第22章、undo日志（上）

标签： MySQL是怎样运行的

事务回滚的需求

我们说过事务需要保证原子性，也就是事务中的操作要么全部完成，要么什么也不做。但是偏偏有时候事务执行到一半会出现一些情况，比如：

- 情况一：事务执行过程中可能遇到各种错误，比如服务器本身的错误，操作系统错误，甚至是突然断电导致的错误。
- 情况二：程序员可以在事务执行过程中手动输入ROLLBACK语句结束当前的事务的执行。

这两种情况都会导致事务执行到一半就结束，但是事务执行过程中可能已经修改了很多东西，为了保证事务的原子性，我们需要把东西改回原先的样子，这个过程就称之为回滚（英文名：rollback），这样就可以造成一个假象：这个事务看起来什么都没做，所以符合原子性要求。

小时候我非常痴迷于象棋，总是想找厉害的大人下棋，赢棋是不可能赢棋的，这辈子都不可能赢棋的，又不想认输，只能偷偷的悔棋才能勉强玩的下去。悔棋就是一种非常典型的回滚操作，比如棋子往前走两步，悔棋对应的操作就是向后走两步；比如棋子往左走一步，悔棋对应的操作就是向右走一步。数据库中的回滚跟悔棋差不多，你插入了一条记录，回滚操作对应的就是把这条记录删除掉；你更新了一条记录，回滚操作对应的就是把该记录更新为旧值；你删除了一条记录，回滚操作对应的自然就是把该记录再插进去。说的貌似很简单的样子[手动偷笑����]。

从上边的描述中我们已经能隐约感觉到，每当我们要对一条记录做改动时（这里的改动可以指INSERT、DELETE、UPDATE），都需要留一手 —— 把回滚时所需的东西都给记下来。比方说：

- 你插入一条记录时，至少要把这条记录的主键值记下来，之后回滚的时候只需要把这个主键值对应的记录删掉就好了。
- 你删除了一条记录，至少要把这条记录中的内容都记下来，这样之后回滚时再把由这些内容组成的记录插入到表中就好了。
- 你修改了一条记录，至少要把修改这条记录前的旧值都记录下来，这样之后回滚时再把这条记录更新为旧值就好了。

设计数据库的大叔把这些为了回滚而记录的这些东东称之为撤销日志，英文名为undo log，我们也可以土洋结合，称之为undo日志。这里需要注意的一点是，由于查询操作（SELECT）并不会修改任何用户记录，所以在查询操作执行时，并不需要记录相应的undo日志。在真实的InnoDB中，undo日志其实并不像我们上边所说的那么简单，不同类型的操作产生的undo日志的格式也是不同的，不过先暂时把这些容易让人脑子糊的具体细节放一放，我们先回过头来看看事务id是个神马玩意儿。

事务id

给事务分配id的时机

我们前边在唠叨事务简介时说过，一个事务可以是一个只读事务，或者是一个读写事务：

- 我们可以通过`START TRANSACTION READ ONLY`语句开启一个只读事务。

  在只读事务中不可以对普通的表（其他事务也能访问到的表）进行增、删、改操作，但可以对临时表做增、删、改操作。

- 我们可以通过`START TRANSACTION READ WRITE`语句开启一个读写事务，或者使用`BEGIN`、`START TRANSACTION`语句开启的事务默认也算是读写事务。

  在读写事务中可以对表执行增删改查操作。

如果某个事务执行过程中对某个表执行了增、删、改操作，那么`InnoDB`存储引擎就会给它分配一个独一无二的`事务id`，分配方式如下：

- 对于只读事务来说，只有在它第一次对某个用户创建的临时表执行增、删、改操作时才会为这个事务分配一个`事务id`，否则的话是不分配`事务id`的。

  > 小贴士： 我们前边说过对某个查询语句执行EXPLAIN分析它的查询计划时，有时候在Extra列会看到Using temporary的提示，这个表明在执行该查询语句时会用到内部临时表。这个所谓的内部临时表和我们手动用CREATE TEMPORARY TABLE创建的用户临时表并不一样，在事务回滚时并不需要把执行SELECT语句过程中用到的内部临时表也回滚，在执行SELECT语句用到内部临时表时并不会为它分配事务id。

- 对于读写事务来说，只有在它第一次对某个表（包括用户创建的临时表）执行增、删、改操作时才会为这个事务分配一个`事务id`，否则的话也是不分配`事务id`的。

  有的时候虽然我们开启了一个读写事务，但是在这个事务中全是查询语句，并没有执行增、删、改的语句，那也就意味着这个事务并不会被分配一个`事务id`。

说了半天，`事务id`有啥子用？这个先保密哈，后边会一步步的详细唠叨。现在只要知道只有在事务对表中的记录做改动时才会为这个事务分配一个唯一的`事务id`。

> 小贴士： 上边描述的事务id分配策略是针对MySQL 5.7来说的，前边的版本的分配方式可能不同～

事务id是怎么生成的

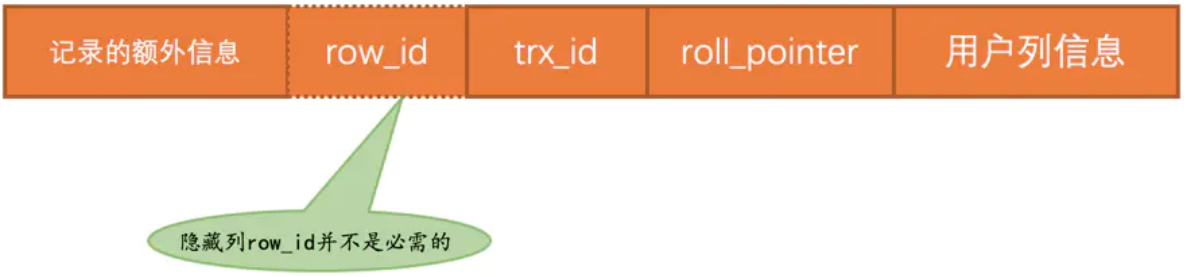
这个`事务id`本质上就是一个数字，它的分配策略和我们前边提到的对隐藏列`row_id`（当用户没有为表创建主键和`UNIQUE`键时`InnoDB`自动创建的列）的分配策略大抵相同，具体策略如下：

- 服务器会在内存中维护一个全局变量，每当需要为某个事务分配一个`事务id`时，就会把该变量的值当作`事务id`分配给该事务，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`5`的页面中一个称之为`Max Trx ID`的属性处，这个属性占用`8`个字节的存储空间。
- 当系统下一次重新启动时，会将上边提到的`Max Trx 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 Trx ID`属性值）。

这样就可以保证整个系统中分配的`事务id`值是一个递增的数字。先被分配`id`的事务得到的是较小的`事务id`，后被分配`id`的事务得到的是较大的`事务id`。

trx_id隐藏列

我们前边唠叨`InnoDB`记录行格式的时候重点强调过：聚簇索引的记录除了会保存完整的用户数据以外，而且还会自动添加名为trx_id、roll_pointer的隐藏列，如果用户没有在表中定义主键以及UNIQUE键，还会自动添加一个名为row_id的隐藏列。所以一条记录在页面中的真实结构看起来就是这样的：



其中的trx_id列其实还蛮好理解的，就是某个对这个聚簇索引记录做改动的语句所在的事务对应的事务id而已（此处的改动可以是INSERT、DELETE、UPDATE操作）。至于roll_pointer隐藏列我们后边分析~

undo日志的格式

为了实现事务的原子性，InnoDB存储引擎在实际进行增、删、改一条记录时，都需要先把对应的undo日志记下来。一般每对一条记录做一次改动，就对应着一条undo日志，但在某些更新记录的操作中，也可能会对应着2条undo日志，这个我们后边会仔细唠叨。一个事务在执行过程中可能新增、删除、更新若干条记录，也就是说需要记录很多条对应的undo日志，这些undo日志会被从0开始编号，也就是说根据生成的顺序分别被称为第0号undo日志、第1号undo日志、...、第n号undo日志等，这个编号也被称之为undo no。

