# undo

参考：

<https://www.cnblogs.com/f-ck-need-u/p/9010872.html>

UNDO LOG的演进与现状：<http://mysql.taobao.org/monthly/2022/10/02/>

B+树数据库故障恢复概述：<http://mysql.taobao.org/monthly/2022/10/04/>

庖丁解InnoDB之UNDO LOG：<http://mysql.taobao.org/monthly/2021/10/01/>

InnoDB之UNDO LOG：<http://mysql.taobao.org/monthly/2021/12/27/bayan/>

InnoDB UNDO LOG写入：<http://mysql.taobao.org/monthly/2020/08/05/>

INNODB UNDO LOG分配：<http://mysql.taobao.org/monthly/2020/08/02/>

undo tablespace 的发展：<http://mysql.taobao.org/monthly/2020/10/02/>

PG undo log存储接口：<http://mysql.taobao.org/monthly/2019/07/02/>

通过 SQL 管理 UNDO TABLESPACE：

<http://mysql.taobao.org/monthly/2019/05/04/>

关于undo表空间的一些新变化：<http://mysql.taobao.org/monthly/2019/04/05/>

## 概述

undo log是为了实现**事务的原子性**，在MySQL数据库InnoDB存储引擎中，采用**undo log来实现多版本并发控制**（简称 MVCC）。

在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地方称为undo log）。然后进行数据的修改。如果出现了错误或者用户执行了ROLLBACK语句，系统可以利用undo log中的备份将数据恢复到事务开始之前的状态。除了可以保证事务的原子性，undo log也可以用来辅助完成事务的持久化。

因此，undo log有两个作用：提供回滚和多个行版本控制(MVCC)。

注意：undo log和redo log记录物理日志不一样，它是逻辑日志。可以理解为：

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

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

当update一条记录时，undo log中会记录一条对应相反的update记录

undo log也会产生redo log，因为undo log也要实现持久性保护。

## 分类

### insert undo log

### update undo log

## 存储

InnoDB存储引擎对undo的管理采用**段**的方式。rollback segment称为回滚段，每个回滚段中有1024个undo log segment。

undo log默认存放在共享表空间中。

[root@\*\*\*\* data]# ll /mydata/data/ib\*

-rw-rw---- 1 mysql mysql 79691776 Mar 31 01:42 ***/mydata/data/ibdata1***

-rw-rw---- 1 mysql mysql 50331648 Mar 31 01:42 /mydata/data/ib\_logfile0

-rw-rw---- 1 mysql mysql 50331648 Mar 31 01:42 /mydata/data/ib\_logfile1

如果开启了 innodb\_file\_per\_table ，将放在每个表的.ibd文件中。

在MySQL5.6中，undo的存放位置还可以通过变量 innodb\_undo\_directory 来自定义存放目录，默认值为"."表示datadir。

默认rollback segment全部写在一个文件中，但可以通过设置变量 innodb\_undo\_tablespaces 平均分配到多少个文件中。该变量默认值为0，即全部写入一个表空间文件。该变量为静态变量，只能在数据库示例停止状态下修改，如写入配置文件或启动时带上对应参数。但是innodb存储引擎在启动过程中提示，不建议修改为非0的值，如下：

**2017**-**03**-**31** **13**:**16**:**00** 7f665bfab720 InnoDB: Expected to open **3** undo tablespaces but was able

**2017**-**03**-**31** **13**:**16**:**00** 7f665bfab720 InnoDB: to find only **0** undo tablespaces.

**2017**-**03**-**31** **13**:**16**:**00** 7f665bfab720 InnoDB: Set the innodb\_undo\_tablespaces parameter to the

**2017**-**03**-**31** **13**:**16**:**00** 7f665bfab720 InnoDB: correct value and retry. ***Suggested value is 0***

## 变量

mysql> show variables like "%undo%";

+-------------------------+-------+

| Variable\_name | Value |

+-------------------------+-------+

| innodb\_undo\_directory | . |

| innodb\_undo\_logs | **128** |

| innodb\_undo\_tablespaces | **0** |

+-------------------------+-------+

### innodb\_undo\_directory

### innodb\_undo\_logs

### innodb\_undo\_tablespaces

## 原理

### mini transaction

参考：

mini transaction详解：<http://mysql.taobao.org/monthly/2021/09/04/>

### insert内部实现

### delete/update内部实现

当事务提交的时候，innodb不会立即删除undo log，因为后续还可能会用到undo log，如隔离级别为repeatable read时，事务读取的都是开启事务时的最新提交行版本，只要该事务不结束，该行版本就不能删除，即undo log不能删除。

但是在事务提交的时候，会将该事务对应的undo log放入到删除列表中，未来通过purge来删除。并且提交事务时，还会判断undo log分配的页是否可以重用，如果可以重用，则会分配给后面来的事务，避免为每个独立的事务分配独立的undo log页而浪费存储空间和性能。

