MySQL——undo日志

〇、大纲

- 什么是 `undo日志` ？它有什么作用?
- 什么时候才会 `分配事务id` ？
- 如何开启 `只读` 事务?
- 如何开启 `读写` 事务?
- 事务id是 `怎么生成` 的?
- 事务id `保存在记录的哪个地方` ？
- `insert操作` 对应的undo日志是怎样的?
- `delete操作` 对应的undo日志是怎样的?

> 【删除记录的操作步骤】1、delete mark阶段； 2、purge阶段

- `update操作` 对应的undo日志是怎样的?

一、事务回滚的需求

- 事务是需要保证原子性的，也就是说，事务中的操作要么全部完成，要么什么也不做。但有如下情况，会造成事务执行不完：

> 【情况一】**事务执行过程中**可能遇到各种错误，比如代码bug出现异常。
> 【情况二】程序员在事务执行过程中**手动输入rollback**语句结束当前事务的执行。
> 遇到上面的情况，为了保证事务的原子性，我们需要把数据还原回原来的样子，这个过程就叫做**回滚**（rollback）。
> 有时候仅需要对部分语句进行回滚，有时间需要对整个事务进行回滚。

- *什么是undo日志呢?*

> 数据库为了回滚而记录的日志，我们就称之为撤销日志（`undo log`）

- 注意一点，由于**SELECT**操作并不会修改任何记录，所以并不需要记录相应的的undo日志。

二、事务id

2.1> 分配事务id的时机

- *何时分配事务id?*

> - 如果是**只读事务**，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，否则是不分配的。
> - 如果是**读写事务**，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，否则是不分配的。

- 综上所述，只有在事务对表中的**记录进行`改动`时**才会为这个事务分配一个唯一的事务id，否则事务id值**默认为`0`**。
- *如何开启只读事务?*

> 通过**START TRANSACTION READ ONLY**语句开启一个只读事务。
> 在只读事务中，不可以对普通表进行增删改操作；但可以对临时表进行增删改操作。

- *如何开启读写事务?*

> 通过**START TRANSACTION READ WRITE**语句开启一个读写事务。
> 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。

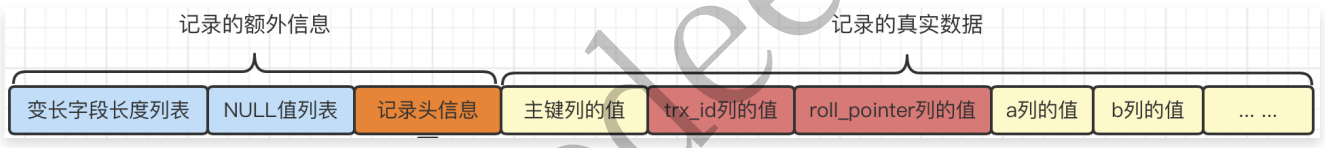
2.2> 事务id是怎么生成的

- 事务id本质上就是一个**数字**。
- 事务id生成策略如下：

> - 内存中维护一个**全局变量**，每当需要为某个事务分配事务id时，就会把该变量值当作事务id分配给该事务，并且**自增1**。
> - 每当这个变量的值为**256的倍数**时，就会将值刷新到**系统表空间中页号为5**的页面中一个名为**Max Trx ID**的属性中（占用8个字节）。
> - 当系统下一次启动时，会将**Max Trx ID**的值加载到到内存中，并**加上256之后**赋值给前面提到的全局变量。
> - 为什么要加256?
>
> 答：因为上次关机时，该全局变量的值可能大于磁盘页面中的Max Trx ID属性值。

2.3> trx_id隐藏列

- 在第5章数据页里，讲过记录行格式，如下所示：



> - `聚簇索引` 的记录会自动添加**trx_id**和**roll_pointer**的隐藏列。
> - 如果用户没有在表中定义主键，并且没有定义 `不允许为NULL` 值的 `UNIQUE键` ，还会自动添加一个名为**row_id**的隐藏列。

- trx_id的含义

> 表示对这个**聚簇索引记录**进行改动的语句所对应的**事务id**。

三、undo日志的格式

- 一般来说，**每对一条记录进行一次改动，就对应着1条undo日志**（某些情况下，也可能会对应着2条undo日志）。
- 在**一个事务中**，这些undo日志会**从0开始编号**，每生成一条undo日志，那么该条日志的undo no就**加1**。即：第0号undo日志、第1号undo日志等等。这个编号也称为**undo no**。
- 这些undo日志被记录到类型为 `FIL_PAGE_UNDO_LOG` 的页面中。
- 我们下面来看看，对表中数据进行不同操作都会产生什么样的undo日志？但是再此之前，我们先创

