什么是事务

事务就是一段 sql 语句的批处理，但是这个批处理是一个 atom（原子），不可分割，要么都执行，要么回滚（ rollback）都不执行。

（保证数据操作的安全性）

为什么出现这种技术

为什么要使用事务这个技术呢？ 现在的很多软件都是多用户，多程序，多线程的，对同一个表可能同时有很多人在用， 为保持数据的一致性， 所以提出了事务的概念。 这样很抽象，举个例子 -- 口述。

如何使用事务

（**1**）

只有 InnoDB /BDB 的之类的 transaction_safe table（commit-or-rollback ，如果是安全的） 才能支持。

默认的 engine MyISAM 是不支持事务的。

建立 InnoDB 表： Create table .... type=InnoDB ；

Alter table table_name type=InnoDB; （如何查看已有表的类型： show create table table_name)

这样我们就可以在 InnoDB 表上进行事务操作了！

（**2**）

启动事务的方法 ---2 种：

1、（推荐） begin， rollback,commit．当然有的人用 begin /begin work．推荐用 START TRANSACTION 是 SQL-99 标准启动一个事务。

start transaction；

update from account set money=money-100 where name='a';

update from account set money=money+100 where name='b';

commit ；

解释： 这样 start transaction 手动开启事务， commit 手动关闭事务。

2、默认的时候 autocommit=1 自动提交是开启的，所以你可以理解为每条语句一输入到 mysql 就 commit 了。当你 set autocommit=0 时候，你可以这样：

update from account set money=money-100 where name='a';

update from account set money=money+100 where name='b';

commit ；// 默认都不提交，只有手动键入 commit 时候才上述都提交。

但注意当你用 set autocommit=0 的时候，你以后所有的 SQL 都将做为事务处理，直到你用 commit 确认或 rollback 结束，注意当你结束这个事务的同时也开启了个新的事务

事务是必须满足 4 个条件（ACID）

原子性（Autmic）：

事务在执行性，要做到 ‘要么不做，要么全做！”，就是说不允许事务部分得执行。即使因为故障而使事务不能完成，在 rollback 时也要消除对数据库得影响！

一致性（Consistency）：

事务得操作应该使使数据库从一个一致状态转变倒另一个一致得状态！

隔离性（Isolation）：

如果多个事务并发执行，应象各个事务独立执行一样！

持久性（Durability）：

一个成功执行得事务对数据库得作用是持久得，即使数据库应故障出错，也应该能够恢复！

不加控制的并发存取会产生以下几种错误

1 丢失修改（**lost updates**）

当多个事务并发修改一个数据时，不加控制会得出错误的结果，一个修改会覆盖掉另一个修改。

2 读的不可重复性

当多个事务按某种时间顺序存取若干数据时，如果对并发存取不加控制，也会产生错误。

3 脏读（**DIRDY DATA**），读的不一致性

4 光标带来的当前值的混乱

事务在执行过程中它在某个表上的当前查找位置是由光标表示的。光标指向当前正处理的记录。当处理完该条记录后，则指向下一条记录。在多个事务并发执行时，某一事务的修改可能产生负作用，使与这些光标有关的事务出错。

5 未释放修改造成连锁退出

一个事务在进行修改操作的过程中可能会发生故障，这时需要将已做的修改回退（Rollback）。如果在已进行过或已发现错误尚未复原之前允许其它事务读已做过修改（脏读），则会导致连锁退出。

6 一事务在对一表更新时，另外的事务却修改或删除此表的定义。

数据库会为每个事务自动地设置适当级别的锁定。对于前面讲述的问题：脏读、未释放修改造成的连锁退出、一事务在对一表更新时另外的事务却修改或删除此表的定义，数据库都会自动解决。而另外的三个问题则需要在编程过程中人为地定义事务或加锁来解决。

事务处理的机制简单地说就是留下更新日志。 数据库会根据这些日志信息， 在必要时将旧数据取回，或者在发生错误时将数据恢复到原先的状态。

事务处理日志文件：

UNDO log 日志

UNDO 日志又被称为回滚段 (Rollback Segment)，在进行数据的捕入、更新、删除的场合，保存变更前的数据， 在表的内容保存了指向 UNDO 日志的指针， ROLLBACK 时根据这个指针米获得旧数据， 并覆盖新数据。 ROLLBACK 后，或者 COMMIT 后 UNDO 日志将被删除。另外， UNDO 日志也使用于保持数据读取的整合性。例如，在用户 A 对数据进行了更新，但还没有提交时， 用户 B 要参照数据怎么办呢 ?此时让用户 B 看到更新后的数据显然是不行的，其实是通过保存的指针让 UNDO 日志中的数据呈现给用户 B 的。

REDO log 日志

REDO 日志根据数据库的不同，有时被称为事务处理日志或日志 .