通过undo log记录delete和update操作的结果发现：

insert操作无需分析，就是插入行而已

delete操作实际上不会直接删除，而是将delete对象打上delete flag，标记为删除，最终的删除操作是purge线程完成的。

update分为两种情况：update的列是否是主键列。

如果不是主键列，在undo log中直接反向记录是如何update的。即update是直接进行的。

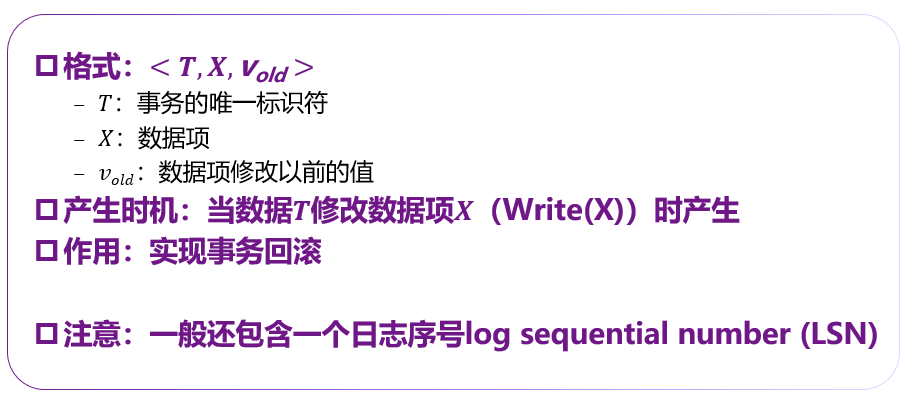
如果是主键列，update分两部执行：先删除该行，再插入一行目标行。

### MVCC

MVCC的版本链上存储的就是各个事务的undo log。

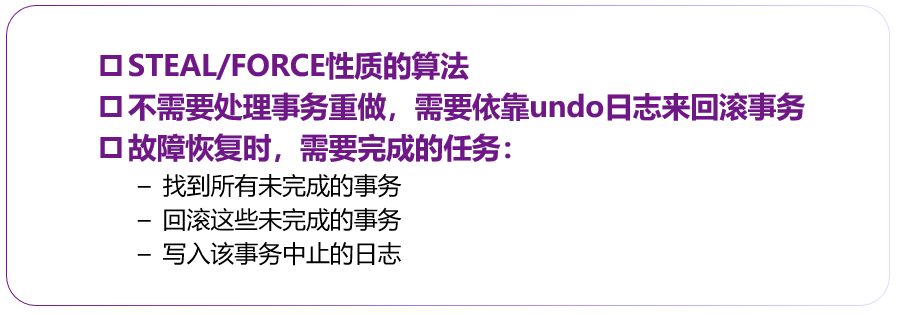
### 回滚

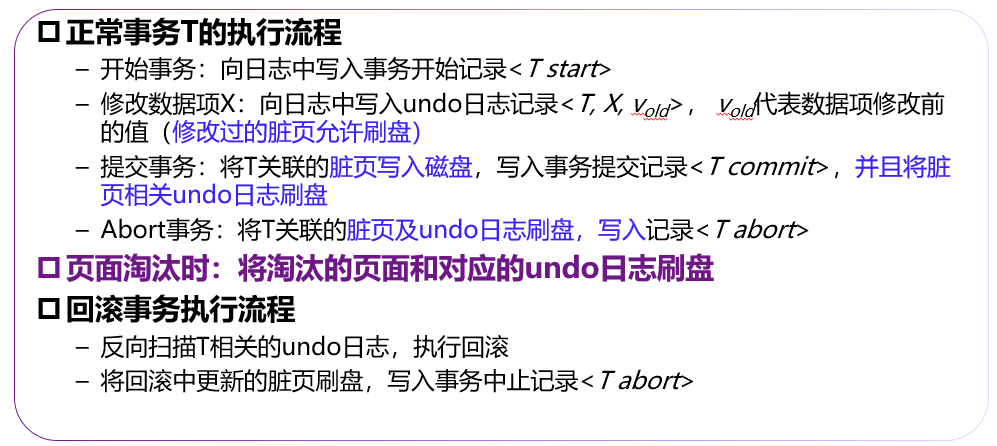
#### Undo回滚日志

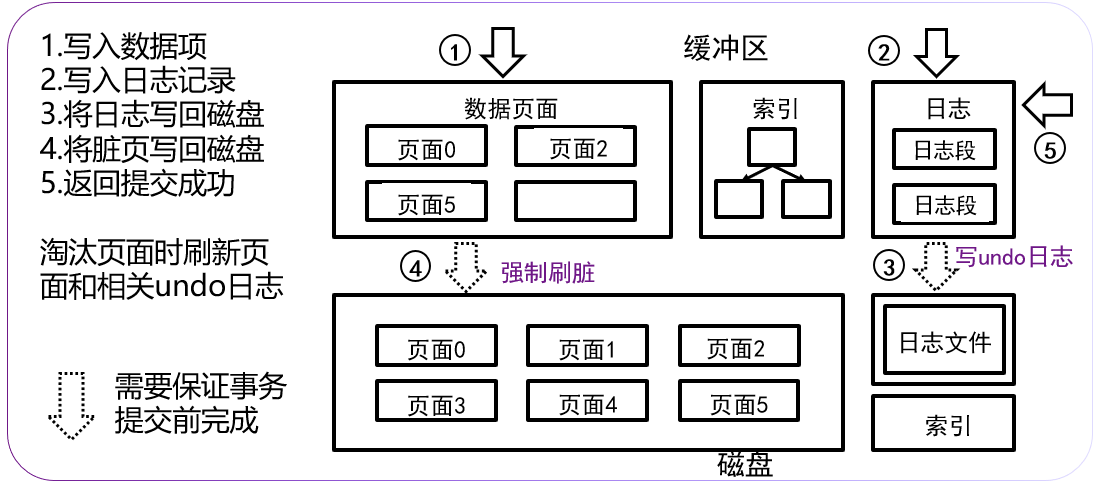




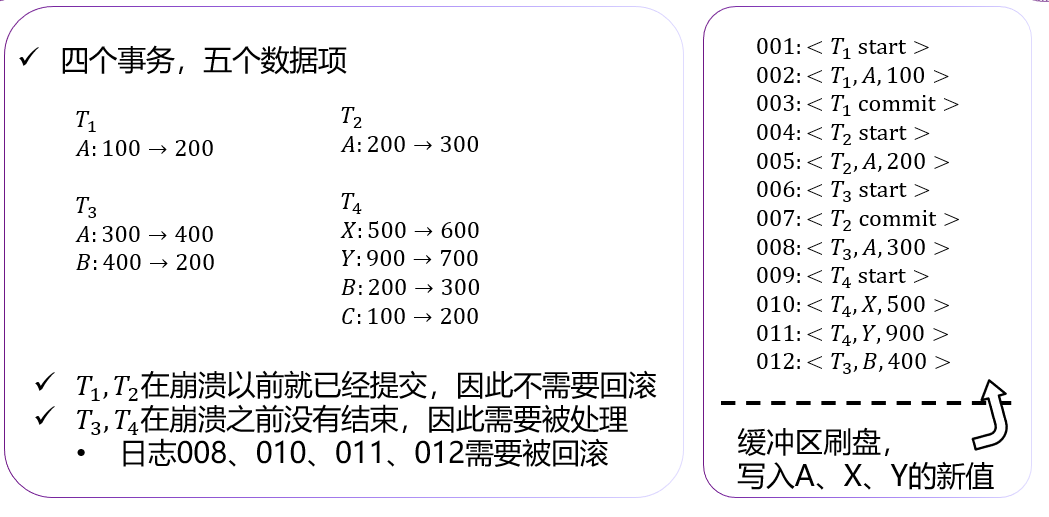
