MySQL——redo日志

〇、大纲

- 关于MySQL发生了故障，导致内存中的数据失效了怎么办?

> 【方案1】在事务提交时，把该事务修改的 `所有页面` 都刷新到磁盘。缺点是什么?
> 【方案2】在事务提交时，只需要把 `修改的内容` 记录一下就好了。

- `redo日志` 的优点有哪些?
- `简单的redo日志` ——物理日志
- `复杂的redo日志`
- 什么是 `MTR` ?
- `事务` 、 `SQL语句` 、 `MTR` 、 `redo日志` 的关系是什么?
- 什么是 `redo log block` ? 大小是多少? 结构是怎样的?
- 什么是 `redo log buffer` ?
- redo日志如何写入到 `log buffer` 中? 默认大小是多少?
- redo日志的 `刷盘时机` 有哪些?
- 什么是 `redo日志文件组` ? `ib_logfile0` 和 `ib_logfile1` ?
- redo日志文件（磁盘上的文件）存储 `内容如何划分` 的?
- 什么是 `lsn` ? `buf_free` ? `buf_next_to_write` ? `checkpoint` ?

一、什么是redo日志

1.1> 关于MySQL故障产生的问题

- 如果我们只在内存的 `Buffer Pool` 中修改了页面，假设在**事务提交后**突然发生了某个**故障**，导致内存中的**数据都失效了**，那么这个已经提交的事务在数据库中所做的更改也就丢失了。针对这种问题，怎么处理呢?
- **方案一**：**在事务提交时，把该事务修改的所有页面都刷新到磁盘。**

> 1> 刷新一个完整的数据页太浪费了

> 虽然我们只修改了一条记录，但是会将这条记录所在的页（`16KB`）都刷新到磁盘上，会造成大量磁盘I/O的浪费。
>
> 2> **随机I/O刷新起来比较慢**
>
> **一个事务**可能包含**很多语句**，即使是**一条语句**也可能修改**许多页面**，并且该事务修改的这些页面可能**并不相邻**。这就意味着将某个事务修改的Buffer Pool中的页面刷新到磁盘时，需要进行很多的**随机I/O**。而随机I/O要比顺序I/O慢，尤其是机械硬盘。

- **方案二：在事务提交时，只需要把修改的内容记录一下就好了。**

> 例如：“将第0号表空间第100号页面中偏移量为1000处的值更新为2。”

1.2> redo日志的定义

- 因为在**系统因崩溃**而重启时需要按照上述内容所记录的步骤重新更新数据页，所以上述内容也成为**重做日志**（redo log）。
- redo日志的优点：

> 1> redo日志占用的**空间非常小**；
> 2> redo日志是**顺序写入磁盘**的；

- 在执行事务的过程中，每执行一条语句，就可能产生若干条redo日志，这些日志是按照产生的**顺序写入磁盘**的，也就是使用**顺序I/O**。

二、redo日志格式

- redo日志本质上只是记录了一下事务对数据库进行了哪些**修改**，因此针对不同修改场景，定义了**多种类型**的redo日志，但是绝大部分类型的redo日志都有如下的**通用格式。**

> 在MySQL5.7.22版本中，共有**53种**不同的类型。其中，【表空间ID + 页号】可以定位到redo日志相关的页。

2.1> 简单的redo日志类型——物理日志

2.1.1> 什么是物理日志?

- 在对页面的**修改是极其简单**的情况下（下面会有例子），redo日志中只需要记录一下在**某个**

页面的**某个偏移量**处**修改了几个字节的值**、具体**修改后的内容是啥**就好了。

2.1.2> 场景举例

- 如果某张表**没有主键**，并且没有定义不允许存储**NULL值**的**UNIQUE键**，那么InnoDB会自动为表添加一个名为**row_id**的隐藏列作为主键。
- 为这个row_id隐藏列进行赋值的方式如下：

> - 内存中维护一个**全局变量**，当向某个包含row_id隐藏列的表中插入一条记录时，就会把这个全局变量的值当做新记录的row_id的值，并且把这个全局变量**+1**；
> - 每当这个全局变量的值为**256的倍数**时，就会将该变量的值刷新到**系统表空间页号为7**的页面中一个名为**Max Row Id**的属性中。（这个写入操作，实际上是在**Buffer Pool**中完成的，我们需要把这次对这个页面的修改以redo日志的形式记录下来）
> - 当系统启动时，会将这个**Max Row Id**属性加载到内存中，并将该值**加上256**之后赋值给前面提到的全局变量（因为在系统上次关机时，如果内存中的全局变量没有到达256的倍数，而没有刷新到BufferPool，那么就会出现该全局变量的值可能大于磁盘页面中Max Row ID属性的值）

- 这种对页面修改是极其简单的。所以该redo日志即为**物理日志**。

2.1.3> 物理日志的几种不同类型

- MLOG_1BYTE（type=1）

> 表示在**页面的某个偏移量**处写入**1字节**的redo日志类型。

- MLOG_2BYTE（type=2）

> 表示在**页面的某个偏移量**处写入**2字节**的redo日志类型。

- MLOG_4BYTE（type=4）

> 表示在**页面的某个偏移量**处写入**4字节**的redo日志类型。

- MLOG_8BYTE（type=8）

> 表示在**页面的某个偏移量**处写入**8字节**的redo日志类型。由于**Max Row ID**占用8字节的空间，所以在修改页面中的这个属性时，会记录一条类型为MLOG_8BYTE的redo日志
>
> 
>

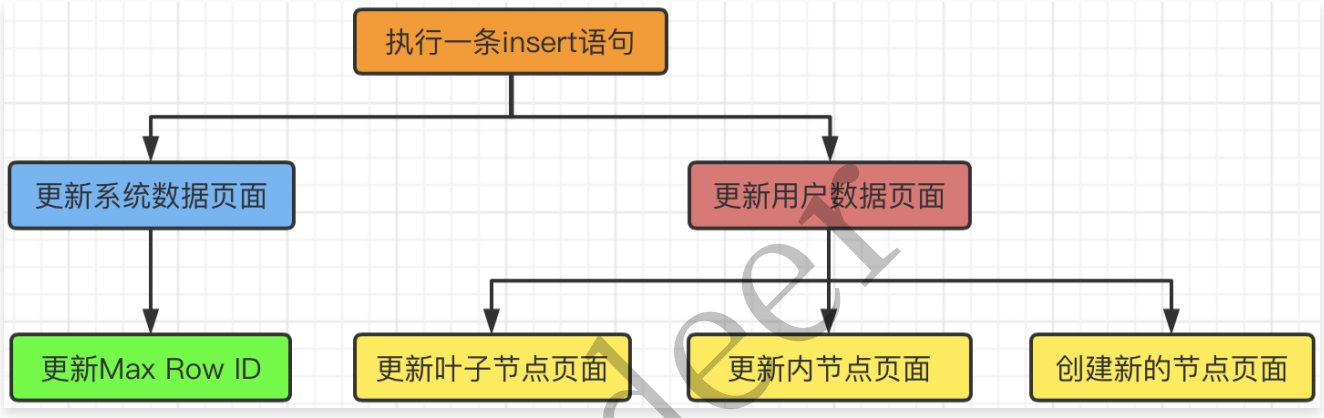
- MLOG_WRITE_STRING（type=30）

表示在**页面的某个偏移量**处写入**一个字节序列**。

【注】只要在【数据占用的字节数】处填入 `1` 、 `2` 、 `4` 、 `8` 这些数字，就可以分别替代MLOG_1BYTE、MLOG_2BYTE、MLOG_4BYTE、MLOG_8BYTE这些类型的redo日志。但是，**为了节省空间**，所以，才特殊定义了这4个类型。

2.2> 复杂一些的redo日志类型

- 执行一条INSERT语句涉及的更新内容



【注】**用户数据**指的就是聚簇索引和二级索引对应的B+树。

- 可能更新 `Page Directory` 中的**槽信息**。
- 可能更新 `Page Header` 中的各种**页面统计信息**。
  - PAGE_N_DIR_SLOTS：表示的槽数量可能会更改。
  - PAGE_HEAP_TOP：代表的还未使用的空间最小地址可能会更改。
  - PAGE_N_HEAP：代表的本页面中的记录数量可能会更改。
- 可能更新记录的**单向链表**
  - 数据页中的记录按照序列从小到大的顺序组成的一个单向链表，每插入一条记录，还需要更新上一条记录的记录头信息中的 `next_record` 属性来维护这个单向链表。
- 还有其他需要**更新的内容**。