事务处理确定后， 由于错误等原因使数据的更新没有正确反映到数据库中的时候 ，REDO 日志提供了数据恢复用的孚段 .

仅仅看上述的说明可能还不能理解。那到底 [事务处理确定后，数据的更新没有反映到数据库中]这种状态是在什么时候发生的呢 ?这个必须首先说明。

我们通常提到 " 更新 (或变更)"就一句话带过，其实在数据库内部是需要经过较复杂的处理的过程的， 一般在数据库中进行更新处理时，并非立即更新数据文件，而是首先在被称缓冲 (buffer cash) 的内存空间中进行数据更新，并将这些保存到 REDO 日志文件中。 到现在为止太家会以为所有的处理都是实时的， 其实在数据库中向数据文件写入时是有延迟 （或称为滞后）的。向数据文件的写入动作是在触发称为检查点 (check point) 时非同步进行的。检查点拨一定的周期触发。

如果频繁进行这样复杂的处理的话， 尤其是对硬盘进行写入处理时将会出现很高的负荷。 因此现在的处理方式是， 数据库将对硬盘的写入操作稍微保留片刻， 当检查点 (check point) 到来时再集中处理，这样就会显著减少访问硬盘的次数，性能得到改善。

问题出现在 REDO 日志的更新与数据文件的更新之间的延迟上。大家想象一下， 如果在这个延迟的当口发生了停电， 硬件错误时会出现什么悄况呢 ?很显然， 内存中的信息将会消失，只剩下在 REDO 日志中保存的信息了。 REDO 日志正是用于复原的 .

数据库在错误排除后的第一次起动时进行称为 Roll forward 的处理，Roll forward 处理具体做法是，从 REDO 日志中抽出从最后的检查点到错误发生时间点间的事务处理，然后重新执行一次 (REDO) 。这样，数据库就恢复到错误发生前的状态了。

Redo log undo log 区别

redo->记录所有操作，用于恢复

undo->记录所有的前印象，用于回滚

redo->已递交的事务 ,实例恢复时要写到数据文件去的

undo->未递交的事务 .

redo 的原因是： 每次 commit 时，将数据的修改立即写到 redo 中，但是并不一定同时将该数据的修改写到数据文件中 。

undo 的原因是：口述。

undo 也是也是 datafile， 可能 dirty buffer 没有写回到磁盘里面去。
只有先 redo apply 成功了，才能保证 undo datafile 里面的东西都是正确的， 然后才能 rollback

做 undo 的目的是使系统恢复到系统崩溃前 (关机前)的状态 ,再进行 redo 是保证系统的一致性 .
不做 undo,系统就不会知道之前的状态 ,redo 就无从谈起

undo log 原理

使用 undo log 时，要求：

1. 记录修改日志时（ redo log），(T，x，v）中 v 为 x 修改前的值，这样才能借助这条日志来回滚；
2. 事务提交后， 必须在事务的所有修改 （包括记录的修改日志） 都持久化后才能写 COMMIT T 日志；这样才能保证，宕机恢复时，已经 COMMIT 的事务的所有修改都已经持久化，不需要回滚。

使用 undo log 时事务执行顺序

1. 记录 START T
2. 记录需要修改的记录的旧值（要求持久化）
3. 根据事务的需要更新数据库（要求持久化）
4. 记录 COMMIT T

使用 undo log 进行宕机回滚

1. 扫描日志，找出所有已经 START, 还没有 COMMIT 的事务。
2. 针对所有未 COMMIT 的日志，根据 redo log 来进行回滚。

如果数据库访问很多，日志量也会很大，宕机恢复时，回滚的工作量也就很大，为了加快回滚，可以通过 checkpoint 机制来加速回滚，

在日志中记录 checkpoint_start (T1,T2 …Tn) (代表做 checkpoint 时，正在进行还未 COMMIT 的事务）
等待所有正在进行的事务（ T1~Tn） COMMIT
在日志中记录 checkpoint_end
借助 checkpoint 来进行回滚

