Redo & Undo in InnoDB

- **InnoDB's Redo**
  - what?
    - DML操作导致的页面变化，均需要记录Redo日志；
    - **大部分为物理日志**；
  - when?
    - 在页面修改完成之后，在脏页刷出磁盘之前，写入Redo日志；
    - 日志先行，日志一定比数据页先写回磁盘；
    - **聚簇索引/二级索引/Undo页面修改，均需要记录Redo日志**；
- **InnoDB's Undo**
  - what?
    - DML操作导致的数据记录变化，均需要将记录的前镜像写入Undo日志；
    - **逻辑日志**；
  - when?
    - **DML操作修改聚簇索引前，记录Undo日志**(Undo日志，先于Redo日志，Why？)；
    - **二级索引记录的修改，不记录Undo日志**；
    - 注意：Undo页面的修改，同样需要记录Redo日志；

Redo & Undo in InnoDB

- DML Operations
  - 用户的三种DML操作：Insert/Delete/Update分别会如何记录日志？

- **Insert**
  - Undo
    - 将插入记录的主键值，写入Undo；
  - Redo
    - 将[space_id, page_no, 完整插入记录, 系统列, ...]写入Redo；
    - space_id, page_no 组合代表了日志操作的页面；

Redo & Undo in InnoDB

- **Delete**
  - Undo
    - 1. Delete，在InnoDB内部为Delete Mark操作，将记录上标识Delete_Bit，而不删除记录；
    - 2. 将当前记录的系统列写入Undo (DB_TRX_ID, ROLLBACK_PTR, ...)；
    - 3. 将当前记录的主键列写入Undo；
    - 4. 将当前记录的所有索引列写入Undo (why? for what?)；
    - 5. 将Undo Page的修改，写入Redo；
  - Redo
    - 将[space_id, page_no, 系统列, 记录在页面中的Slot, ...]写入Redo；

Redo & Undo in InnoDB

- **Update(*未修改聚簇索引键值，属性列长度未变化*)**
  - **Undo (聚簇索引)**
    - 1. 将当前记录的系统列写入Undo (DB_TRX_ID, ROLLBACK_PTR, ...)；
    - 2. 将当前记录的主键列写入Undo；
    - 3. 将当前Update列的前镜像写入Undo；
    - 4. 若Update列中包含二级索引列，则将二级索引其他未修改列写入Undo；
    - 5. 将Undo页面的修改，写入Redo；
  - **Redo**
    - 进行**In Place Update**，记录Update Redo日志(**聚簇索引**)；
    - 若更新列包含**二级索引**列，二级索引肯定不能进行In Place Update，记录**Delete Mark** + Insert Redo日志；

Redo & Undo in InnoDB

- **Update(*未修改聚簇索引键值，属性列长度发生变化*)**
  - **Undo (聚簇索引)**
    - 1. 将当前记录的系统列写入Undo (DB_TRX_ID, ROLLBACK_PTR, ...)；
    - 2. 将当前记录的主键列写入Undo；
    - 3. 将当前Update列的前镜像写入Undo；
    - 4. 若Update列中包含二级索引列，则将二级索引其他未修改列写入Undo；
    - 5. 将Undo页面的修改，写入Redo；
  - **Redo**
    - 不可进行**In Place Update**，记录**Delete + Insert Redo**日志(**聚簇索引**)；
    - 若更新列包含**二级索引**列，二级索引肯定不能进行In Place Update，记录**Delete Mark** + Insert Redo日志；

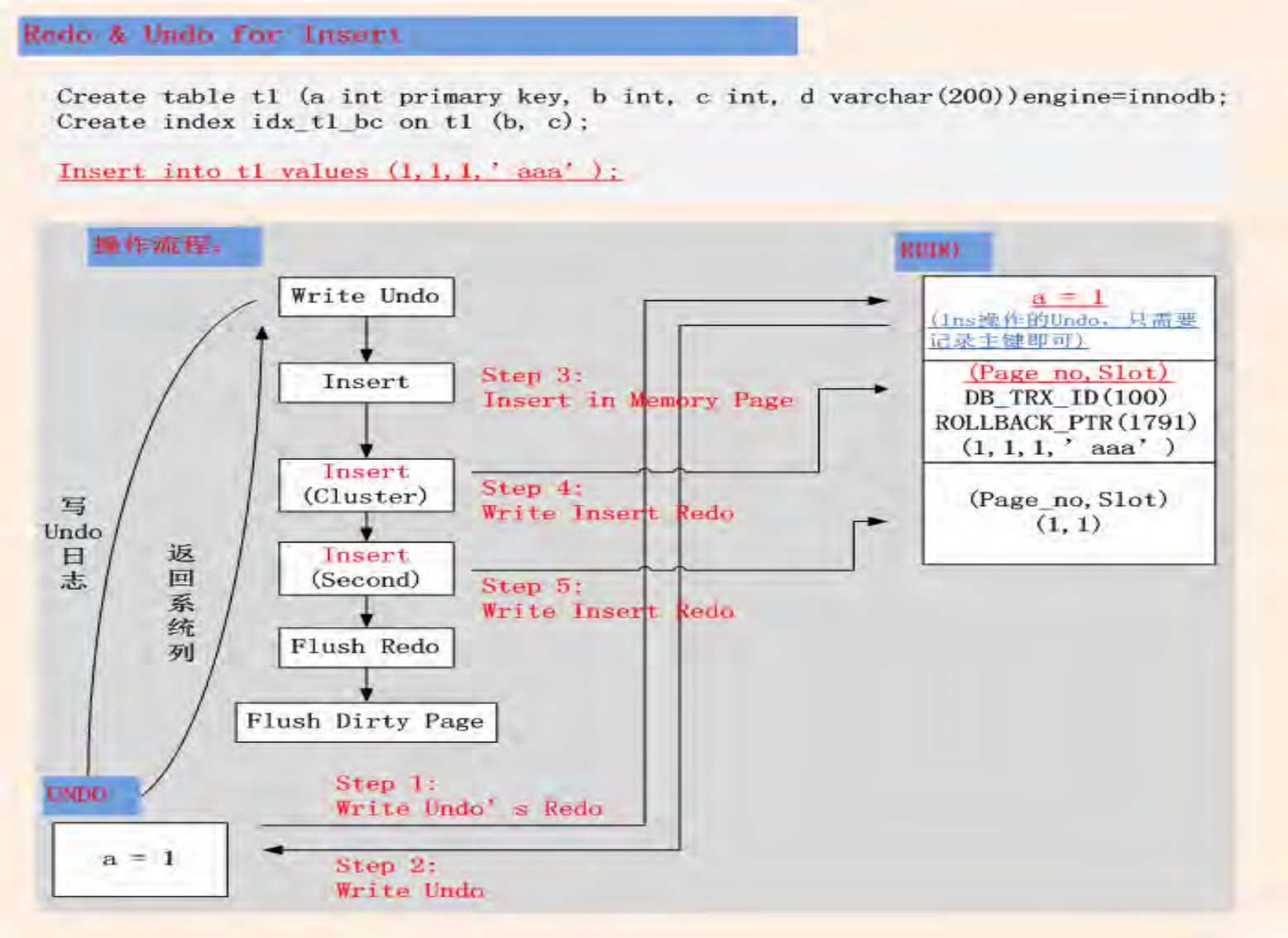
Redo & Undo in InnoDB

- **Update(*修改聚簇索引键值*)**
  - **Undo (聚簇索引)**
    - 1. 不可进行In Place Update。Update = **Delete Mark + Insert**；
    - 2. 对原有记录进行Delete Mark操作，写入Delete Mark操作Undo；
    - 3. 将新纪录插入聚簇索引，写入Insert操作Undo；
    - 4. 将Undo页面的修改，写入Redo；
  - **Redo**
    - 不可进行In Place Update，记录**Delete Mark + Insert Redo**日志(**聚簇索引**)；
    - 若更新列包含**二级索引**列，二级索引肯定不能进行In Place Update，记录Delete Mark + Insert Redo日志；