建一张表作为下面实验用的基表。

tb_user　table_id=1065		
id<int>（主键）	name<varchar(100)>（二级索引）	city<varchar(100)>

- *InnoDB为每张表都分配了一个唯一的**table_id**，那么如何查询表id?*

mysql 5.x查询**innodb_sys_tables**

mysql 8.x查询**innodb_tables**

```
33 •  select * from information_schema.innodb_tables;
```

100%　26:26

Result Grid　Filter Rows: Search　Export:

TABLE_ID	NAME	FLAG	N_COLS	SPACE	ROW_FORMAT	ZIP_PAGE_SIZE	SPACE_TYPE	INSTANT_COLS
1065	muse/tb_user	33	6	8	Dynamic	0	Single	0
1066	muse/tb_user_contact	33	9	9	Dynamic	0	Single	0
1067	muse/tb_message	33	10	10	Dynamic	0	Single	0
1068	muse/tb_message_detail	33	8	11	Dynamic	0	Single	0

3.1> INSERT操作对应的undo日志

- 如果希望 `回滚一个插入操作` ，无论是乐观插入还是悲观插入，那么只需要把插入的这条**记录删除**就可以了。也就是说，在写对应的undo日志时，只要把这条记录的**主键信息记上**就好了。对应的undo日志类型为**TRX_UNDO_INSERT_REC**。
- TRX_UNDO_INSERT_REC类型的undo日志结构

- end of record
  本条undo日志结束，下一条开始时在页面中的地址。
- undo type
  本条undo**日志的类型**，也就是 `TRX_UNDO_INSERT_REC`
- undo no
  本条undo日志对应的**编号**
- table id
  本条undo日志对应的记录所在表的 `table_id`
- **主键各列信息 <len, value>列表**
  主键的每个列占用的**存储空间大小**和**真实值**
- start of record

> 上一条undo日志结束，**本条开始时**在页面中的地址

- 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二级索引都插入记录。不过在记录undo日志时，我们只需要**针对聚簇索引**记录来记录一条undo日志就好了。如果回滚，会根据主键信息进行对应的删除操作。**在执行删除操作时，就会把聚簇索引和二级索引中相应的记录都删掉**。
- 演示插入操作生成undo日志。

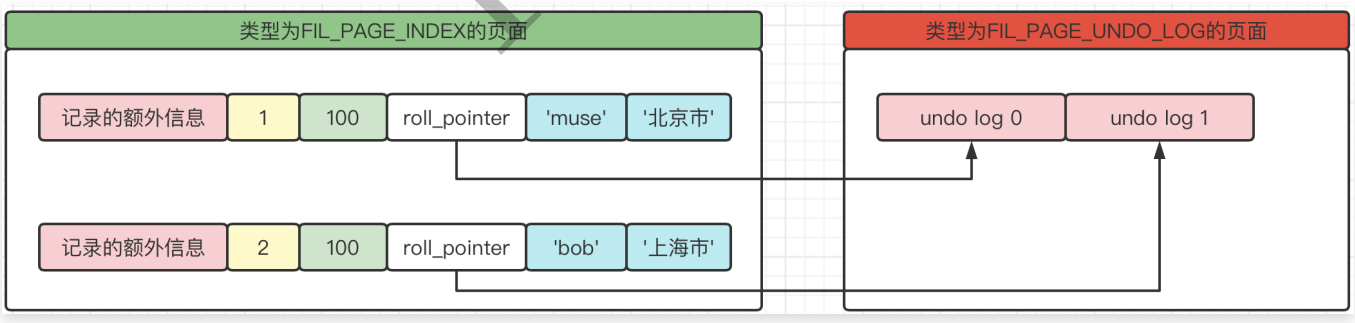
> 我们先插入两条记录：
> BEGIN; # 显示开启一个事务，假设该事务的事务id为100
>
> # 插入两条记录
> INSERT INTO tb_user(id, name, city) VALUES(1, 'muse','北京市'), (2, 'bob','上海市');

结束地址	TRX_UNDO_INSERT_REC	0	1065	<4, 1>	开始地址
结束地址	TRX_UNDO_INSERT_REC	1	1065	<4, 2>	开始地址

> 【注】因为id主键的类型为**INT**，存储长度为**4个字节**。所以主键列表值为<4, 1> 和 <4, 2>。

- **roll_pointer**本质上就是一个指向记录对应的**undo日志的指针**。

> - 聚簇索引记录存放到类型为 `FIL_PAGE_INDEX` 的页面；
> - undo日志存放到类型为 `FIL_PAGE_UNDO_LOG` 的页面。
> - 聚簇索引记录和undo日志的存放位置，如下图所示：



