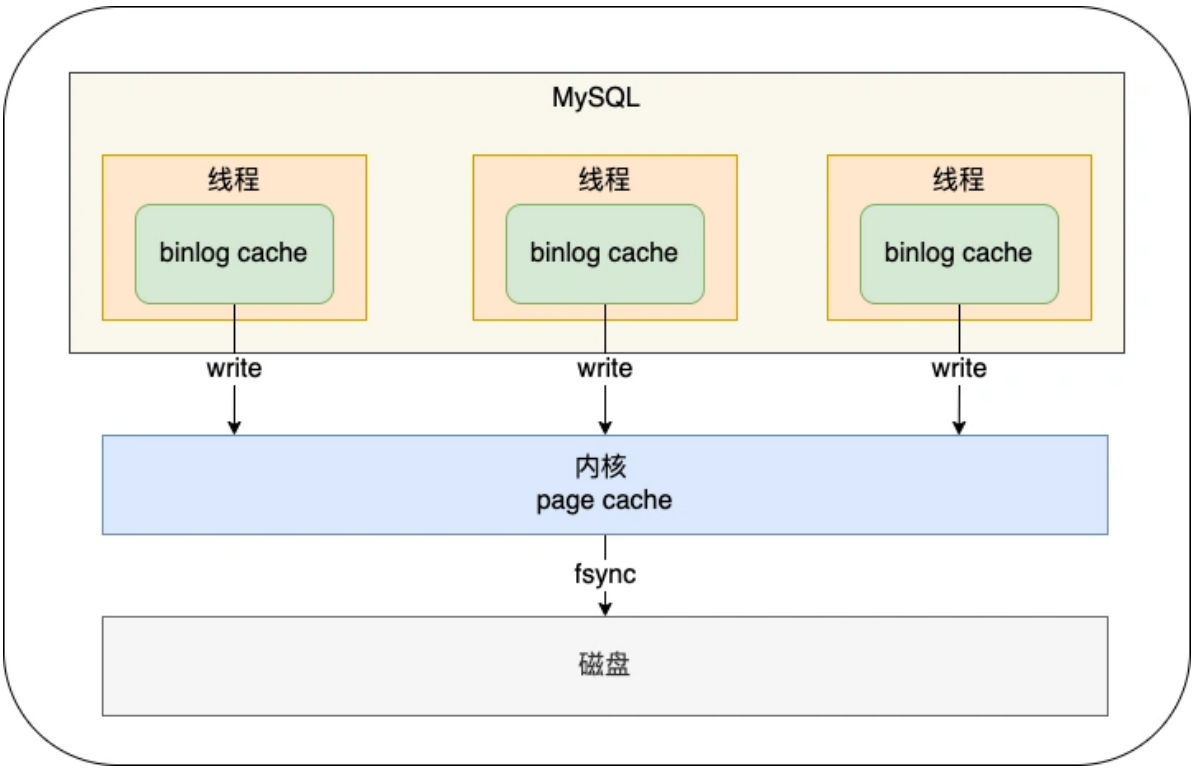
## binlog 什么时候刷盘?

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。**

**一个事务的 binlog 不能被拆开**，因为被拆开的话在从库执行时就会出现一个事务被分段的问题，破坏了原子性。

MySQL 给**每个线程分配了一片内存用于缓冲 binlog** ，该内存叫 **binlog cache**，参数 binlog_cache_size 用于控制单个线程内 binlog cache 所占内存的大小。如果超过了这个参数规定的大小，就要暂存到磁盘。

当提交一个事务时，会将 binlog cache 的完整事务写入 binlog 文件中，并清空 binlog cache，**虽然每个线程有自己 binlog cache，但是最终都写到同一个 binlog 文件**




**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。

## 面试介绍：binlog

bin log是server层实现的日志,因为最开始MySQL使用的是MyISAM存储引擎,此时并没有redo log。并且binlog **没有数据恢复的能力,只有数据归档的能力**，也就是说不能知道哪些记录被执行了，哪些没被执行。