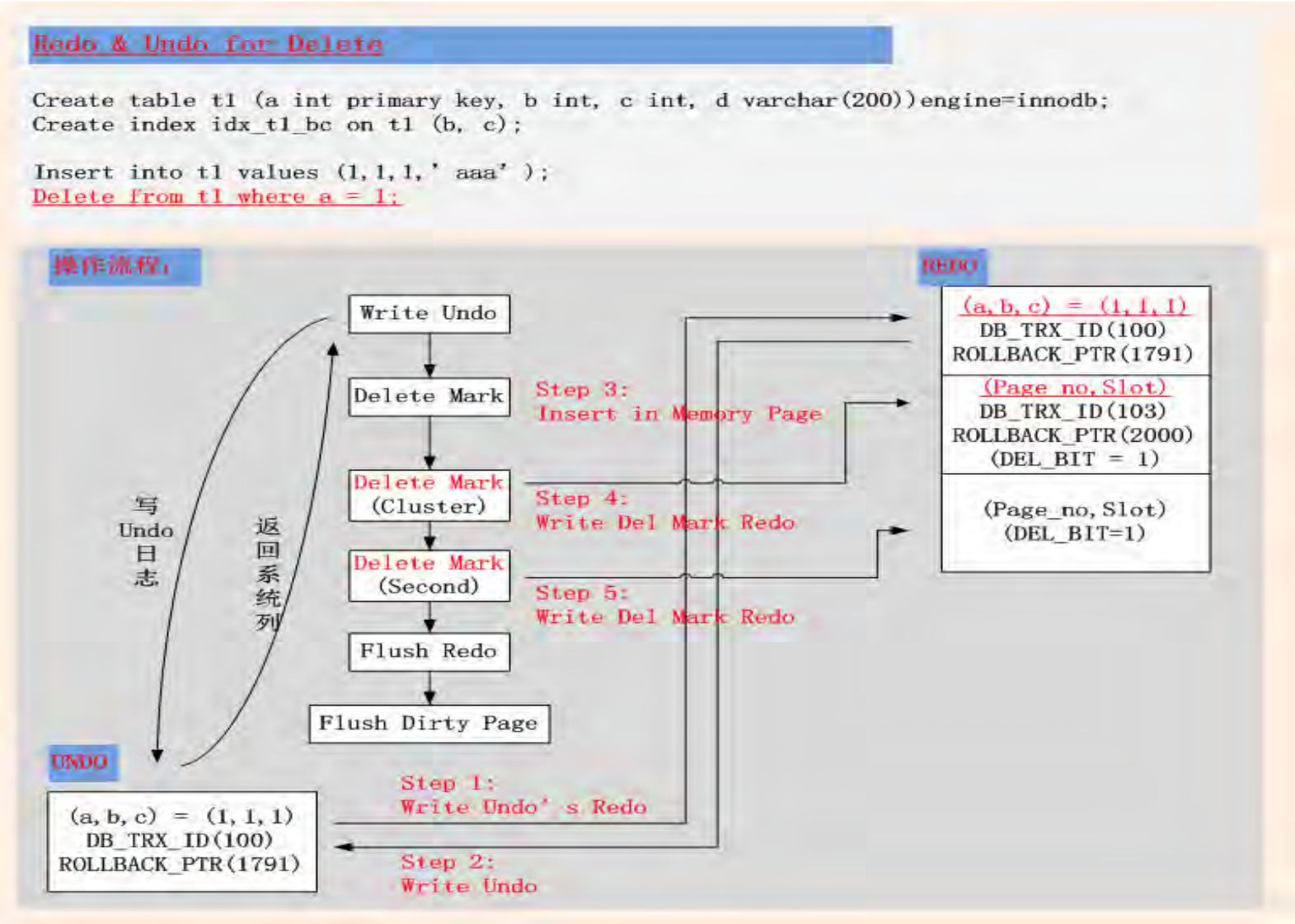
Redo & Undo in InnoDB

- Examples
  - 测试准备
    - create table t1 (a int primary key, b int, c int, d varchar(200))engine=innodb;
    - create index idx_t1_bc on t1 (b, c);
  - **DML语句**
    - 以下不同的DML语句，分别如何记录 Redo/Undo？
    - 语句1：**insert** into t1 values (1,1,1,'aaa');
    - 语句2：**delete** from t1 where a = 1;
    - 语句3：**update** t1 set b = 2, **d = 'bbb'** where a = 1;
    - 语句4：**update** t1 set b = 2, **d= 'bbbb'** where a = 1;
    - 语句5：**update** t1 set **a = 10**, b = 2, d = 'bbbb' where a = 1;