3.2> DELETE操作对应的undo日志

- 正常记录链表

> 记录的头信息中的 `next_record` 属性组成一个单向链表，我们把这个链表称为**正常记录链表**。

- 垃圾链表

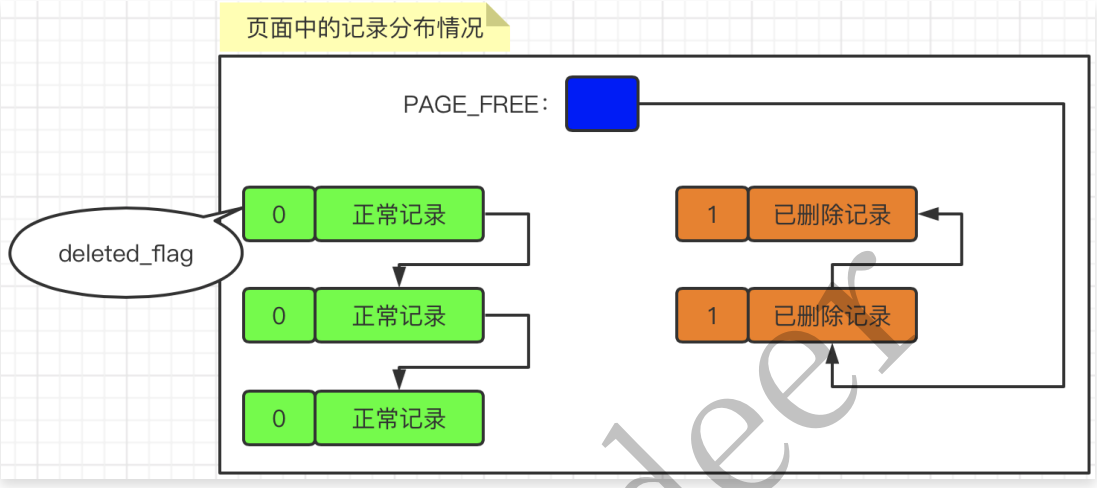
被删除的记录其实也会根据记录头信息中的`next_record`属性组成一个链表，只不过这个链表中的记录所占用的存储空间可以被重新利用，所以也称这个链表为**垃圾链表**。

- *PAGE_FREE的作用是什么？*

Page Header部分中有一个名为**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**。

每删除一条记录，则该记录都会插入到垃圾链表的**头节点处**。

- 举例，有3条正常记录和2条被删除记录，他们在页中的记录分布情况如下所示：

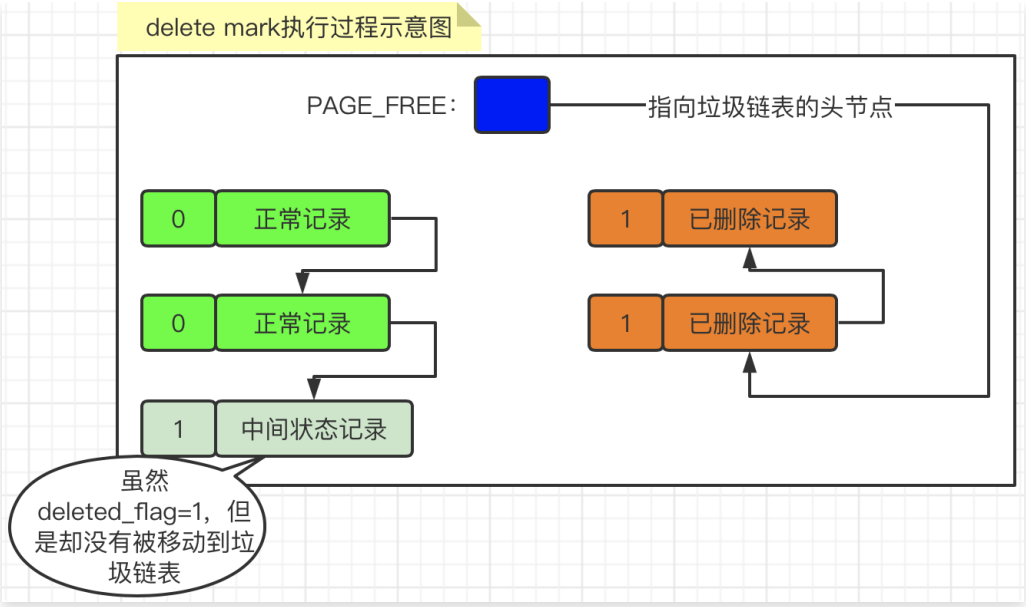


在垃圾链表中，这些记录占用的存储空间**可以被重新利用**。

- 如果要删除一条记录，则需要两个步骤
  - 第一步：delete mark阶段

    仅仅将记录的`deleted_flag`标识位**设置为1**，但是这条记录**并没有加入到垃圾链表中**。也就是说，这条记录即不是正常记录，也不是已删除记录。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**中间状态**（其实主要是为了实现MVCC的功能才这样处理的）。

