第20章、redo日志（上）

标签： MySQL是怎样运行的

事先说明

本文以及接下来的几篇文章将会频繁的使用到我们前边唠叨的InnoDB记录行格式、页面格式、索引原理、表空间的组成等各种基础知识，如果大家对这些东西理解的不透彻，那么阅读下边的文字可能会有些吃力，为保证您的阅读体验，请确保自己已经掌握了我前边唠叨的这些知识。

redo日志是个啥

我们知道InnoDB存储引擎是以页为单位来管理存储空间的，我们进行的增删改查操作其实本质上都是在访问页面（包括读页面、写页面、创建新页面等操作）。我们前边唠叨Buffer Pool的时候说过，在真正访问页面之前，需要把在磁盘上的页缓存到内存中的Buffer Pool之后才可以访问。但是在唠叨事务的时候又强调过一个称之为持久性的特性，就是说对于一个已经提交的事务，在事务提交后即使系统发生了崩溃，这个事务对数据库中所做的更改也不能丢失。但是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，那么这个已经提交了的事务对数据库中所做的更改也就跟着丢失了，这是我们所不能忍受的（想想ATM机已经提示狗哥转账成功，但之后由于服务器出现故障，重启之后猫爷发现自己没收到钱，猫爷就被砍死了）。那么如何保证这个持久性呢？一个很简单的做法就是在事务提交完成之前把该事务所修改的所有页面都刷新到磁盘，但是这个简单粗暴的做法有些问题：

- 刷新一个完整的数据页太浪费了

  有时候我们仅仅修改了某个页面中的一个字节，但是我们知道在InnoDB中是以页为单位来进行磁盘IO的，也就是说我们在该事务提交时不得不将一个完整的页面从内存中刷新到磁盘，我们又知道一个页面默认是16KB大小，只修改一个字节就要刷新16KB的数据到磁盘上显然是太浪费了。

- 随机IO刷起来比较慢

  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，倒霉催的是该事务修改的这些页面可能并不相邻，这就意味着在将某个事务修改的Buffer Pool中的页面刷新到磁盘时，需要进行很多的随机IO，随机IO比顺序IO要慢，尤其对于传统的机械硬盘来说。

咋办呢？再次回到我们的初心：我们只是想让已经提交了的事务对数据库中数据所做的修改永久生效，即使后来系统崩溃，在重启后也能把这种修改恢复出来。所以我们其实没有必要在每次事务提交时就把该事务在内存中修改过的全部页面刷新到磁盘，只需要把修改了哪些东西记录一下就好，比方说某个事务将系统表空间中的第100号页面中偏移量为1000处的那个字节的值1改成2

我们只需要记录一下：

> 将第0号表空间的100号页面的偏移量为1000处的值更新为2。

这样我们在事务提交时，把上述内容刷新到磁盘中，即使之后系统崩溃了，重启之后只要按照上述内容所记录的步骤重新更新一下数据页，那么该事务对数据库中所做的修改又可以被恢复出来，也就意味着满足持久性的要求。因为在系统崩溃重启时需要按照上述内容所记录的步骤重新更新数据页，所以上述内容也被称之为重做日志，英文名为redo log，我们也可以土洋结合，称之为redo日志。与在事务提交时将所有修改过的内存中的页面刷新到磁盘中相比，只将该事务执行过程中产生的redo日志刷新到磁盘的好处如下：

- redo日志占用的空间非常小

  存储表空间ID、页号、偏移量以及需要更新的值所需的存储空间是很小的，关于redo日志的格式我们稍后会详细唠叨，现在只要知道一条redo日志占用的空间不是很大就好了。

- 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IO。

redo日志格式

通过上边的内容我们知道，redo日志本质上只是记录了一下事务对数据库做了哪些修改。 设计InnoDB的大叔们针对事务对数据库的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日志都有下边这种通用的结构：

redo日志通用结构

各个部分的详细释义如下：

- type：该条redo日志的类型。

  在MySQL 5.7.21这个版本中，设计InnoDB的大叔一共为redo日志设计了53种不同的类型，稍后会详细介绍不同类型的redo日志。

- space ID：表空间ID。
- page number：页号。
- data：该条redo日志的具体内容。

简单的redo日志类型

我们前边介绍InnoDB的记录行格式的时候说过，如果我们没有为某个表显式的定义主键，并且表中也没有定义Unique键，那么InnoDB会自动的为表添加一个称之为row_id的隐藏列作为主键。为这个row_id隐藏列赋值的方式如下：

- 服务器会在内存中维护一个全局变量，每当向某个包含隐藏的row_id列的表中插入一条记录时，就会把该变量的值当作新记录的row_id列的值，并且把该变量自增1。
- 每当这个变量的值为256的倍数时，就会将该变量的值刷新到系统表空间的页号为7的页面中一个称之为Max Row ID的属性处（我们前边介绍表空间结构时详细说过）。
- 当系统启动时，会将上边提到的Max Row ID属性加载到内存中，将该值加上256之后赋值给我们前边提到的全局变量（因为在上次关机时该全局变量的值可能大于Max Row ID属性值）。

这个Max Row ID属性占用的存储空间是8个字节，当某个事务向某个包含row_id隐藏列的表插入一条记录，并且为该记录分配的row_id值为256的倍数时，就会向系统表空间页号为7的页面的相应偏移量处写入8个字节的值。但是我们要知道，这个写入实际上是在Buffer Pool中完成的，我们需要为这个页面的修改记录一条redo日志，以便在系统崩溃后能将已经提交的该事务对该页面所做的修改恢复出来。这种情况下对页面的修改是极其简单的，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，具体被修改的内容是啥就好了，设计InnoDB的大叔把这种极其简单的redo日志称之为物理日志，并且根据在页面中写入数据的多少划分了几种不同的redo日志类型：

- MLOG_1BYTE（type字段对应的十进制数字为1）：表示在页面的某个偏移量处写入1个字节的redo日志类型。
- MLOG_2BYTE（type字段对应的十进制数字为2）：表示在页面的某个偏移量处写入2个字节的redo日志类型。
- MLOG_4BYTE（type字段对应的十进制数字为4）：表示在页面的某个偏移量处写入4个字节的redo日志类型。
- MLOG_8BYTE（type字段对应的十进制数字为8）：表示在页面的某个偏移量处写入8个字节的redo日志类型。
- MLOG_WRITE_STRING（type字段对应的十进制数字为30）：表示在页面的某个偏移量处写入一串数据。

我们上边提到的Max Row ID属性实际占用8个字节的存储空间，所以在修改页面中的该属性时，会记录一条类型为MLOG_8BYTE的redo日志，MLOG_8BYTE的redo日志结构如下所示：

其余MLOG_1BYTE、MLOG_2BYTE、MLOG_4BYTE类型的redo日志结构和MLOG_8BYTE的类似，只不过具体数据中包含对应个字节的数据罢了。MLOG_WRITE_STRING类型的redo日志表示写入一串数据，但是因为不能确定写入的具体数据占用多少字节，所以需要在日志结构中添加一个len字段：

小贴士： 只要将MLOG_WRITE_STRING类型的redo日志的len字段填充上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，为啥还要多此一举设计这么多类型呢？还不是因为省空间啊，能不写len字段就不写len字段，省一个字节算一个字节。