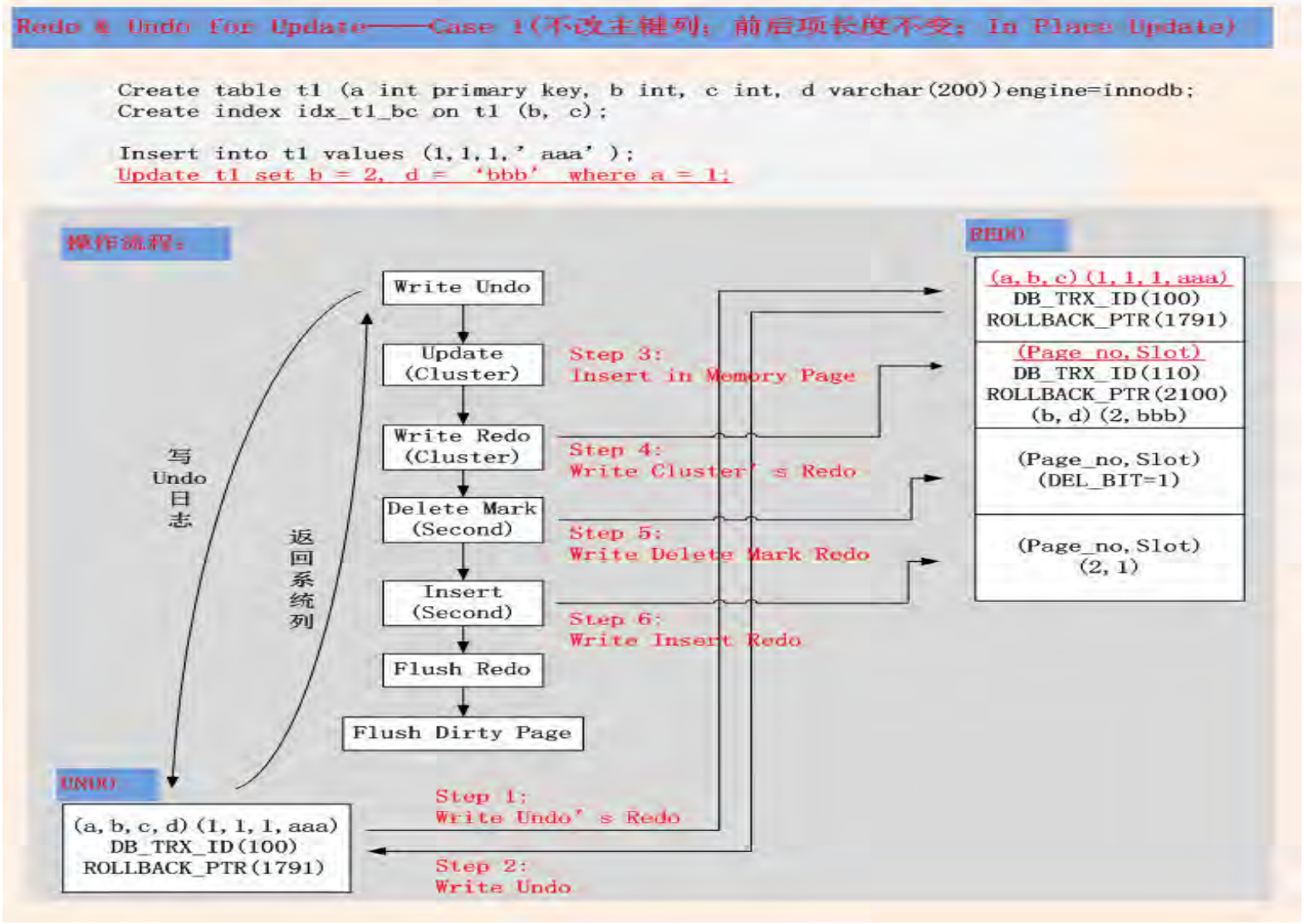
Redo & Undo in InnoDB



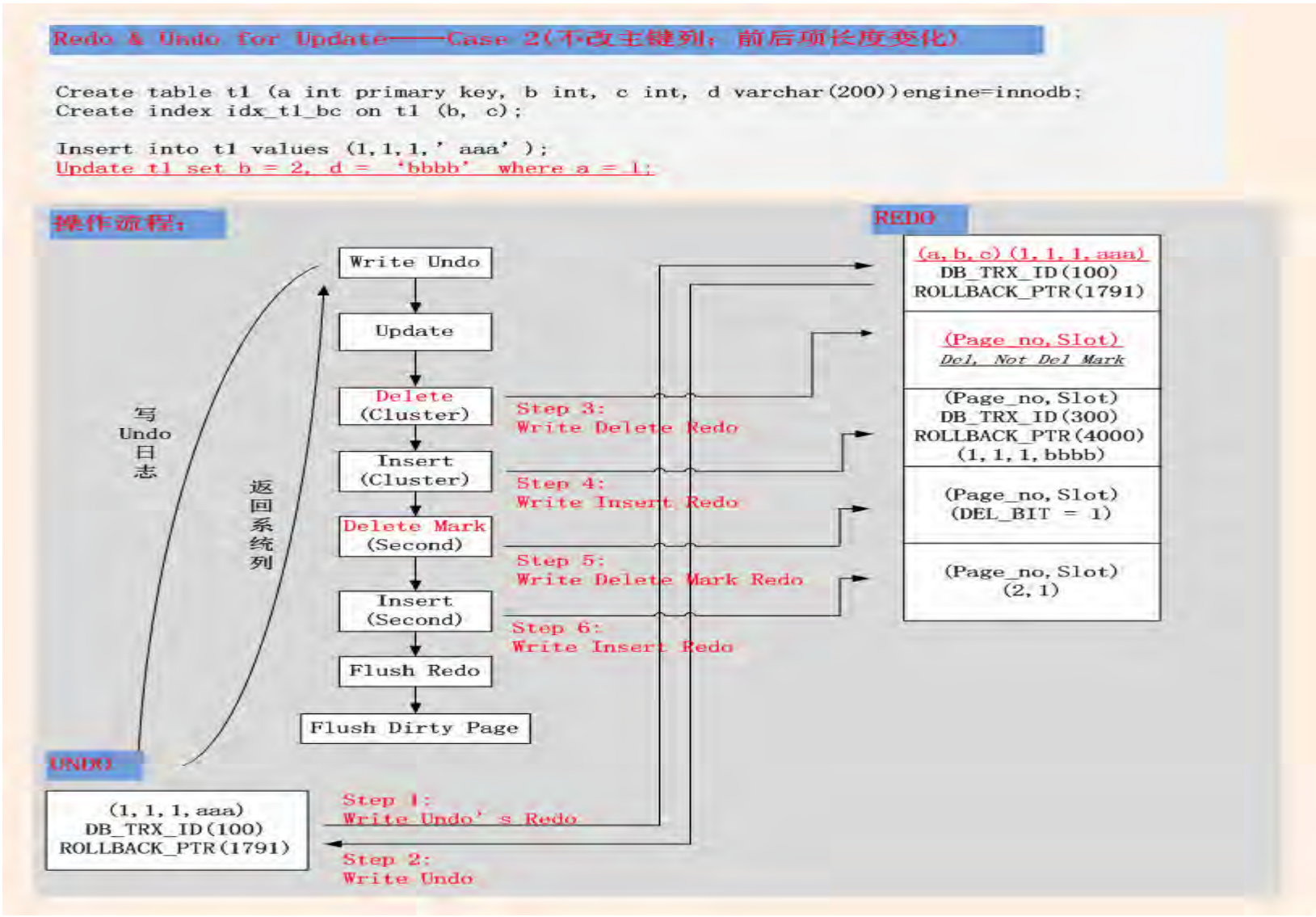
Redo & Undo in InnoDB



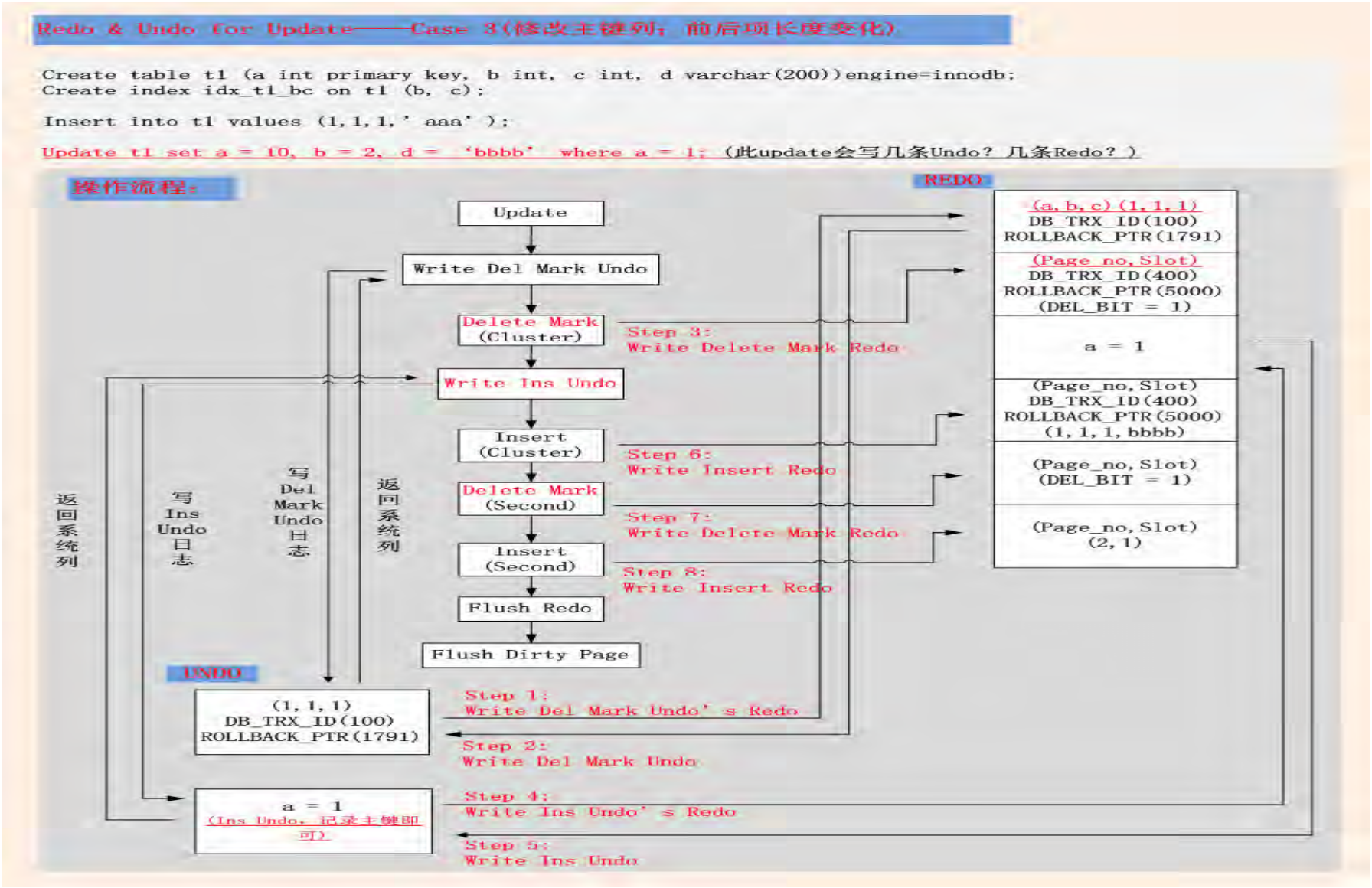
Redo & Undo in InnoDB



Redo & Undo in InnoDB



Redo & Undo in InnoDB



Redo & Undo Summary

- InnoDB遵循WAL协议，在日志持久化到磁盘之后，才会将日志对应的脏页刷回磁盘；
- InnoDB内存中，DML操作顺序如下：
  - 写Undo(**获取ROLLBACK_PTR系统列**)
  - 修改Page
  - 写Redo的顺序
- 不同的Update语句，写的日志量有较大差异(三种Update Case)；
  - In Place Update日志量最小，操作最简单；
  - 不修改主键列，日志量其次；
  - 修改主键列，日志量最大，操作最复杂；

Redo Types in InnoDB

- Redo Types
  - InnoDB的Redo Types，按照维度的不同，有多种分类方式
    - Physical vs Logical
    - Single vs Multiple
    - Data Page's Redo vs Undo Page's Redo
    - ...
- InnoDB中最主要的三种Redo日志类型
  - Physical Redo
    - **MLOG_SINGLE_REC**
    - **MLOG_MULTI_REC**
  - **Logical Redo**
    - 例如：MLOG_PAGE_CREATE；MLOG_UNDO_HDR_CREATE；...

Redo Types in InnoDB

- **Physical Redo**
  - **MLOG_SINGLE_REC**
    - **当前日志，记录的是一个Page的一个Redo日志；**
    - 对应操作：**简单的I/U/D，Undo的Redo**等；
    - 例如：一个Insert操作，会产生3个MLOG_SINGLE_REC，分别对应：聚簇索引页；二级索引页；Undo页；
    - MLOG_SINGLE_REC日志，一定是有效的；
  - **MLOG_MULTI_REC**
    - **当前日志，是一组日志中的一个，这一组日志，包含了多个Page的多条Redo日志；**
    - 对应操作：I/U/D导致的索引分裂，合并；Varchar/LOB导致的链接行等；
    - 例如：Insert使得聚簇索引分裂，分裂操作需要涉及至少3个Page，这三个Pages上的所有修改日志，均为MLOG_MULTI_REC中的一部分；
    - **MLOG_MULTI_REC日志组，只有当最后一条MLOG_MULTI_REC_END写出之后，才起作用；否则全部丢弃；**

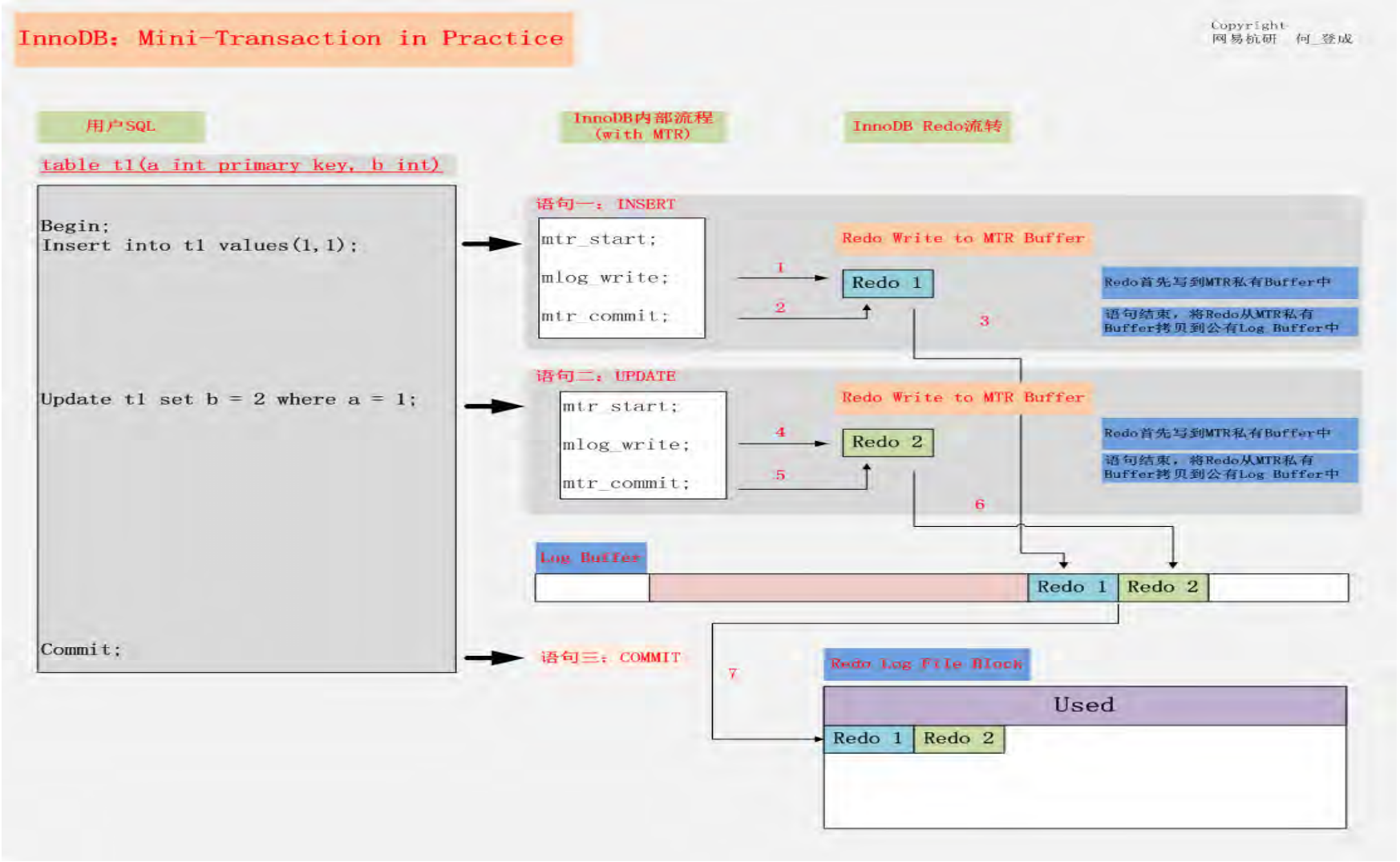
Redo Types in InnoDB

- **Logical Redo**
  - **逻辑Redo，不是记录页面的实际修改，而是记录修改页面的一类固定操作；**
  - 例如：如何写页面初始化日志？
    - 写MLOG_COMP_PAGE_CREATE日志；
    - 重做此日志，只需再次调用page0page.c::page_create方法初始化对应的Page即可；
  - MLOG_COMP_PAGE_CREATE；MLOG_UNDO_HDR_CREATE；MLOG_IBUF_BITMAP_INIT；...
  - 这类动作是固定的，减少Redo的一个优化；

Mini-Transaction

- Mini-TRansaction(MTR)
  - 定义
    - mini-transaction不属于事务；InnoDB内部使用
    - 对于InnoDB内所有page的访问(I/U/D/S)，都需要mini-transaction支持
  - 功能
    - 访问page，对page加latch (*只读访问：S latch；写访问：X latch*)
    - 修改page，写redo日志 (*mtr本地缓存*)
    - page操作结束，提交mini-transaction (*非事务提交*)
      - 将redo日志写入log buffer
      - 将脏页加入Flush List链表
      - 释放页面上的 S/X latch
  - 总结
    - mini-transaction，保证单page操作的原子性(读/写单一page)
    - mini-transaction，保证多pages操作的原子性(索引SMO/记录链出，多pages访问的原子性)