- 综上所述：我们想要说明的一点就是——在把一条记录插入到一个页面时，**需要更改的地方非常的多**。这时，如果使用前面介绍的简单的物理redo日志来记录这些修改，可能会有如下两种解决方案

- 方案一：**在每个修改的地方都记录一条redo日志**

这种方式的缺点是显而易见的，因为被修改的地方实在太多了，可能redo日志占用的空间都要比整个页面占用的空间多。

- 方案二：**将整个页面第一个被修改的字节到最后一个被修改的字节之间所有的数据当成一条物理redo日志中的具体内容**

这种方案所涉及的数据中，会掺杂很多本来没有被修改的数据，这样都加到redo日志中，太浪费空间了。

- 为了解决上面的问题，我们来介绍一下新的redo日志类型

- MLOG_REC_INSERT（type=9）

表示**插入**一条使用**非紧凑行格式**（REDUNDANT）的**记录**时，redo日志的类型。

- MLOG_COMP_REC_INSERT（type=38）

表示**插入**一条使用**紧凑行格式**（COMPACT、DYNAMIC、COMPRESSED）的**记录**时，redo日志的类型。

- MLOG_COMP_PAGE_CREATE（type=58）

表示**创建**一个存储**紧凑行格式**记录的**页面**时，redo日志的类型。

- MLOG_COMP_REC_DELETE（type=42）

表示**删除**一条使用**紧凑行格式**的**记录**时，redo日志的类型。

- MLOG_COMP_LIST_START_DELETE（type=44）

表示在从某条给定记录开始**删除**页面中一系列使用**紧凑行格式**的**记录**时，redo日志的类型。

- MLOG_COMP_LIST_END_DELETE（type=43）

与MLOG_COMP_LIST_START_DELETE类型的redo日志**相呼应**，表示**删除一系列记录**，直到MLOG_COMP_LIST_END_DELETE类型的redo日志对应的**记录为止**。

- MLOG_ZIP_PAGE_COMPRESS（type=51）

表示**压缩一个数据页**时，redo日志的类型。

- 还有很多很多种类型，这里就不列举了，等用到时再说。

- 上面这些类型的redo日志包含两个层面的意思：

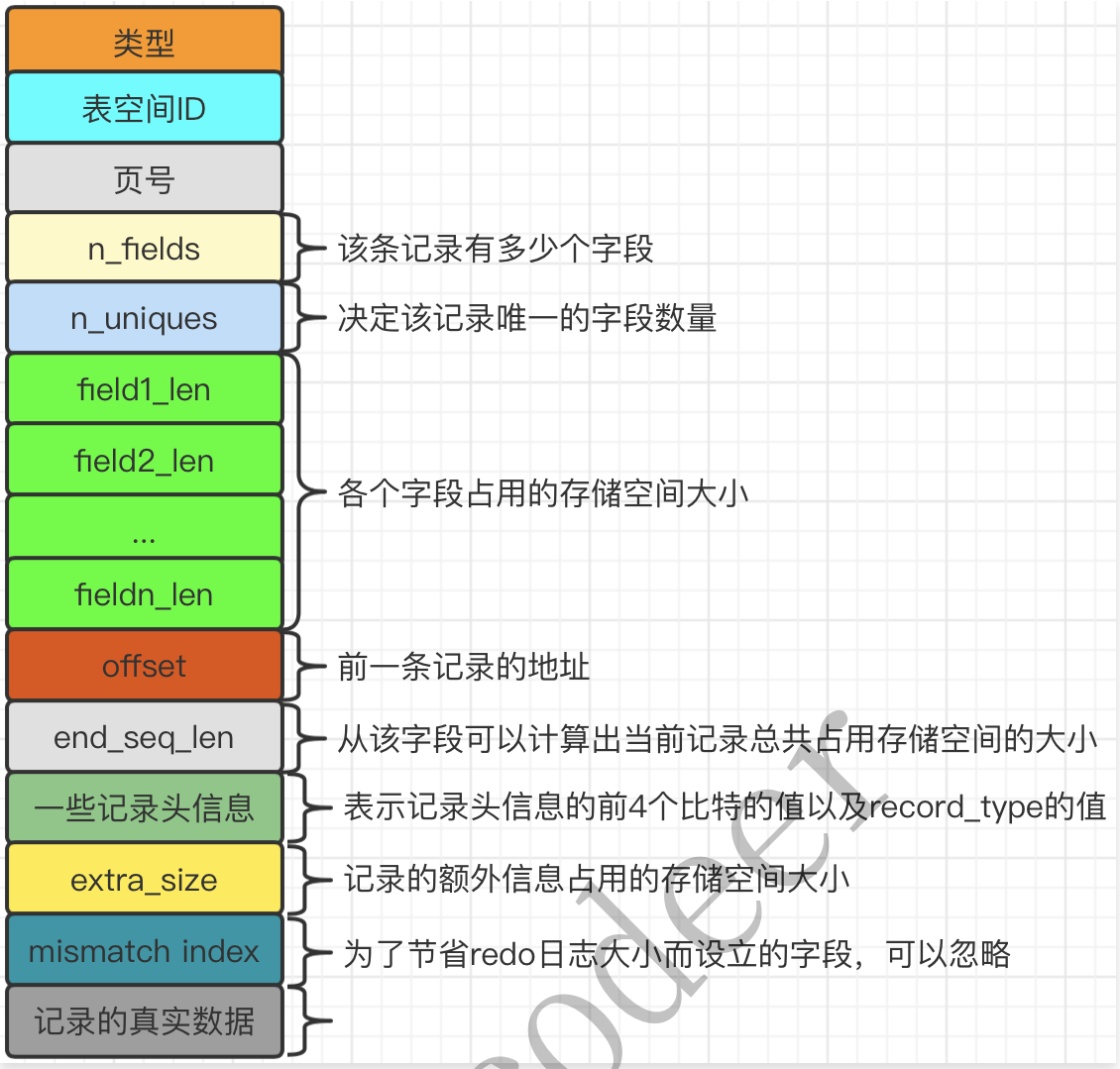
- 从**物理层面**来看

这些日志都指明了对哪个**表空间**的哪个**页**进行修改。

- 从**逻辑层面**来看

在系统崩溃后重启时，并不能直接根据这些日志中的记载，在页面内的某个偏移量处恢复某个数据，而是需要**调用一些事先准备好的函数**，在执行完这些函数后才可以将页面恢复成系统崩溃前的样子。

- 上面解释可能有些懵，我们还是以MLOG_COMP_REC_INSERT类型的redo日志为例，解释一下物理层面和逻辑层面到底是啥意思。



三、Mini-Transaction

3.1> 以组的形式写入redo日志

- 在执行语句的过程中产生的redo日志，被InnoDB划分成了若干个**不可分割的组**。比如：

> - 更新 `Max Row ID` 属性时产生的redo日志为一组，是不可分割的。
> - 向**聚簇索引**对应B+树的页面中插入一条记录时产生的redo日志为一组，是不可分割的。
> - 向**二级索引**对应B+树的页面中插入一条记录时产生的redo日志为一组，是不可分割的。
> - 还有其他的一些不可分割的组。

- 什么是**不可分割**呢?