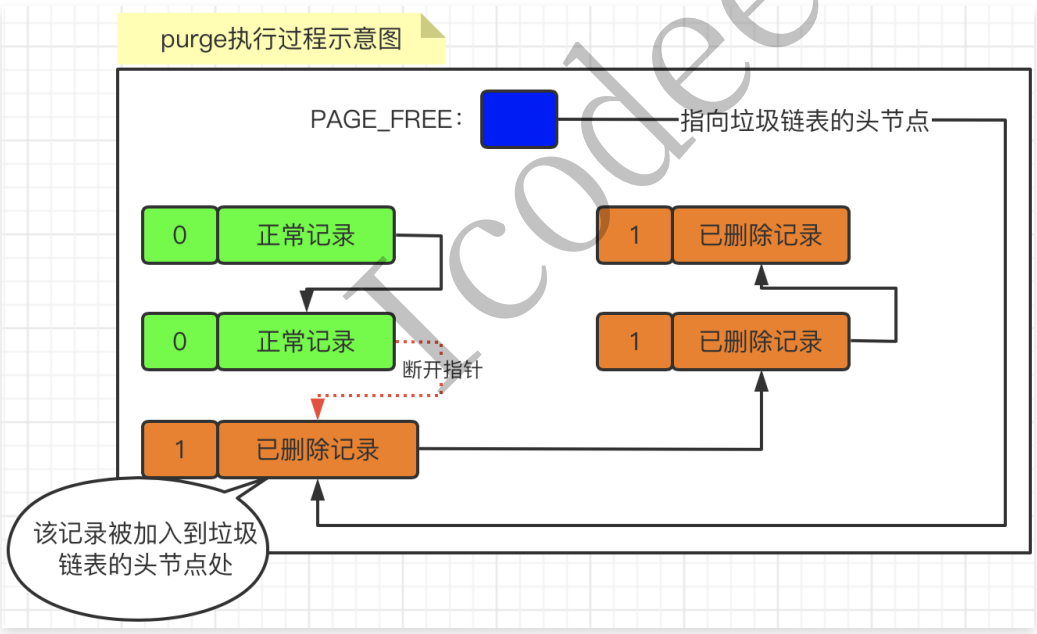
    如下所示：



- 第二步：purge阶段

  当该删除语句所在的`事务提交后`，会有**专门的线程**来把该记录从正常记录链表中移除，并加入到垃圾链表中作为**头节点**。

  如下所示：



- 关于其它垃圾链的重用空间的知识点补充介绍

  - *PAGE_GARBAGE是做什么的?*

    Page Header部分有一个名为`PAGE_GARBAGE`的属性。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**。每当有已删除记录加入到垃圾链表后，都会把这个`PAGE_GARBAGE`属性的值加上已删除记录占用的存储空间大小。

  - *如何重用垃圾链表的存储空间?*

`PAGE_FREE` 指向垃圾链表的头节点，每当新插入数据的时候：

- 首先：判断垃圾链表**头节点记录**的存储空间是否足够容纳这条新插入的记录。如果可以容纳则直接重用这条已删除记录的存储空间。
- 其次：如果不能容纳，则**直接向页面申请新的空间**来存储这条记录。（**是的，你没看错！并不会尝试遍历垃圾链表，以找到可以容纳新记录的节点**）

- *如果新插入的那条记录记录小于重用的记录空间，那么会有一部分空间用不到，怎么处理呢？直接浪费掉吗？*

这种情况会频繁发生，也就会随着记录越插越多而产生越来越多的空间碎片。只有当**页面块满**的时候，如果**再插入**一条新记录，**无法分配一条完整的记录空间**时，会先查看 `PAGE_GARBAGE` 的空间和**剩余空间相加**是否可以容纳这条新的记录，如果可以，InnoDB则会尝试**重新组织页内的记录**。即：先开辟一个临时页面，把原页面内的记录依次挨着插入一遍到临时页，之后，再把临时页的内容复制到本页面，这样就可以把那些碎片空间都释放出来了。但是该操作比较耗费性能。