复杂一些的redo日志类型

有时候执行一条语句会修改非常多的页面，包括系统数据页面和用户数据页面（用户数据指的就是聚簇索引和二级索引对应的B+树）。以一条INSERT语句为例，它除了要向B+树的页面中插入数据，也可能更新系统数据Max Row ID的值，不过对于我们用户来说，平时更关心的是语句对B+树所做更新：

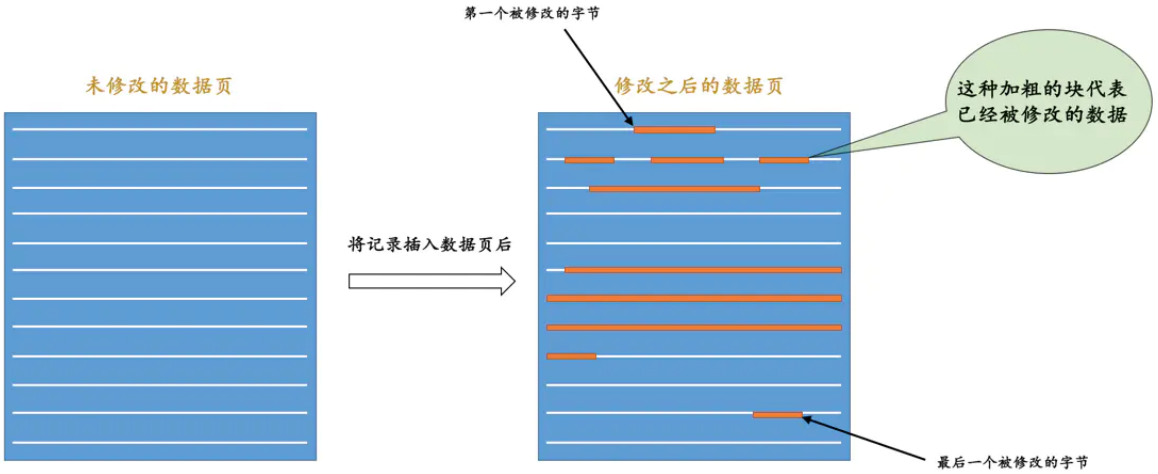
- 表中包含多少个索引，一条INSERT语句就可能更新多少棵B+树。
- 针对某一棵B+树来说，既可能更新叶子节点页面，也可能更新内节点页面，也可能创建新的页面（在该记录插入的叶子节点的剩余空间比较少，不足以存放该记录时，会进行页面的分裂，在内节点页面中添加目录项记录）。

在语句执行过程中，INSERT语句对所有页面的修改都得保存到redo日志中去。这句话说的比较轻巧，做起来可就比较麻烦了，比方说将记录插入到聚簇索引中时，如果定位到的叶子节点的剩余

空间足够存储该记录时，那么只更新该叶子节点页面就好，那么只记录一条MLOG_WRITE_STRING类型的redo日志，表明在页面的某个偏移量处增加了哪些数据就好了么？那就too young too naive了~ 别忘了一个数据页中除了存储实际的记录之后，还有什么File Header、Page Header、Page Directory等等部分（在唠叨数据页的章节有详细讲解），所以每往叶子节点代表的数据页里插入一条记录时，还有其他很多地方会跟着更新，比如说：

- 可能更新Page Directory中的槽信息。
- Page Header中的各种页面统计信息，比如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，吧啦吧啦，各种信息都可能会被修改。
- 我们知道在数据页里的记录是按照索引列从小到大的顺序组成一个单向链表的，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。
- 还有别的吧啦吧啦的更新的地方，就不一一唠叨了...

画一个简易的示意图就像是这样：



说了这么多，就是想表达：把一条记录插入到一个页面时需要更改的地方非常多。这时我们如果使用上边介绍的简单的物理redo日志来记录这些修改时，可以有两种解决方案：

- 方案一：在每个修改的地方都记录一条redo日志。

  也就是如上图所示，有多少个加粗的块，就写多少条物理redo日志。这样子记录redo日志的缺点是显而易见的，因为被修改的地方是在太多了，可能记录的redo日志占用的空间都比整个页面占用的空间都多了~

- 方案二：将整个页面的第一个被修改的字节到最后一个修改的字节之间所有的数据当成是一条物理redo日志中的具体数据。

  从图中也可以看出来，第一个被修改的字节到最后一个修改的字节之间仍然有许多没有修改过的数据，我们把这些没有修改的数据也加入到redo日志中去岂不是太浪费了~

正因为上述两种使用物理redo日志的方式来记录某个页面中做了哪些修改比较浪费，设计InnoDB的大叔本着勤俭节约的初心，提出了一些新的redo日志类型，比如：

- MLOG_REC_INSERT（对应的十进制数字为9）：表示插入一条使用非紧凑行格式的记录时的redo日志类型。
- MLOG_COMP_REC_INSERT（对应的十进制数字为38）：表示插入一条使用紧凑行格式的记录时的redo日志类型。

小贴士： Redundant是一种比较原始的行格式，它就是非紧凑的。而Compact、Dynamic以及Compressed行格式是较新的行格式，它们是紧凑的（占用更小的存储空间）。

- MLOG_COMP_PAGE_CREATE（type字段对应的十进制数字为58）：表示创建一个存储紧凑行格式记录的页面的redo日志类型。
- MLOG_COMP_REC_DELETE（type字段对应的十进制数字为42）：表示删除一条使用紧凑行格式记录的redo日志类型。
- MLOG_COMP_LIST_START_DELETE（type字段对应的十进制数字为44）：表示从某条给定记录开始删除页面中的一系列使用紧凑行格式记录的redo日志类型。
- MLOG_COMP_LIST_END_DELETE（type字段对应的十进制数字为43）：
  与MLOG_COMP_LIST_START_DELETE类型的redo日志呼应，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。

小贴士： 我们前边唠叨InnoDB数据页格式的时候重点强调过，数据页中的记录是按照索引列大小的顺序组成单向链表的。有时候我们会有删除索引列的值在某个区间范围内的所有记录的需求，这时候如果我们每删除一条记录就写一条redo日志的话，效率可能有点低，所以提出MLOG_COMP_LIST_START_DELETE和MLOG_COMP_LIST_END_DELETE类型的redo日志，可以很大程度上减少redo日志的条数。

- MLOG_ZIP_PAGE_COMPRESS（type字段对应的十进制数字为51）：表示压缩一个数据页的redo日志类型。
- ……还有很多很多种类型，这就不列举了，等用到再说哈～

这些类型的redo日志既包含物理层面的意思，也包含逻辑层面的意思，具体指：

- 物理层面看，这些日志都指明了对哪个表空间的哪个页进行了修改。
- 逻辑层面看，在系统崩溃重启时，并不能直接根据这些日志里的记载，将页面内的某个偏移量处恢复成某个数据，而是需要调用一些事先准备好的函数，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。

大家看到这可能有些懵逼，我们还是以类型为MLOG_COMP_REC_INSERT这个代表插入一条使用紧凑行格式的记录时的redo日志为例来理解一下我们上边所说的物理层面和逻辑层面到底是个啥意思。废话少说，直接看一下这个类型为MLOG_COMP_REC_INSERT的redo日志的结构（由于字段太多了，我们把它们竖着看效果好些）：