> 我们以向某个索引对应的B+树中插入一条记录为例进行解释。
>
> - **乐观插入**

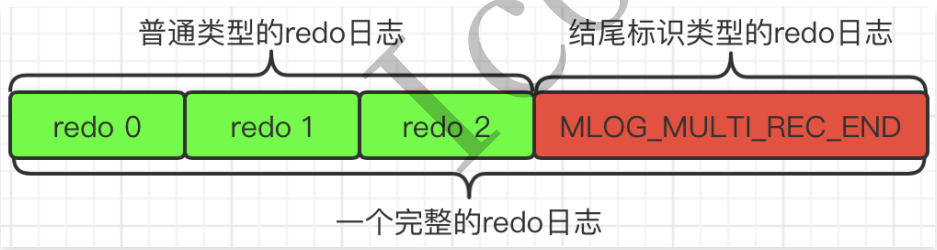
该数据页剩余的空闲空间相当**充足**，足够容纳这一条待插入记录。这样很简单，直接把记录插入到这个数据页中，然后记录一条MLOG_COMP_REC_INSERT类型的redo日志就好了。

- **悲观插入**

该数据页剩余的**空间不足**，那么就涉及到了**页分裂**操作——即：创建一个叶子节点，把原先数据页中的一部分记录复制到这个新的数据页中，然后再把记录插入进去；再把这个叶子节点插入到叶子节点链表中，最后还要在内节点中添加一条目录项记录来指向这个新创建的页面。很显然，这个过程需要对多个页面进行修改，着意味着会产生多条redo日志。还要涉及修改各种段、区的统计信息，修改各种链表的统计信息等（比如：FREE链表、FREE_FRAG链表等），反正**总共需要记录的redo日志有二三十条**。

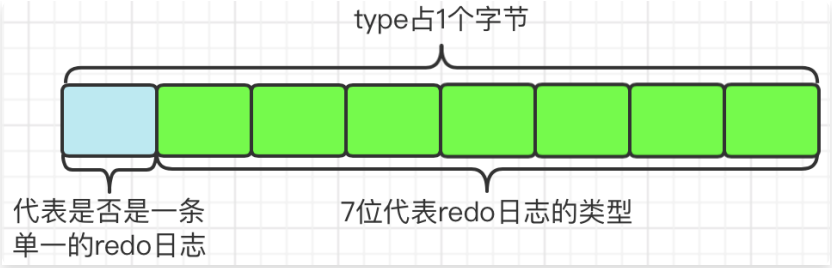
- InnoDB认为，向某个索引对应的B+树中插入一条记录的过程必须是原子的，不能说插入了一半之后就停止了。否则就会形成一棵不正确的B+树。所以他们规定在执行这些需要**保证原子性**的操作时，必须以**组**的形式来记录redo日志。
- 在进行恢复时，针对某个组中的redo日志，要么把全部的日志都恢复，要么一条也不恢复。
- 如何把这些redo日志划分到一个组里呢?

在该组中的最后一条redo日志后面加上一条特殊类型的redo日志。该类型的redo日志的名称为MLOG_MULTI_REC_END，结构很简单，只有一个type字段（type=31）。
所以，某个需要保证原子性的操作所产生的一系列redo日志，必须以一条类型为MLOG_MULTI_REC_END的redo日志结尾，如下所示：



- 当系统因崩溃而重启恢复时，只有解析到类型为 `MLOG_MULTI_REC_END` 的redo日志时，才认为解析到了一组完整的redo日志，才会进行恢复；否则直接放弃前面解析到的redo日志。
- 有些需要保证原子性的操作**只生成一条**redo日志，那是否也需要MLOG_MULTI_REC_END结尾呢?

答：不是的。为了勤俭节约，通过type字段即可表示。因为一个type字段其实占用1个字节（8位）。所以，当**最高位为1**的时候，代表这个需要保证原子的操作且只产生了一条**单一的redo日志**。否则，就表示这个需要保证原子性的操作产生了一系列的redo日志。而剩下的7个位，足够可以表达所有的redo类型日志（redo日志有几十种）。如下所示：

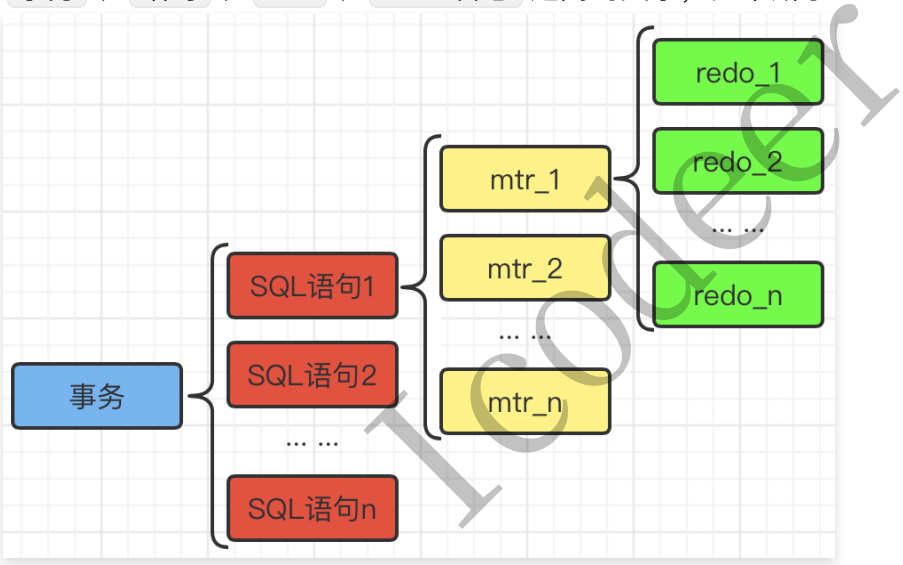


3.2> Mini-Transaction的概念

- 什么是MTR

> 对底层**页面**进行一次**原子访问**的过程被称为一个 `Mini-Transaction` （**MTR**）。
> 比如，前文说的修改Max Row ID的值，就算是一个MTR；
> 比如，向某个索引对应的B+树中插入一条记录的过程也算是一个MTR；