这些undo日志是被记录到类型为FIL_PAGE_UNDO_LOG（对应的十六进制是0x0002，忘记了页面类型是个啥的同学需要回过头再看看前边的章节）的页面中。这些页面可以从系统表空间中分配，也可以从一种专门存放undo日志的表空间，也就是所谓的undo tablespace中分配。不过关于如何分配存储undo日志的页面这个事情我们稍后再说，现在先来看看不同操作都会产生什么样子的undo日志吧~ 为了故事的顺利发展，我们先来创建一个名为undo_demo的表：

```
CREATE TABLE undo_demo (
    id INT NOT NULL,
    key1 VARCHAR(100),
    col VARCHAR(100),
    PRIMARY KEY (id),
    KEY idx_key1 (key1)
)Engine=InnoDB CHARSET=utf8;
```

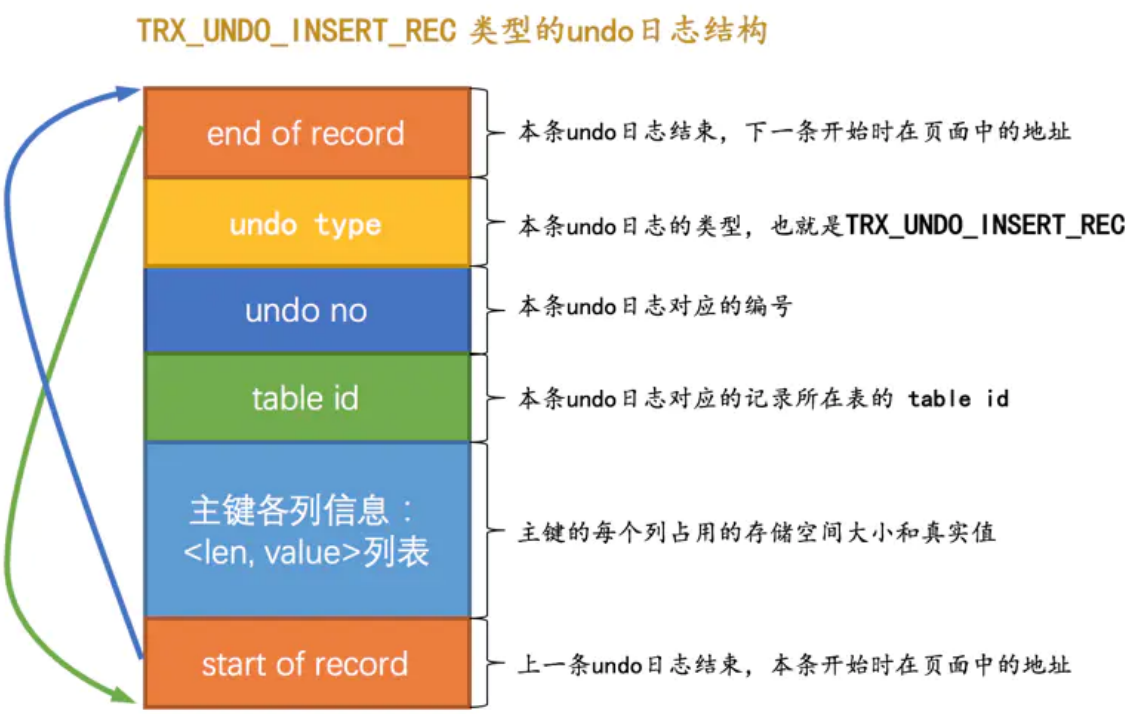
这个表中有3个列，其中id列是主键，我们为key1列建立了一个二级索引，col列是一个普通的列。我们前边介绍InnoDB的数据字典时说过，每个表都会被分配一个唯一的table id，我们可以通过系统数据库information_schema中的innodb_sys_tables表来查看某个表对应的table id是什么，现在我们查看一下undo_demo对应的table id是多少：

```
mysql> SELECT * FROM information_schema.innodb_sys_tables WHERE name = 'xiaohaizi/undo_demo';
+----------+---------------------+------+--------+-------+-------------+------------+---------------+------------+
| TABLE_ID | NAME                | FLAG | N_COLS | SPACE | FILE_FORMAT | ROW_FORMAT | ZIP_PAGE_SIZE | SPACE_TYPE |
+----------+---------------------+------+--------+-------+-------------+------------+---------------+------------+
|      138 | xiaohaizi/undo_demo |   33 |      6 |   482 | Barracuda   | Dynamic    |             0 | Single     |
+----------+---------------------+------+--------+-------+-------------+------------+---------------+------------+
1 row in set (0.01 sec)
```

从查询结果可以看出，undo_demo表对应的table id为138，先把这个值记住，我们后边有用。

INSERT操作对应的undo日志

我们前边说过，当我们向表中插入一条记录时会有乐观插入和悲观插入的区分，但是不管怎么插入，最终导致的结果就是这条记录被放到了一个数据页中。如果希望回滚这个插入操作，那么把这条记录删除就好了，也就是说在写对应的undo日志时，主要是把这条记录的主键信息记上。所以设计InnoDB的大叔设计了一个类型为TRX_UNDO_INSERT_REC的undo日志，它的完整结构如下图所示：



根据示意图我们强调几点：

- undo no在一个事务中是从0开始递增的，也就是说只要事务没提交，每生成一条undo日志，那么该条日志的undo no就增1。
- 如果记录中的主键只包含一个列，那么在类型为TRX_UNDO_INSERT_REC的undo日志中只需要把该列占用的存储空间大小和真实值记录下来，如果记录中的主键包含多个列，那么每个列占用的存储空间大小和对应的真实值都需要记录下来（图中的len就代表列占用的存储空间大小，value就代表列的真实值）。

小贴士： 当我们向某个表中插入一条记录时，实际上需要向聚簇索引和所有的二级索引都插入一条记录。不过记录undo日志时，我们只需要考虑向聚簇索引插入记录时的情况就好了，因为其实聚簇索引记录和二级索引记录是一一对应的，我们在回滚插入操作时，只需要知道这条记录的主键信息，然后根据主键信息做对应的删除操作，做删除操作时就会顺带着把所有二级索引中相应的记录也删除掉。后边说到的DELETE操作和UPDATE操作对应的undo日志也都是针对聚簇索引记录而言的，我们之后就不强调了。

现在我们向undo_demo中插入两条记录：

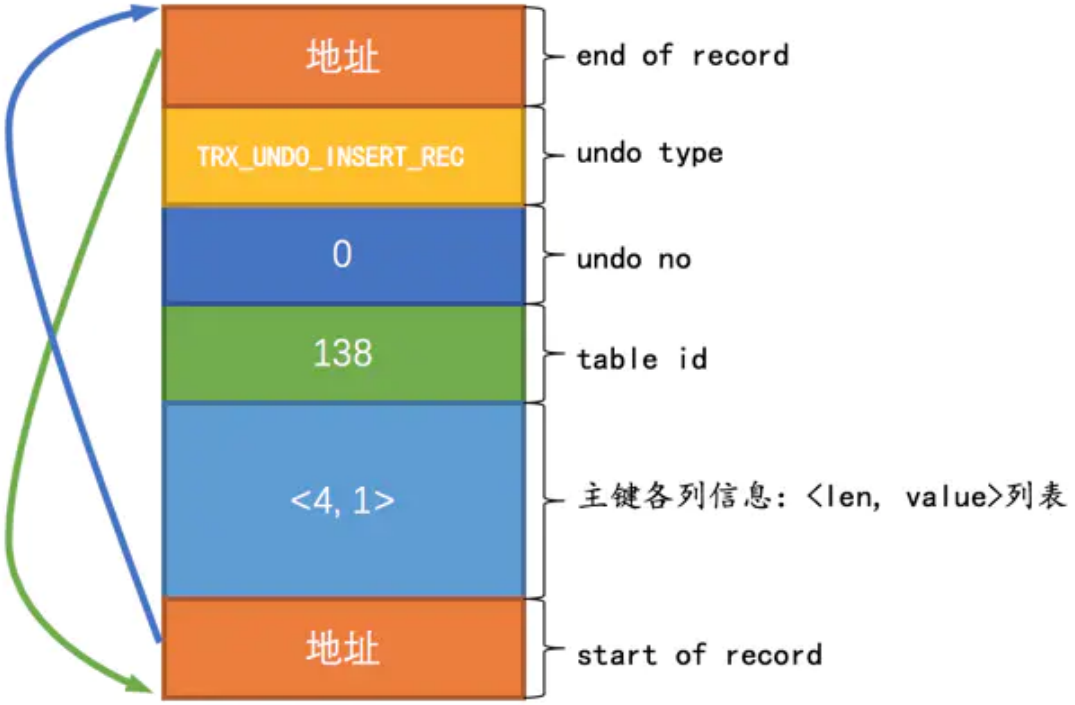
```
BEGIN;  # 显式开启一个事务，假设该事务的id为100

# 插入两条记录
INSERT INTO undo_demo(id, key1, col)
    VALUES (1, 'AWM', '狙击枪'), (2, 'M416', '步枪');
```