这个类型为`MLOG_COMP_REC_INSERT`的`redo`日志结构有几个地方需要大家注意：

- 我们前边在唠叨索引的时候说过，在一个数据页里，不论是叶子节点还是非叶子节点，记录都是按照索引列从小到大的顺序排序的。对于二级索引来说，当索引列的值相同时，记录还需要按照主键值进行排序。图中`n_uniques`的值的含义是在一条记录中，需要几个字段的值才能确保记录的唯一性，这样当插入一条记录时就可以按照记录的前`n_uniques`个字段进行排序。对于聚簇索引来说，`n_uniques`的值为主键的列数，对于其他二级索引来说，该值为索引列数+主键列数。这里需要注意的是，唯一二级索引的值可能为`NULL`，所以该值仍然为索引列数+主键列数。
- `field1_len ~ fieldn_len`代表着该记录若干个字段占用存储空间的大小，需要注意的是，这里不管该字段的类型是固定长度大小的（比如`INT`），还是可变长度大小（比如`VARCHAR(M)`）的，该字段占用的大小始终要写入`redo`日志中。
- `offset`代表的是该记录的前一条记录在页面中的地址。为啥要记录前一条记录的地址呢？这

是因为每向数据页插入一条记录，都需要修改该页面中维护的记录链表，每条记录的记录头信息中都包含一个称为next_record的属性，所以在插入新记录时，需要修改前一条记录的next_record属性。

- 我们知道一条记录其实由额外信息和真实数据这两部分组成，这两个部分的总大小就是一条记录占用存储空间的总大小。通过end_seg_len的值可以间接的计算出一条记录占用存储空间的总大小，为啥不直接存储一条记录占用存储空间的总大小呢？这是因为写redo日志是一个非常频繁的操作，设计InnoDB的大叔想方设法想减小redo日志本身占用的存储空间大小，所以想了一些弯弯绕的算法来实现这个目标，end_seg_len这个字段就是为了节省redo日志存储空间而提出来的。至于具体设计InnoDB的大叔到底是用了什么神奇魔法减小redo日志大小的，我们这就不多唠叨了，因为的确有那么一丢丢小复杂，说清楚还是有一点点麻烦的，而且说明白了也没啥用。
- mismatch_index的值也是为了节省redo日志的大小而设立的，大家可以忽略。

很显然这个类型为MLOG_COMP_REC_INSERT的redo日志并没有记录PAGE_N_DIR_SLOTS的值修改为了啥，PAGE_HEAP_TOP的值修改为了啥，PAGE_N_HEAP的值修改为了啥等等这些信息，而只是把在本页面中插入一条记录所有必备的要素记了下来，之后系统崩溃重启时，服务器会调用相关向某个页面插入一条记录的那个函数，而redo日志中的那些数据就可以被当成是调用这个函数所需的参数，在调用完该函数后，页面中的PAGE_N_DIR_SLOTS、PAGE_HEAP_TOP、PAGE_N_HEAP等等的值也就都被恢复到系统崩溃前的样子了。这就是所谓的逻辑日志的意思。

redo日志格式小结

虽然上边说了一大堆关于redo日志格式的内容，但是如果你不是为了写一个解析redo日志的工具或者自己开发一套redo日志系统的话，那就没必要把InnoDB中的各种类型的redo日志格式都研究的透透的，没那个必要。上边我只是象征性的介绍了几种类型的redo日志格式，目的还是想让大家明白：redo日志会把事务在执行过程中对数据库所做的所有修改都记录下来，在之后系统崩溃重启后可以把事务所做的任何修改都恢复出来。

> 小贴士： 为了节省redo日志占用的存储空间大小，设计InnoDB的大叔对redo日志中的某些数据还可能进行压缩处理，比方说space ID和page number一般占用4个字节来存储，但是经过压缩后，可能使用更小的空间来存储。具体压缩算法就不唠叨了。

Mini-Transaction

以组的形式写入redo日志

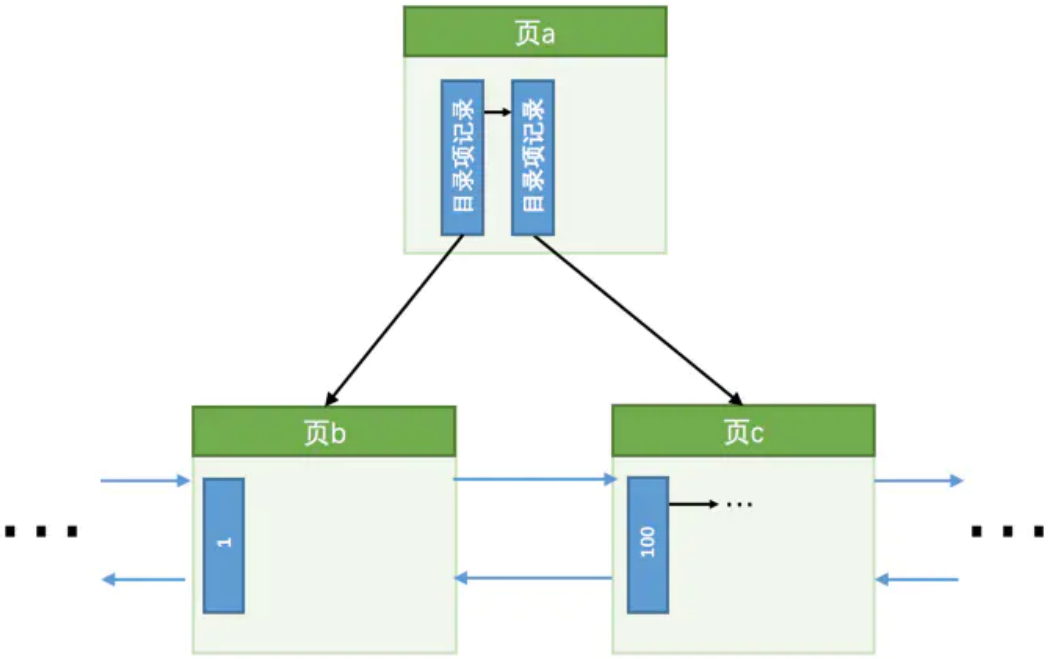
语句在执行过程中可能修改若干个页面。比如我们前边说的一条INSERT语句可能修改系统表空间页号为7的页面的Max Row ID属性（当然也可能更新别的系统页面，只不过我们没有都列举出来而已），还会更新聚簇索引和二级索引对应B+树中的页面。由于对这些页面的更改都发生在Buffer Pool中，所以在修改完页面之后，需要记录一下相应的redo日志。在执行语句的过程中产生的