binlog是使用追加写的方式写入文件，和 redolog 的循环写不同。有三种写入方式：statement、row、mixed

如果全部数据删除，可以使用binlog恢复，但是不能用redo log，因为redo log会在恢复数据后将日志记录抹除（循环写）

binlog日志先写到binlog cache，事务提交后再写入binlog文件，每个线程具备一个内存作为binlog cache

参数sync_binlog用于控制刷盘时机

- 0时：提交事务只write到os缓存
- 1时：直接write并fsync
- N时：每次都write，但是N个事务才fsync

## 什么是两阶段提交?

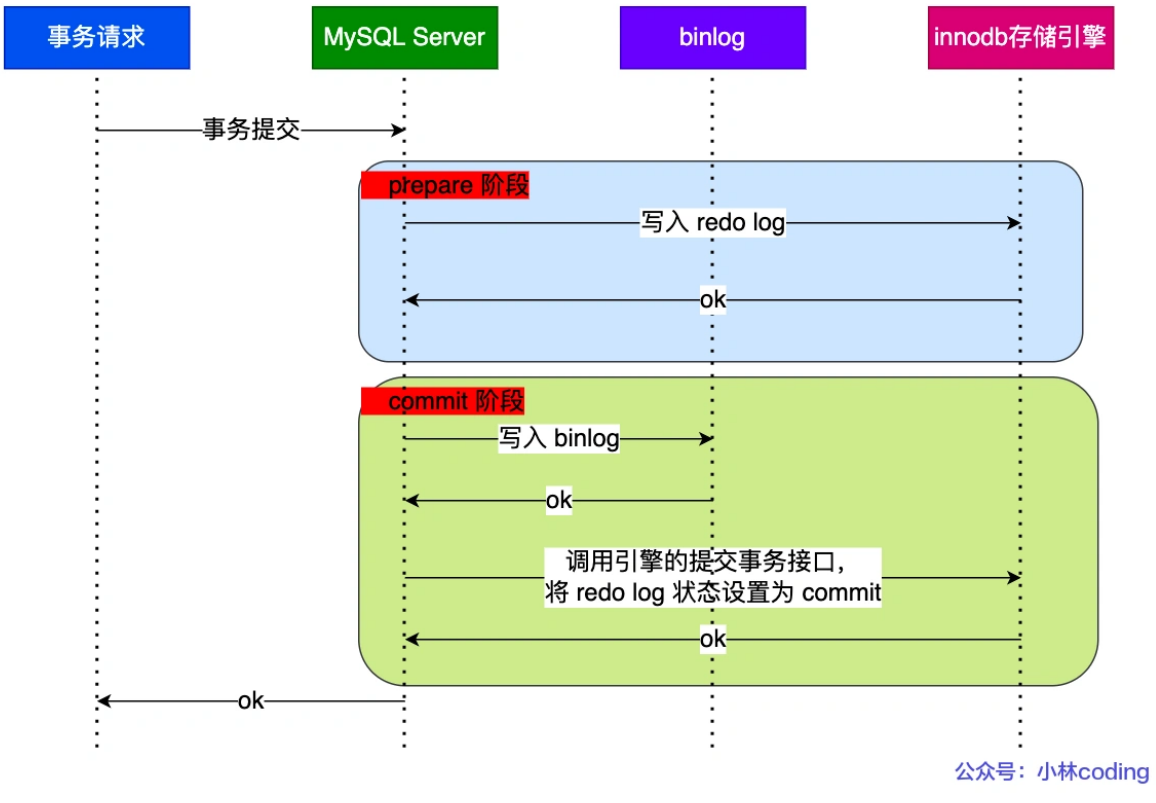
事务提交后，redo log 和 binlog 都要持久化到磁盘，但是这两个是独立的逻辑，可能出现半成功的状态，这样就造成两份日志之间的逻辑不一致。

举个例子，假设 id = 1 这行数据的字段 name 的值原本是 'jay'，然后执行 `UPDATE t_user SET name = 'xiaolin' WHERE id = 1;` 如果在持久化 redo log 和 binlog 两个日志的过程中，出现了半成功状态，那么就有两种情况：

- **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 Buffer Pool 中 id = 1 这行数据的 name 字段恢复到新值 xiaolin，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 name 字段是旧值 jay，与主库的值不一致性；
- **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 id = 1 这行数据的 name 字段还是旧值 jay，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，那么这一行 name 字段是新值 xiaolin，与主库的值不一致性；