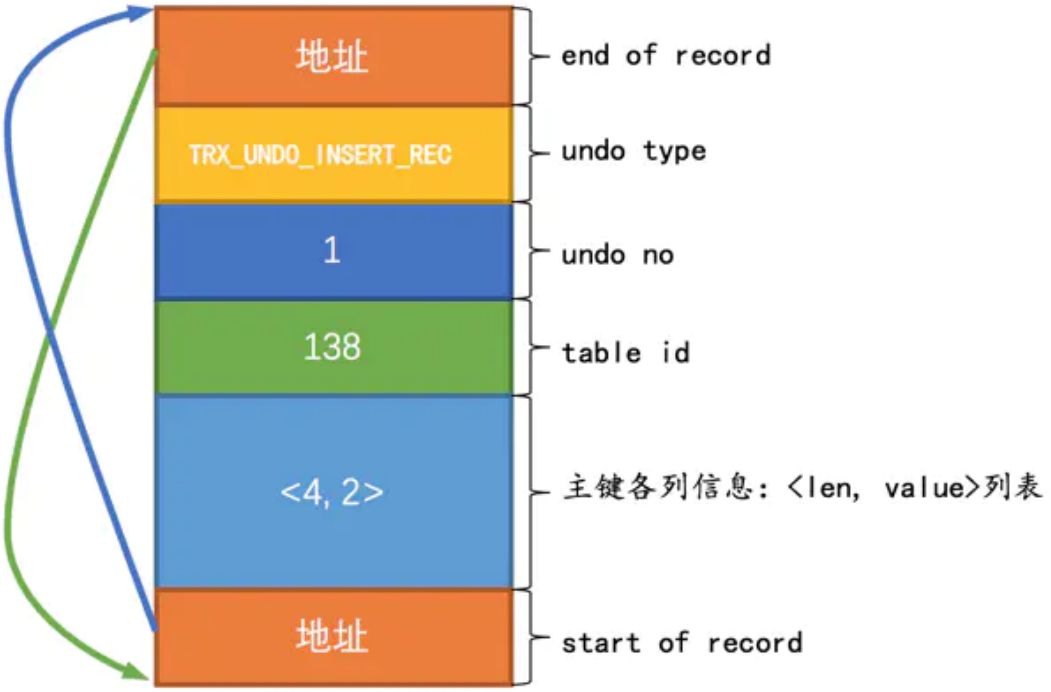
因为记录的主键只包含一个id列，所以我们在对应的undo日志中只需要将待插入记录的id列占用的存储空间长度（id列的类型为INT，INT类型占用的存储空间长度为4个字节）和真实值记录下来。本例中插入了两条记录，所

以会产生两条类型为`TRX_UNDO_INSERT_REC`的`undo日志`:

- 第一条`undo日志`的`undo no`为`0`，记录主键占用的存储空间长度为`4`，真实值为`1`。画一个示意图就是这样：



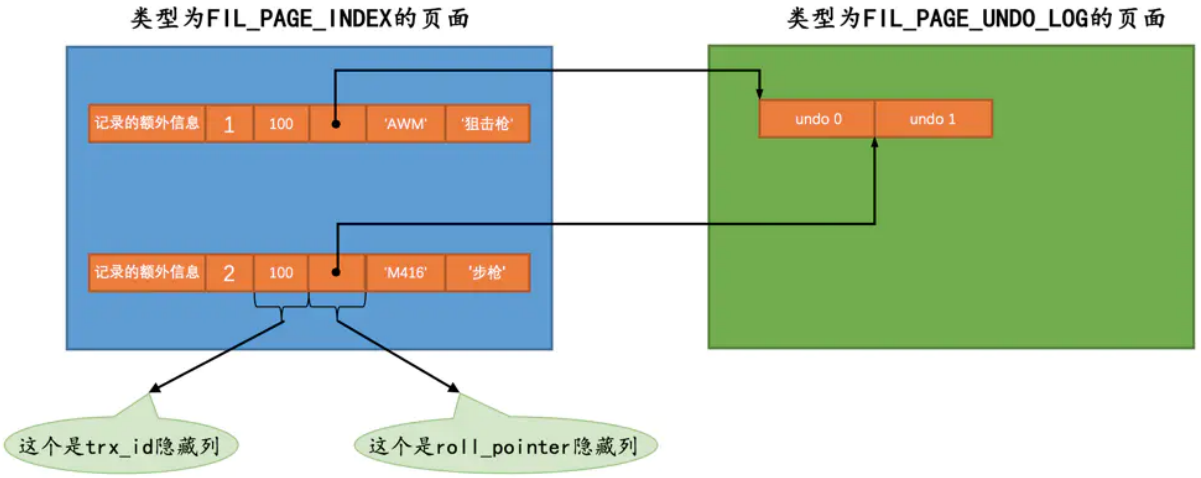
- 第二条`undo日志`的`undo no`为`1`，记录主键占用的存储空间长度为`4`，真实值为`2`。画一个示意图就是这样（与第一条`undo日志`对比，`undo no`和主键各列信息有不同）：



小贴士： 为了最大限度的节省undo日志占用的存储空间，和我们前边说过的redo日志类似，设计InnoDB的大叔会给undo日志中的某些属性进行压缩处理，具体的压缩细节我们就不唠叨了。

roll_pointer隐藏列的含义

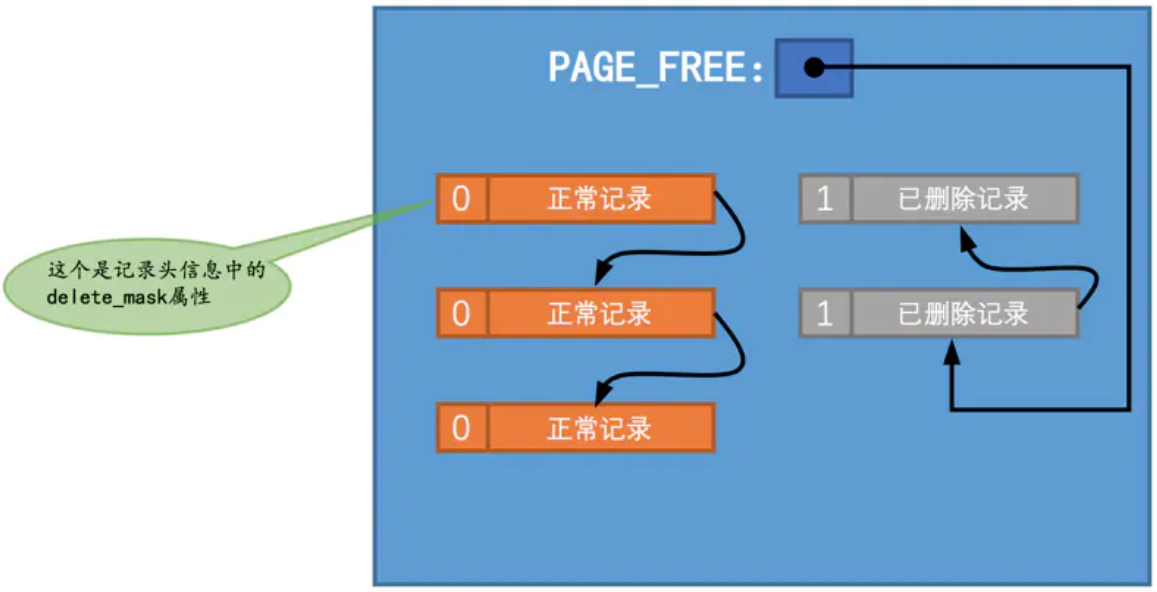
是时候揭开roll_pointer的真实面纱了，这个占用7个字节的字段其实一点都不神秘，本质上就是一个指向记录对应的undo日志的一个指针。比方说我们上边向undo_demo表里插入了2条记录，每条记录都有与其对应的一条undo日志。记录被存储到了类型为FIL_PAGE_INDEX的页面中（就是我们前边一直所说的数据页），undo日志被存放到了类型为FIL_PAGE_UNDO_LOG的页面中。效果如图所示：