- `事务` 、 `语句` 、 `MTR` 、 `redo日志` 之间的关系，如下所示：



> 1个事务可以包含n条SQL语句；
> 1条SQL语句可以包含n个MTR；
> 1条MTR可以包含n条redo日志；

四、redo日志的写入过程

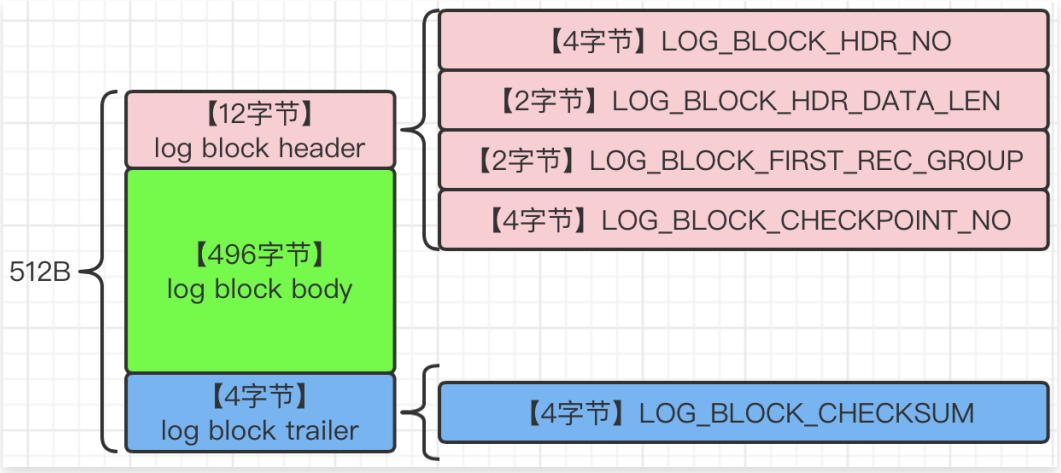
4.1> redo log block

- 什么是redo log block

> 为了更好地管理redo日志，设计InnoDB的大叔把**通过MTR生成的redo日志**都放在了大小为**512字节**的页中。

> 为了与前文提到的表空间中的页进行区别，我们这里把用来存储redo日志的页称为**block**。（其实“页”和“block”的意思差不多）

- 一个redo log block的结构示意图

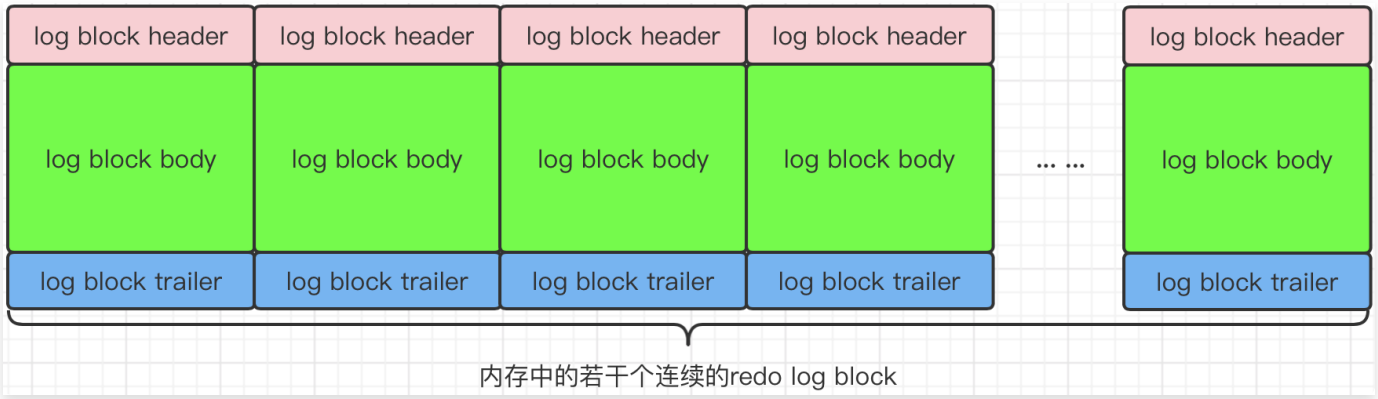


> - 【4字节】LOG_BLOCK_HDR_NO
>
> 每个block都有一个**大于0**的唯一编号，该属性就表示该**编号值**。
> - 【2字节】LOG_BLOCK_HDR_DATA_LEN
>
> 表示block中已经**使用了多少字节**；
>
> 初始值为**12**（因为log block body从第12个字节处开始）。
>
> 随着往block中写入的redo日志越来越多，该属性值页跟着增长。
>
> 如果log block body已经被全部写满，那么该属性的值被**设置为512**。
> - 【2字节】LOG_BLOCK_FIRST_REC_GROUP
>
> 代表该block中**第一个MTR**生成的redo日志记录组的偏移量，其实也就是这个block中第一个MTR生成的第一条redo日志记录的偏移量（如果一个MTR生成的redo日志横跨了好多个block，那么最后一个block中的LOG_BLOCK_FIRST_REC_GROUP属性就表示这个MTR对应的redo日志结束的地方，也就是下一个MTR生成的redo日志开始的地方）。
> - 【4字节】LOG_BLOCK_CHECKPOINT_NO
>
> 表示**checkpoint**的序号。
> - 【4字节】LOG_BLOCK_CHECKSUM
>
> 表示该**block的校验值**，用于正确性校验。

4.2> redo日志缓冲区

- 与 `Buffer Pool` 类似，写入redo日志时也**不能直接写到磁盘**中，实际上在服务器启动时就向操作系统申请了一大片称为**redo log buffer**（redo日志缓冲区）的**连续内存空间**，也可

以将其简称为 `log buffer` 。这片内存空间被划分成若干个连续的 `redo log block` 。如下所示：



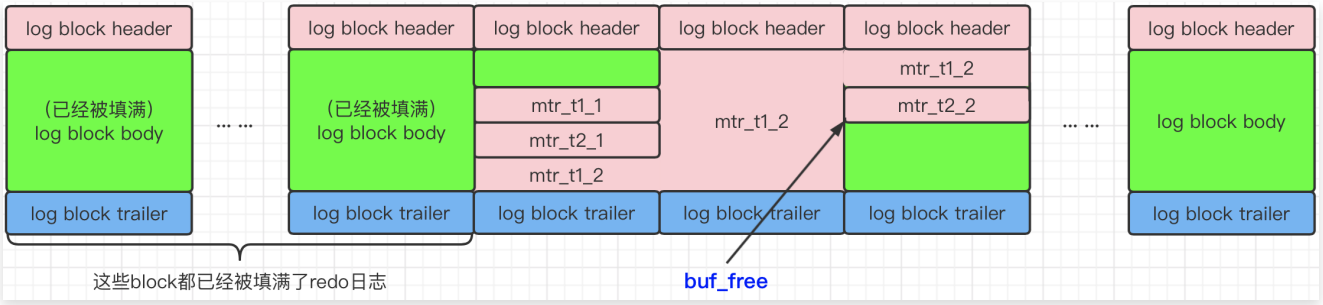
- `innodb_log_buffer_size` 用来指定log buffer的大小。在MySQL 5.7.22版本中，该启动选项的默认值为16MB。

4.3> redo日志写入log buffer

- 向log buffer中写入redo日志的过程是**顺序写入**的。
- **buf_free**

> 全局变量，该变量指明后续写入的redo日志应该写到log buffer中的哪个位置。

- 一个MTR执行过程中可能产生若干条redo日志，这些redo日志是一个不可分割的组，所以**并不是每生成一条redo日志就将其插入到log buffer中**，而是将每个MTR运行过程中产生的日志**先暂时存到一个地方**；当该MTR结束的时候，再将过程中产生的一组redo日志全部复制到log buffer中。
- log buffer结构示意图



> 事务T1的两个MTR：**mtr_t1_1** 和 **mtr_t1_2**
> 事务T2的两个MTR：**mtr_t2_1** 和 **mtr_t2_2**
> **不同的事务是可能并发执行的**，所以T1、T2的MTR可能是交替执行的。

五、redo日志文件

5.1> redo日志刷盘时机

- MTR运行过程中产生的一组redo日志在MTR结束时会被复制到log buffer中。可是这些日志总在内存里也不是办法，在一些情况下它们会被刷新到磁盘中。
- 哪些情况下会被刷新到磁盘中呢?

1> log buffer**空间不足50%**的时候

2> **事务提交的时候**

引入redo日志后，虽然在事务提交时可以不把修改过的Buffer Pool页面立即刷新到磁盘，但是为了保证持久性，必须要把页面修改时所对应的**redo日志刷新到磁盘**。否则假如系统崩溃后，无法将该事务对页面所做的修改恢复过来。

3> 后台有一个线程，大约以**每秒一次**的频率将log buffer中的redo日志刷新到磁盘。

4> **正常关闭服务器**时

5> 做**checkpoint**时

5.2> redo日志文件组

- MySQL的数据目录（使用show variables like 'datadir';可查看）下默认有名为**ib_logfile0**和**ib_logfile1**的两个文件，log buffer中的日志在默认情况下就是刷新到这两个磁盘文件中。可以通过下面几个启动选项来调节：

```
mysql> show variables like 'datadir';
+---------------+-----------------------+
| Variable_name | Value                 |
+---------------+-----------------------+
| datadir       | /usr/local/mysql/data/ |
+---------------+-----------------------+
1 row in set (0.01 sec)
```

```
muse@muse:/usr/local/mysql/data> ll /usr/local/mysql/data/ | grep ib_logfile
-rw-r-----    1 _mysql   _mysql   50331648  9 21 19:42 ib_logfile0
-rw-r-----    1 _mysql   _mysql   50331648  9 21 19:42 ib_logfile1
```

1> innodb_log_group_home_dir ： 指定了redo日志文件所在的目录，默认值为当前的数据目录。

```
mysql> show variables like 'innodb_log_group_home_dir';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| innodb_log_group_home_dir | ./    |
+---------------------------+-------+
1 row in set (0.00 sec)
```