#### 基于undo日志的恢复

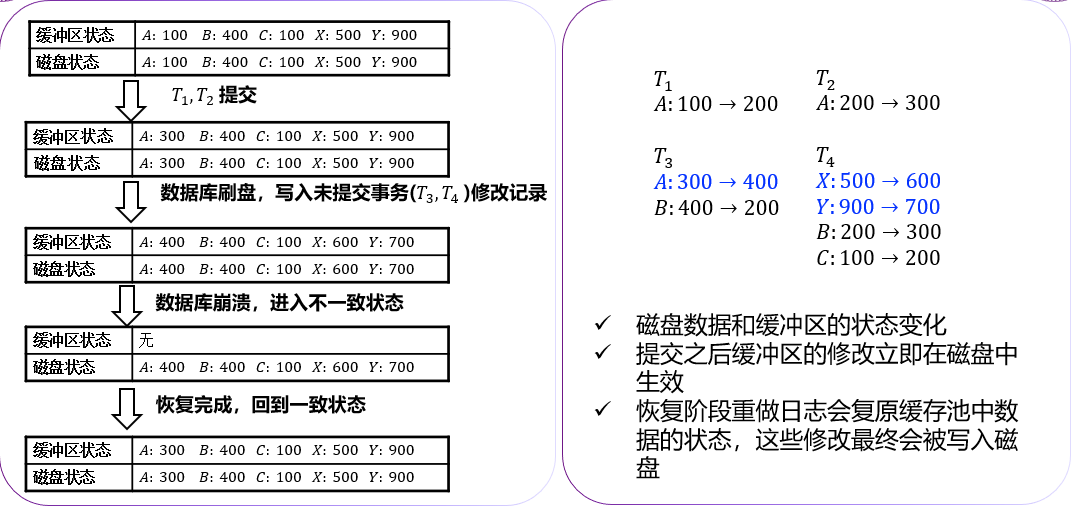




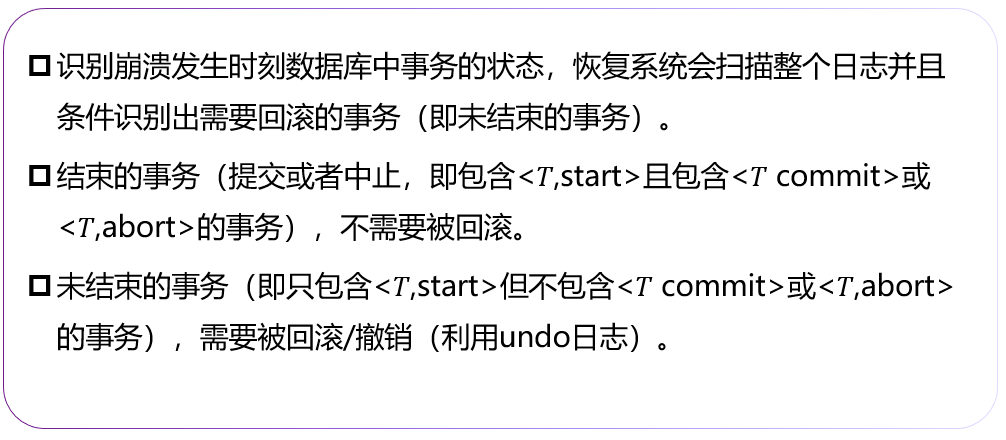


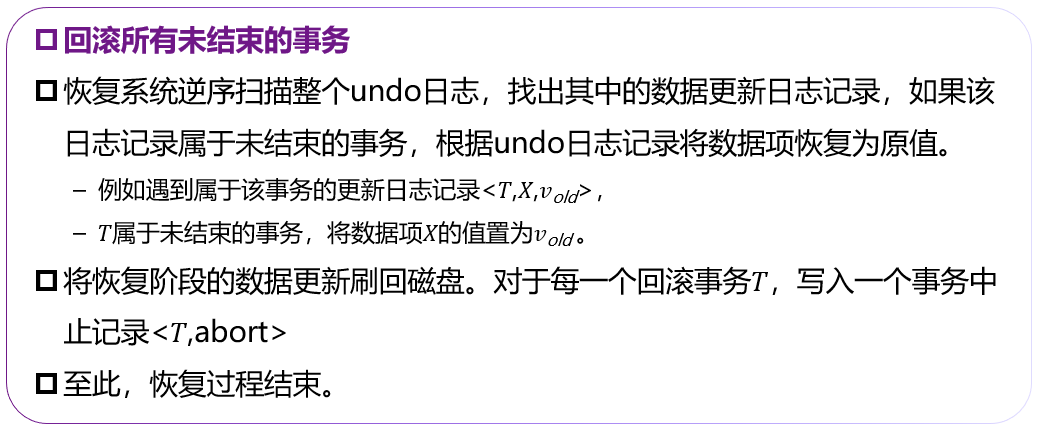
基于undo日志的恢复举例：



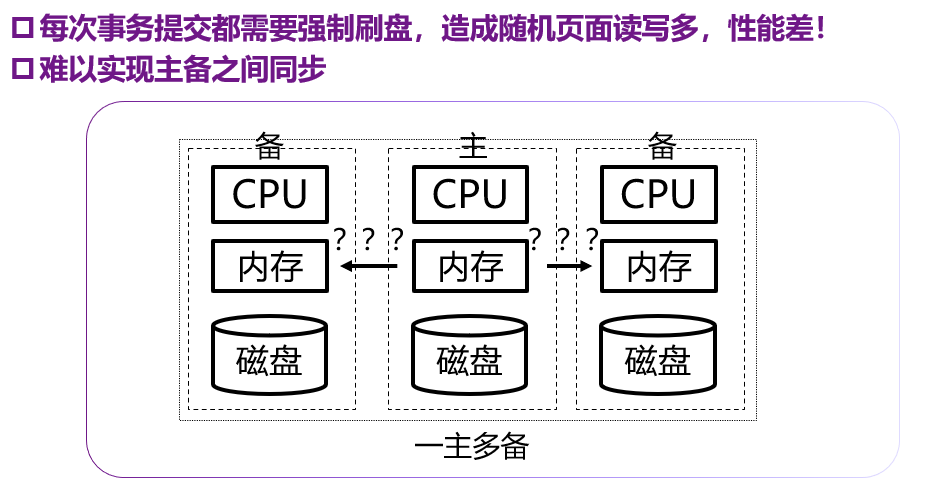


##### 基于undo日志的恢复流程





##### 基于undo日志的问题



## 优化

参考：<http://mysql.taobao.org/monthly/2021/06/01/>

# WAL

InnoDB WAL物理格式：<http://mysql.taobao.org/monthly/2020/01/06/>

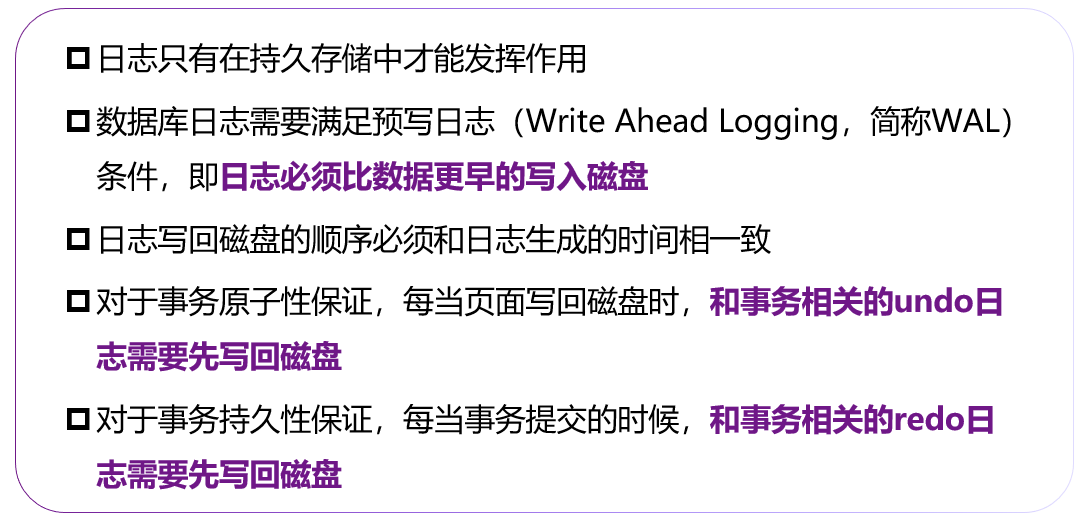
8.0 对WAL的设计修改：<http://mysql.taobao.org/monthly/2018/06/01/>