从后往前，扫描 undo log
1，如果先遇到 checkpoint_start，则将 checkpoint_start 之后的所有未提交的事务进行回滚；
2. 如果先遇到 checkpoint_end，则将前一个 checkpoint_start 之后所有未提交的事务进行回滚；（在 checkpoint 的过程中，可能有很多新的事务 START 或者 COMMIT）。
使用 undo log，在写 COMMIT 日志时，要求 redo log 以及事务的所有修改都必须已经持久化，这种做法通常很影响性能。

redo log 原理

redo log 是指在回放日志的时候把已经 COMMIT 的事务重做一遍， 对于没有 commit 的事务按照 abort 处理，不进行任何操作。

使用 redo log 时，要求：

1. 记录 redo log 时，（T,x， v）中的 v 必须是 x 修改后的值，否则不能通过 redo log 来恢复已经 COMMIT 的事务。
2. 写 COMMIT T 日志之前，事务的修改不能进行持久化，否则恢复时，对于未 COMMIT 的操作，可能有数据已经修改，但重放 redo log 不会对该事务做任何处理，从而不能保证事务的原子性。

使用 redo log 时事务执行顺序

1. 记录 START T
2. 记录事务需要修改记录的新值（要求持久化）
3. 记录 COMMIT T （要求持久化）
4. 将事务相关的修改写入数据库

使用 redo log 重做事务

1. 扫描日志，找到所有已经 COMMIT 的事务；
2. 对于已经 COMMIT 的事务，根据 redo log 重做事务；

在日志中使用 checkpoint

1. 在日志中记录 checkpoint_start (T1,T2...Tn) (Tx 代表做 checkpoint 时，正在进行还未 COMMIT 的日志）
2. 将所有已提交的事务的更改进行持久化；
3. 在日志中记录 checkpoint_end

根据 checkpoint 来加速恢复

从后往前，扫描 redo log

1, 如果先遇到 checkpoint_start，则把 T1~Tn 以及 checkpoint_start 之后的所有已经 COMMIT 的事务进行重做；

2. 如果先遇到 checkpoint_end，则 T1~Tn 以及前一个 checkpoint_start 之后所有已经 COMMIT 的事务进行重做；

与 undo log 类似，在使用时对持久化以及事务操作顺序的要求都比较高，可以将两者结合起来使用，在恢复时，对于已经 COMMIT 的事务使用 redo log 进行重做，对于没有 COMMIT 的事务，使用 undo log 进行回滚。 redo/undo log 结合起来使用时，要求同时记录操作修改前和修改后的值，如（ T，x，v，w），v 为 x 修改前的值， w 为 x 修改后的值，具体操作顺序为：

1. 记录 START T
2. 记录修改日志（ T，x，v，w）（要求持久化，其中 v 用于 undo，w 用于 redo）
3. 更新数据库
4. 记录 COMMIT T

4 和 3 的操作顺序没有严格要求，并且都不要求持久化；因为如果宕机时 4 已经持久化，则恢复时可通过 redo log 来重做；如果宕机时 4 未持久化，则恢复时可通过 undo log 来回滚；

在处理 checkpoint 时，可采用与 redo log 相同的处理方式。

什么是 binlog

binlog 日志用于记录所有更新且提交了数据或者已经潜在更新提交了数据（例如，没有匹配任何行的一个 DELETE ）的所有语句。语句以“事件”的形式保存，它描述数据更改。

binlog 作用

1.恢复使能够最大可能地更新数据库，因为二进制日志包含备份后进行的所有更新。

2.在主复制服务器上记录所有将发送给从服务器的语句。

binlog 未打卡状态下的提交流程

测试前置条件：

```
set autocommit=0;
create table tt(col1 int, col2 varchar(100));
```

测试语句：

```
insert into tt values(1, 'abcdef');
commit;
```

无论对于 dml 语句【insert ，update ，delete 等】还是 dcl 语句【commit ，rollback 】，mysql 提供了公共接口 mysql_execute_command ，我们先分析 mysql_execute_command 接口的基本流程：

```
<span style= "font-size:
14px;" >mysql_execute_command
{
    switch (command)
  {
        case SQLCOM_INSERT:
         mysql_insert();
                break ;

        case SQLCOM_UPDATE:
         mysql_update();
                break ;

        case SQLCOM_DELETE:
         mysql_delete();
                break ;
    ......
 }
```

```
    if   thd->is_error()              //  语句执行错误
    trans_rollback_stmt(thd);
   else
   trans_commit_stmt(thd);
}
</span>
```