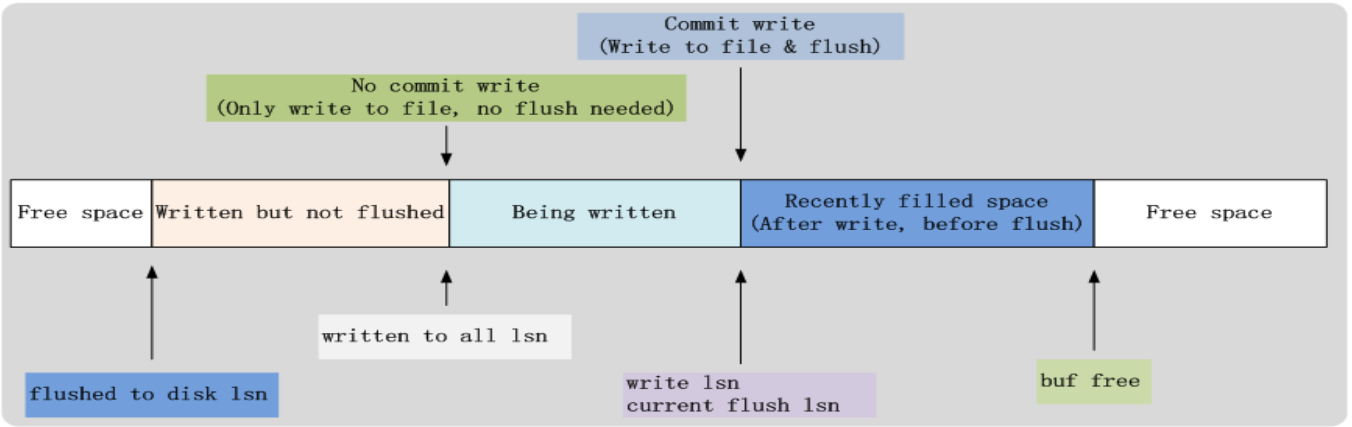
Mini-Transaction in Practice



Log Buffer Structure

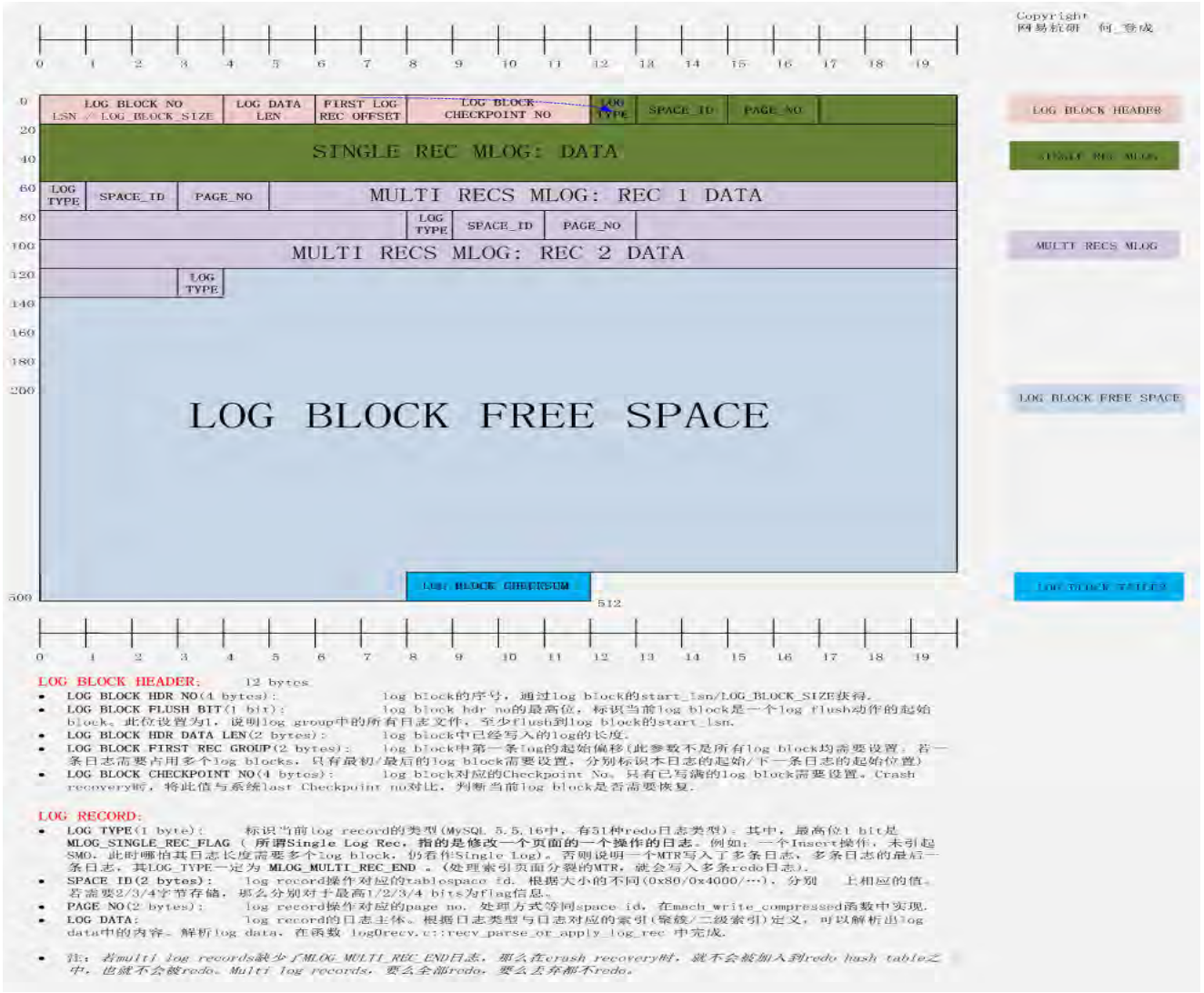
- Log Buffer
  - 参数：innodb_log_buffer_size；默认值：**8388608 (8M)**

- Log Buffer 结构



- Important Pointers
  - flushed_to_disk_lsn：　此指针之前的日志已经flush到磁盘
  - written_to_all_lsn：　此指针之前的日志已经写文件，但是未flush
  - write_lsn/current_flush_lsn：　此指针之前的日志，正在写文件
  - buf free：　log buffer的空闲起始位置

Redo Block Structure



LOG BLOCK HEADER: 12 bytes

- **LOG BLOCK HDR NO**(4 bytes): log block的序号，通过log block的start_lsn/LOG_BLOCK_SIZE获得.
- **LOG BLOCK FLUSH BIT**(1 bit): log block hdr no的最高位，标识当前log block是一个log flush动作的起始block。此位设置为1，说明log group中的所有日志文件，至少flush到log block的start_lsn.
- **LOG BLOCK HDR DATA LEN**(2 bytes): log block中已经写入的log的长度.
- **LOG BLOCK FIRST REC GROUP**(2 bytes): log block中第一条log的起始偏移(此参数不是所有log block均需要设置，若一条日志需要占用多个log blocks，只有最初/最后的log block需要设置，分别标识本日志的起始/下一条日志的起始位置)
- **LOG BLOCK CHECKPOINT NO**(4 bytes): log block对应的Checkpoint No。只有已写满的log block需要设置。Crash recovery时，将此值与系统last Checkpoint no对比，判断当前log block是否需要恢复.

LOG RECORD:

- **LOG TYPE**(1 byte): 标识当前log record的类型(MySQL 5.5.16中，有51种redo日志类型)。其中，最高位1 bit是**MLOG_SINGLE_REC_FLAG**（**所谓Single Log Rec，指的是修改一个页面的一个操作的日志**。例如：一个Insert操作，未引起SMO，此时哪怕其日志长度需要多个log block，仍看作Single Log)。否则说明一个MTR写入了多条日志，多条日志的最后一条日志，其LOG_TYPE一定为 **MLOG_MULTI_REC_END** 。(处理索引页面分裂的MTR，就会写入多条redo日志)。
- **SPACE ID**(2 bytes): log record操作对应的tablespace id. 根据大小的不同(0x80/0x4000/…)，分别 上相应的值。若需要2/3/4字节存储，那么分别对于最高1/2/3/4 bits为flag信息。
- **PAGE NO**(2 bytes): log record操作对应的page no. 处理方式等同space id，在mach_write_compressed函数中实现.
- **LOG DATA**: log record的日志主体。根据日志类型与日志对应的索引(聚簇/二级索引)定义，可以解析出log data中的内容。解析log data，在函数 log0recv.c::recv_parse_or_apply_log_rec 中完成.

- 注：*若multi log records缺少了MLOG_MULTI_REC_END日志，那么在crash recovery时，就不会被加入到redo hash table之中，也就不会被redo。Multi log records，要么全部redo，要么丢弃都不redo。*

Redo Log File

- Redo Log File
  - 顾名思义，保存InnoDB Redo日志的文件；
- 控制参数
  - innodb_log_file_size
    - 单个日志文件大小；
    - 默认值：**5242880 (5M)**
  - innodb_log_files_in_group：
    - 一个日志组中，日志文件的个数；
    - 默认值：2
- 日志文件空间总量
  - 可用的日志文件空间总量 = 单个日志文件大小 * 日志组中的日志文件个数

LSN

- LSN
  - Log Sequence Number，日志序列号；
  - LSN递增产生；
  - LSN可唯一标识一条Redo日志；
  - LSN有重要的意义；
    - Checkpoint LSN：标识数据库崩溃恢复的Redo起点；
  - LSN与日志文件位置，一一对应；

- LSN与日志文件位置关系
  - **日志文件可用空间总量**
    - group_size = (group->file_size – LOG_FILE_HDR_SIZE) * group->n_files;
    - 每个日志文件，去除LOG_FILE_HDR_SIZE (4 * 512 bytes)，余下的既为日志文件可用空间；
  - LSN to log file position
    - **log file number** = (LSN % group_size) / group->file_size
    - **log file position** = (LSN % group_size) % group->file_size

Redo Durability

- Log Write触发
  - 事务提交/回滚
    - 参数：innodb_flush_log_at_trx_commit
    - 事务提交，一定写日志，此参数控制写完是否flush
  - log buffer的log free指针超过max_buf_free