从图中也可以更直观的看出来，roll_pointer本质就是一个指针，指向记录对应的undo日志。不过这7个字节的roll_pointer的每一个字节具体的含义我们后边唠叨完如何分配存储undo日志的页面之后再具体说哈～

DELETE操作对应的undo日志

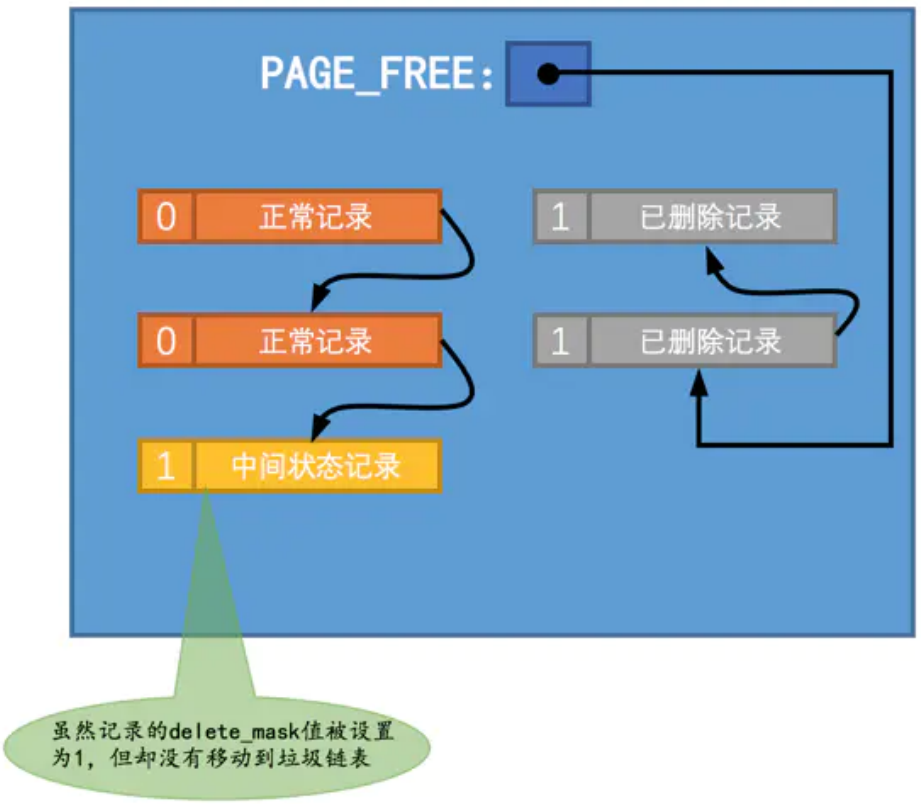
我们知道插入到页面中的记录会根据记录头信息中的next_record属性组成一个单向链表，我们把这个链表称之为正常记录链表；我们在前边唠叨数据页结构的时候说过，被删除的记录其实也会根据记录头信息中的next_record属性组成一个链表，只不过这个链表中的记录占用的存储空间可以被重新利用，所以也称这个链表为垃圾链表。Page Header部分有一个称之为PAGE_FREE的属性，它指向由被删除记录组成的垃圾链表中的头节点。为了故事的顺利发展，我们先画一个图，假设此刻某个页面中的记录分布情况是这样的（这个不是undo_demo表中的记录，只是我们随便举的一个例子）：



为了突出主题，在这个简化版的示意图中，我们只把记录的delete_mask标志位展示了出来。从图中可以看出，正常记录链表中包含了3条正常记录，垃圾链表里包含了2条已删除记录，在垃圾链表中的这些记录占用的存储空间可以被重新利用。页面的Page Header部分的PAGE_FREE属性的值代表指向垃圾链表头节点的指针。假设现在我们准备使用DELETE语句把正常记录链表中的最后一条记录给删除掉，其实这个删除的过程需要经历两个阶段：

- 阶段一：仅仅将记录的delete_mask标识位设置为1，其他的不做修改（其实会修改记录的trx_id、roll_pointer这些隐藏列的值）。设计InnoDB的大叔把这个阶段称之为delete mark。

  把这个过程画下来就是这样：

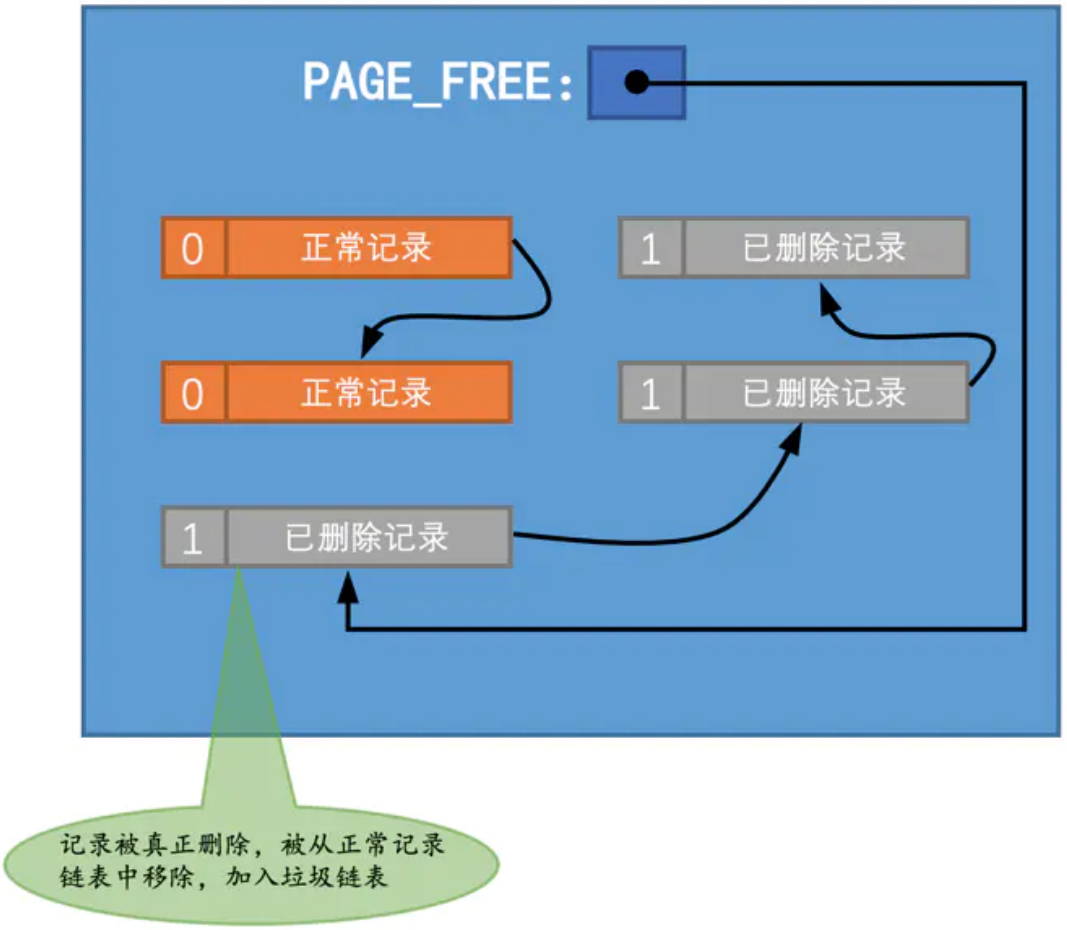


可以看到，正常记录链表中的最后一条记录的delete_mask值被设置为1，但是并没有被加入到垃圾链表。也就是此时记录处于一个中间状态，跟猪八戒照镜子——里外不是人似的。在删除语句所在的事务提交之前，被删除的记录一直都处于这种所谓的中间状态。