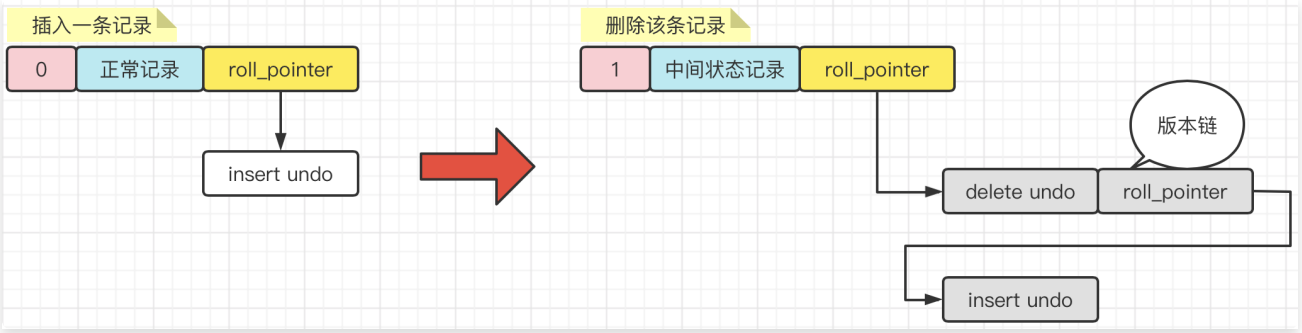
- 由于一旦事务提交，我们也就不需要再回滚这个事务了，所以在设计undo日志时，**只需要考虑delete mark这个阶段所做的影响进行回滚就可以了**。
- **TRX_UNDO_DEL_MARK_REC**类型的undo日志结构如下所示：

- **info bits**
  记录头信息的前4个比特的值。
- **trx_id**
  旧记录的trx_id值。
- **roll_pointer**
  旧记录的roll_pointer值。
- **len of index_col_info**
  也就是下边的【索引列各列信息】部分和本部分占用的存储空间总和。
- **索引列各列信息 <pos, len, value>列表**
  凡是被索引的列的各列信息。

- *为什么TRX_UNDO_DEL_MARK_REC类型的undo日志保存旧记录的**trx_id**值和**roll_pointer**值？*

保存旧记录的trx_id值——为了采用 `事务id` 作为版本号，记录每个undo日志所对应的版本是多少。
保存旧记录的roll_pointer值——可以通过undo日志的 `roll_pointer属性` 找到上一次对该记录进行改动时产生的undo日志，因此可以将日志串成链表。这个链表就是**版本链**。

- 我们模拟一下，新增一条记录，然后再删除这条记录的完整操作过程，如下所示：



- 演示删除操作生成undo日志。

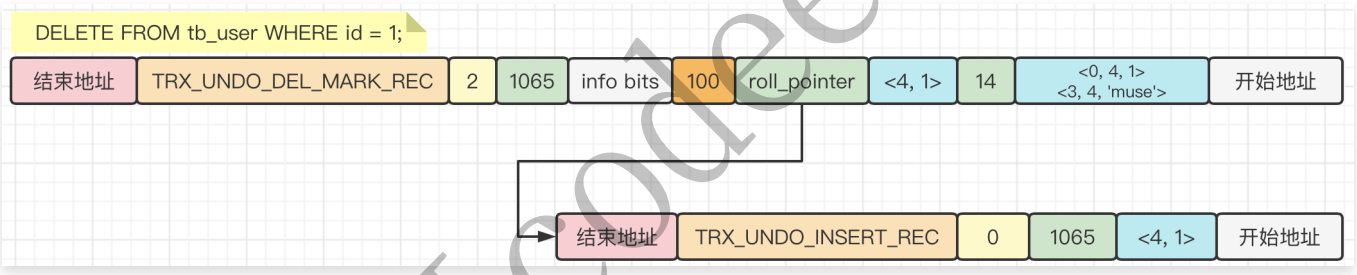
我们先插入两条记录：

BEGIN; # 显示开启一个事务，假设该事务的事务id为100

插入两条记录

INSERT INTO tb_user(id, name, city) VALUES(1, 'muse','北京市'), (2, 'bob','上海市');

删除一条记录DELETE FROM tb_user WHERE id = 1;



其中需要说明两个字段：

- 索引列各列信息 <pos, len, value>列表

  <0, 4, 1>：

  由于id列是主键，所以pos=0；

  由于id列的类型是INT，所以len=4；

  由于id=1，所以value=1；

  <3, 4, 'muse'>：

  由于name列是二级索引，它排在 `id列` 、 `trx_id列` 、 `roll_pointer列` 之后，所以pos=3；

  由于id列的类型是INT，所以len=4；

  由于name='muse'，所以value='muse'；

- len of index_col_info

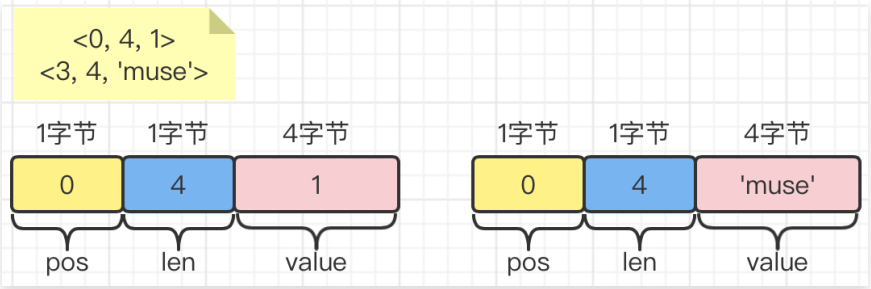
  pos使用1字节来存储。

  len使用1字节来存储。

value根据具体值，来判断。比如：id=1，主键是INT占4个字节，所以value使用4字节存储。name='muse'，VARCHAR类型，所以value用4字节存储。