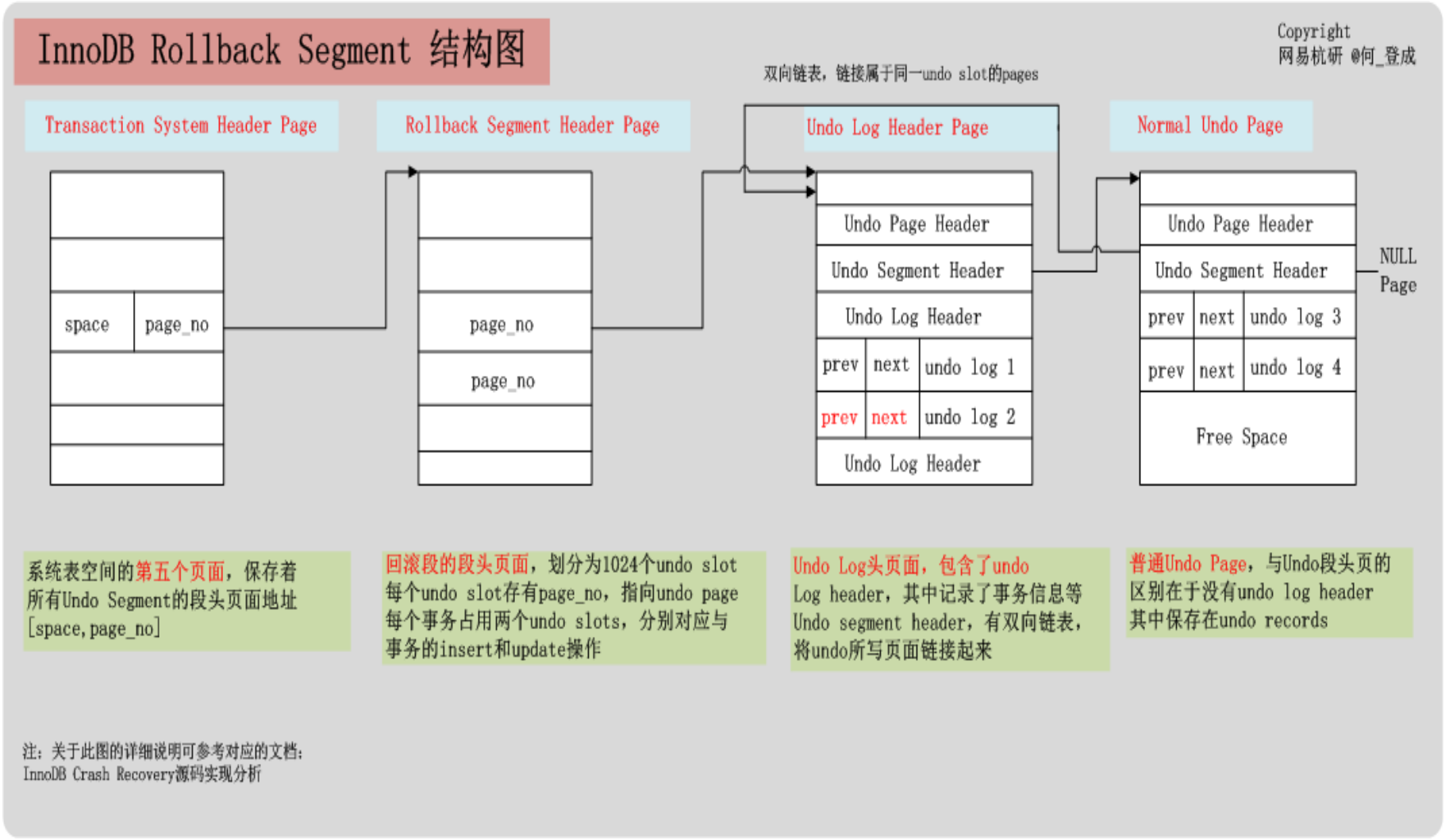
```
#define OS_FILE_LOG_BLOCK_SIZE            512
#define UNIV_PAGE_SIZE                    (2 * 8192)

/* A margin for free space in the log buffer before a log entry is catenated */
#define LOG_BUF_WRITE_MARGIN              (4 * OS_FILE_LOG_BLOCK_SIZE)

/* Margins for free space in the log buffer after a log entry is catenated */
#define LOG_BUF_FLUSH_RATIO 2


#define LOG_BUF_FLUSH_MARGIN              (LOG_BUF_WRITE_MARGIN + 4 * UNIV_PAGE_SIZE)

Log_sys->max_buf_free = log_sys->buf_size / LOG_BUF_FLUSH_RATIO - LOG_BUF_FLUSH_MARGIN;
```

    - **InnoDB在写完日志之后，均会检查log buffer前面的空闲空间，若前面的空闲空间超过max_buf_free的一半，就会将log buffer内容向前移动**
  - 后台线程，1S检查一次
  - 用户线程，在修改页面时进行检查

InnoDB Undo

- **Undo的存储**
  - InnoDB的Undo，存储于回滚段(**Rollback Segment**)之中(与Oracle类似)；
  - 回滚段，使用的是普通数据文件(Tablespace 0)；
    - MySQL 5.6.3之后，可通过**innodb_undo_tablespace**设置undo存储的位置；
  - Undo Page的修改，同样需要记录Redo日志；
- **Undo的功能**
  - 用户的DML操作，均需要记录Undo；
  - 对于回滚的事务，Undo可用来将事务的操作全部撤销；
    - **Rollback**
  - 对于提交的事务，Undo可用来将事务产生的过期版本回收；
    - **Purge**

Rollback Segment



Transaction System Header Page

- **Transaction System Header Page**
  - 系统表空间的第五个Page[TRX_SYS_SPACE，TRX_SYS_PAGE_NO]
  - **事务系统头页面；**
  - **存储内容**
    - TRX_SYS_TRX_ID_STORE [0..8]
      - InnoDB定期持久化的最大事务ID(非实时)；
      - **系统Crash Recovery后，第一个事务ID = 此ID + TRX_SYS_TRX_ID_UPDATE_MARGIN；**
    - **回滚段段头页地址** [**TRX_SYS_RSEG_SPACE, TRX_SYS_RSEG_PAGE_NO**]；
      - 参数：innodb_rollback_segments
    - MySQL Binlog信息；
    - Double Write信息；
    - ...

Transaction System Header Page

- Transaction System Header Page
  - 功能
    - InnoDB系统中，最为重要的一个页面；
    - 恢复时，哪些事务需要回滚；
    - 启动后，第一个事务ID的分配起点；
    - 运行中，回滚段的管理；
    - MySQL Binlog信息管理；
    - Double Write信息管理；
    - ...

Rollback Segment Header Page

- Rollback Segment Header Page
  - 回滚段段头页
  - InnoDB支持多回滚段
    - 参数： innodb_rollback_segments　　默认：128
  - 每个回滚段，占用一个Rollback Segment Header Page
  - 每个回滚段段头页地址[space_id, page_no]，存储于Transaction System Header Page中

Rollback Segment Header Page

- 存储内容
  - **Transaction Rollback Segment Header**
    - **TRX_RSEG_HISTORY_SIZE**
      - 已提交，但是未Purge的事务所占用的Undo Pages数量 (Update/Delete操作);
      - 参数：innodb_max_purge_lag (Purge页详细介绍)
    - **TRX_RSEG_HISTORY**
      - 统一管理提交之后的Undo Log Pages；
      - 双向链表，链表项为Undo Log Header Page上的Undo Log Header；
    - **TRX_RSEG_UNDO_SLOTS**
      - 当前回滚段，包含的Undo Slots数组的起始位置；
      - **TRX_RSEG_N_SLOTS** (UNIV_PAGE_SIZE / 16)：1024个Undo Slots；
      - 每个段头页，维护1024个Undo Slots；
  - **Undo Slot**
    - 每个Undo Slot，存储一个Page No，指向Undo Log Header Page (**存储Undo日志的Undo Data Page**)；
    - 每个更新事务，至少占用一个Undo Slot；最多占用两个Undo Slots; (事务部分详细介绍)
    - Undo Slot使用：Page_no存在；Undo Slot未使用：Page_no = NULL；

Rollback Segment Header Page

- 功能
  - 统一管理一个回滚段；
  - 事务 分配Undo Page的管理接口；
    - 寻找空闲Undo Slot
  - 标识每个Undo Header Page的位置；
    - Undo Slot指向的Page
  - 事务回收Undo Page的管理接口
    - 将Undo Page链接到TRX_RSEG_HISTORY链表中

Undo Log Header Page

- **Undo Log Header Page**
  - 实际存储Undo记录的页面类型之一(另一个为Normal Undo Page)
  - 每一个使用的Undo Slot，都指向一个Undo Log Header Page
  - 更新事务，至少占用一个Undo Log Header Page (最多两个)
  - Undo Log Header Page，在同一时刻，只能被一个事务使用
  - 关于事务方面，后面详细分析

Undo Log Header Page

- **存储内容**
  - 头结构
    - **TRX_UNDO_PAGE_HDR**
      - 事务操作类型(Update/Insert)；Page的Free空间等；
    - **TRX_UNDO_SEG_HDR**
      - 事务状态；undo log header位置；undo page链表；
    - **Undo Log Header**
      - 事务ID；XID；下一个Undo Log Header位置等；
  - Undo记录
    - 存储实际的Undo数据；
    - 每条Undo Rec，包含**prev/next**两个offset，分别指向前后Undo记录；
- **功能**
  - 保存事务同一类型的Undo (为什么区分类型？后续揭晓)
  - 将事务同一类型的Undo Page，链接管理
  - 保存事务的信息；例如：**状态**