可以看到执行任何语句最后，都会执行 trans_rollback_stmt ，或 trans_commit_stmt ，这两个调用分别是语句级提交和语句级回滚。

语句级提交，对于非自动模式提交情况下，主要作两件事情，

一是释放 autoinc 锁(什么是他 -- 口述)，这个锁主要用来处理多个事务互斥地获取自增列值， 因此，无论最后该语句提交或是回滚， 该资源都是需要而且可以立马放掉的。

二是标识语句在事务中位置，方便语句级回滚。执行 commit 后，可以进入 commit 流程，现在来看看具体事务提交的流程是怎样的。

```
<span style=     "font-size: 14px;"          >mysql_execute_command
   trans_commit
ha_commit_trans(thd, FALSE);
{
   TC_LOG_DUMMY:ha_commit_low
      ha_commit_low()
         innobase_commit
         {
                        //  获取 innodb  层对应的事务结构
            trx = check_trx_exists(thd);

                        if  ( 单个语句，且非自动提交  )
            {
                           //  释放自增列占用的  autoinc  锁资源
               lock_unlock_table_autoinc(trx);
                           //  标识 sql  语句在事务中的位置，方便语句级回滚
               trx_mark_sql_stat_end(trx);
            }
                        else  事务提交
```

```
{
innobase_commit_low()
{
trx_commit_for_mysql();
<span style= "color:
#ff0000;" >trx_commit</span>(trx);
}

// 确定事务对应的 redo 日志是否落盘【根
据 flush_log_at_trx_commit 参数，确定 redo 日志落盘方式】
trx_commit_complete_for_mysql(trx);

trx_flush_log_if_needed_low(trx->commit_lsn);
log_write_up_to(lsn);
}
}
}</span><br><br>
```

```
trx_commit
trx_commit_low
{
trx_write_serialisation_history
{
trx_undo_update_cleanup // 供 purge 线程处理，清理回滚页
}
trx_commit_in_memory
{
lock_trx_release_locks // 释放锁资源
trx_flush_log_if_needed(lsn) // 刷日志
trx_roll_savepoints_free // 释放 savepoints
}
}
```

mysql 通过 WAL 方式，来保证数据库事务的一致性和持久性，即 ACID 特性中的 C(consistent) 和 D(durability) 。WAL （Write-Ahead Logging ）是一种实现事务日志的标准方法，具体而言就是修改记录前，一定要先写日志；事务提交过程中，一定要保证日志先落盘，才能算事务提交完成。通过 WAL 方式，在保证事务特性的情况下，可以提交数据库的性能。从上述流程可以看出，提交过程中， 主要做了 4 件事情，首先是 清理 undo 段信息，对于 innodb 存储引擎的更新操作来说， undo 段需要 purge ，这里的 purge 主要职能是，真正

删除物理记录。在执行 delete 或 update 操作时，实际旧记录没有真正删除，只是在记录上打了一个标记，而是在事务提交后， purge 线程真正删除，释放物理页空间。因此，提交过程中会将 undo 信息加入 purge 列表，供 purge 线程处理。然后是 释放锁资源 ， mysql 通过锁互斥机制保证不同事务不同时操作一条记录， 事务执行后才会真正释放所有锁资源， 并唤醒等待其锁资源的其他事务；再就是 刷 redo 日志，前面我提到了， mysql 实现事务一致性和持久性的机制。通过 redo 日志落盘操作，保证了即使修改的数据页没有即使更新到磁盘，只要日志是完成了， 就能保证数据库的完整性和一致性； 最后就是 清理保存点列表，每个语句实际都会有一个 savepoint(保存点)，保存点作用是为了可以回滚到事务的任何一个语句执行前的状态， 由于事务都已经提交了， 所以保存点列表可以被清理了。