len of index_col_info本身占2个字节。

综上所述，<0, 4, 1><3, 4, 'muse'>占用空间等于：(1+1+4)+(1+1+4) =12，如下图所示：



最后，再加上len of index_col_info属性本身占2个字节，所以总共14字节。 即：len of index_col_info=14。

3.3> 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式。

3.3.1> 不更新主键

- **就地更新**

在更新记录时，对于**被更新的每个列**来说，如果更新后的列与更新前的列占用的存储空间**一样大**，那么可以进行就地更新，也就是直接在原记录的基础上修改对应列的值。

- **先删除旧记录，再插入新记录**

  - 如果有**任何一个**被更新的列在更新前后占用的存储空间**大小不一致**，那么就需要先把这条旧记录从聚簇索引页面中删除，然后再根据更新后列的值**创建一条新的记录**并插入到页面中。
  - 这里的删除，是**真正的删除**，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**。
  - 是由**用户线程**同步执行真正的删除操作，而不是DELETE语句中进行purge操作时使用的专门线程。
  - 如果新创建的记录占用的存储空间**不超过旧记录占用的空间**，那么可以**直接重用**加入到**垃圾链表**中的旧记录所占用的存储空间，否则，需要在页面中**新申请**一块空间供新记录使用。
  - 如果本页面已经没有可用的空间，就需要进行**页面分裂**操作，然后再插入新的记录。

- 更新操作对应TRX_UNDO_UPD_EXIST_REC类型的undo日志结构，如下图所示：

其中大部分属性与DELETE操作的redo日志相同。其中不同的如下说明：

- **n_updated**
  表示本条UPDATE语句执行后将有几个列被更新
- **被更新的列更新前信息 <pos, old_len, old_value>列表**
  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。

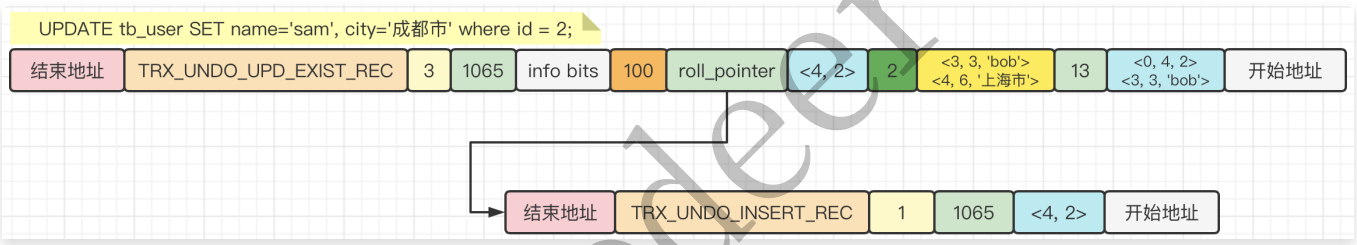
- 演示更新操作生成undo日志。

我们先插入两条记录：

```
BEGIN; # 显示开启一个事务，假设该事务的事务id为100

# 插入两条记录
INSERT INTO tb_user(id, name, city) VALUES(1, 'muse','北京市'), (2, 'bob','上海市');

# 删除一条记录DELETE FROM tb_user WHERE id = 1;
# 更新一条记录UPDATE tb_user SET name='sam', city='成都市' where id = 2;
```



这个UPDATE语句更新的列的大小都没有改动，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前，会先记录一条类型为 `TRX_UNDO_UPD_EXIST_REC` 的undo日志。
其中，<0, 4, 2><3, 3, 'bob'>占用空间等于：(1+1+4)+(1+1+3) =11，最后，再加上len of index_col_info属性本身占2个字节，所以总共13字节。 即：len of index_col_info=13。

3.3.2> 更新主键

- **步骤一：将旧记录进行delete mark操作**

此时**仅执行delete mark操作**。而在**事务提交后**，才由专门的线程**执行purge操作**，从而把它加入到垃圾链表中。
【注】之所以只对旧记录执行delete mark操作，是因为别的事务也可能同时访问这条记录，如果把它真正删除并加入到垃圾链表后，别的事务就访问不到了。这个功能就是MVCC。