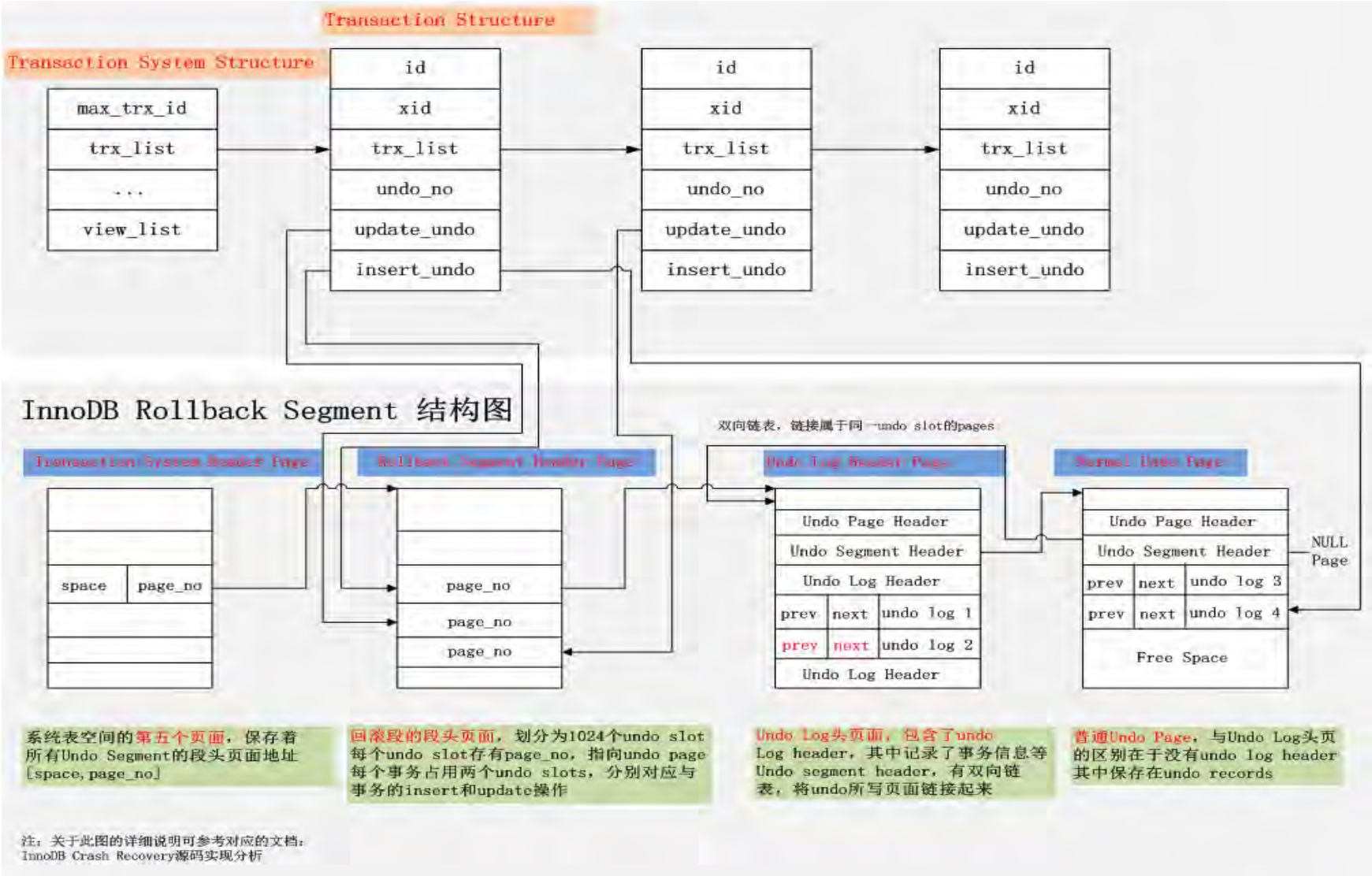
Normal Undo Page

- Normal Undo Page
  - Rollback Segment中，最后一种Page类型；
  - 实际存储Undo记录的页面类型之一 (另一个是Undo Log Header Page)
  - Normal Undo Page，通过Undo Log Header Page上**TRX_UNDO_SEG_HDR**结构中的双向链表，链接起来
  - 若事务Undo较小，则可能不会产生Normal Undo Page (只有Undo Log Header Page)

Undo Log Header vs Normal Undo

- Normal Undo Page与Undo Log Header Page的区别
  - 区别一
    - Undo Slot指向的是Undo Log Header Page，而非Normal Undo Page
  - 区别二
    - Normal Undo Page不包含TRX_UNDO_SEG_HDR(段头)与undo log header(日志头)
  - 区别三
    - 每个更新事务，至少会使用一个Undo Log Header Page，但是不一定会产生Normal Undo Page
  - 区别四
    - Undo Log Header Page可以被多个事务使用(串行使用)，但是Normal Undo Page只属于一个事务
  - 区别五
    - 事务如何使用这两类Undo Page，接下来详细分析

Undo plus Transaction



Transaction Structure

- Transaction与Undo相关的重要数据结构
  - **undo_no**
    - 标识事务写的Undo记录数量，递增；
  - **last_sql_stat_start**
    - 事务上一条成功执行的语句写的最后一条Undo记录的undo_no
    - 语句级rollback所需
  - **rseg**
    - 事务所使用的rollback segment
  - **insert_undo**
    - 指向Rollback Segment Header Page中的一个Undo Slot；
    - 事务(内部)Insert操作所写Undo，分配的Undo Page，链入此Undo Slot
  - **update_undo**
    - 指向Rollback Segment Header Page中的一个Undo Slot；
    - 事务(内部)Update/Delete(Delete Mark)操作所写Undo，分配的Undo Page，链入此Undo Slot

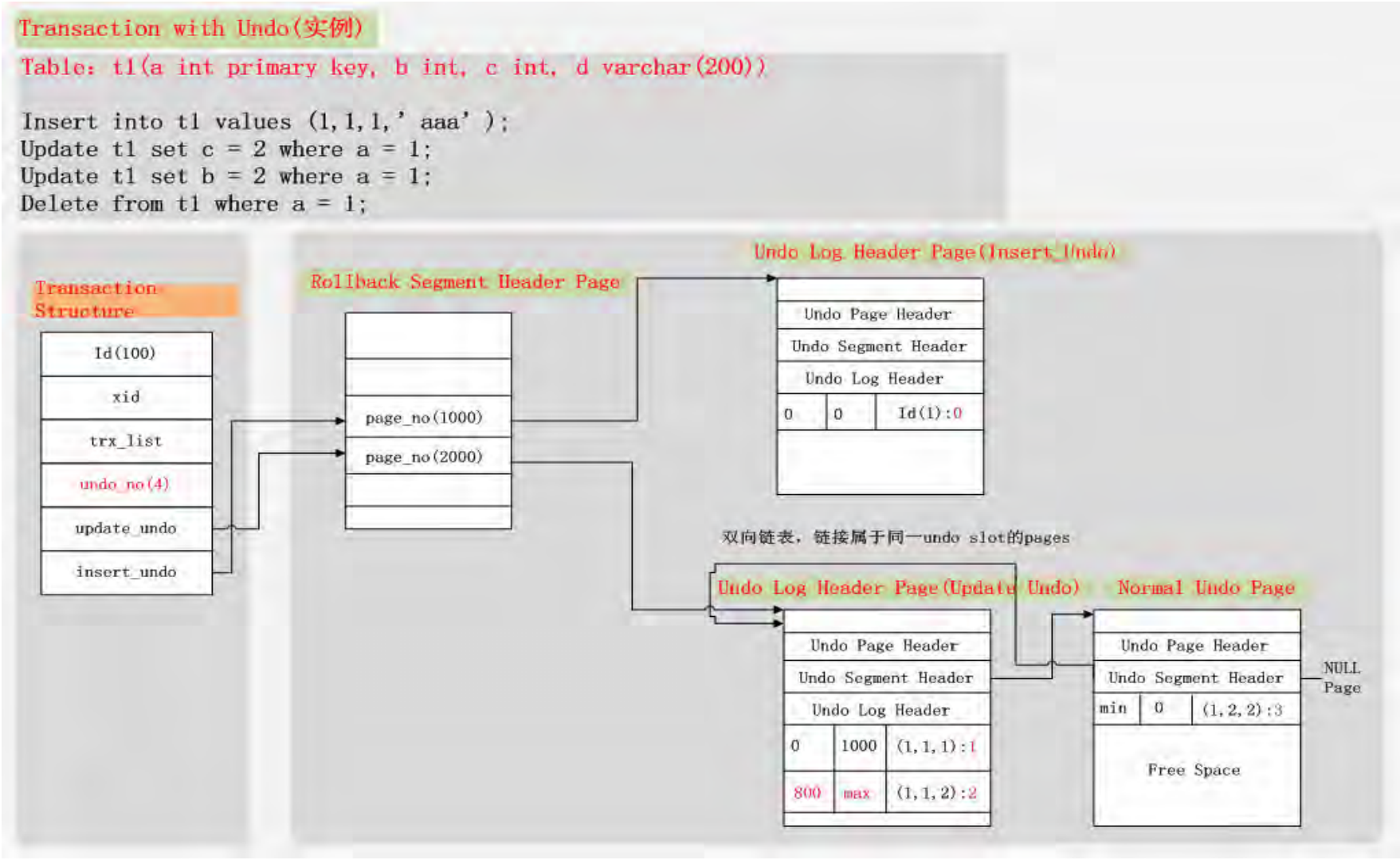
Transaction with Undo Questions

- **Q1：如何为事务指定Rollback Segment？**
  - **Round Robin**策略，指定事务的Rollback Segment

- **Q2：如何在Rollback Segment中找空闲的Undo Slot？**
  - 1. 在Rollback Segment Header Page的1024个Undo Slots中寻找空闲Undo Slot
  - 2. 为Undo Slot分配一个空闲的Undo Log Header Page (可优化，如何优化？)
  - 3. 所有这些操作，需要记录Redo日志

- **Q3：为何需要按照操作类型，分为insert_undo与update_undo？**
  - 目标：**为了实现Undo Page的分类回收**；
  - insert_undo
    - 所有的Undo Page，在事务提交后，可直接回收释放；
  - update_undo
    - 所有的Undo Page，事务提交后，不可立即回收释放；
    - 需要遍历其中的Undo日志，删除索引中的过期版本记录(**Purge**)；然后才可释放；

Transaction with Undo(实例)



Transaction with Undo(实例)

- 实例解析
  - **undo_no(4)**
    - 一共写了4条undo记录；
  - **insert_undo/update_undo**
    - 消耗了两个undo_slot
    - undo log header page分别为1000，2000
    - insert_undo写了1条；update_undo写了3条；
    - 用户delete语句属于内部update(delete mark)
  - **update_undo**
    - update_undo使用了两个undo_page
    - update_undo中的3条undo记录，通过prev/next偏移，链接起来(双向)
  - (1,1,1):**1**
    - 当前Undo，属于事务的第二条Undo记录(undo_no)；

Transaction Commit

- 事务提交，与Undo相关的操作
  - **Insert_Undo**
    - 释放Rollback Segment Header Page中的Undo Slot；
    - 直接释放Insert_Undo对应的所有Undo Page，回收空间；
  - **Update_Undo**
    - 释放Rollback Segment Header Page中的Undo Slot；
    - 将事务在Undo Log Header Page上当前事务的Undo Log Header链接到Rollback Segment Header Page上 (等待Purge，如何Purge？)
  - 优化
    - 若insert_undo/update_undo只使用了一个Undo Log Header Page，则将此Page Cache起来，留作下次事务的分配；
    - 好处：无需重新分配/初始化Undo Log Header Page，降低开销；

Transaction Rollback

- **Thoughts**
  - 反向使用事务的所有Undo日志，回滚整个事务所做的修改；
- InnoDB Implementation
  - The Same as our Thoughts
  - 反向
    - Undo记录，都维护着前一条Undo记录的偏移(prev)，可以反向遍历所有的Undo记录；
  - 使用
    - Undo是逻辑操作，根据Undo中的信息，回滚聚簇索引/二级索引记录 (Search & Undo)；
    - 由于需要回滚聚簇索引/二级索引，因此Undo中必须将未修改的二级索引列记录；
  - Tips
    - 反向使用Undo日志，因此必须将insert_undo/update_undo排序后选择；
    - 每条Undo记录，都记录了Undo的Number；