关于里面提到的 mysql 的锁机制， purge 原理， redo 日志， undo 段等内容，以后再说。。

```
<span style= "font-size: 14px;" >struct trx_t{

    trx_rseg_t* rseg;                    /*!< rollback segment assigned to the
                                         transaction, or NULL if not assigned

    trx_undo_t* insert_undo;  /*!< pointer to the insert undo log, or
                                         NULL if no inserts performed yet */
    trx_undo_t* update_undo;             /*!< pointer to the update undo log, or
                                         NULL if no update performed yet */
    const char * mysql_log_file_name;
                                         /*!< if MySQL binlog is used, this field
                                         contains a pointer to the latest file
                                         name; this is NULL if binlog is not
                                         used */
    ib_int64_t  mysql_log_offset;
                                         /*!< if MySQL binlog is used, this
                                         field contains the end offset of the
                                         binlog entry */
}


/* The rollback segment memory object */
```

```
struct     trx_rseg_t{
        /* Fields for update undo logs */
   UT_LIST_BASE_NODE_T(trx_undo_t) update_undo_list;
                                   /* List of update undo logs */
   UT_LIST_BASE_NODE_T(trx_undo_t) update_undo_cached;
                                   /* List of update undo log segments
                                   cached for fast reuse */
        /*-------------------------------------------------------*/
        /* Fields for insert undo logs */
   UT_LIST_BASE_NODE_T(trx_undo_t) insert_undo_list;
                                   /* List of insert undo logs */
   UT_LIST_BASE_NODE_T(trx_undo_t) insert_undo_cached;
                                   /* List of insert undo log segments
                                   cached for fast reuse */
}
</span>
```

binlog 打开状态下的提交流程

开启 binlog 后事务提交流程会变成两阶段提交，这里的两阶段提交并不涉及分布式事务，当然 mysql 把它称之为内部 xa 事务（Distributed Transactions ），与之对应的还有一个外部 xa 事务。内部 xa 事务我理解主要是 mysql 内部为了保证 binlog 与 redo log 之间数据的一致性而存在的，这也是由其架构决定的 (binlog 在 mysql 层，而 redo log 在存储引擎层)；而外部 xa 事务则是指支持多实例分布式事务，这个才算是真正的分布式事务。既然是 xa 事务，必然涉及到两阶段提交，对于内部 xa 而言，同样存在着提交的两个阶段。下文会结合源码详细解读内部 xa 的两阶段提交过程，以及各种情况下，mysqld crash 后，mysql 如何恢复来保证事务的一致性。

测试环境：
OS ：windows
DB ：mysql 5.6.12
engine ：innodb

配置文件参数：
log-bin=D:\chuck\mysql\log\5-6-12\mysql-bin

binlog_format=ROW
set autocommit=0;
sync_binlog=1 ;
innodb_flush_log_at_trx_commit=1 ;
innodb_support_xa=1 ;

测试前置条件：
create table tt(col1 int, col2 varchar(100));

测试语句：
insert into tt values(1, 'abcdef');
commit;

打开 binlog 选项后，执行事务提交命令时，就会进入两阶段提交模式。两阶段提交分为 prepare 阶段和 commit 两个阶段 。
流程如下 ：这里面涉及到两个重要的参数：innodb_flush_log_at_trx_commit 和 sync_binlog ，参数可以设置不同的值，具体可以查看 mysql 的帮助手册。我这里设置的是双一模式（innodb_flush_log_at_trx_commit=1 ，sync_binlog=1 ），不同的模式区别在于，写文件调用 write 和落盘 fsync 调用的频率不同，所导致的后果是 mysqld 或 os crash 后，不严格的设置可能会丢失事务的更新。 双一模式是最严格的模式，这种设置情况下，单机在任何情况下不会丢失事务更新。
prepare 阶段：
1. 设置 undo state=TRX_UNDO_PREPARED ；//trx_undo_set_state_at_prepare 调用
2. 刷事务更新产生的 redo 日志；【步骤 1 产生的 redo 日志也会刷入】

commit 阶段：
1. 将事务产生的 binlog 写入文件，刷入磁盘；
2. 设置 undo 页的状态 ,置为 TRX_UNDO_TO_FREE 或 TRX_UNDO_TO_PURGE; // trx_undo_set_state_at_finish 调用
3. 记录事务对应的 binlog 偏移，写入系统表空间; //trx_sys_update_mysql_binlog_offset 调用

下面这部分是我抽象出来的源码调用部分，大家可以通过单步调试方式，在关键函数中设置断点，来详细了解这个过程。
==========
prepare 阶段
==========
MYSQL_BIN_LOG::prepare
ha_prepare_low
{
engine:
binlog_prepare

innobase_xa_prepare
mysql:
trx_prepare_for_mysql
{
1.trx_undo_set_state_at_prepare // 设置 undo 段的标记为 TRX_UNDO_PREPARED
2. 设置事务状态为 TRX_STATE_PREPARED
3.trx_flush_log_if_needed // 将产生的 redolog 刷入磁盘
}
}

==========
commit 阶段
==========
MYSQL_BIN_LOG::commit
ordered_commit
{
1.FLUSH_STAGE
flush_cache_to_file // 刷 binlog

2.SYNC_STAGE
sync_binlog_file //Call fsync() to sync the file to disk.