- **步骤二：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**

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操作时，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进行改动，都会记录**2条undo日志**。

3.4> 增删改操作对二级索引的影响

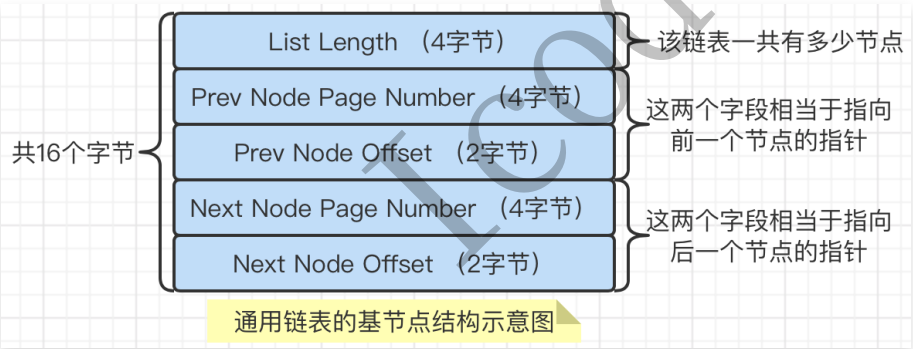
- 对于二级索引，INSERT操作和DELETE操作与在聚簇索引中执行时产生的影响差不多；但是对于UPDATE操作稍微有点不同。
- 如果在UPDATE语句中涉及了二级索引，即：**更新了二级索引的值**，那么意味着要进行下面两步骤的操作：

> 步骤一：对旧的二级索引记录执行delete mark操作。
> 步骤二：根据更新后的值创建一条新的二级索引记录，然后在二级索引对应的B+树中重新定位到它的位置并插进去。

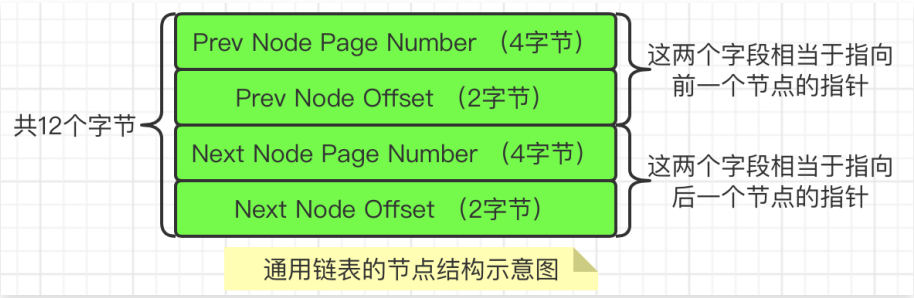
- 需要注意的是，每当我们增删改一条二级索引记录时，都会影响这条二级索引记录所在页面的Page Header部分中一个名为PAGE_MAX_TRX_ID的属性。这个属性代表修改当前页的最大事务id。后面会用到该值。

四、通用链表结构

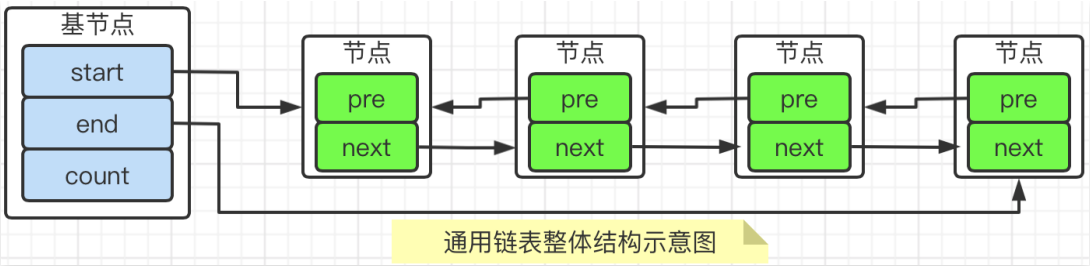
- 在某个表空间内，我们可以通过一个页的**页号**(Page Number)与页内的**偏移量**(Offset)来唯一定位一个节点的位置。
- 通用链表**基节点**结构示意图