WAL那些事儿：<http://mysql.taobao.org/monthly/2018/07/01/>

Write-Ahead Logging机制浅析：<http://mysql.taobao.org/monthly/2017/03/02/>

MySQL的WAL（Write Ahead Log）在InnoDB中被称作redo log。

## 概述



## 原理

# Redo

参考：

InnoDB之redo log：<http://mysql.taobao.org/monthly/2020/02/01/>

redo log record编码格式：<http://mysql.taobao.org/monthly/2022/01/02/>

redo log重构：<http://mysql.taobao.org/monthly/2022/09/03/>

redo log之write ahead：<http://mysql.taobao.org/monthly/2020/01/05/>

redo log写入性能问题分析：<http://mysql.taobao.org/monthly/2020/01/04/>

InnoDB redo log thread cpu usage：<http://mysql.taobao.org/monthly/2020/07/05/>

link buf：<http://mysql.taobao.org/monthly/2020/06/04/>

Redo Logging动态开关：<http://mysql.taobao.org/monthly/2020/08/03/>

mini transaction详解：<http://mysql.taobao.org/monthly/2021/09/04/>

InnoDB Physiological logging 原理分析：

<http://mysql.taobao.org/monthly/2022/05/01/>

The design of mysql8.0 redolog：<http://mysql.taobao.org/monthly/2019/02/05/>

动态元信息持久化：<http://mysql.taobao.org/monthly/2019/12/01/>

Innodb redo log record 源码分析：<http://mysql.taobao.org/monthly/2019/08/03/>

增加系统文件追踪space ID和物理文件的映射：

<http://mysql.taobao.org/monthly/2019/04/07/>

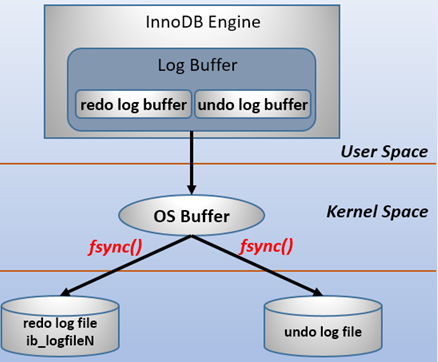
Innodb 引擎Redo日志存储格式简介：

<http://mysql.taobao.org/monthly/2017/09/07/>

## 概述

和undo log相反，redo log记录的是新数据的备份。**在事务提交前，只要将redo log持久化即可，不需要将数据持久化**。当系统崩溃时，虽然数据没有持久化，但是redo log已经持久化。系统可以根据redo log的内容，将所有数据恢复到最新的状态。

redo log包括两部分：一是内存中的日志缓冲(redo log buffer)，该部分日志是易失性的；二是磁盘上的重做日志文件(redo log file)，该部分日志是持久的。

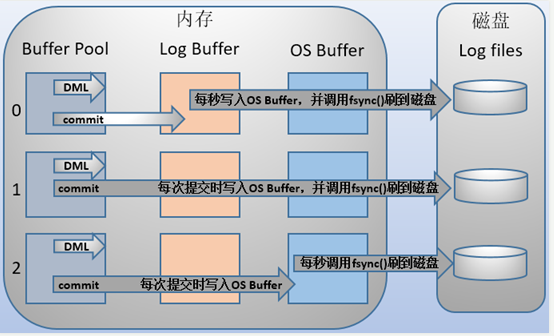


MySQL支持用户自定义在commit时如何将log buffer中的日志刷log file中。这种控制通过变量 innodb\_flush\_log\_at\_trx\_commit 的值来决定。该变量有3种值：0、1、2，默认为1。但注意，这个变量只是控制commit动作是否刷新log buffer到磁盘。

当设置为1的时候，事务每次提交都会将log buffer中的日志写入os buffer并调用fsync()刷到log file on disk中。这种方式即使系统崩溃也不会丢失任何数据，但是因为每次提交都写入磁盘，IO的性能较差。

当设置为0的时候，事务提交时不会将log buffer中日志写入到os buffer，而是每秒写入os buffer并调用fsync()写入到log file on disk中。也就是说设置为0时是(大约)每秒刷新写入到磁盘中的，当系统崩溃，会丢失1秒钟的数据。

当设置为2的时候，每次提交都仅写入到os buffer，然后是每秒调用fsync()将os buffer中的日志写入到log file on disk。



## 源码

### 文件

### 相关数据结构

## 原理

### InnoDB实现

#### 格式/物理存储结构

Redo log record编码方式：<http://mysql.taobao.org/monthly/2022/01/02/>

#### 日志块

innodb存储引擎中，redo log以块为单位进行存储的，每个块占512字节，这称为redo log block。所以不管是log buffer中还是os buffer中以及redo log file on disk中，都是这样以512字节的块存储的。

每个redo log block由3部分组成：日志块头、日志块尾和日志主体。其中日志块头占用12字节，日志块尾占用8字节，所以每个redo log block的日志主体部分只有512-12-8=492字节。

#### mini transaction

参考：

mini transaction详解：<http://mysql.taobao.org/monthly/2021/09/04/>

#### 日志刷盘规则

参考：<http://mysql.taobao.org/monthly/2014/08/02/>

log buffer中未刷到磁盘的日志称为脏日志(dirty log)。

默认情况下事务每次提交的时候都会刷事务日志到磁盘中，这是因为变量 innodb\_flush\_log\_at\_trx\_commit 的值为1。但是innodb不仅仅只会在有commit动作后才会刷日志到磁盘，这只是innodb存储引擎刷日志的规则之一。

刷日志到磁盘有以下几种规则：

1、发出commit动作时。已经说明过，commit发出后是否刷日志由变量 innodb\_flush\_log\_at\_trx\_commit 控制。

2、每秒刷一次。这个刷日志的频率由变量 innodb\_flush\_log\_at\_timeout 值决定，默认是1秒。要注意，这个刷日志频率和commit动作无关。