3.COMMIT_STAGE
ha_commit_low
{
binlog_commit
innobase_commit
trx_commit(trx)
{
trx_write_serialisation_history(trx, mtr); // 更新 binlog 位点，设置 undo 状态
trx_commit_in_memory(trx, lsn); // 释放锁资源，清理保存点列表，清理回滚段
}
}
}

mysqld 可能在任何情况下 crash ，os 也有可能出现问题，另外若机器掉电，mysqld 也会同样挂掉。但是即使这样，mysql 仍然能保证数据库的一致性。接下来，我会结合上述流程，分析二阶段提交如何保证这点的。 下面给出几种常见的场景，
1.prepare 阶段，redo log 落盘前，mysqld crash
2.prepare 阶段，redo log 落盘后，binlog 落盘前，mysqld crash

3.commit 阶段，binlog 落盘后，mysqld crash

对于第一种情况，由于 redo 没有落盘，毫无疑问，事务的更新肯定没有写入磁盘，数据库的一致性受影响；对于第二种情况，这时候 redo log 写入完成，但 binlog 还未写入，事务处于 TRX_STATE_PREPARED 状态，这是提交还是回滚呢？对于第三种情况，此时，redo log 和 binlog 都已经落盘，只是 undo 状态没有更新，这种情况也应该提交，因为 redo log 和 binlog 已经一致了，当然这只是我的假设，需要通过源码逻辑来验证。

下面给出了 mysqld 异常重启后的执行逻辑以及关键的源代码。对于第三种情况，我们可以搜集到未提交事务的 binlog event ，所以需要提交，与我们假设相符；而对于第二种情况，由于 binlog 未写入，需要通过执行回滚操作来保证数据库的一致性。

异常重启后，如何判断事务该提交还是回滚

1. 读 binlog 日志，获取崩溃时没有提交的 event ； //info->commit_list 中含有该元素
2. 若存在，则对应的事务要提交；否则需要回滚。

判断事务提交或回滚源码如下：

```
1791          // recovery mode
1792          if (info->commit_list ?
1793              my_hash_search(info->commit_list, (uchar *)&x, sizeof(x)) != 0 :
1794              tc_heuristic_recover == TC_HEURISTIC_RECOVER_COMMIT)
1795          {
1796  #ifndef DBUG_OFF
1797            char buf[XIDDATASIZE*4+6]; // see xid_to_str
1798            sql_print_information("commit xid %s", xid_to_str(buf, info->list+i
1799  #endif
1800            hton->commit_by_xid(hton, info->list+i);
1801          }
1802          else
1803          {
1804  #ifndef DBUG_OFF
1805            char buf[XIDDATASIZE*4+6]; // see xid_to_str
1806            sql_print_information("rollback xid %s",
1807                                  xid_to_str(buf, info->list+i));
1808  #endif
1809            hton->rollback_by_xid(hton, info->list+i);
1810          }
```