Record Rollback

- **InnoDB MVCC**
  - InnoDB是行级多版本，快照读需要将记录回滚到可见版本；
  - InnoDB的聚簇索引记录，新增了两个系统字段[DB_TRX_ID, ROLLBACK_PTR]；
  - ROLLBACK_PTR指向记录的Undo，根据Undo回滚记录到可见版本；
- **ROLLBACK_PTR**
  - 7 Bytes
  - **最低位2 bytes**
    - Undo记录在Undo Page中的偏移；
  - **中间4 bytes**
    - Undo记录所属Undo Page的Page_No；
  - **最高位1 bytes**
    - **低 7 bits**： Rollback Segment Id (128个)
    - **最高1 bit**： 标识Undo类型；Insert or Update操作；

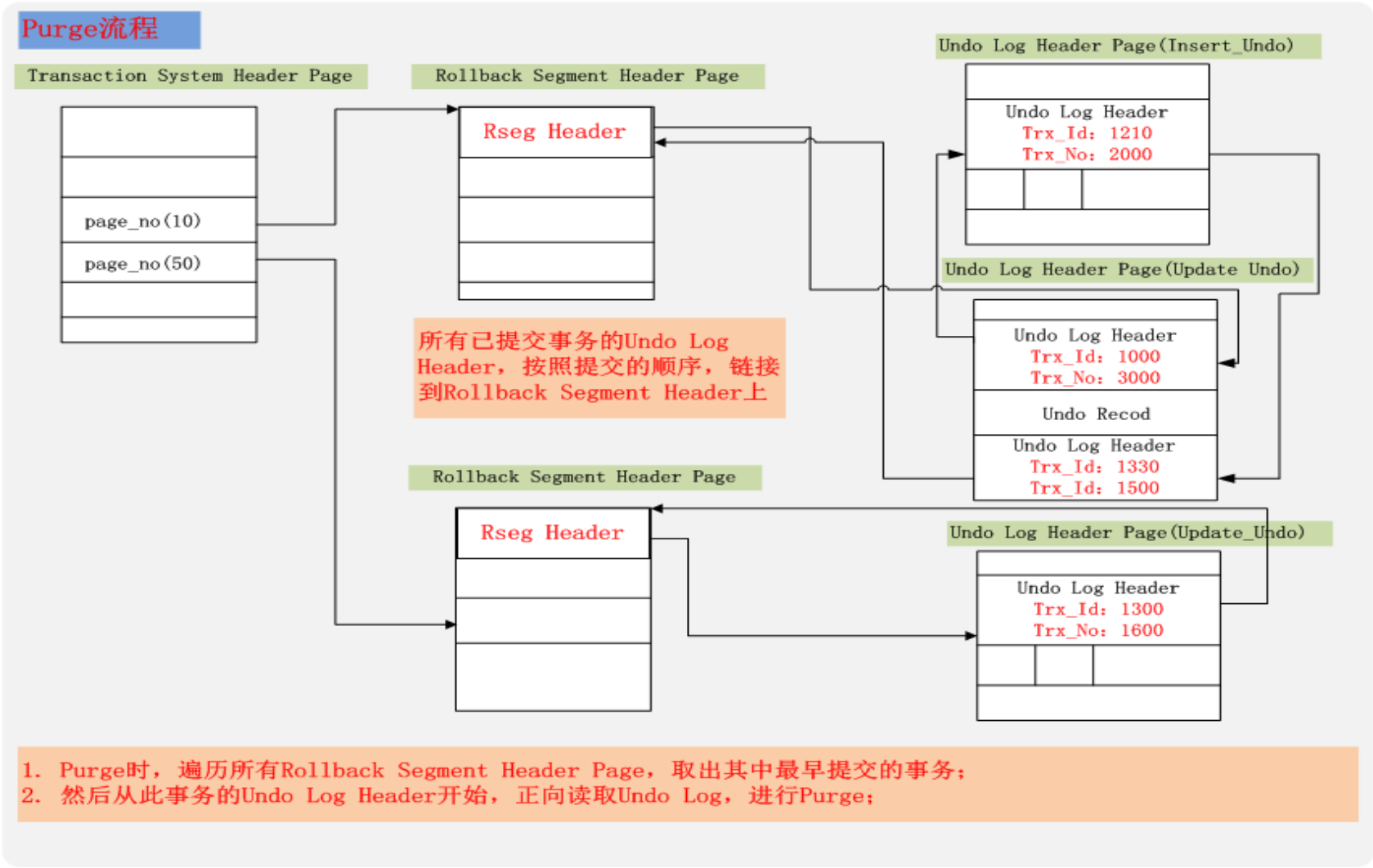
Purge

- Purge功能
  - 根据**Undo日志**，回收聚簇索引/二级索引上的**被标记为删除(DEL_BIT = 1)**，并且不会被当前活跃事务及新事务看到的**过期版本记录**；
  - **Undo日志**
    - Insert操作，不会产生DEL_BIT = 1的删除项，因此Purge不需要使用Insert_Undo;
    - 事务提交时，Insert_Undo可直接回收；
  - **DEL_BIT = 1**
    - 用户Delete操作/非In Place Update，均会将原有记录DEL_BIT标识为1；
    - 所有这些操作产生的Undo记录，均存储于Upate_Undo；
  - **过期版本记录**
    - 已标识为DEL_BIT = 1的记录，可能对活跃事务仍旧可见，因此不能立即删除；
    - 事务提交时，Update_Undo不可直接回收；
    - 后台Purge线程，根据系统中事务的提交顺序，逐个Purge提交事务，删除过期版本记录，回收Update_Undo；

Purge(续)

- Purge流程
  - 选择系统中最老的提交事务(所有Rollback Segment中最老提交事务)
  - 正向遍历事务的Update_Undo记录，删除聚簇/二级索引上对应的DEL_BIT=1的项
    - 正向遍历Undo，因此每条Undo也维护了**next** offset，用于定位下一条Undo记录
    - Purge时的删除，是彻底从数据页面中删除
- Purge相关系统参数
  - **innodb_max_purge_lag**
    - 当系统中积累的已提交，但未Purge的事务超过此限制时，前台DML操作等待；(**默认不开启**)
    - **慎用此参数**
  - **innodb_max_purge_lag_delay**
    - 设置等待的最长时间；(Since 5.6.5；innodb_max_purge_lag参数开启时有效)
  - **innodb_purge_threads**
    - 设置purge线程的数量；(Since 5.5.4；可设置Purge线程的数量)
  - **innodb_purge_batch_size**
    - 设置每次Purge，回收的事务数量；(Since 5.5.4)

Purge(Purge流程)



Undo Record遍历

- 正向 vs 反向
  - 事务Rollback
    - 从最新的Undo出发，反向遍历Undo，回滚事务的所有操作；
  - 事务Purge
    - 从第一条Undo出发，正向遍历Undo，根据Undo日志Purge过期删除版本；
- **定位事务最后一条Undo**
  - 根据Undo Log Header Page上的undo pages链表(TRX_UNDO_SEG_HDR)，定位最后一个Undo Page
  - 最后一个Undo Page上的TRX_UNDO_PAGE_HDR结构中的TRX_UNDO_PAGE_FREE，确定了最后一条Undo的offset
- **定位事务第一条Undo**
  - **Undo Log Header Page**
    - 事务的第一条Undo，一定在Undo Log Header Page上
    - 位于Undo Log Header结构中的TRX_UNDO_LOG_START处
  - **Normal Undo Page**
    - Normal Undo Page的第一条Undo，位于相对页面起始TRX_UNDO_PAGE_HDR + TRX_UNDO_PAGE_HDR_SIZE处

Questions about Crash Recovery

- 关于InnoDB Crash Recovery最主要的问题
  - Q1：Crash Recovery的起点，Checkpoint LSN存储于何处？
  - Q2：InnoDB如何完成Redo日志的重做？
  - Q3：InnoDB如何定位哪些事务需要Rollback？
  - Q4：Crash Recovery需要等待Rollback完成吗？
  - Q5：InnoDB各版本，在Crash Recovery流程上做了哪些优化？
  - …

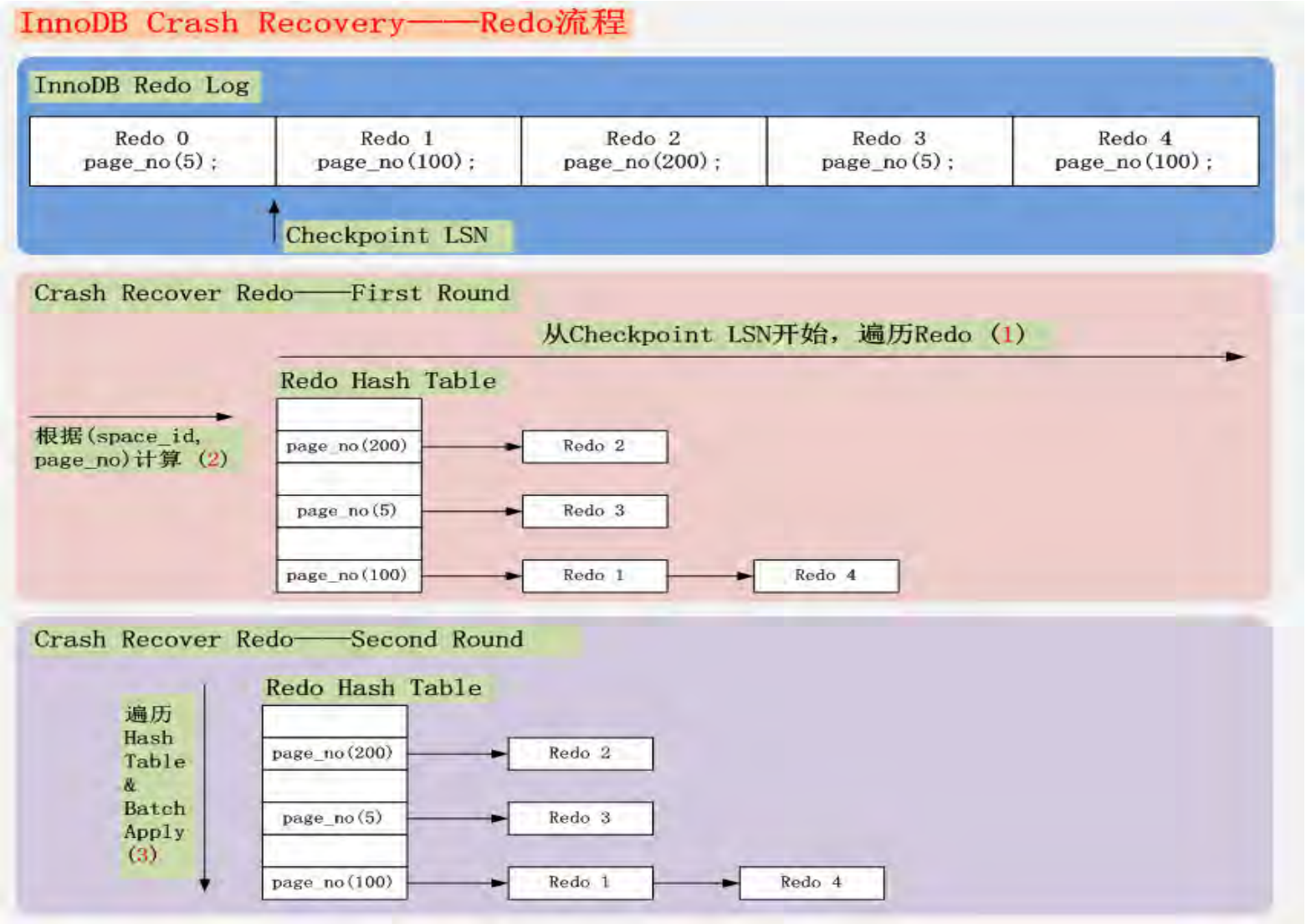
Crash Recovery流程

- Crash Recovery主要流程
  - 读取Checkpoint LSN
  - 从Checkpoint LSN开始向前遍历Redo Log File
    - 重做从Checkpoint LSN开始的所有Redo日志
  - 重新构造系统崩溃时的事务
    - Commit事务
      - 等待Purge线程回收
    - Prepare事务
      - 由MySQL Server控制提交/回滚
    - Active事务
      - 回滚活跃事务
  - 新建各种后台线程，Crash Recovery完成返回