redo日志被设计InnoDB的大叔人为的划分成了若干个不可分割的组，比如：

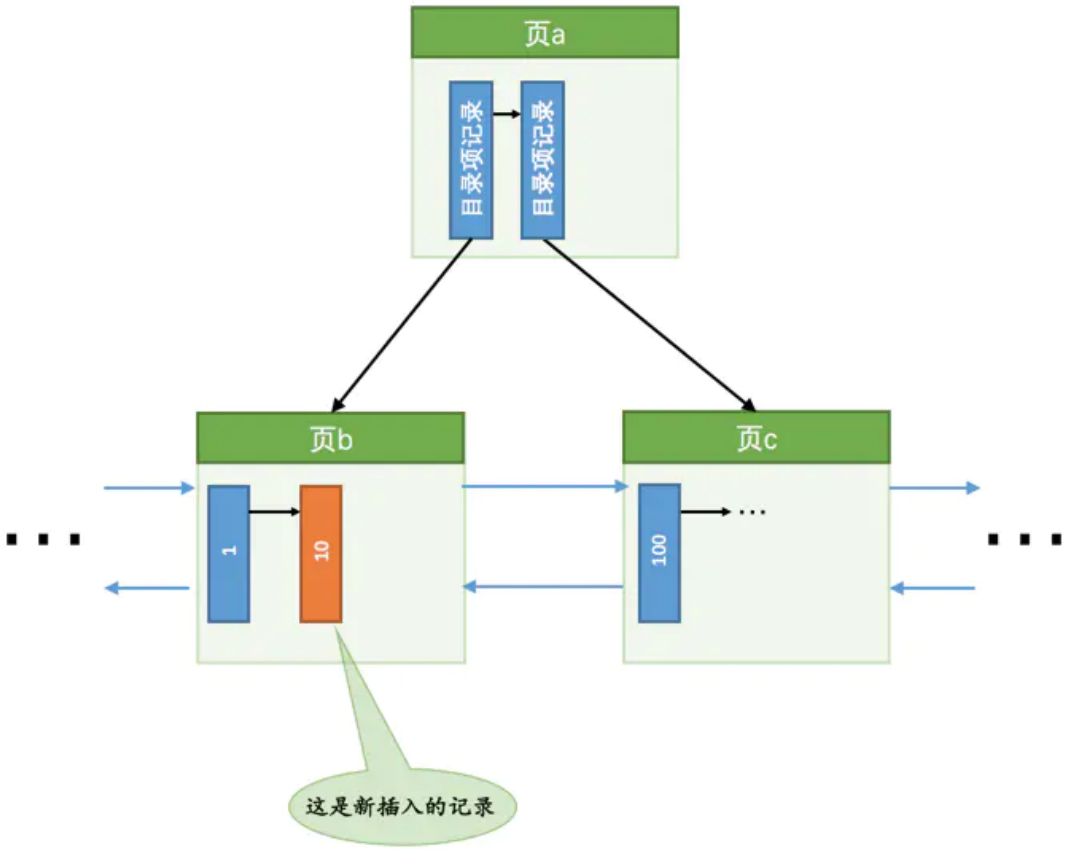
- 更新Max Row ID属性时产生的redo日志是不可分割的。
- 向聚簇索引对应B+树的页面中插入一条记录时产生的redo日志是不可分割的。
- 向某个二级索引对应B+树的页面中插入一条记录时产生的redo日志是不可分割的。
- 还有其他的一些对页面的访问操作时产生的redo日志是不可分割的。。。

怎么理解这个不可分割的意思呢？我们以向某个索引对应的B+树插入一条记录为例，在向B+树中插入这条记录之前，需要先定位到这条记录应该被插入到哪个叶子节点代表的数据页中，定位到具体的数据页之后，有两种可能的情况：

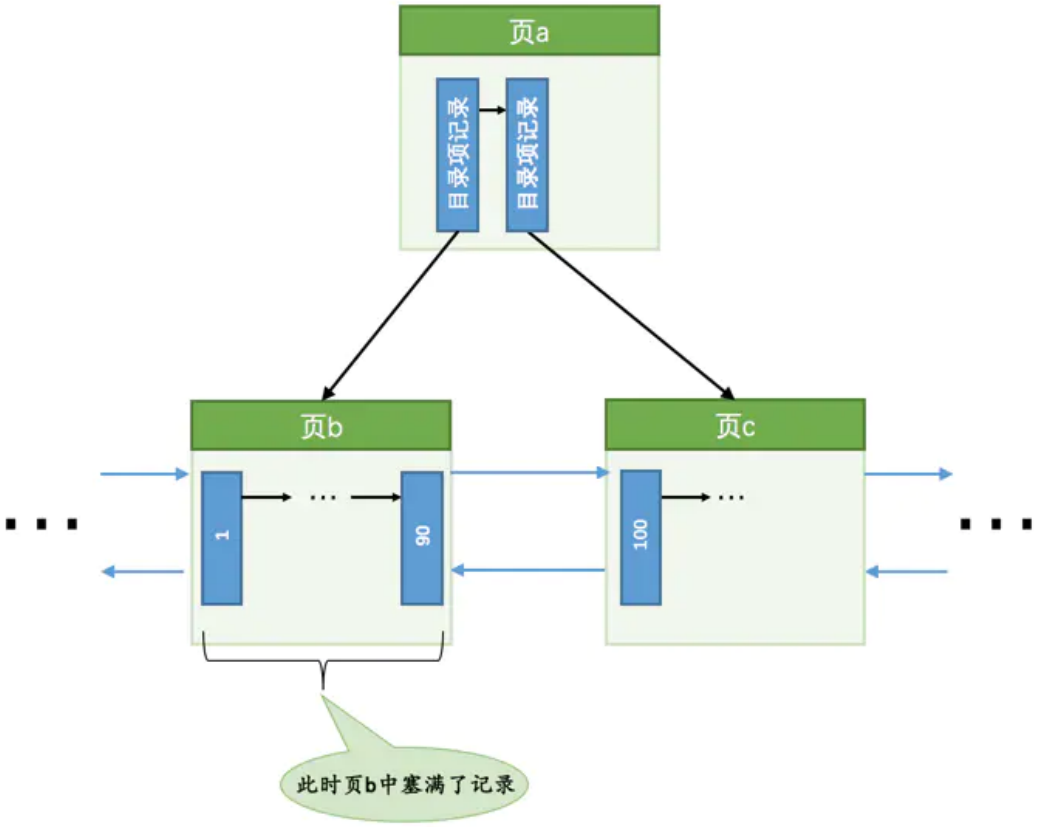
- 情况一：该数据页的剩余的空闲空间充足，足够容纳这一条待插入记录，那么事情很简单，直接把记录插入到这个数据页中，记录一条类型为MLOG_COMP_REC_INSERT的redo日志就好了，我们把这种情况称之为乐观插入。假如某个索引对应的B+树长这样：