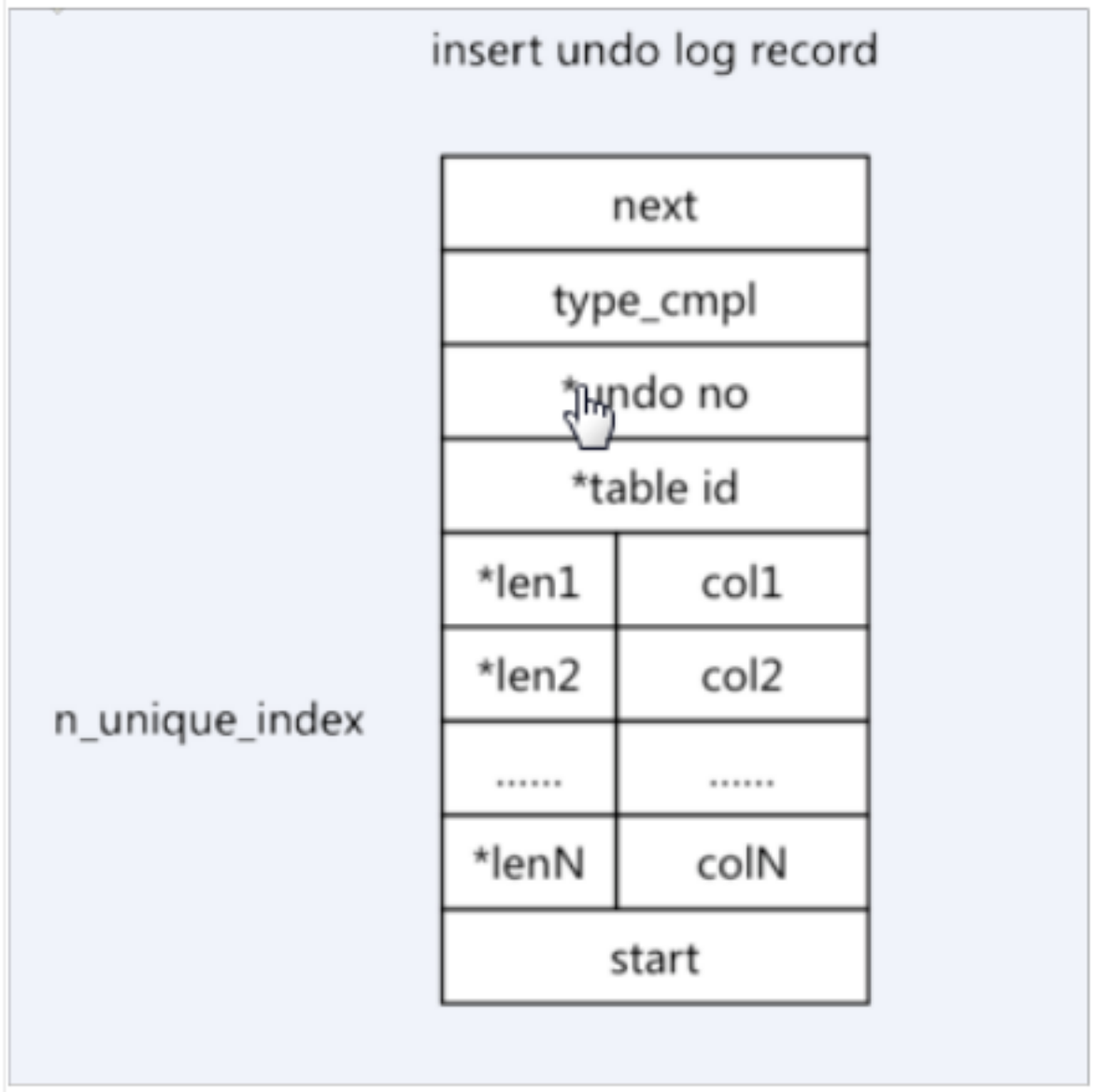
上面讨论了两阶段提交的基本流程，以及服务器异常 crash 后，mysql 如何重启恢复保证 binlog 和数据的一致性。简而言之，对于异常的 xa 事务，若 binlog 已落盘，则事务应该提交； binlog 未落盘，则事务就应该回滚。由于这块涉及到的源代码较多， 我也没有看完所有源代码， 如有不正确的地方， 欢迎指正。