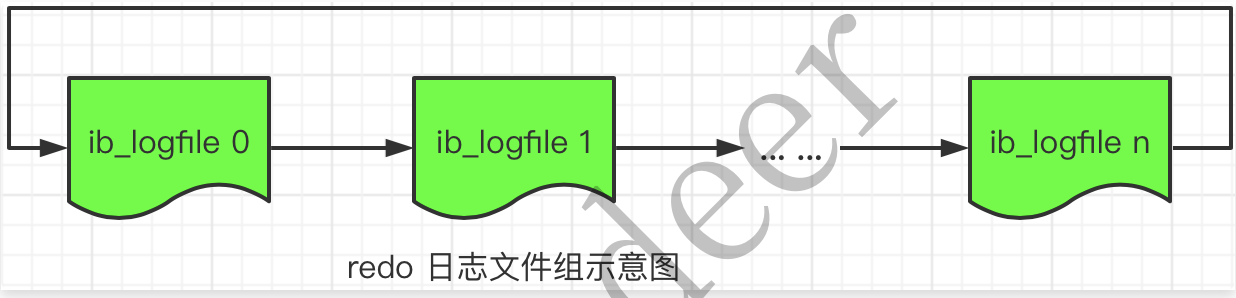
2> innodb_log_file_size ： 指定了每个redo日志文件的大小，在MySQL 5.7.22版本中的默认值为48MB。

```
mysql> show variables like 'innodb_log_file_size';
+----------------------+----------+
| Variable_name        | Value    |
+----------------------+----------+
| innodb_log_file_size | 50331648 |
+----------------------+----------+
1 row in set (0.00 sec)
```

3> innodb_log_files_in_group：指定了redo日志文件的个数，默认值为2，最大值为100。

```
mysql> show variables like 'innodb_log_files_in_group';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| innodb_log_files_in_group | 2     |
+---------------------------+-------+
1 row in set (0.01 sec)
```

- redo 日志文件组示意图

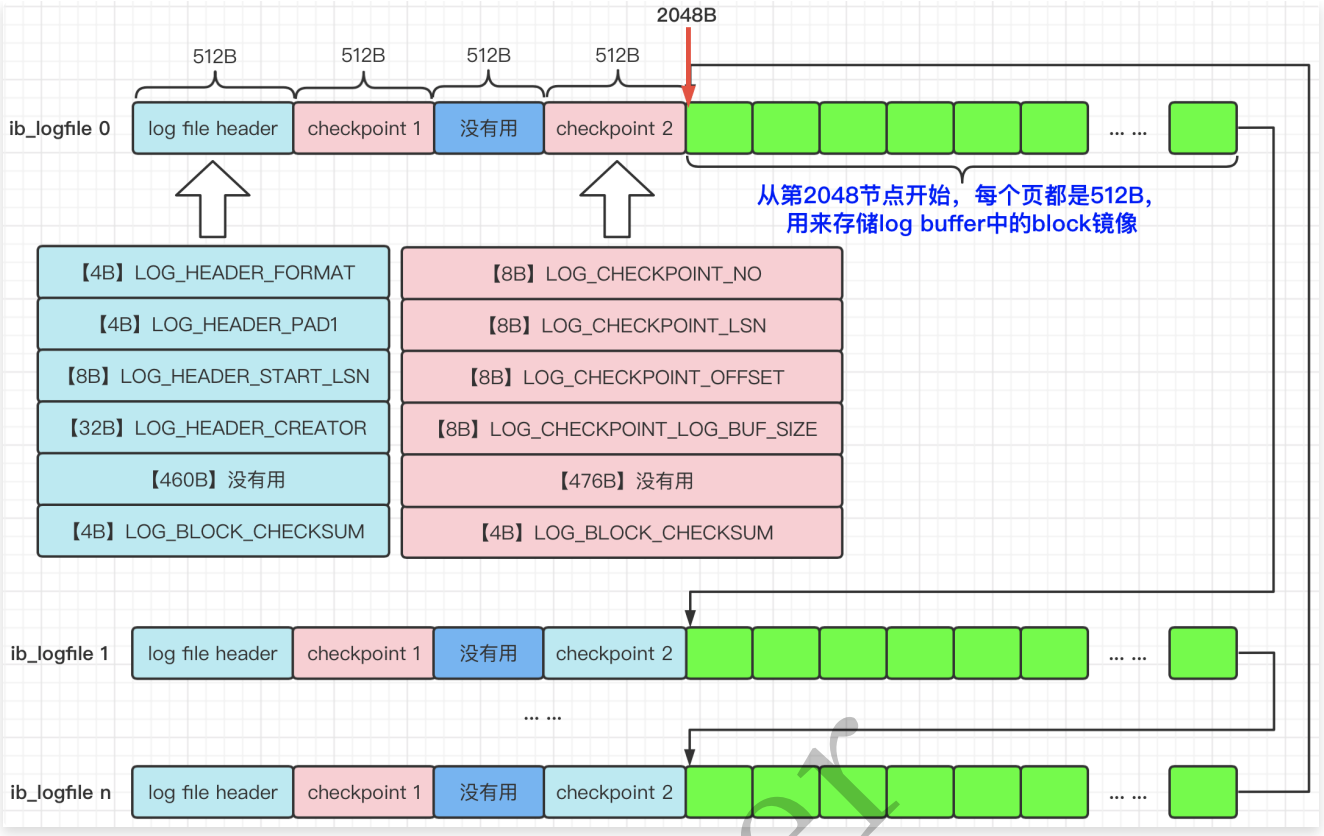


redo 日志文件组示意图

【注】按箭头的**顺序**进行日志写入。

5.3> redo日志文件（磁盘上的文件）格式

- 由于 `log buffer` 本质是一片**连续**的内存空间，被划分成若干个**512字节**大小的block；将log buffer中的redo日志刷新到磁盘的本质就是**把block的镜像写入日志文件**中，所以redo日志文件其实也是由若干个512字节大小的block组成的。`redo日志文件结构图` 如下图所示：



> - redo日志文件组中，每个文件的**大小都一样**，**格式页一样**。
> - 前2048个字节（也就是前4个block），用来存储一些**管理信息**。
> - 从第2048字节往后的字节，用来存储log buffer中的**block镜像**。

- 所以前面所说的循环使用redo日志文件，其实是从每个日志文件的前2048个字节开始算起的。
- log file header的参数说明如下：

> 1> 【4B】LOG_HEADER_FORMAT
>
> redo日志的版本，在MySQL 5.7.22中永远为1。
>
> 2> 【4B】LOG_HEADER_PAD1
>
> 用于字节填充，没什么实际意义
>
> 3> **【8B】LOG_HEADER_START_LSN**
>
> **标记本redo log文件偏移量为2048字节处对应的lsn值。**
>
> 4> 【32B】LOG_HEADER_CREATOR
>
> 标记本redo日志文件的创建者是谁。正常运行时该值为MySQL的版本号，如“SQL 5.7.22”
>
> 5>【4B】LOG_BLOCK_CHECKSUM
>
> 本block的校验值；所有block都有该值，我们不用关心。

- checkpoint1或checkpoint2的参数说明如下：

> 1> **【8B】LOG_CHECKPOINT_NO**

服务器执行checkpoint的编号，每执行一次checkpoint，该值就加1。

2> 【8B】LOG_CHECKPOINT_LSN

服务器在结束checkpoint时对应的lsn值；系统在崩溃后恢复时将从该值开始。

3> 【8B】LOG_CHECKPOINT_OFFSET

上个属性中的lsn值在redo日志文件组中的偏移量。

4> 【8B】LOG_CHECKPOINT_LOG_BUF_SIZE

服务器在执行checkpoint操作时对应的log buffer的大小。

5> 【4B】LOG_BLOCK_CHECKSUM

本block的校验值；所有block都有该值，我们不用关心。

- 系统中**checkpoint**的相关信息其实只存储在redo日志文件组的**第一个日志文件中**。

六、log sequence number

- 什么是lsn

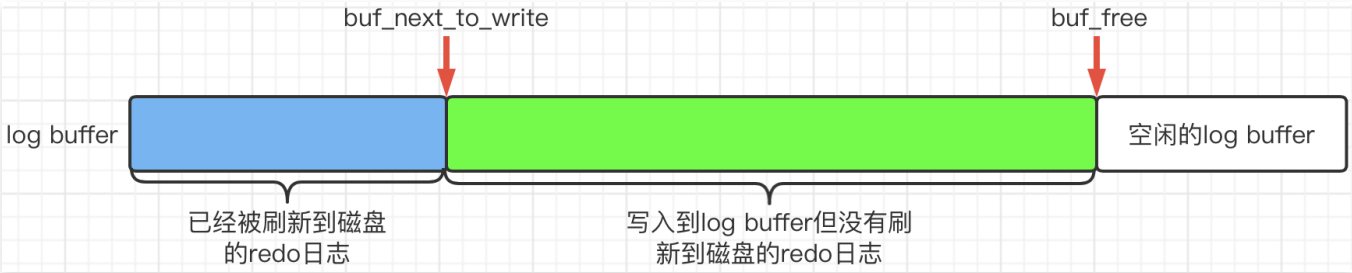
lsn全称为 `log sequence number` ，是一个**全局变量**，用来**记录当前总共已经写入的redo日志量**。lsn初始值为**8704**，也就是说，一条redo日志也没写入的时候，lsn的值就是 `8704` 。