读取Checkpoint LSN

- Checkpoint
  - 关于InnoDB的Checkpoint实现，可参考数据库内核分享——第一期
- Checkpoint LSN读取
  - Checkpoint LSN，存储于每个日志组，第一个日志文件的LOG FILE Header内，两处冗余存储
  - 位置一
    - **LOG_CHECKPOINT_1** (= OS_FILE_LOG_BLOCK_SIZE = 512)
  - 位置二
    - **LOG_CHECKPOINT_2** (= 3 * OS_FILE_LOG_BLOCK_SIZE = 3 * 512)
  - **LOG_FILE_HDR**
    - 大小：4 * OS_FILE_LOG_BLOCK_SIZE
    - log0log.h

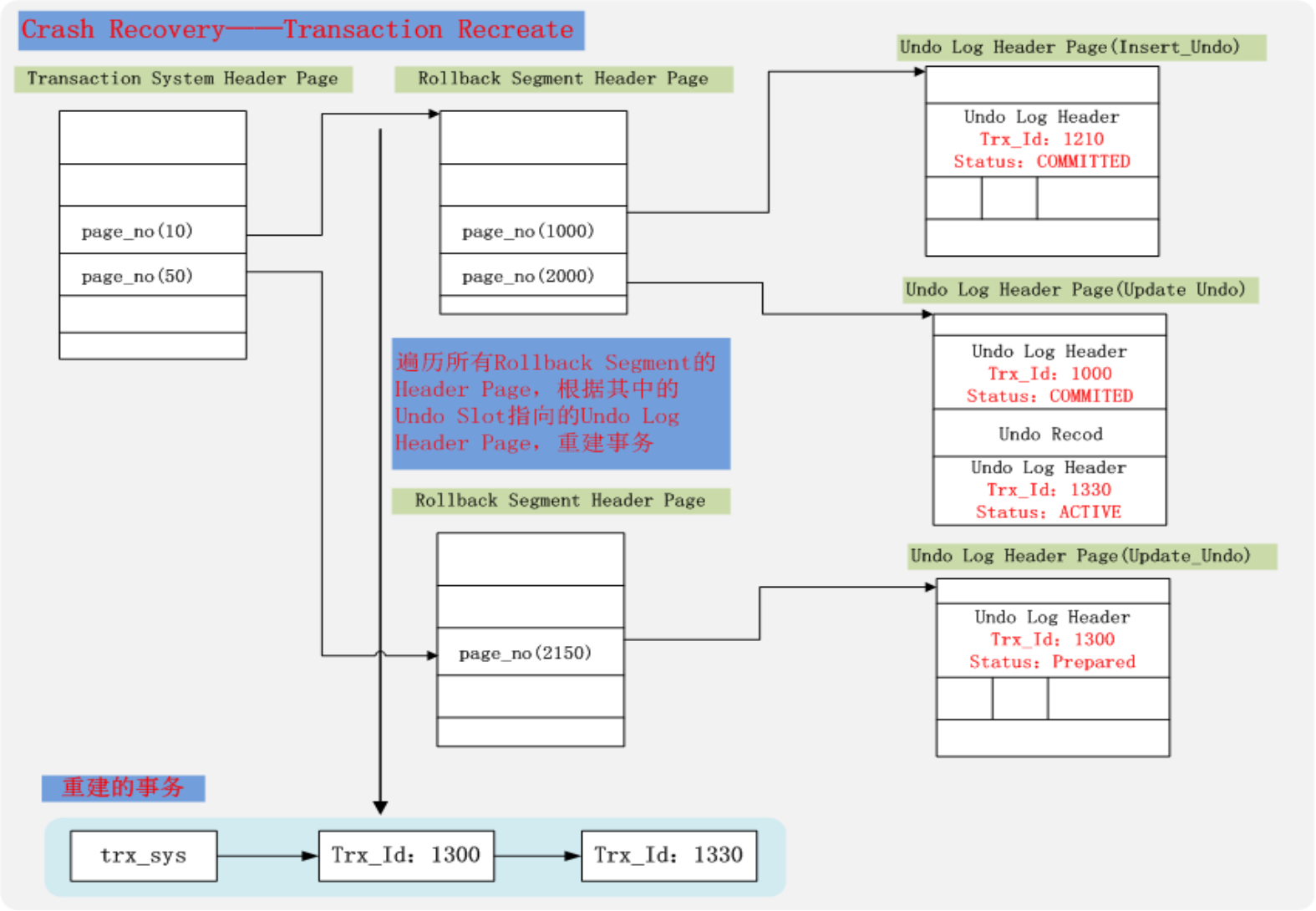
Redo流程


InnoDB Crash Recovery——Redo流程
InnoDB Redo Log
Redo 0
page_no(5);
Redo 1
page_no(100);
Redo 2
page_no(200);
Redo 3
page_no(5);
Redo 4
page_no(100);
Checkpoint LSN
Crash Recover Redo——First Round
从Checkpoint LSN开始，遍历Redo (1)
Redo Hash Table
根据(space_id, page_no)计算 (2)
page_no(200)
Redo 2
page_no(5)
Redo 3
page_no(100)
Redo 1
Redo 4
Crash Recover Redo——Second Round
Redo Hash Table
遍历Hash Table & Batch Apply (3)
page_no(200)
Redo 2
page_no(5)
Redo 3
page_no(100)
Redo 1
Redo 4

Redo流程(续)

- 关键词
  - **Batch Apply**
    - 收集Redo，存入内存Hash Table；
    - 同一个页面的Redo，按照先后顺序链接
    - 当Hash Table大小达到上限时
      - 遍历Hash Table，每个Page，Batch Apply所有的Redo
  - **Rollback Segment Redo**
    - 回滚段Page的修改，同样需要Redo
    - 回滚段页面Redo到最新，可用于下一阶段的Undo流程

Undo流程


Crash Recovery——Transaction Recreate
Transaction System Header Page
Rollback Segment Header Page
page_no(10)
page_no(50)
page_no(1000)
page_no(2000)
Undo Log Header Page(Insert_Undo)
Undo Log Header
Trx_Id: 1210
Status: COMMITTED
Undo Log Header Page(Update Undo)
Undo Log Header
Trx_Id: 1000
Status: COMMITED
Undo Recod
Undo Log Header
Trx_Id: 1330
Status: ACTIVE
遍历所有Rollback Segment的Header Page，根据其中的Undo Slot指向的Undo Log Header Page，重建事务
Rollback Segment Header Page
page_no(2150)
Undo Log Header Page(Update_Undo)
Undo Log Header
Trx_Id: 1300
Status: Prepared
重建的事务
trx_sys
Trx_Id: 1300
Trx_Id: 1330

Undo流程(续)

- 关键流程
  - **Recreate Transactions**
    - 遍历系统所有的Rollback Segment Header Page，读取其中的Undo Slot指向的Undo Log Header Page；
    - 读取Undo Log Header Page中的Undo Log Header
      - 读取事务ID
      - 读取事务状态，重建ACTIVE/Prepared状态的事务 (Committed状态的事务，无需重建)
  - **Transaction Rollback**
    - ACTIVE状态的事务，Rollback；
    - Prepared状态的事务，由MySQL Server决定最终Commit/Rollback (**根据Binlog**)
  - **Rollback处理(异步)**
    - InnoDB新建后台线程处理Rollback
    - Rollback操作本身，不属于Crash Recovery的一部分
    - 大事务的Rollback，会持续到Crash Recovery结束，MySQL提供服务之后

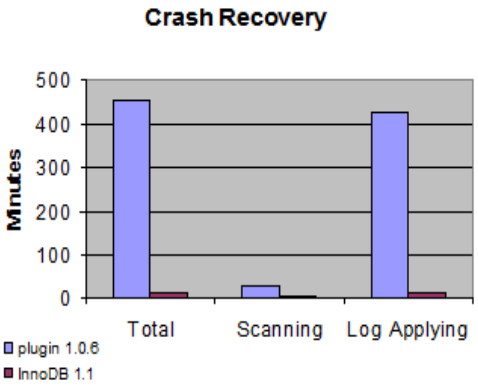
Crash Recovery优化

- 两个主要优化，均集中于 Crash Recovery的Redo过程
  - 优化一
    - Bug #49535
    - 控制Hash Table大小
  - 优化二
    - Bug #29847
    - 维护Flush List链表
    - 引入红黑树
  - 优化效果
    - 右图所示

Improved Recovery Performance - BenchMark

- 60m sysbench OLTP Read/Write test
  - innodb-buffer-pool-size=18g
  - innodb-log-file-size=2047m
  - Kill the server after 20 minutes
  - Modified DB pages 1007907
  - Redo bytes: 3050455773

	Total (min)	Scanning	Log Applying
Plugin 1.0.6	456	32	426
InnoDB 1,1	14	2	12
Improvement	32	16	35.5



ORACLE

InnoDB Crash Recovery Summary

- **如何读取Checkpoint LSN**？
  - 日志组第一个日志文件的LOG_FILE_HDR内
- **如何Redo**？
  - Batch Apply
- **如何Undo**？
  - 构造崩溃时的所有事务
  - 后台线程，回滚活跃事务