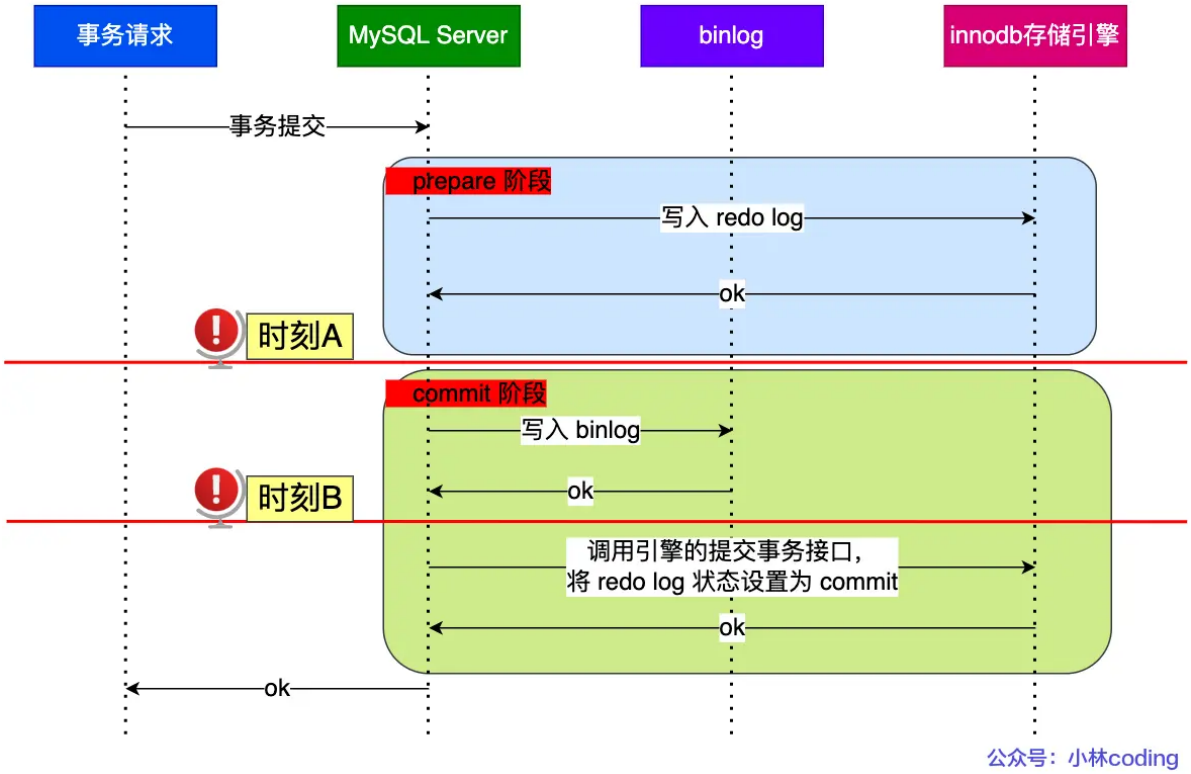
**两阶段提交把单个事务的提交拆分成了 2 个阶段，分别是「准备（Prepare）阶段」和「提交（Commit）阶段」**

- **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；



1. 执行器调用存储引擎接口，存储引擎将修改更新到内存中后，将修改操作写到 redo log 里面，此时 redo log 处于prepare状态；
2. 存储引擎告知执行器执行完毕，执行器开始将操作写入到 bin log 中；
3. bin log写完后调用存储引擎的接口提交事务，存储引擎将redo log的状态置为commit。

## 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 redolog 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 redolog 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。

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。这样就可以保证 redo log 和 binlog 这两份日志的一致性了。

所以，**两阶段提交是以 binlog 写成功为事务提交成功的标识**

## 两阶段提交有什么问题？

虽然保证了数据一致性，但是性能较差

- **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。

## 面试介绍：两阶段提交

两阶段提交是用于保持redo log和bin log一致性的，因为如果出现mysql崩溃重启而导致两个日志不一致时，会出现主库和从库数据不一致的问题。比如redo log刷入磁盘，binlog未写入，此时主库会是新值，从库为旧值。

为了保证日志一致性，使用了两阶段提交，分为prepare和commit阶段，

1. 首先当执行器调用存储引擎接口时，当数据在内存中被修改后，会将修改记录写入redo log，此时redolog为prepare状态，并返回相应给执行器。
2. 执行器收到响应后，将其记录写入到binlog中，写完后给存储引擎发出响应，redolog变为commit状态

如果故障重启时，会检查所有的在prepare状态下的redolog文件，并且取出其中的事务id，如果对应的binlog没有该事务id，则需要回滚（说明binlog没有写入成功）；如果有该事务id，则直接提交