- MTR写入log buffer后，lsn的变化示意图：

6.1> flushed_to_disk_lsn

- 如何知道有哪些**日志被刷新到磁盘**中了

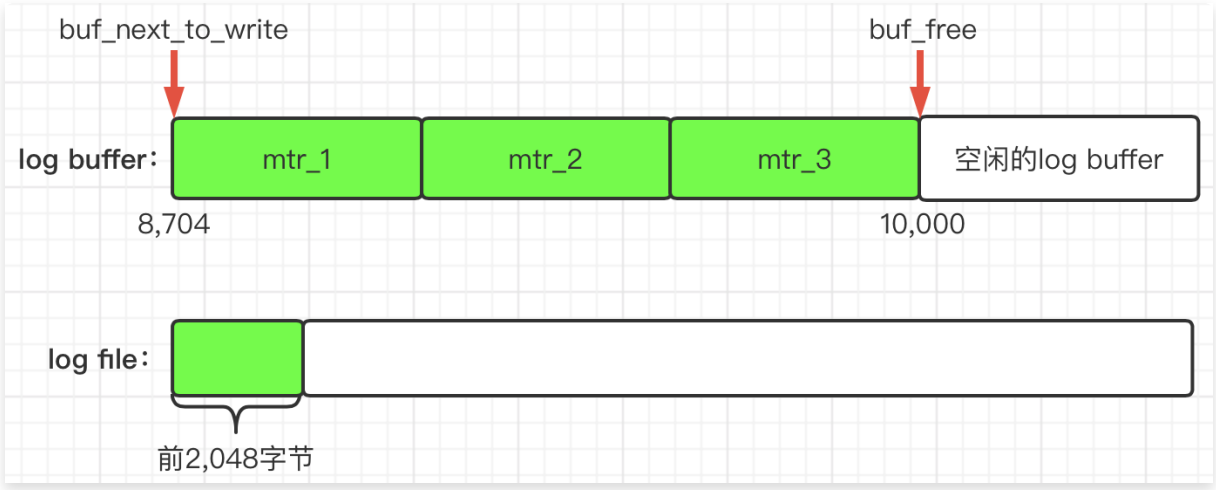
  - 一个名为 `buf_next_to_write` 的全局变量，用来标记当前log buffer中已经有哪些日志被刷新到磁盘中了。如下所示：



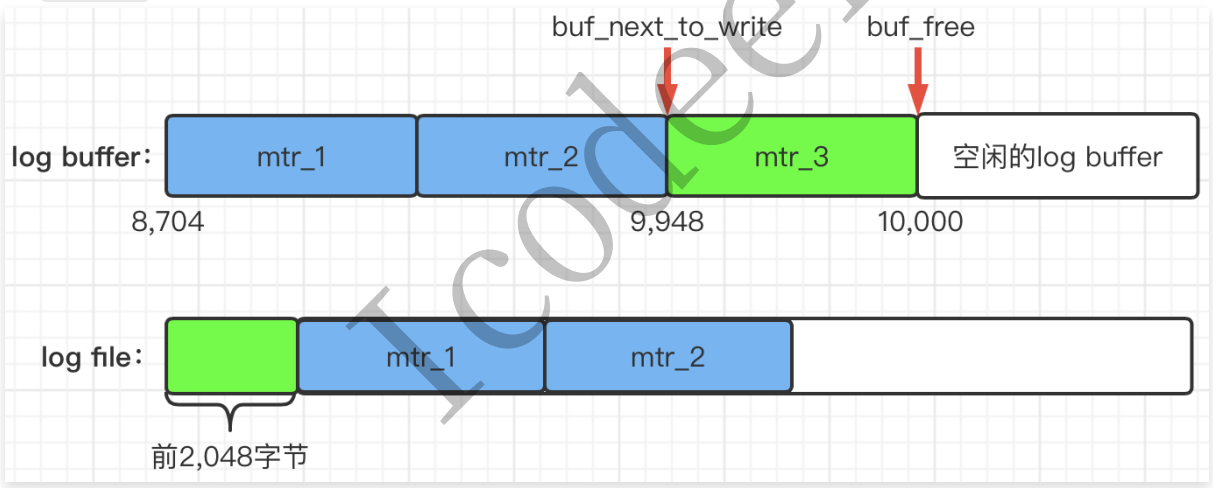
- 演示一下block写入redo日志文件的过程

  - 首先：系统在第一次启动后，向log buffer中写入了mtr_1(`8,716~8,916`)、mtr_2(`8,916~`

`9,948`)、mtr_3(`9,948~10,000`)这3个MTR产生的redo日志。此时的lsn已经增长到了10,000，由于没有刷新操作，此时`flushed_to_disk_lsn`的值仍为初始值`8,704`。如下图所示：

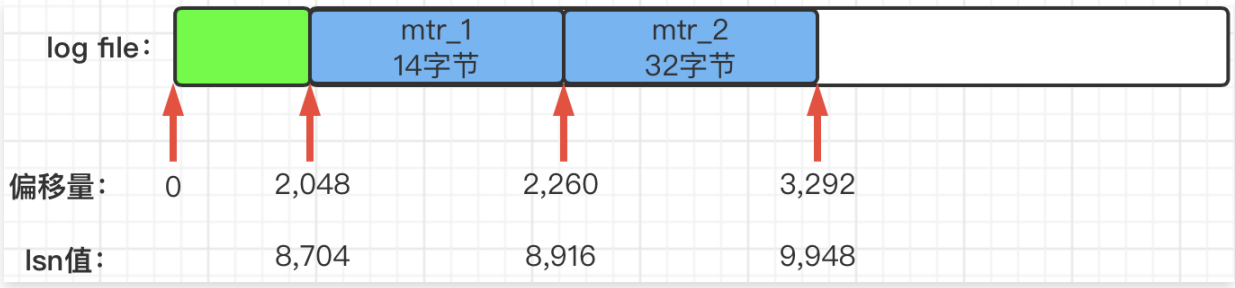


- 随后：将log buffer中的block刷新到redo日志文件中。假设讲`mtr_1`和`mtr_2`的redo日志刷新到磁盘，那么flush_to_disk_len就应该增长mtr_1和mtr_2写入的日志量，即：增长到`9,948`。如下图所示：