> 小贴士： 为啥会有这种奇怪的中间状态呢？其实主要是为了实现一个称之为MVCC的功能，哈哈，稍后再介绍。

- 阶段二：当该删除语句所在的事务提交之后，会有专门的线程后来真正的把记录删除掉。所谓真正的删除就是把该记录从正常记录链表中移除，并且加入到垃圾链表中，然后还要调整一些页面的其他信息，比如页面中的用户记录数量PAGE_N_RECS、上次插入记录的位置PAGE_LAST_INSERT、垃圾链表头节点的指针PAGE_FREE、页面中可重用的字节数量PAGE_GARBAGE、还有页目录的一些信息等等。设计InnoDB的大叔把这个阶段称之为purge。

  把阶段二执行完了，这条记录就算是真正的被删除掉了。这条已删除记录占用的存储空间也可以被重新利用了。画下来就是这样：

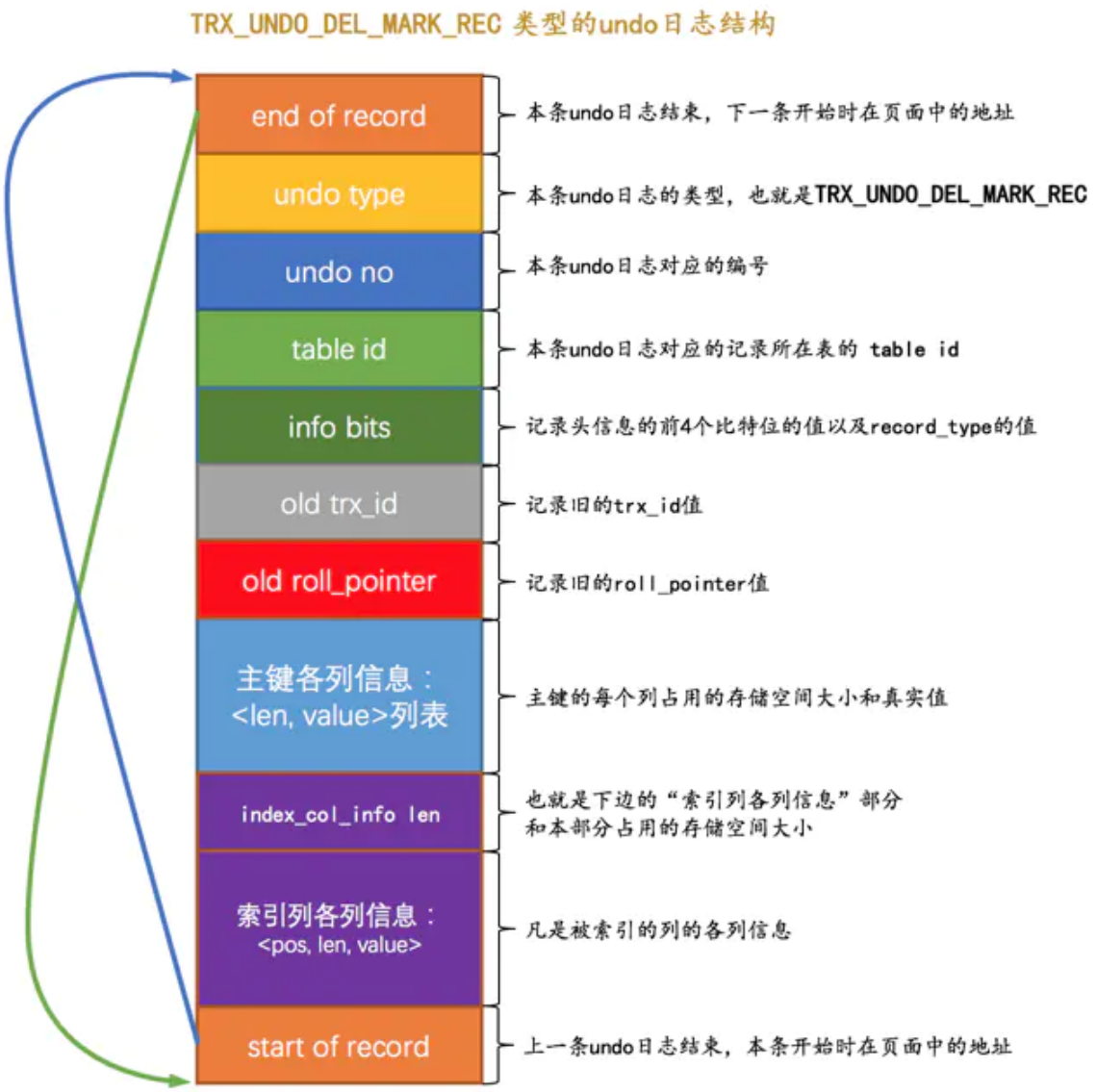


对照着图我们还要注意一点，将被删除记录加入到垃圾链表时，实际上加入到链表的头节点处，会跟着修改PAGE_FREE属性的值。

> 小贴士： 页面的Page Header部分有一个PAGE_GARBAGE属性，该属性记录着当前页面中可重用存储空间占用的总字节数。每当有已删除记录被加入到垃圾链表后，都会把这个PAGE_GARBAGE属性的值加上该已删除记录占用的存储空间大小。PAGE_FREE指向垃圾链表的头节点，之后每当新插入记录时，首先判断PAGE_FREE指向的头节点代表的已删除记录占用的存储空间是否足够容纳这条新插入的记录，如果不可以容纳，就直接向页面中申请新的空间来存储这条记录（是的，你没看错，并不会尝试遍历整个垃圾链表，找到一个可以容纳新记录的节点）。如果可以容纳，那么直接重用这条已删除记录的存储空间，并且把PAGE_FREE指向垃圾链表中的下一条已删除记录。但是这里有一个问题，如果新插入的那条记录占用的存储

空间大小小于垃圾链表的头节点占用的存储空间大小，那就意味头节点对应的记录占用的存储空间里有一部分空间用不到，这部分空间就被称之为碎片空间。那这些碎片空间岂不是永远都用不到了么？其实也不是，这些碎片空间占用的存储空间大小会被统计到PAGE_GARBAGE属性中，这些碎片空间在整个页面快使用完前并不会被重新利用，不过当页面快满时，如果再插入一条记录，此时页面中并不能分配一条完整记录的空间，这时候会首先看一看PAGE_GARBAGE的空间和剩余可利用的空间加起来是不是可以容纳下这条记录，如果可以的话，InnoDB会尝试重新组织页内的记录，重新组织的过程就是先开辟一个临时页面，把页面内的记录依次插入一遍，因为依次插入时并不会产生碎片，之后再把临时页面的内容复制到本页面，这样就可以把那些碎片空间都解放出来（很显然重新组织页面内的记录比较耗费性能）。

从上边的描述中我们也可以看出来，在删除语句所在的事务提交之前，只会经历阶段一，也就是delete mark阶段（提交之后我们就不用回滚了，所以只需考虑对删除操作的阶段一做的影响进行回滚）。设计InnoDB的大叔为此设计了一种称之为TRX_UNDO_DEL_MARK_REC类型的undo日志，它的完整结构如下图所示：



额滴个神呐，这个里边的属性也太多了点儿吧～（其实大部分属性的意思我们上边已经介绍过了）是的，的确有点多，不过大家千万不要在意，如果记不住千万不要勉强自己，我这里把它们都列出来让大家混个脸熟而已。劳烦大家先克服一下密集恐急症，再抬头大致看一遍上边的这个类型为TRX_UNDO_DEL_MARK_REC的undo日志中的属性，特别注意一下这几点：

- 在对一条记录进行delete mark操作前，需要把该记录的旧的trx_id和roll_pointer隐藏列的值都给记到对应的undo日志中来，就是我们图中显示的old trx_id和old roll_pointer属性。这样有一个好处，那就是可以

通过undo日志的old roll_pointer找到记录在修改之前对应的undo日志。比方说在一个事务中，我们先插入了一条记录，然后又执行对该记录的删除操作，这个过程的示意图就是这样：