3、当log buffer中已经使用的内存超过一半时。

4、当有checkpoint时，checkpoint在一定程度上代表了刷到磁盘时日志所处的LSN位置。

#### 数据页刷盘规则

#### Purge

参考：<http://mysql.taobao.org/monthly/2022/03/01/>

<http://mysql.taobao.org/monthly/2022/05/04/>

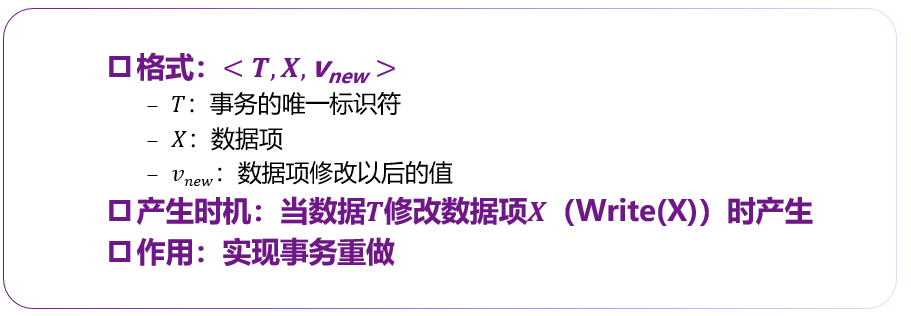
#### Checkpoint

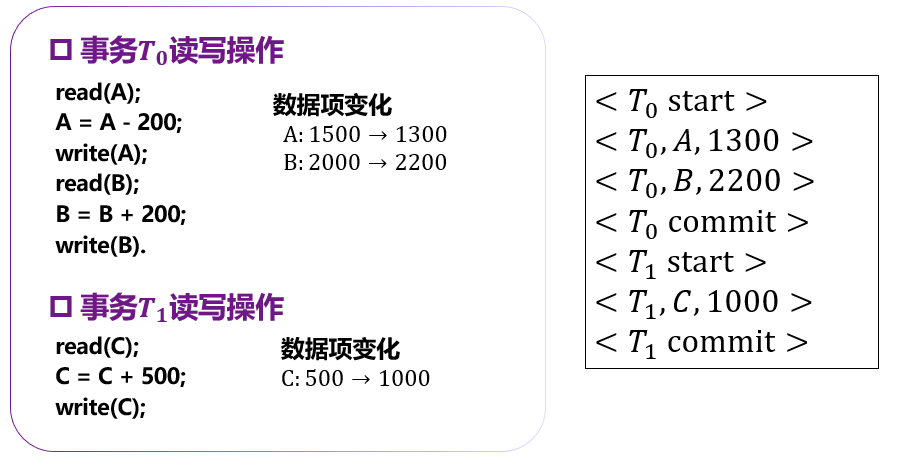
#### LSN

#### Innodb恢复行为

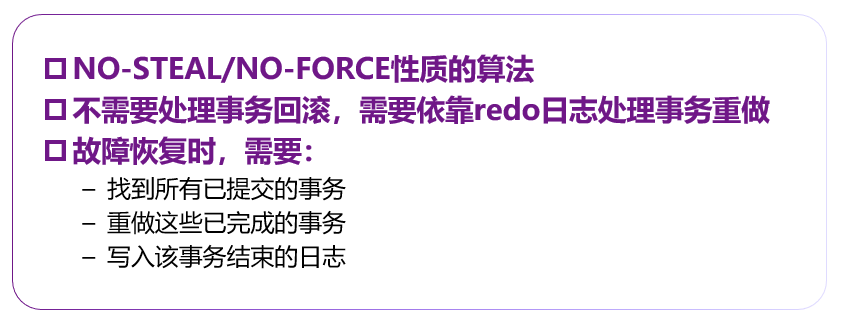
### 恢复

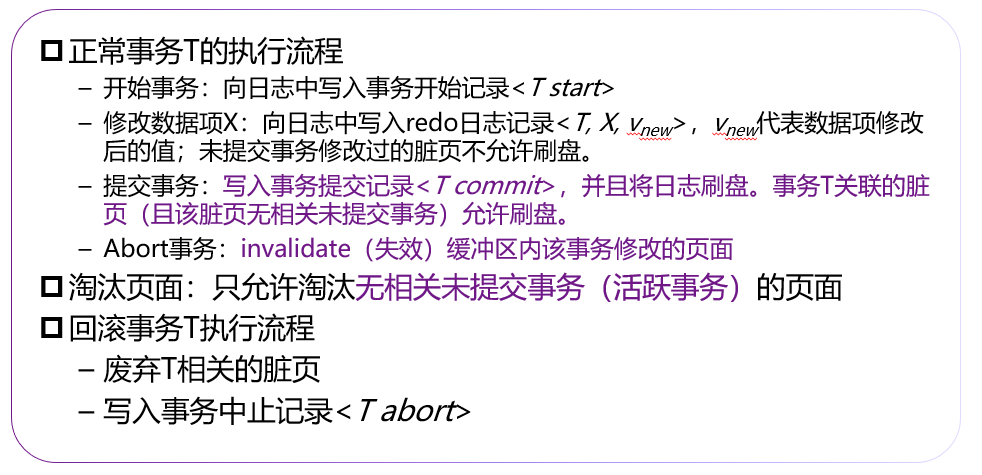
#### Redo重做日志

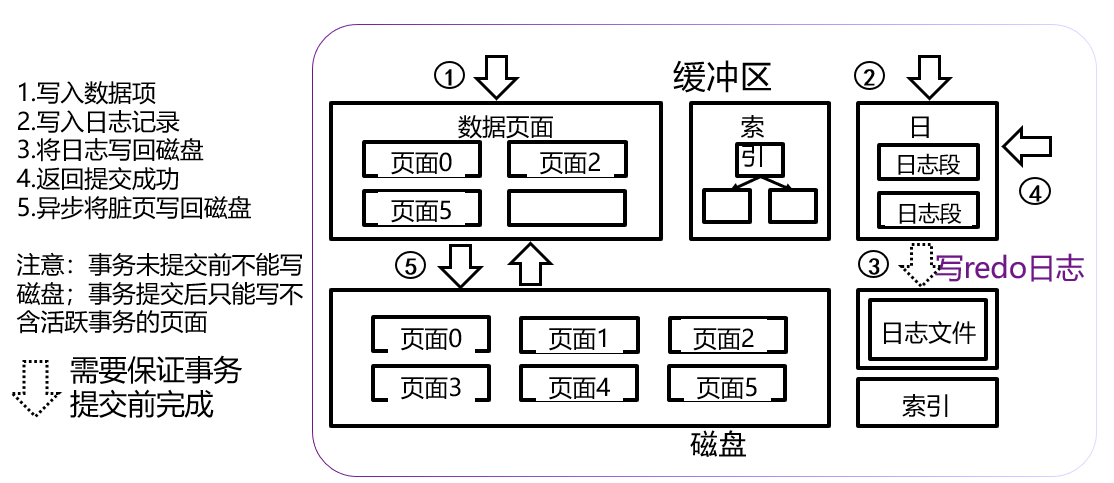




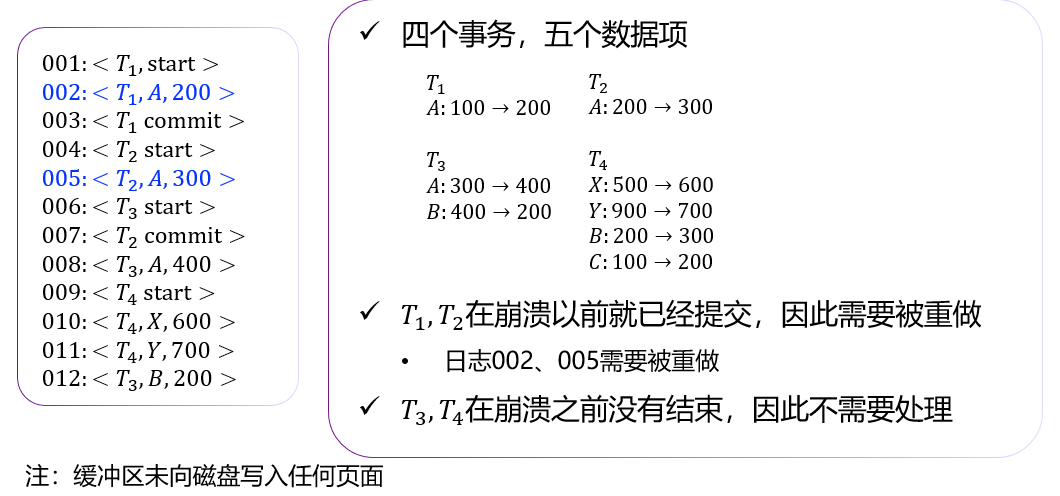
#### 基于redo日志的恢复

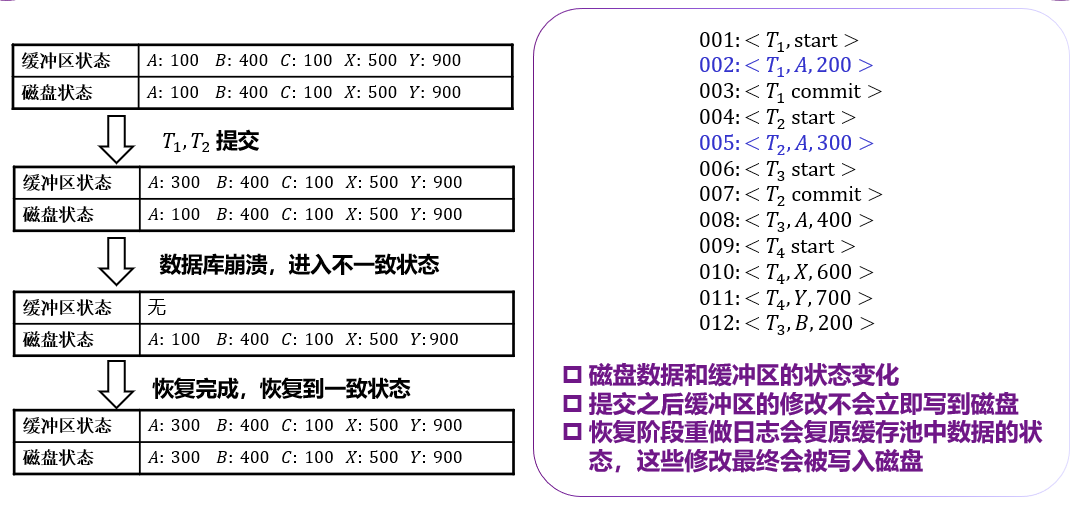