6.2> lsn值与redo日志文件组中的偏移量的对应关系

- lsn值和redo日志文件组偏移量的对应关系



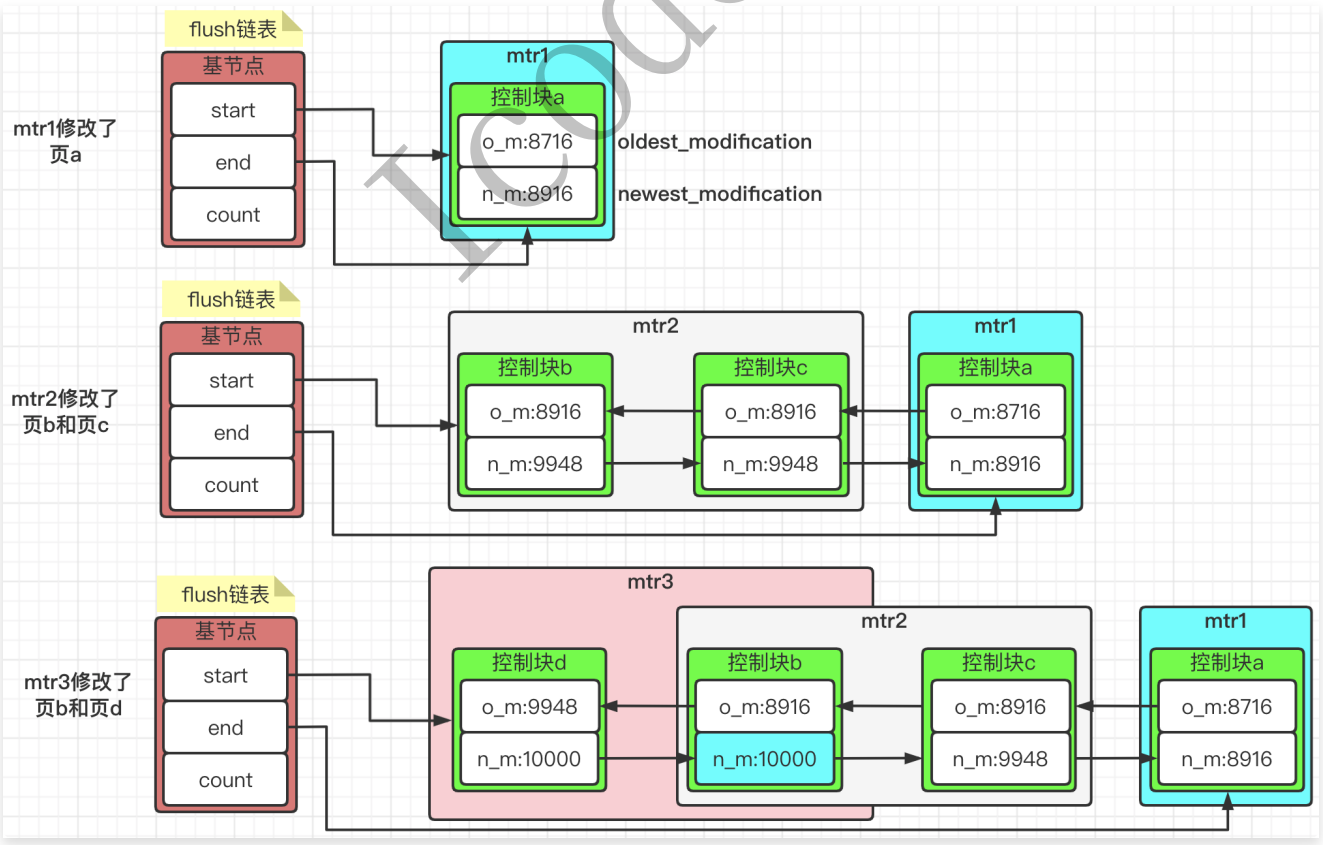
【注】偏移量从`2,048`开始；lsn值从`8,704`开始；

6.3> flush链表中的lsn

- 一个MTR代表对底层页面的一次原子访问，在访问过程中可能会产生一组不可分割的redo日志；在MTR结束时，会把这一组redo日志写入到log buffer中。除此之外，在MTR结束时还有一件非常重要的事情要做，就是**把在MTR执行过程中修改过的页面加入到Buffer Pool的flush链表中**。
- 第一次修改某个已经加载到Buffer Pool中的页面时，就会把这个页面对应的**控制块**插入到flush链表的头部；之后再次修改该页面时，由于它已经在flush链表中，所以就**不再次插入**了。也就是说，flush链表中的脏页是按照页面的**第一次修改时间**进行排序的。
- 在这个过程中，会在缓冲页对应的控制块中记录两个关于页面何时修改的属性：

> - oldest_modification
>
> **第一次修改**Buffer Pool中的某个缓冲页时，就将修改该页面的MTR**开始时对应的lsn值**写入这个属性。
>
> - newest_modification
>
> **每修改**一次页面，都会将修改该页面的MTR**结束时对应的lsn**值写入这个属性。也就是说，该属性表示页面最近一次修改后对应的lsn值。

- 下面进行举例演示，如图所示：

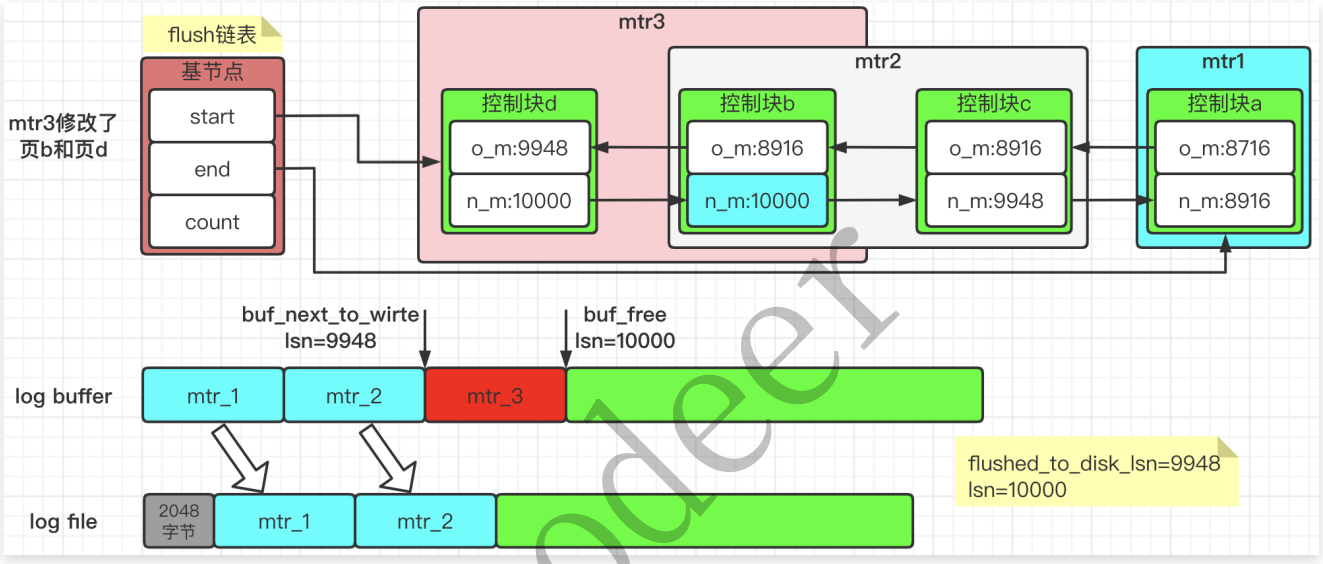


【解释】页b第二次被修改了，所以只改变newest_modification的值，oldest_modification的值不变化。

- 所以，综上所述，在flush链表中，前面的脏页修改的时间比较晚，后面的脏页修改的时间比较早。

七、checkpoint

- 如果redo日志对应的**脏页已经刷新到磁盘中**了，那么它也就失去了存在的意义了。它所占用的磁盘空间就可以被后续的redo日志所重用。
- 针对如下图：



【解释】

- mtr1和mtr2生成的redo日志虽然已经写到磁盘上的log file中了，但是它们修改的脏页仍然留在Buffer Pool中，所以它们的redo日志不可以被覆盖。
- 随着系统运行，如果页a从Buffer Pool中刷到了磁盘上，那么页a对应的控制块就会**从flush链表中移除掉**。而且，它的redo日志占用的空间就可以被覆盖掉了。

- InnoDB通过全局变量 `checkpoint_lsn` ，来表示当前系统中**可以被覆盖的redo日志总量是多少**。这个变量的初始值也是**8704**（因为lsn的初始值就是8704）。
- 比如，现在页a被刷新到了磁盘上，mtr1生成的redo日志就可以被覆盖了，所以可以进行一个增加checkpoint_lsn的操作。我们把这个过程称为**执行一次checkpoint**。
- 执行一次checkpoint可以分为两个步骤