- 通用链表**普通节点**结构示意图



- 通用链表**整体**结构示意图



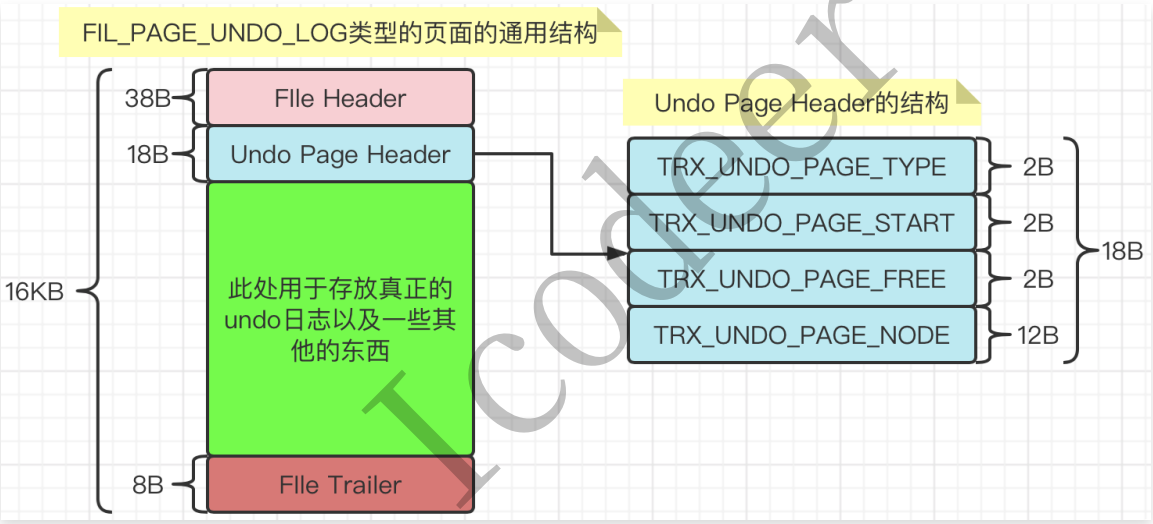
五、FIL_PAGE_UNDO_LOG页面

- 表空间其实是由许许多多的页面构成的，页面默认大小为16KB。
- 页面有不同的类型，比如：

  FIL_PAGE_INDEX——用于存储聚簇索引以及二级索引的页面。

  FIL_PAGE_TYPE_FSP_HDR——用于存储表空间头部信息的页面。

  FIL_PAGE_UNDO_LOG——用于存储undo日志的页面，也叫Undo页面。
- FIL_PAGE_UNDO_LOG类型的页面的通用结构如下所示：



- Undo Page Header是Undo页面特有的。其中属性意义如下：

  TRX_UNDO_PAGE_TYPE
  - 表示本页面准备存储什么类型的undo日志。可选值为：TRX_UNDO_INSERT或者TRX_UNDO_UPDATE
  - 之所以把undo日志分成2个大类，是因为类型为TRX_UNDO_INSERT_REC的undo日志在事务提交后就可以直接删除掉，而其他类型的undo日志还需要为MVCC服务，不能直接删除掉，因此对它们的处理需要区别对待。
  - TRX_UNDO_INSERT（使用1表示），称为insert undo日志

    一般由insert语句产生，当update语句中有更新主键的情况时，也会产生此类型的undo日志。

    类型为TRX_UNDO_INSERT_REC的undo日志属于这个大类。
  - TRX_UNDO_UPDATE（使用2表示），称为update undo日志

出了insert undo日志之外，其他类型的undo日志都属于这个大类。一般由delete、update语句产生的undo日志。

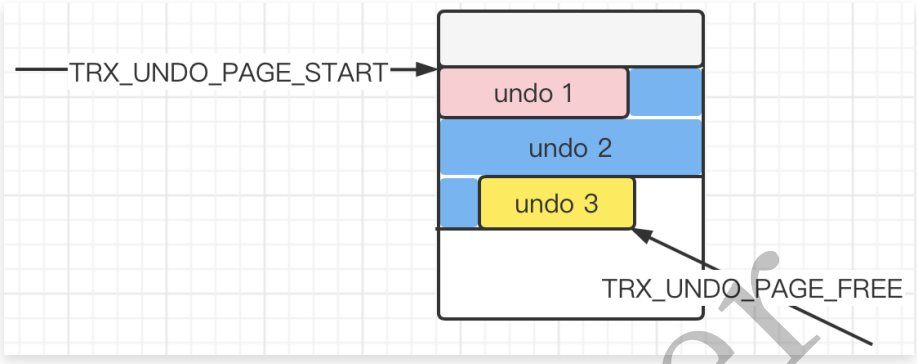
类型为TRX_UNDO_DEL_MARK_REC，TRX_UNDO_UPD_EXIST_REC等

TRX_UNDO_PAGE_START

- 表示第一条undo日志在本页面中的起始偏移量

TRX_UNDO_PAGE_FREE

- 表示最后一条undo日志结束时的偏移量
- 假设写入了3条undo日志，TRX_UNDO_PAGE_START和TRX_UNDO_PAGE_FREE的示意图如下所示：



TRX_UNDO_PAGE_NODE

- 代表一个链表节点结构。

吾尝终日而思矣，不如须臾之所学也；
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」