从图中可以看出来，执行完delete mark操作后，它对应的undo日志和INSERT操作对应的undo日志就串成了一个链表。这个很有意思啊，这个链表就称之为版本链，现在貌似看不出这个版本链有啥用，等我们再往后看看，讲完UPDATE操作对应的undo日志后，这个所谓的版本链就慢慢的展现出它的牛逼之处了。

- 与类型为TRX_UNDO_INSERT_REC的undo日志不同，类型为TRX_UNDO_DEL_MARK_REC的undo日志还多了一个索引列各列信息的内容，也就是说如果某个列被包含在某个索引中，那么它的相关信息就应该被记录到这个索引列各列信息部分，所谓的相关信息包括该列在记录中的位置（用pos表示），该列占用的存储空间大小（用len表示），该列实际值（用value表示）。所以索引列各列信息存储的内容实质上就是<pos, len, value>的一个列表。这部分信息主要是用在事务提交后，对该中间状态记录做真正删除的阶段二，也就是purge阶段中使用的，具体如何使用现在我们可以忽略~

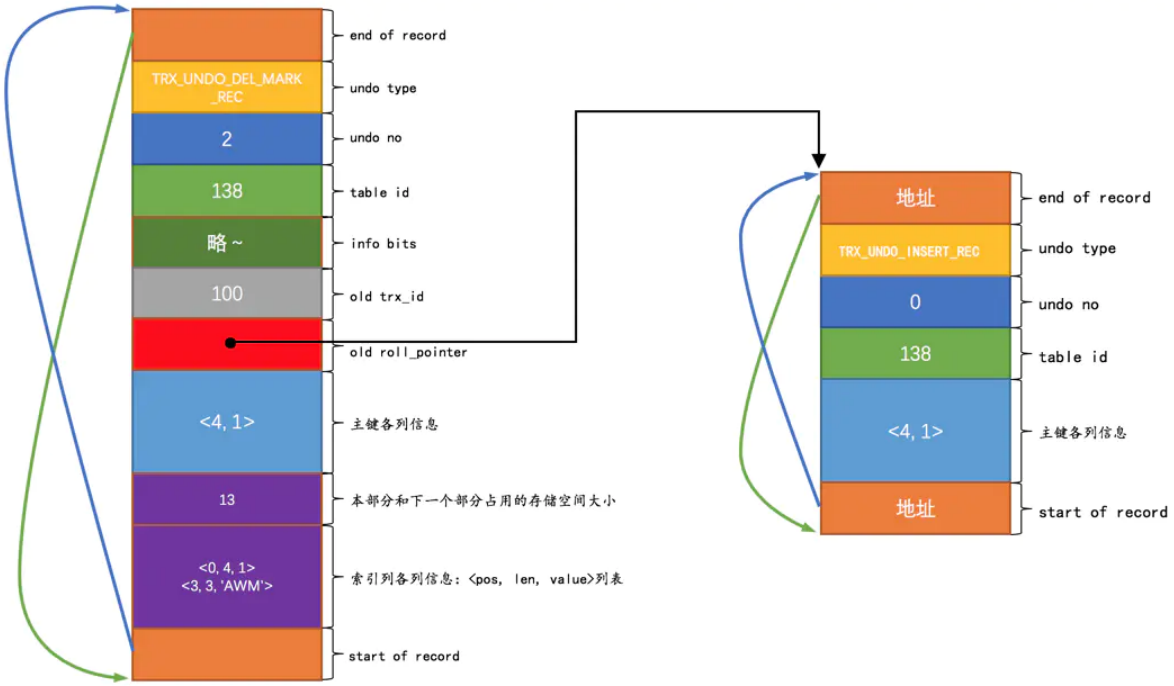
该介绍的我们介绍完了，现在继续在上边那个事务id为100的事务中删除一条记录，比如我们把id为1的那条记录删除掉：

```
BEGIN;  # 显式开启一个事务，假设该事务的id为100

# 插入两条记录
INSERT INTO undo_demo(id, key1, col)
    VALUES (1, 'AWM', '狙击枪'), (2, 'M416', '步枪');

# 删除一条记录
DELETE FROM undo_demo WHERE id = 1;
```

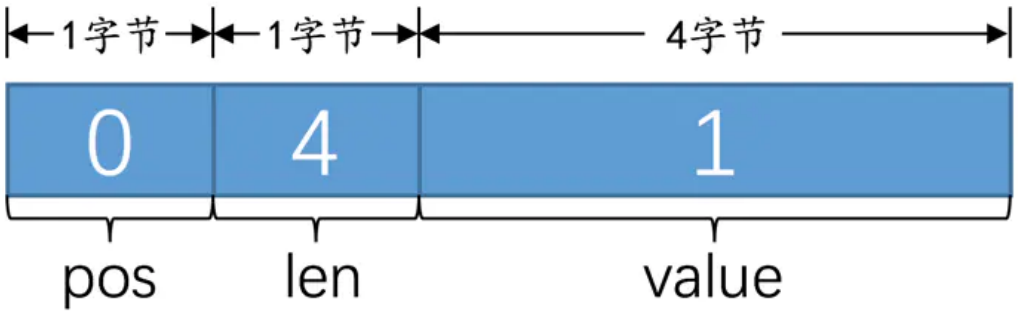
这个delete mark操作对应的undo日志的结构就是这样：



对照着这个图，我们得注意下边几点：

- 因为这条`undo`日志是`id`为`100`的事务中产生的第3条`undo`日志，所以它对应的`undo no`就是`2`。
- 在对记录做`delete mark`操作时，记录的`trx_id`隐藏列的值是`100`（也就是说对该记录最近的一次修改就发生在本事务中），所以把`100`填入`old trx_id`属性中。然后把记录的`roll_pointer`隐藏列的值取出来，填入`old roll_pointer`属性中，这样就可以通过`old roll_pointer`属性值找到最近一次对该记录做改动时产生的`undo`日志。
- 由于`undo_demo`表中有2个索引：一个是聚簇索引，一个是二级索引`idx_key1`。只要是包含在索引中的列，那么这个列在记录中的位置（`pos`），占用存储空间大小（`len`）和实际值（`value`）就需要存储到`undo`日志中。
  - 对于主键来说，只包含一个`id`列，存储到`undo`日志中的相关信息分别是：
    - `pos`：`id`列是主键，也就是在记录的第一个列，它对应的`pos`值为`0`。`pos`占用1个字节来存储。
    - `len`：`id`列的类型为`INT`，占用4个字节，所以`len`的值为`4`。`len`占用1个字节来存储。
    - `value`：在被删除的记录中`id`列的值为`1`，也就是`value`的值为`1`。`value`占用4个字节来存储。