- 步骤一：计算当前系统中**可以被覆盖**的redo日志对应的lsn值最大是多少

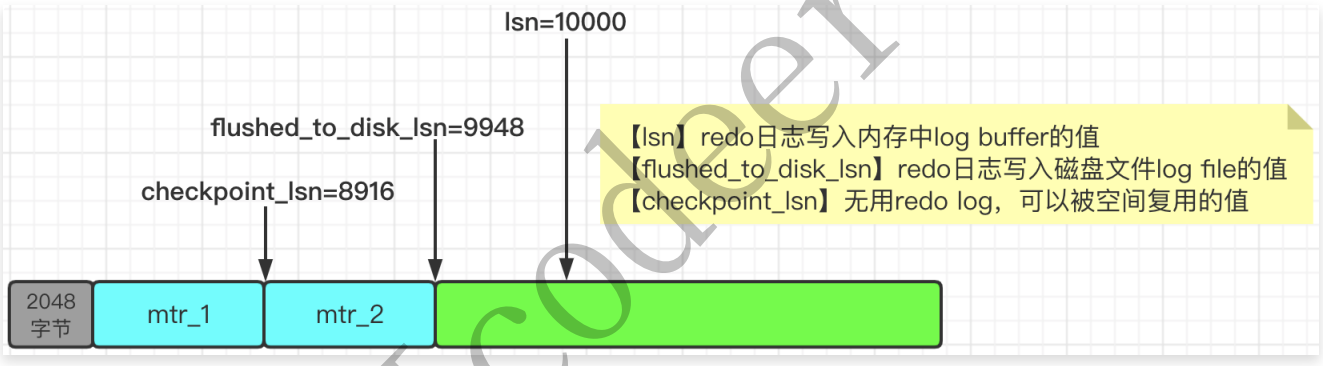
  比如，当前系统中页a已经被刷新到磁盘了，那么**flush链表的尾节点**就是页c。该节点就是当前系统中最早修改的脏页了，它的oldest_modification=8916，我们就 `把8916赋值给`

> `checkpoint_lsn`（也就是说**在redo日志对应的lsn值小于8916时，就可以被覆盖掉**）。
>
> - 步骤二：将checkpoint_lsn与对应的redo日志文件组偏移量以及此次checkpoint的编号写到日志文件的管理信息（checkpoint1或者checkpoint2）中**checkpoint_no**变量，`用来统计目前系统执行了多少次checkpoint，每执行一次checkpoint，该变量就+1`；前面我们也说过，计算一个lsn值对应的redo日志文件组偏移量是很容易的，所以可以计算出该checkpoint_lsn在redo日志文件组中对应的偏移量**checkpoint_offset**；然后将checkpoint_lsn、checkpoint_no、checkpoint_offset这3个值写到redo日志文件组的管理信息中。

- 每个redo日志文件都有2048字节的管理信息，但是上述关于checkpoint的信息只会被写到日志文件组中的**第一个日志文件**的管理信息中。
- *不过，应该存储到checkpoint1还是checkpoint2中呢?*

> 答：当checkpoint_no的值为**偶数**时，就写到`checkpoint1`中；是**奇数**时，就写到`checkpoint2`中。

- redo日志文件组中各个lsn值的关系



八、用户线程批量从flush链表中刷出脏页

- 一般情况下，针对Buffer Pool中的刷脏页操作，都是**后台线程**对LRU链表和**flush链表**进行刷脏页操作的，主要是因为刷脏操作比较慢，不想影响用户线程处理请求。
- 但是，如果修改操作十分频繁，导致redo日志操作频繁，系统lsn值增长过快。如果后台线程的刷脏操作不能将脏页快速刷出，系统将无法及时执行checkpoint，可能就需要**用户线程**从flush链表中把那些**最早修改的脏页**（oldest_modification较小的脏页）同步刷新到磁盘。这样这些脏页对应的redo日志就没有用了，然后就可以去执行checkpoint了。

九、查看系统中的各种lsn值

- 可以使用SHOW ENGINE INNODB STATUS \G 命令查看当前各种lsn值的情况。

```
---
LOG
---
Log sequence number          21673923
Log buffer assigned up to    21673923
Log buffer completed up to   21673923
Log written up to            21673923
Log flushed up to            21673923
Added dirty pages up to      21673923
Pages flushed up to          21673923
Last checkpoint at           21673923
13 log i/o's done, 0.00 log i/o's/second
```

- Log sequence number

> lsn值，即：当前系统已经写入的redo日志量。包括写入到log buffer中的redo日志量。

- Log flushed up to

> flushed_to_disk_lsn，即：写入磁盘log file文件的redo日志量。

- Pages flushed up to

> 表示flush链表中被最早修改的那个页面对应的oldest_modification属性值。

- Last checkpoint at

> checkpoint_lsn值。

十、innodb_flush_log_at_trx_commit的用法

- 为了保证事务的持久性，用户线程在事务提交时，需要将该事务执行过程中产生的所有redo日志都刷新到磁盘中。这个规则我们可以通过系统变量innodb_flush_log_at_trx_commit来进行配置修改，该变量有如下3个可选值:

> 0：表示在事务提交时，不立即向磁盘同步redo日志，这个任务交给后台线程来处理；
> 1：表示在事务提交时，需要将redo日志同步到磁盘。（默认值）
> 2：表示在事务提交时，需要将redo日志写到操作系统的缓冲区中，但并不需要保证将日志真正刷新到磁盘。如果操作系统挂掉了，则数据丢失。

十一、崩溃恢复

11.1> 确认恢复的起点

- 我们只需要把checkpoint1和checkpoint2这两个block中的checkpoint_no值读出来并比较一下大

小，**哪个checkpoint_no值更大，就说明哪个block存储的就是最近一次的checkpoint信息**。这样就能拿到最近发生的checkpoint对应的checkpoint_lsn值以及它在redo日志文件组中的偏移量checkpoint_offset。

11.2> 确认恢复的终点

- 前面说过，redo日志是顺序写入的，写满一个block之后再往下一个block中写，所以，根据block的log block header部分中有一个名为 `LOG_BLOCK_HDR_DATA_LEN` 的属性值，该值记录了当前block中使用了多少字节的空间。**对于被填满的block来说，该值永远为512**，如果不为512，则表示此次崩溃恢复中需要扫描的最后一个block。
- 综上所述：当因崩溃而恢复系统时，**只需要从checkpoint_lsn在日志文件组中对应的偏移量开始，一直扫描redo日志文件中的block，直到某个block的LOG_BLOCK_HDR_DATA_LEN值不等于512为止**。

11.3> 怎么恢复

- 使用哈希表

  - 根据redo日志的 `spaceID` 和 `page number` 属性计算出哈希值，把spaceID和page number相同的redo日志放到哈希表的同一个槽位中。如果哈希值相同，则使用链表连接起来。（链表按照生成的先后顺序连接，redo1——>redo2——redo3... ——>redoN）
  - 这样就可以在恢复过程中，**针对同一个页面一次性的修复好，避免了很多读取页面的随机I/O，加快恢复速度**。

- 跳过已经刷新到磁盘中的页面

  - 对于lsn值不小于checkpoint_lsn的redo日志，它所对应的脏页不能确定是否已经刷到磁盘中。
  - 原因是在最近执行的一次checkpoint后，后台线程可能又不断地从LRU链表和flush链表中将一些脏页刷出Buffer Pool。
  - 解决办法：

  在File Header中有一个称为 `FIL_PAGE_LSN` 的属性，该属性记载了最近一次修改页面时对应的 `lsn` 值（其实就是页面控制块中的 `newest_modification` 值）。如果在执行了某次checkpoint之后，有脏页被刷新到磁盘中，那么**该页对应的FIL_PAGE_LSN代表的lsn值肯定大于checkpoint_lsn的值**。所以符合这种情况的页面，就不需要进行恢复了。

吾尝跂而望矣，不如登高之博见也。
登高而招，臂非加长也，而见者远；
顺风而呼，声非加疾也，而闻者彰。
假舆马者，非利足也，而致千里；
假舟楫者，非能水也，而绝江河。
君子生非异也，善假于物也。
------ 摘自《劝学》

愿本文可以成为大家的“山”、“风”、“马”、“舟”，助大家在技术之路上乘风破浪，大展宏图~~

同时，也欢迎大家关注我的公众号“爪哇缪斯” ~\(^o^)/~ 「干货分享，每天更新」