// 异常重启后，回滚流程

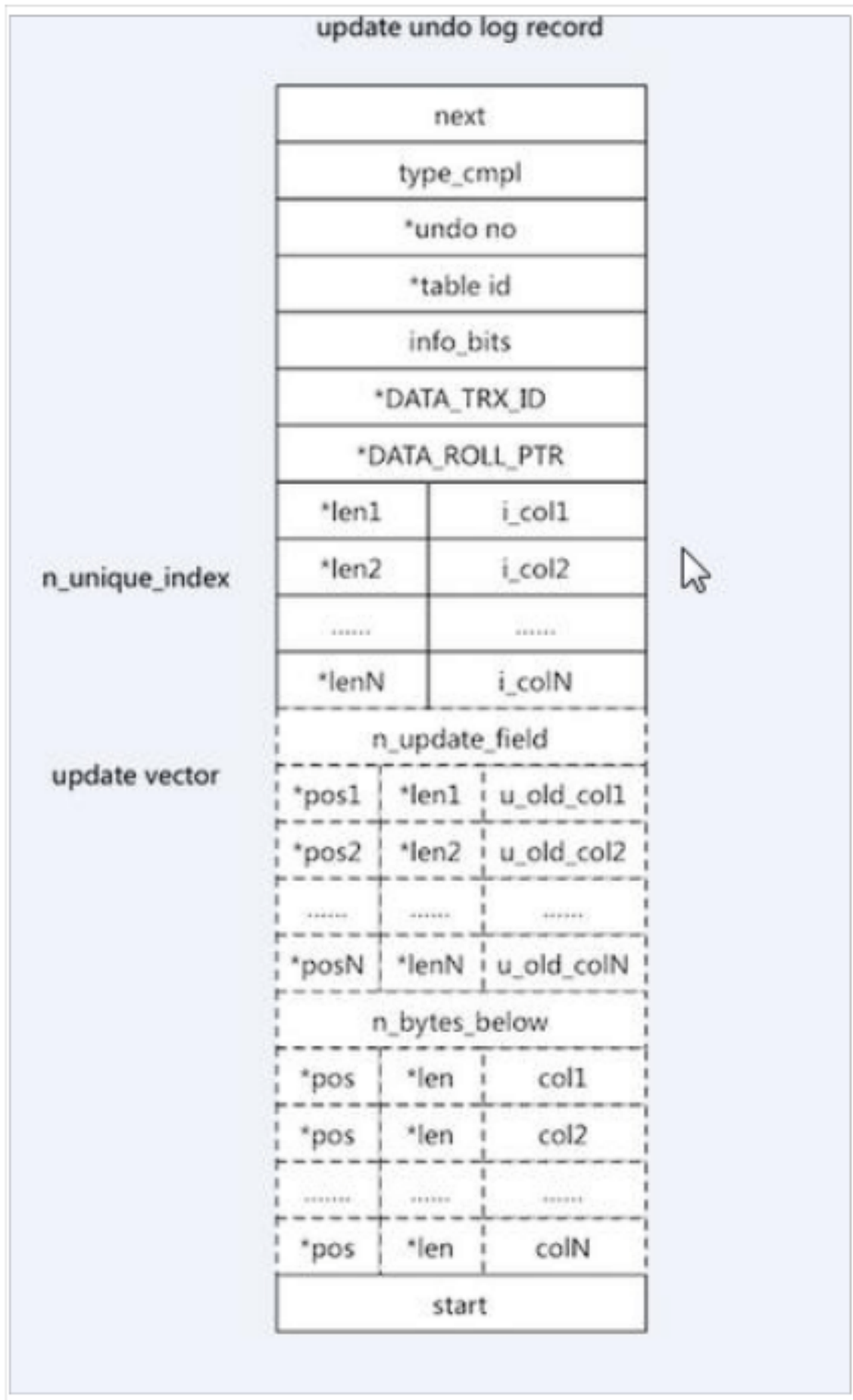
```
innobase_rollback_by_xid
    rollback_by_xid
        trx_rollback_resurrected
            trx_rollback_active
                row_undo
                {
                    //          从回滚页获取  undo 记录
                    //          分析 undo 记录类型
                    if (insert)
                        row_undo_ins
                    else
                        row_undo_mod
                }
```

```
// 异常重启后，提交流程
commit_by_xid
    trx_commit_for_mysql

// 写 binlog  接口
handler.cc:binlog_log_row
sql/binlog.cc:commit
mysys/my_sync:my_sync
sql/binlog.cc:sync_binlog_file
handler/ha_innodb.cc:innobase_xa_prepare
```



图中＊表示对存储的字段进行了压缩。insert undo log 开始的前两个字节 next 记录的是下一个 undo log 的位置，通过该 next 的字节可以知道一个 undo log 所站的空间字节数。 类似地，尾部的两个字节记录的是 undo log 的开始位置。 type_cmpl 占用一个字节，记录的是 undo 地类型，对于 insert undo log，该值总是为 11。undo_no 记录事务的 ID，table_id 记录 undo log 所对应的表对象。这两个值都是用于进行压缩的。接着的部分记录了所有主键的列和值。在进行 rollback 操作时，根据这些值可以定位到具体的记录，然后进行删除即可。



该 undo log 可能需要提供 MVCC 多版本并发控制机制，因此不能在事务提交时就进行删除。提交时放入 undo log 链表，等待 purge 线程进行最后的删除。

这里的 type_cmpl，由于 update undo log 本身还有分类，故其可能的值如下：

12 （TRX_UNDO_UPD_EXIST_REC ）

13 （TRX_UNDO_UPD_DEL_REC ）

14 （TRX_UNDO_DEL_MARK_REC ）

undo log 可以分为上述 3 种类型，其分别代表了更新 non-delete-mark 的记录，将 delete 的记录标记为 not delete，将记录标记为 delete。

紧接着的部分记录 update_vector 信息， update_vector 表示 update 操作导致发生改变的列。每个修改的列信息都要记录的 undo log 中。对于不同的 undo log 类型，可能还需要记录对索引列所做的修改。