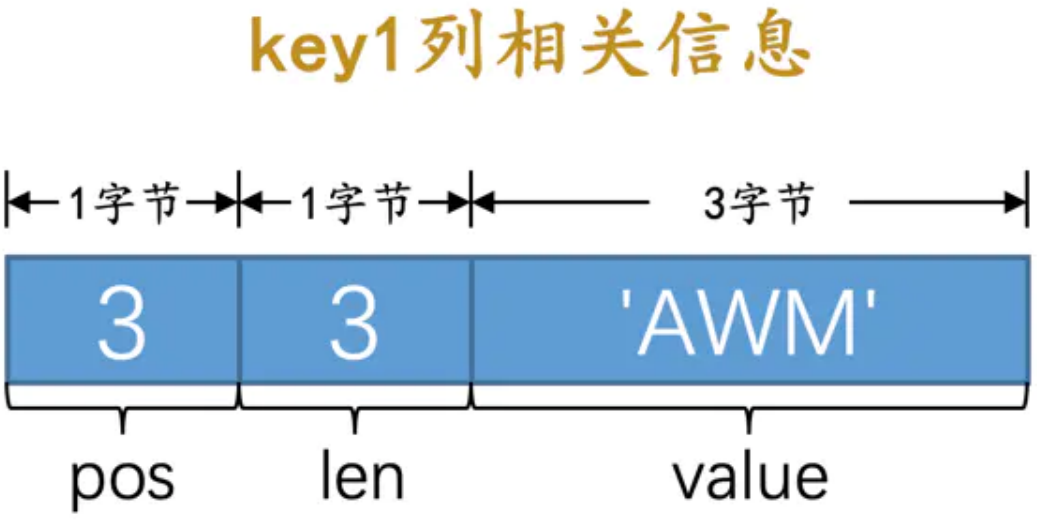
    画一个图演示一下就是这样：



所以对于`id`列来说，最终存储的结果就是`<0, 4, 1>`，存储这些信息占用的存储空间大小为`1 + 1 + 4 = 6`个字节。

- 对于`idx_key1`来说，只包含一个`key1`列，存储到`undo日志`中的相关信息分别是：
  - `pos`：`key1`列是排在`id`列、`trx_id`列、`roll_pointer`列之后的，它对应的`pos`值为`3`。`pos`占用1个字节来存储。
  - `len`：`key1`列的类型为`VARCHAR(100)`，使用`utf8`字符集，被删除的记录实际存储的内容是`AWM`，所以一共占用3个字节，也就是所以`len`的值为`3`。`len`占用1个字节来存储。
  - `value`：在被删除的记录中`key1`列的值为`AWM`，也就是`value`的值为`AWM`。`value`占用3个字节来存储。

  画一个图演示一下就是这样：



所以对于`key1`列来说，最终存储的结果就是`<3, 3, 'AWM'>`，存储这些信息占用的存储空间大小为`1 + 1 + 3 = 5`个字节。

从上边的叙述中可以看到，`<0, 4, 1>`和`<3, 3, 'AWM'>`共占用`11`个字节。然后`index_col_info len`本身占用`2`个字节，所以加起来一共占用`13`个字节，把数字`13`就填到了`index_col_info len`的属性中。

UPDATE操作对应的undo日志

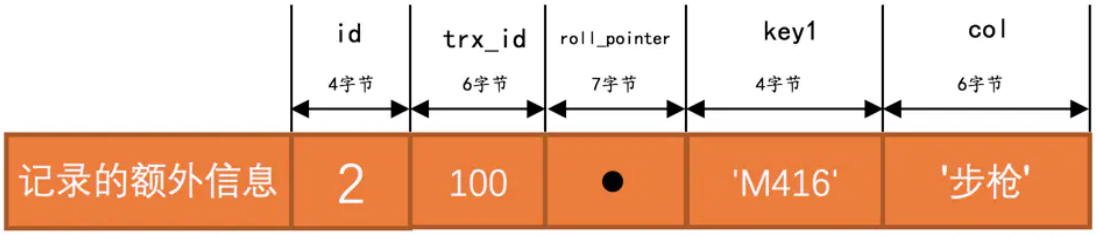
在执行UPDATE语句时，InnoDB对更新主键和不更新主键这两种情况有截然不同的处理方案。

不更新主键的情况

在不更新主键的情况下，又可以细分为被更新的列占用的存储空间不发生变化和发生变化的情况。

- 就地更新（in-place update）

  更新记录时，对于被更新的每个列来说，如果更新后的列和更新前的列占用的存储空间都一样大，那么就可以进行就地更新，也就是直接在原记录的基础上修改对应列的值。再次强调一边，是每个列在更新前后占用的存储空间一样大，有任何一个被更新的列更新前比更新后占用的存储空间大，或者更新前比更新后占用的存储空间小都不能进行就地更新。比方说现在undo_demo表里还有一条id值为2的记录，它的各个列占用的大小如图所示（因为采用utf8字符集，所以'步枪'这两个字符占用6个字节）：



  假如我们有这样的UPDATE语句：

```
UPDATE undo_demo
    SET key1 = 'P92', col = '手枪'
    WHERE id = 2;
```

  在这个UPDATE语句中，col列从步枪被更新为手枪，前后都占用6个字节，也就是占用的存储空间大小未改变；key1列从M416被更新为P92，也就是从4个字节被更新为3个字节，这就不满足就地更新需要的条件了，所以不能进行就地更新。但是如果UPDATE语句长这样：

```
UPDATE undo_demo
    SET key1 = 'M249', col = '机枪'
    WHERE id = 2;
```

  由于各个被更新的列在更新前后占用的存储空间是一样大的，所以这样的语句可以执行就地更新。

- 先删除掉旧记录，再插入新记录

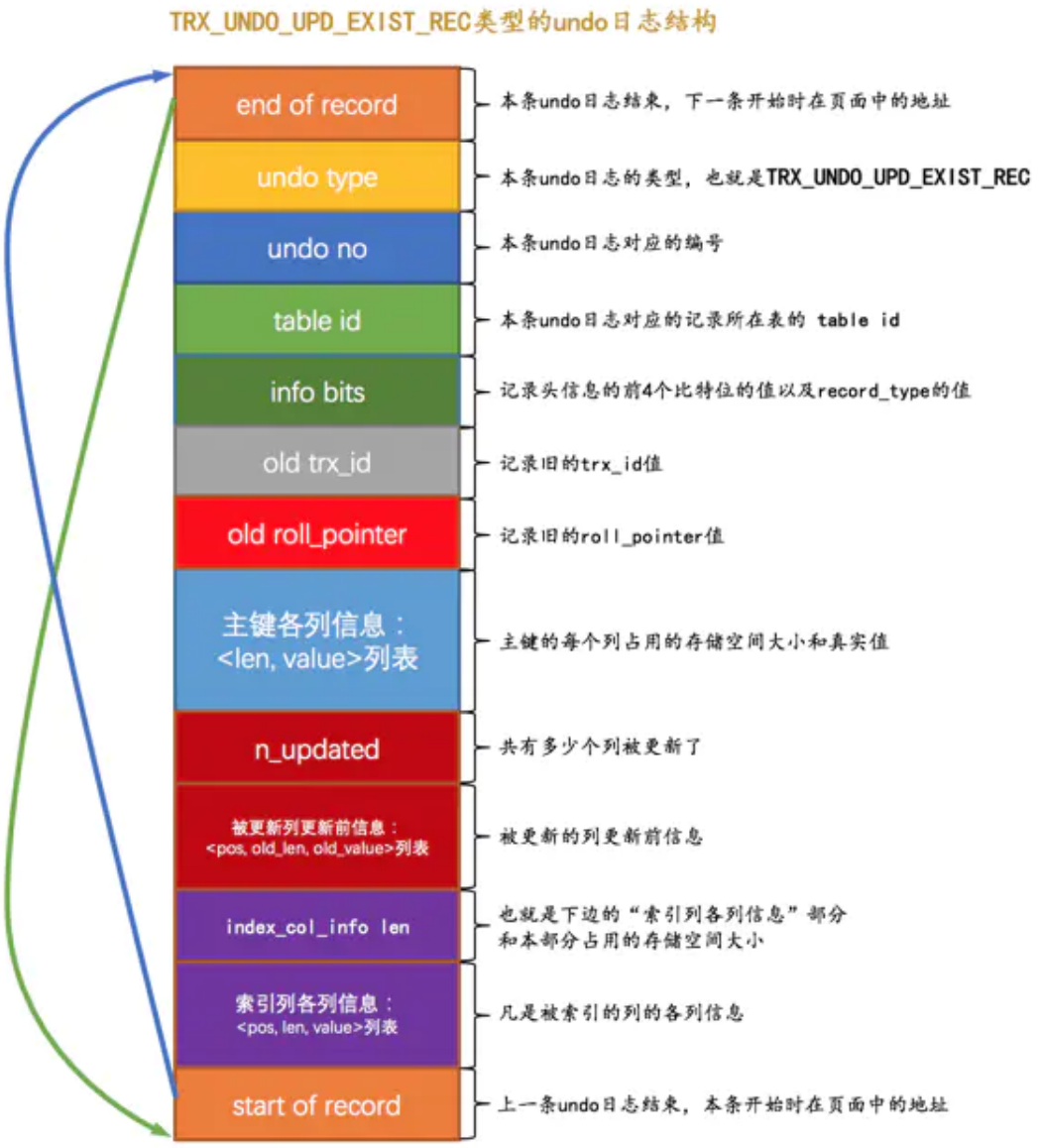
  在不更新主键的情况下，如果有任何一个被更新的列更新前和更新后占用的存储空间大小不一致，那么就需要先把这条旧的记录从聚簇索引页面中删除掉，然后再根据更新后列的值创建一条新的记录插入到页面中。