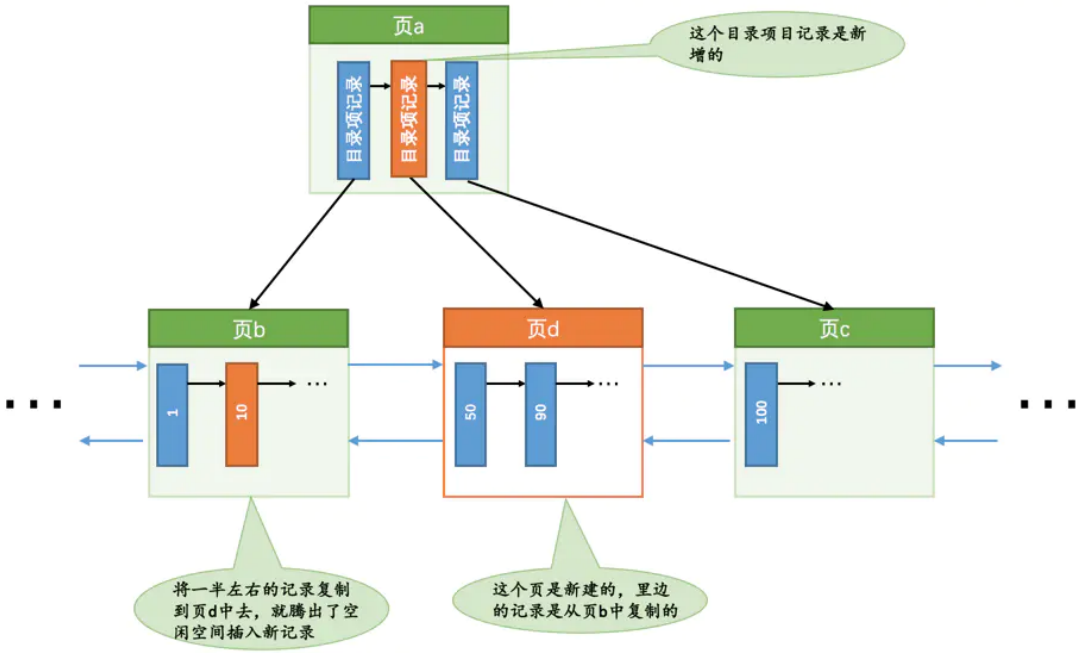
现在我们要插入一条键值为10的记录，很显然需要被插入到页b中，由于页b现在有足够的空间容纳一条记录，所以直接将该记录插入到页b中就好了，就像这样：



- 情况二：该数据页剩余的空闲空间不足，那么事情就悲剧了，我们前边说过，遇到这种情况要进行所谓的页分裂操作，也就是新建一个叶子节点，然后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，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指向这个新创建的页面。很显然，这个过程要对多个页面进行修改，也就意味着会产生多条redo日志，我们把这种情况称之为悲观插入。假如某个索引对应的B+树长这样：



现在我们要插入一条键值为10的记录，很显然需要被插入到页b中，但是从图中也可以看出来，此时页b已经塞满了记录，没有更多的空闲空间来容纳这条新记录了，所以我们需要进行页面的分裂操作，就像这样：



如果作为内节点的页a的剩余空闲空间也不足以容纳增加一条目录项记录，那需要继续做内节点页a的分裂操作，也就意味着会修改更多的页面，从而产生更多的redo日志。另外，对于悲观插入来说，由于需要新申请数据页，还需要改动一些系统页面，比方说要修改各种段、区的统计信息信息，各种链表的统计信息（比如什么FREE链表、FSP_FREE_FRAG链表吧啦吧啦我

们在唠叨表空间那一章中介绍过的各种东东）等等等等，反正总共需要记录的redo日志有二、三十条。

小贴士： 其实不光是悲观插入一条记录会生成许多条redo日志，设计InnoDB的大叔为了其他的一些功能，在乐观插入时也可能产生多条redo日志（具体是为了什么功能我们就不多说了，要不篇幅就受不了了~ ）。

设计InnoDB的大叔们认为向某个索引对应的B+树中插入一条记录的这个过程必须是原子的，不能说插了一半之后就停止了。比方说在悲观插入过程中，新的页面已经分配好了，数据也复制过去了，新的记录也插入到页面中了，可是没有向内节点中插入一条目录项记录，这个插入过程就是不完整的，这样会形成一棵不正确的B+树。我们知道redo日志是为了在系统崩溃重启时恢复崩溃前的状态，如果在悲观插入的过程中只记录了一部分redo日志，那么在系统崩溃重启时会将索引对应的B+树恢复成一种不正确的状态，这是设计InnoDB的大叔们所不能忍受的。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的redo日志，在进行系统崩溃重启恢复时，针对某个组中的redo日志，要么把全部的日志都恢复掉，要么一条也不恢复。怎么做到的呢？这得分情况讨论：

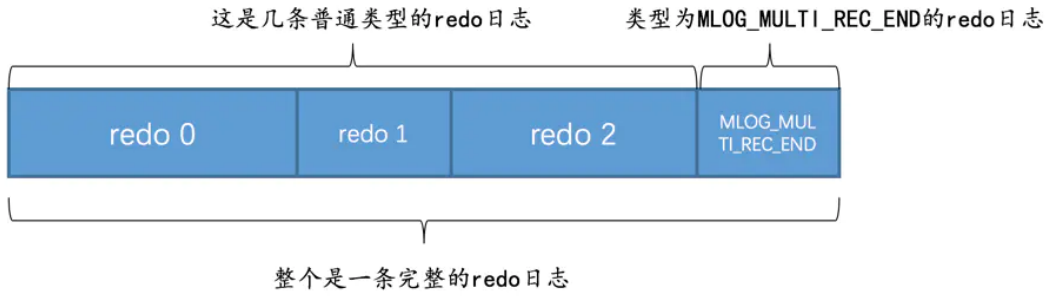
- 有的需要保证原子性的操作会生成多条redo日志，比如向某个索引对应的B+树中进行一次悲观插入就需要生成许多条redo日志。

  如何把这些redo日志划分到一个组里边儿呢？设计InnoDB的大叔做了一个很简单的小把戏，就是在该组中的最后一条redo日志后边加上一条特殊类型的redo日志，该类型名称为MLOG_MULTI_REC_END，type字段对应的十进制数字为31，该类型的redo日志结构很简单，只有一个type字段：

  MLOG_MULTI_REC_END 类型的redo日志结构

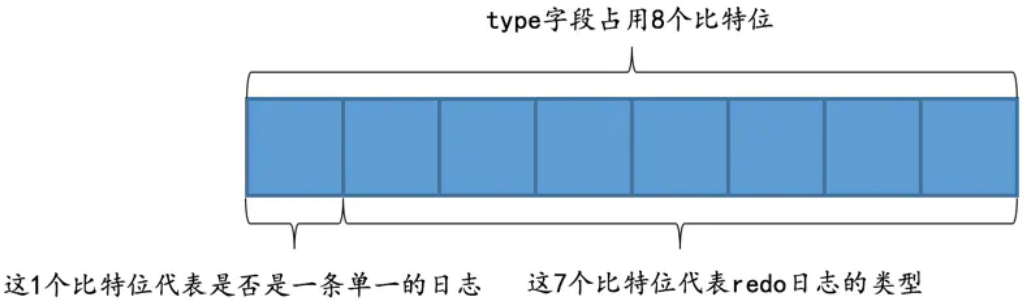
  所以某个需要保证原子性的操作产生的一系列redo日志必须要以一个类型为MLOG_MULTI_REC_END结尾，就像这样：



这样在系统崩溃重启进行恢复时，只有当解析到类型为MLOG_MULTI_REC_END的redo日志，才认为解析到了一组完整的redo日志，才会进行恢复。否则的话直接放弃前边解析到的redo日志。

- 有的需要保证原子性的操作只生成一条redo日志，比如更新Max Row ID属性的操作就只会生成一条redo日志。

  其实在一条日志后边跟一个类型为MLOG_MULTI_REC_END的redo日志也是可以的，不过设计InnoDB的大叔比较勤俭节约，他们不想浪费一个比特位。别忘了虽然redo日志的类型比较多，但撑死了也就是几十种，是小于127这个数字的，也就是说我们用7个比特位就足以包括所有的redo日志类型，而type字段其实是占用1个字节的，也就是说我们可以省出来一个比特位用来表示该需要保证原子性的操作只产生单一的一条redo日志，示意图如下：