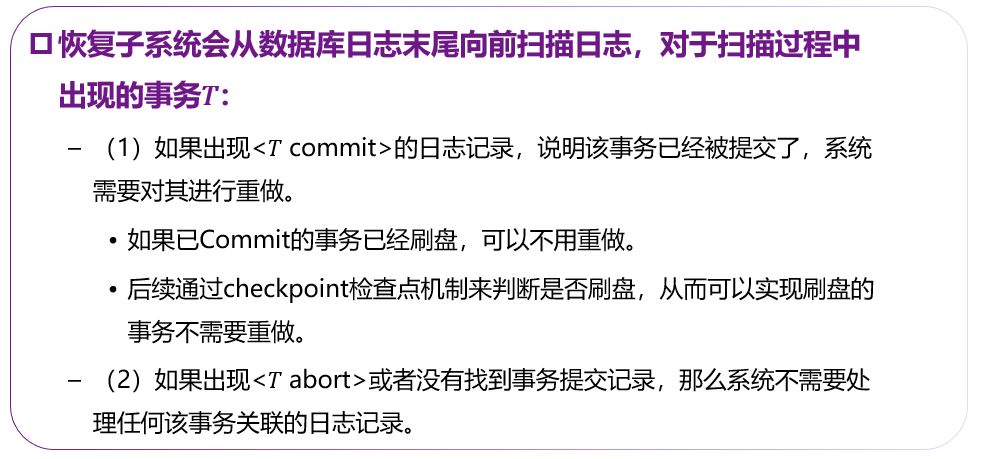


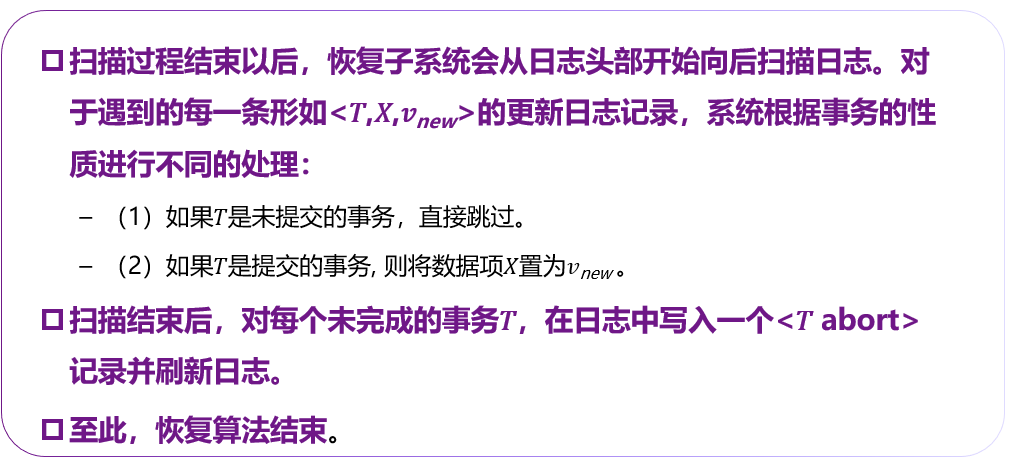
基于redo日志恢复举例：



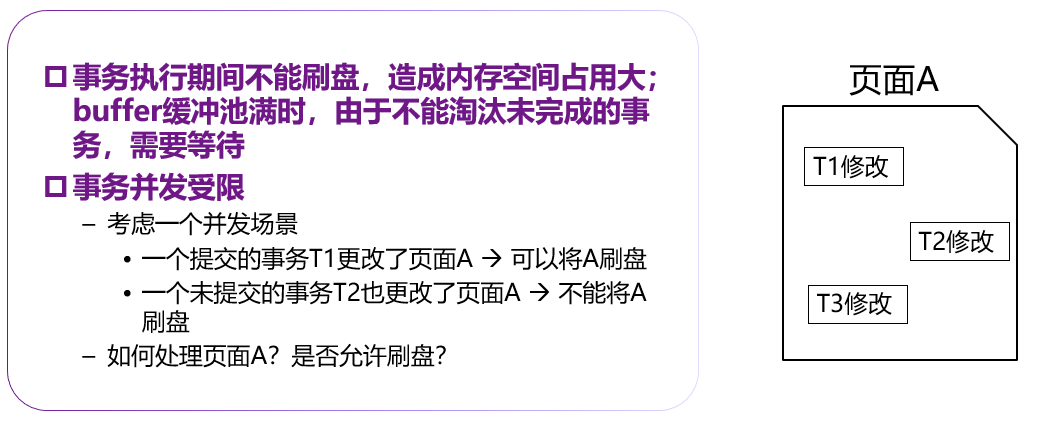


##### 基于redo日志的恢复流程

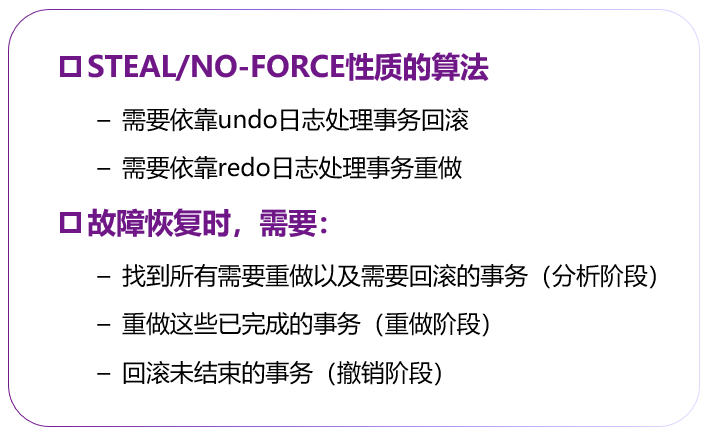


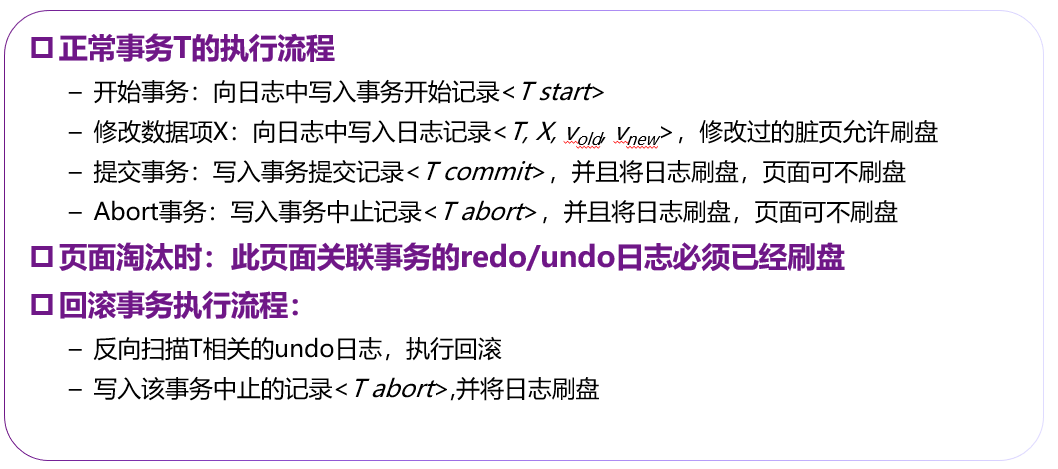


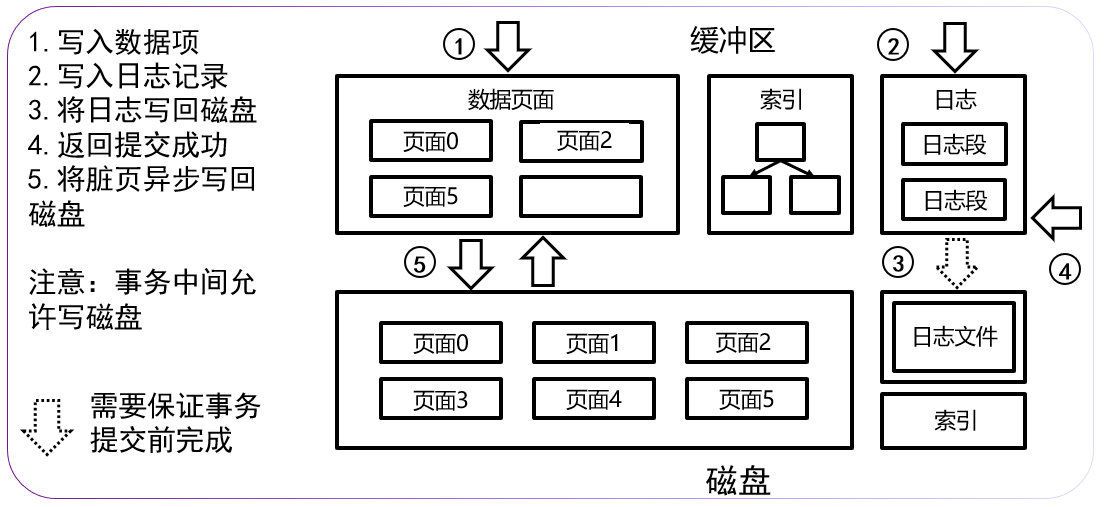
##### 基于redo日志的问题



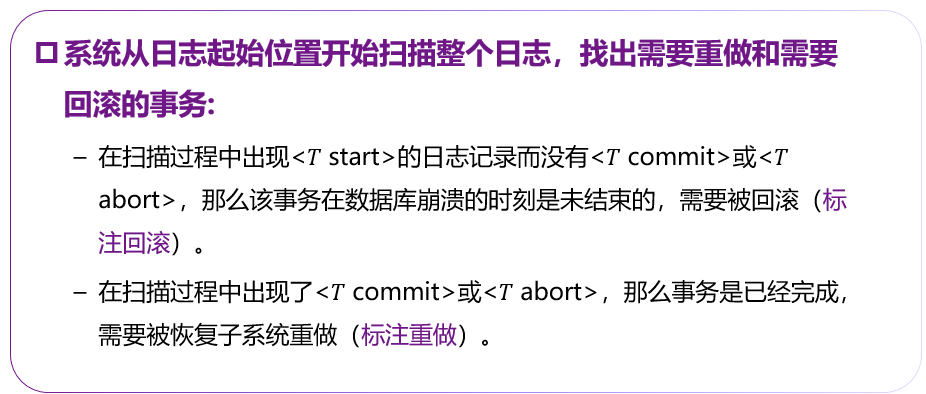
#### 基于undo/redo日志的恢复



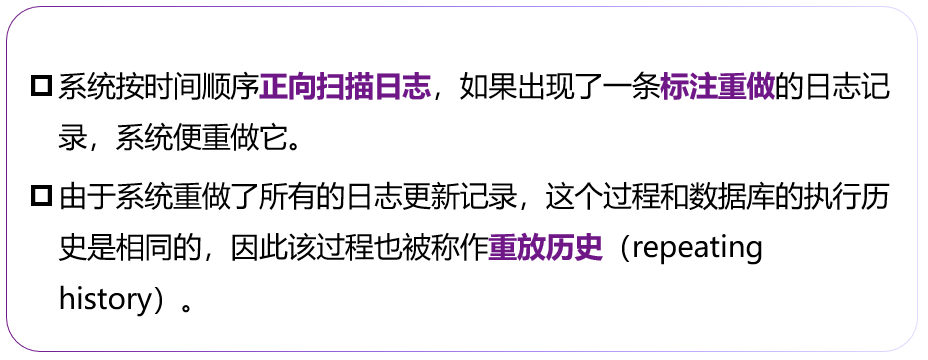




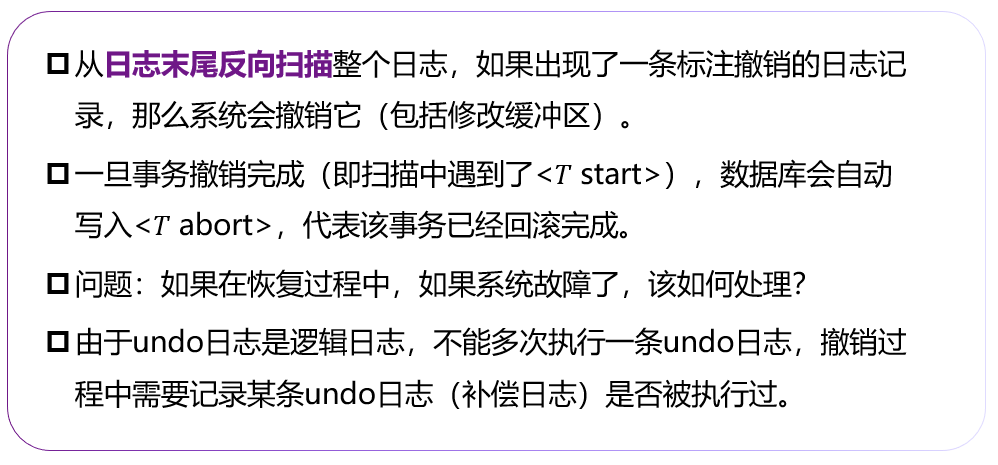
##### undo/redo恢复的分析阶段



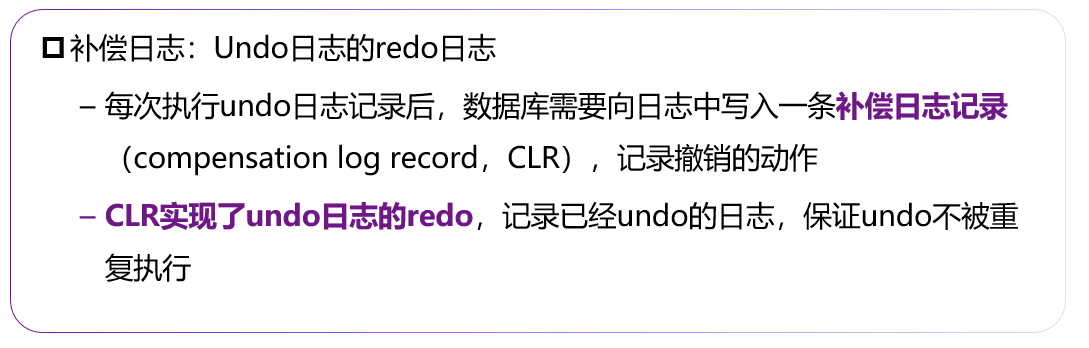
##### undo/redo恢复的重做阶段

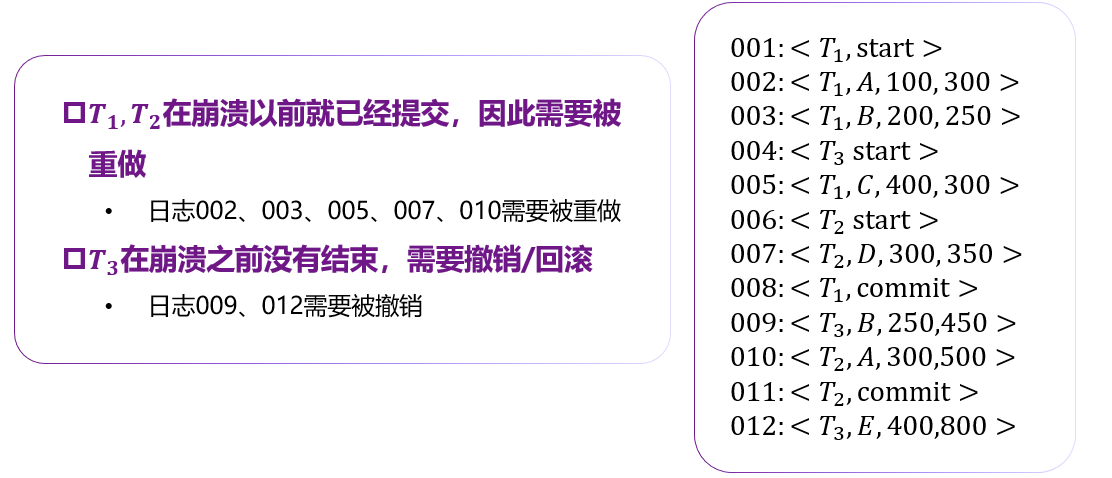