  请注意一下，我们这里所说的删除并不是delete mark操作，而是真正的删除掉，也就是把这条记录从正常记录链表中移除并加入到垃圾链表中，并且修改页面中相应的统计信息（比如PAGE_FREE、PAGE_GARBAGE等这些信息）。不过这里做真正删除操作的线程并不是在唠叨DELETE语句中做purge操作时使用的另外专门的线程，而是由用户线程同步执行真正的删除操作，真正删除之后紧接着就要根据各个列更新后的值创建的新记录插入。

  这里如果新创建的记录占用的存储空间大小不超过旧记录占用的空间，那么可以直接重用被加入到垃圾链表

中的旧记录所占用的存储空间，否则的话需要在页面中新申请一段空间以供新记录使用，如果本页面内已经没有可用的空间的话，那就需要进行页面分裂操作，然后再插入新记录。

针对UPDATE不更新主键的情况（包括上边所说的就地更新和先删除旧记录再插入新记录），设计InnoDB的大叔们设计了一种类型为TRX_UNDO_UPD_EXIST_REC的undo日志，它的完整结构如下：



其实大部分属性和我们介绍过的TRX_UNDO_DEL_MARK_REC类型的undo日志是类似的，不过还是要注意这么几点：

- n_updated属性表示本条UPDATE语句执行后将有几个列被更新，后边跟着的<pos, old_len, old_value>分别表示被更新列在记录中的位置、更新前该列占用的存储空间大小、更新前该列的真实值。
- 如果在UPDATE语句中更新的列包含索引列，那么也会添加索引列各列信息这个部分，否则的话是不会添加这个部分的。

现在继续在上边那个事务id为100的事务中更新一条记录，比如我们把id为2的那条记录更新一下：

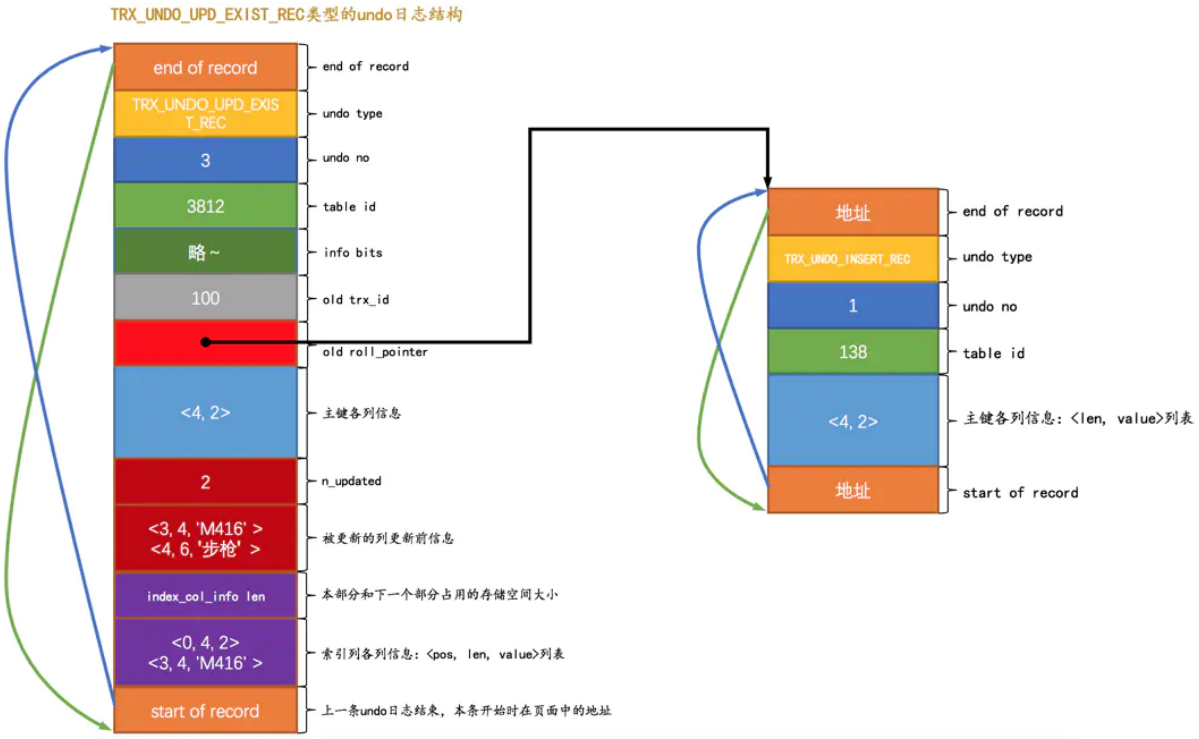
```
BEGIN;  # 显式开启一个事务，假设该事务的id为100

# 插入两条记录
INSERT INTO undo_demo(id, key1, col)
    VALUES (1, 'AWM', '狙击枪'), (2, 'M416', '步枪');
```

```
# 删除一条记录
DELETE FROM undo_demo WHERE id = 1;

# 更新一条记录
UPDATE undo_demo
    SET key1 = 'M249', col = '机枪'
    WHERE id = 2;
```

这个UPDATE语句更新的列大小都没有改动，所以可以采用就地更新的方式来执行，在真正改动页面记录时，会先记录一条类型为TRX_UNDO_UPD_EXIST_REC的undo日志，长这样：



对照着这个图我们注意一下这几个地方：

- 因为这条undo日志是id为100的事务中产生的第4条undo日志，所以它对应的undo no就是3。
- 这条日志的roll_pointer指向undo no为1的那条日志，也就是插入主键值为2的记录时产生的那条undo日志，也就是最近一次对该记录做改动时产生的undo日志。
- 由于本条UPDATE语句中更新了索引列key1的值，所以需要记录一下索引列各列信息部分，也就是把主键和key1列更新前的信息填入。

更新主键的情况

在聚簇索引中，记录是按照主键值的大小连成了一个单向链表的，如果我们更新了某条记录的主键值，意味着这条记录在聚簇索引中的位置将会发生改变，比如你将记录的主键值从1更新为10000，如果还有非常多的记录的主键值分布在1 ~ 10000之间的话，那么这两条记录在聚簇索引中就有可能离得非常远，甚至中间隔了好多个页面。针对UPDATE语句中更新了记录主键值的这种情况，InnoDB在聚簇索引中分了两步处理：

- 将旧记录进行delete mark操作

  高能注意：这里是delete mark操作！这里是delete mark操作！这里是delete mark操作！也就是说在UPDATE语句所在的事务提交前，对旧记录只做一个delete mark操作，在事务提交后才由专门的线程做purge操作，把它加入到垃圾链表中。这里一定要和我们上边所说的在不更新记录主键值时，先真正删除旧

记录，再插入新记录的方式区分开！

> 小贴士： 之所以只对旧记录做delete mark操作，是因为别的事务同时也可能访问这条记录，如果把它真正的删除加入到垃圾链表后，别的事务就访问不到了。这个功能就是所谓的MVCC，我们后边的章节中会详细唠叨什么是个MVCC。

- 根据更新后各列的值创建一条新记录，并将其插入到聚簇索引中（需重新定位插入的位置）。

  由于更新后的记录主键值发生了改变，所以需要重新从聚簇索引中定位这条记录所在的位置，然后把它插进去。

针对UPDATE语句更新记录主键值的这种情况，在对该记录进行delete mark操作前，会记录一条类型为TRX_UNDO_DEL_MARK_REC的undo日志；之后插入新记录时，会记录一条类型为TRX_UNDO_INSERT_REC的undo日志，也就是说每对一条记录的主键值做改动时，会记录2条undo日志。这些日志的格式我们上边都唠叨过了，就不赘述了。

> 小贴士： 其实还有一种称为TRX_UNDO_UPD_DEL_REC的undo日志的类型我们没有介绍，主要是想避免引入过多的复杂度，如果大家对这种类型的undo日志的使用感兴趣的话，可以额外查一下别的资料。