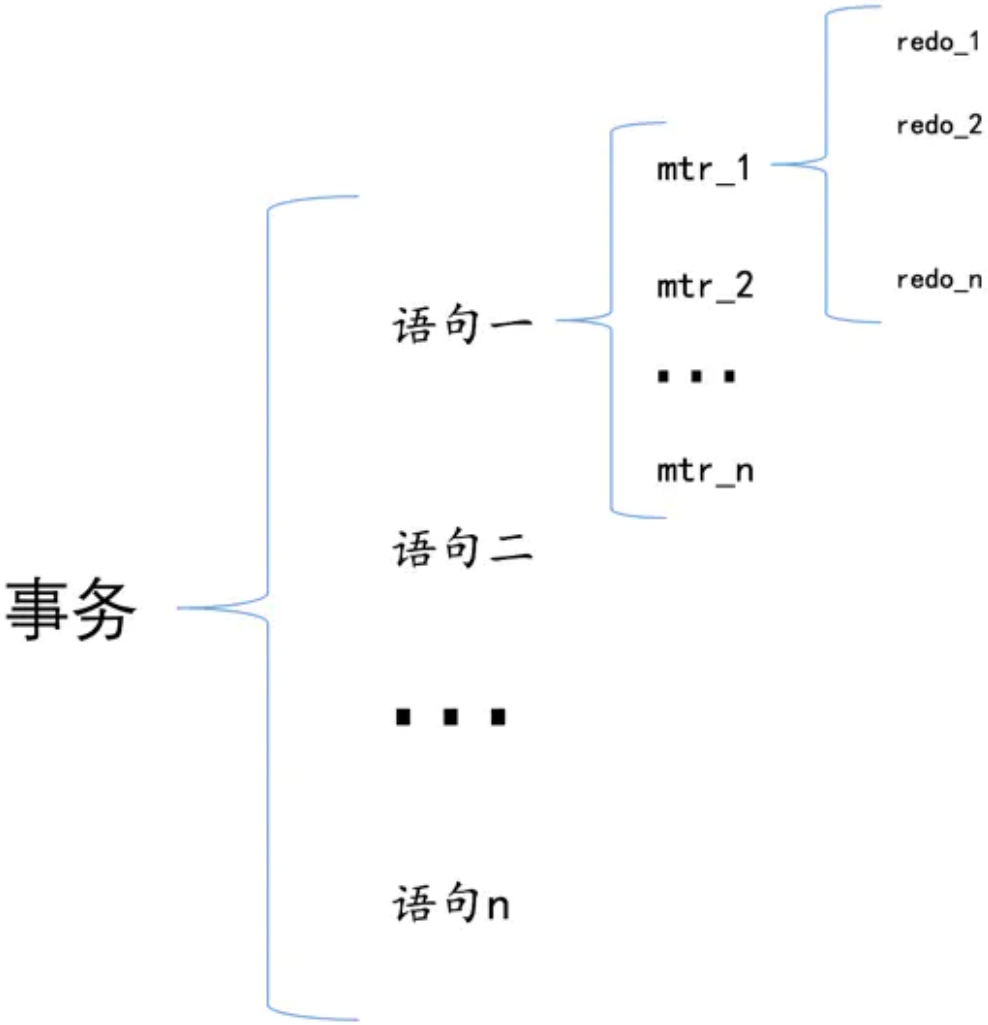


如果type字段的第一个比特位为1，代表该需要保证原子性的操作只产生了单一的一条redo日志，否则表示该需要保证原子性的操作产生了一系列的redo日志。

Mini-Transaction的概念

设计MySQL的大叔把对底层页面中的一次原子访问的过程称之为一个Mini-Transaction，简称mtr，比如上边所说的修改一次Max Row ID的值算是一个Mini-Transaction，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Mini-Transaction。通过上边的叙述我们也知道，一个所谓的mtr可以包含一组redo日志，在进行崩溃恢复时这一组redo日志作为一个不可分割的整体。

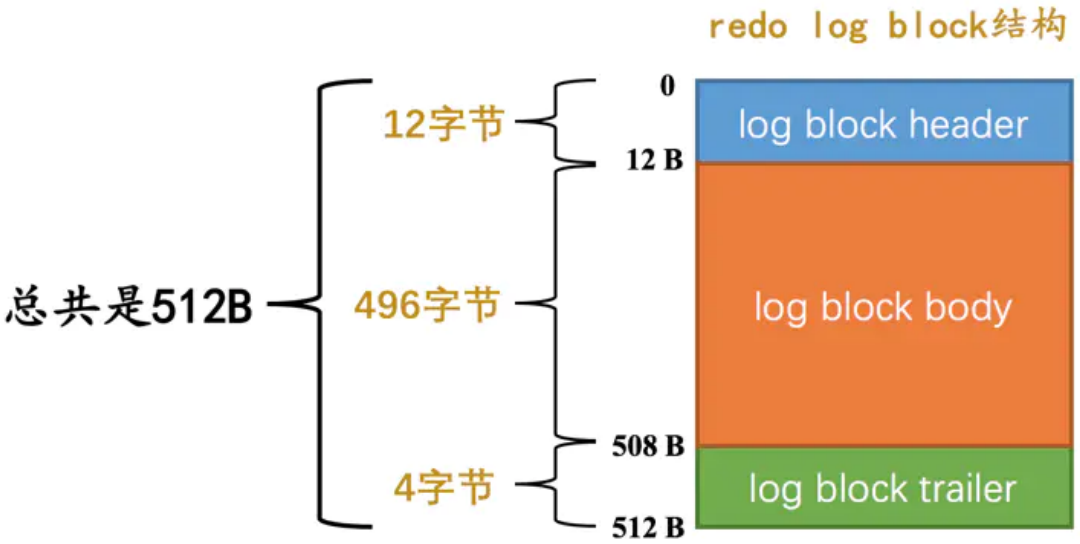
一个事务可以包含若干条语句，每一条语句其实是由若干个mtr组成，每一个mtr又可以包含若干条redo日志，画个图表示它们的关系就是这样：