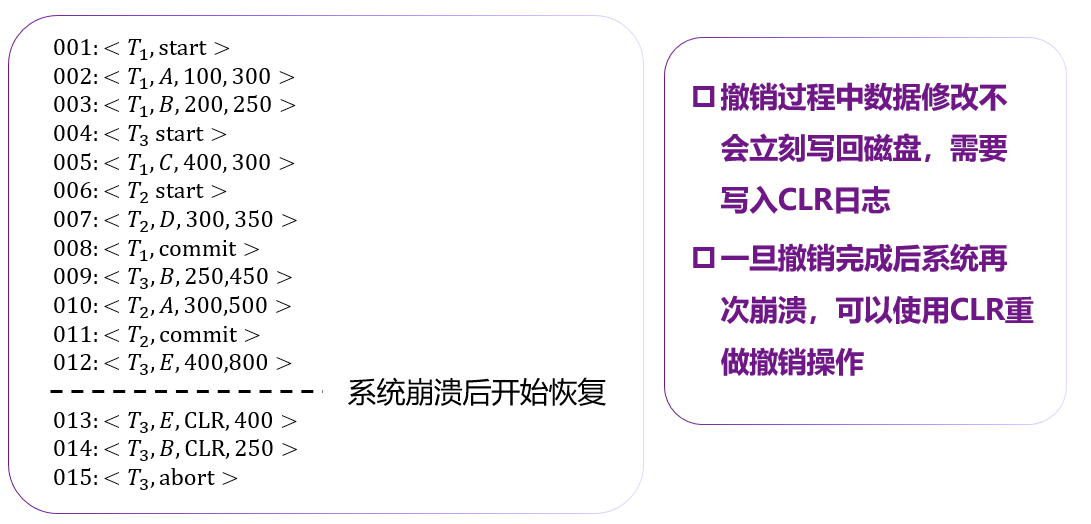
##### undo/redo恢复的撤销阶段



##### 基于undo/redo日志的恢复







## 组提交

## redo log与binlog

提交阶段InnoDB存储引擎要落盘redo log（undo log是不落盘的），MySQL Server服务器层要落盘binlog。

说明：这里binlog与redo log出于的层级完全不同，binlog位于Server层，而redo log是Engine层。

binlog和redo log之间的数据一致性问题

**必要性：**

**保证binlog存在的事务一定在redo log里面存在**。**主从复制架构中，主机崩溃恢复依赖redo log和binlog，从机数据来源是主机binlog**。

保证binlog里面事务顺序与redo log事务顺序一致。

**解决方案：**

引入XA协议

prepare阶段：

持锁prepare\_commit\_mutex

write/sync redo log

undo设置为prepared状态

commit阶段：

write/sync binlog

innodb commit，写入commit标记，释放prepare\_commit\_mutex锁

说明：

1. 以binlog写入与否作为事务提交成功与否的标志
2. 由于prepare\_commit\_mutex锁存在，保证binlog和redo log之间顺序一致，但是却导致每个事物都需要一个fsync操作，导致性能急剧下降。

**MySQL怎么知道binlog是完整的？**

一个事务的 binlog是有完整格式的：

* statement格式的binlog，最后会有COMMIT；
* row格式的binlog，最后会有一个XID event。

另外，在 MySQL 5.6.2 版本以后，还引入了 binlog-checksum 参数，用来验证binlog内容的正确性。对于binlog日志由于磁盘原因，可能会在日志中间出错的情况，MySQL 可以通过校验 checksum 的结果来发现。所以，MySQL 还是有办法验证事务 binlog 的完整性的。

**redo log和binlog是怎么关联起来的？**

这就是xid的存在的意义。它们有一个共同的数据字段，叫XID。崩溃恢复的时候，会按顺序扫描 redo log：

* 如果碰到既有prepare，又有有commit的redo log，就直接提交；
* 如果碰到只有parepare，而没有commit的redo log，就拿着XID去 binlog找对应的事务