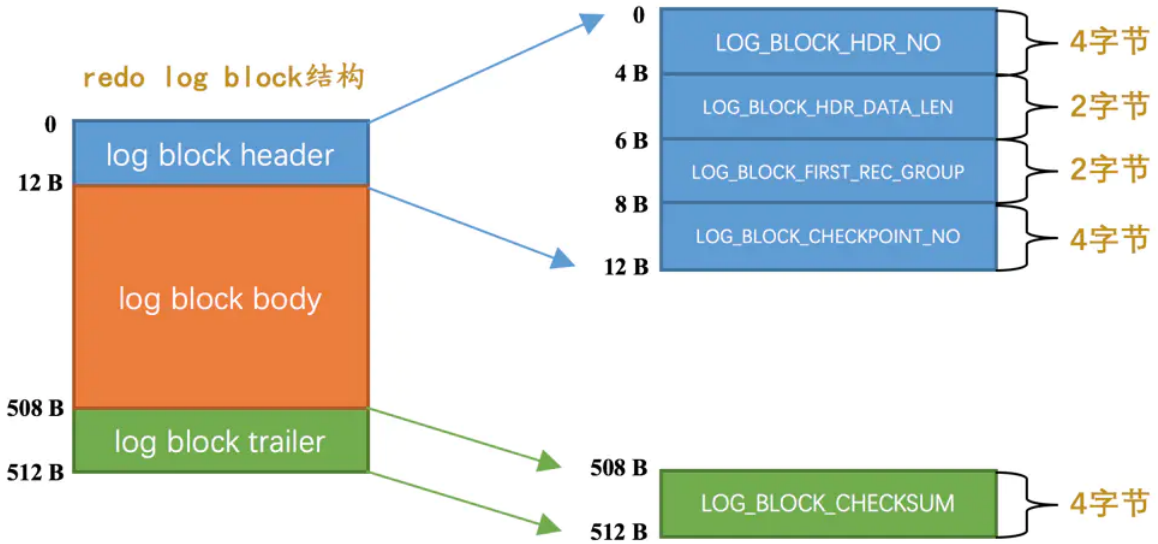
redo日志的写入过程

redo log block

设计InnoDB的大叔为了更好的进行系统崩溃恢复，他们把通过mtr生成的redo日志都放在了大小为512字节的页中。为了和我们前边提到的表空间中的页做区别，我们这里把用来存储redo日志的页称为block（你心里清楚页和block的意思其实差不多就行了）。一个redo log block的示意图如下：



真正的redo日志都是存储到占用496字节大小的log block body中，图中的log block header和log block trailer存储的是一些管理信息。我们来看看这些所谓的管理信息都是啥：



其中log block header的几个属性的意思分别如下：

- LOG_BLOCK_HDR_NO：每一个block都有一个大于0的唯一标号，本属性就表示该标号值。
- 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日志越来也多，本属性值也跟着增长。如果log block body已经被全部写满，那么本属性的值被设置为512。
- LOG_BLOCK_FIRST_REC_GROUP：一条redo日志也可以称之为一条redo日志记录（redo log record），一个mtr会生产多条redo日志记录，这些redo日志记录被称之为一个redo日志记录组（redo log record group）。LOG_BLOCK_FIRST_REC_GROUP就代表该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里第一个mtr生成的第一条redo日志的偏

移量）。

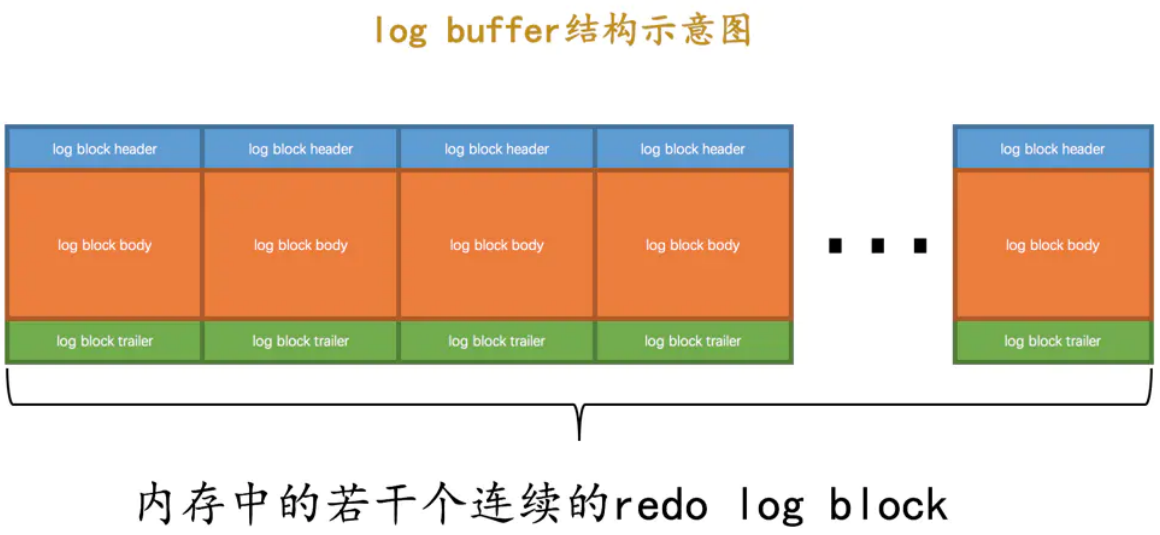
- LOG_BLOCK_CHECKPOINT_NO：表示所谓的checkpoint的序号，checkpoint是我们后续内容的重点，现在先不用清楚它的意思，稍安勿躁。

log block trailer中属性的意思如下：

- LOG_BLOCK_CHECKSUM：表示block的校验值，用于正确性校验，我们暂时不关心它。

redo日志缓冲区

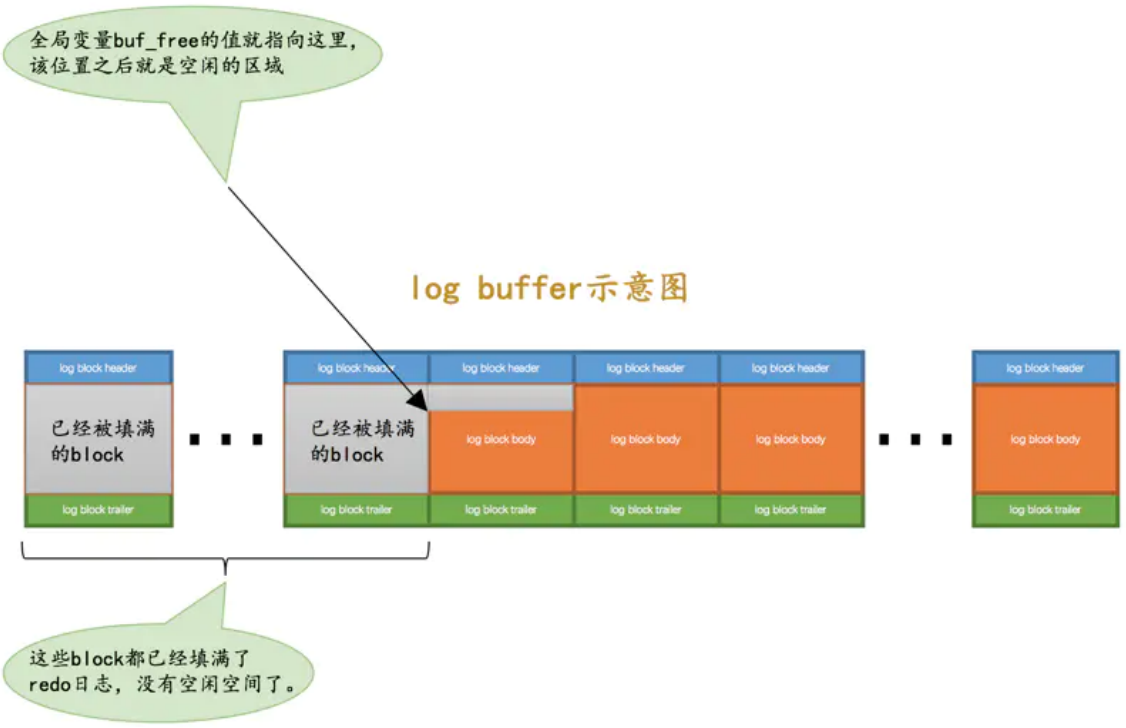
我们前边说过，设计InnoDB的大叔为了解决磁盘速度过慢的问题而引入了Buffer Pool。同理，写入redo日志时也不能直接直接写到磁盘上，实际上在服务器启动时就向操作系统申请了一大片称之为redo log buffer的连续内存空间，翻译成中文就是redo日志缓冲区，我们也可以简称为log buffer。这片内存空间被划分成若干个连续的redo log block，就像这样：



我们可以通过启动参数innodb_log_buffer_size来指定log buffer的大小，在MySQL 5.7.21这个版本中，该启动参数的默认值为16MB。

redo日志写入log buffer

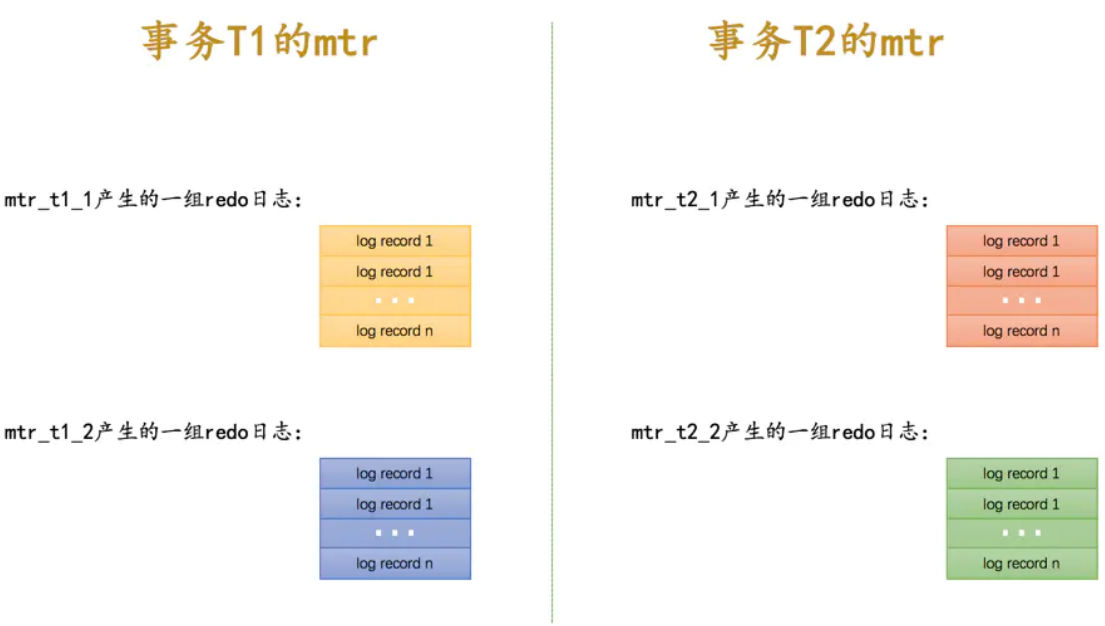
向log buffer中写入redo日志的过程是顺序的，也就是先往前边的block中写，当该block的空闲空间用完之后再往下一个block中写。当我们想往log buffer中写入redo日志时，第一个遇到的问题就是应该写在哪个block的哪个偏移量处，所以设计InnoDB的大叔特意提供了一个称之为buf_free的全局变量，该变量指明后续写入的redo日志应该写入到log buffer中的哪个位置，如图所示：



我们前边说过一个`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，所以其实并不是每生成一条`redo`日志，就将其插入到`log buffer`中，而是每个`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，将过程中产生的一组`redo`日志再全部复制到`log buffer`中。我们现在假设有两个名为`T1`、`T2`的事务，每个事务都包含2个`mtr`，我们给这几个`mtr`命名一下：

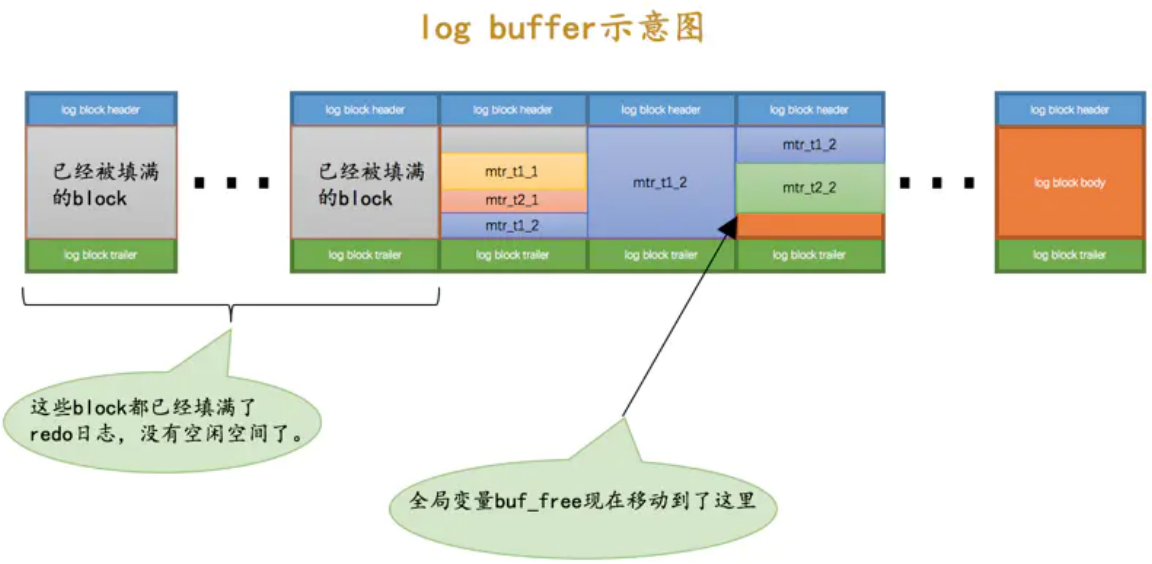
- 事务`T1`的两个`mtr`分别称为`mtr_T1_1`和`mtr_T1_2`。
- 事务`T2`的两个`mtr`分别称为`mtr_T2_1`和`mtr_T2_2`。

每个`mtr`都会产生一组`redo`日志，用示意图来描述一下这些`mtr`产生的日志情况：



不同的事务可能是并发执行的，所以`T1`、`T2`之间的`mtr`可能是交替执行的。每当一个`mtr`执行完成

时，伴随该mtr生成的一组redo日志就需要被复制到log buffer中，也就是说不同事务的mtr可能是交替写入log buffer的，我们画个示意图（为了美观，我们把一个mtr中产生的所有的redo日志当作一个整体来画）：



从示意图中我们可以看出来，不同的mtr产生的一组redo日志占用的存储空间可能不一样，有的mtr产生的redo日志量很少，比如mtr_t1_1、mtr_t2_1就被放到同一个block中存储，有的mtr产生的redo日志量非常大，比如mtr_t1_2产生的redo日志甚至占用了3个block来存储。

> 小贴士： 对照着上图，自己分析一下每个block的LOG_BLOCK_HDR_DATA_LEN、LOG_BLOCK_FIRST_REC_GROUP属性值都是什么哈～