第23章、undo日志（下）

标签： MySQL是怎样运行的

上一章我们主要唠叨了为什么需要undo日志，以及INSERT、DELETE、UPDATE这些会对数据做改动的语句都会产生什么类型的undo日志，还有不同类型的undo日志的具体格式是什么。本章会继续唠叨这些undo日志会被具体写到什么地方，以及在写入过程中需要注意的一些问题。

通用链表结构

在写入undo日志的过程中会使用到多个链表，很多链表都有同样的节点结构，如图所示：

在某个表空间内，我们可以通过一个页的页号和在页内的偏移量来唯一定位一个节点的位置，这两个信息也就相当于指向这个节点的一个指针。所以：

- Pre Node Page Number和Pre Node Offset的组合就是指向前一个节点的指针
- Next Node Page Number和Next Node Offset的组合就是指向后一个节点的指针。

整个List Node占用12个字节的存储空间。

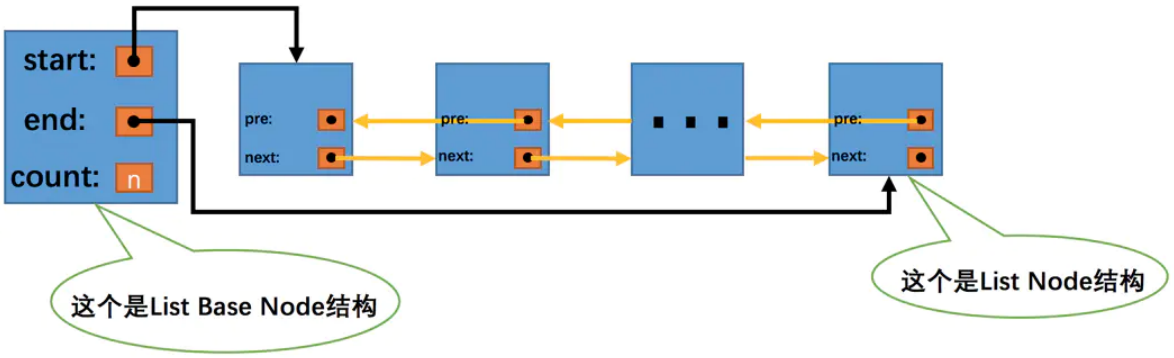
为了更好的管理链表，设计InnoDB的大叔还提出了一个基节点的结构，里边存储了这个链表的头节点、尾节点以及链表长度信息，基节点的结构示意图如下：

其中：

- `List Length`表明该链表一共有多少节点。
- `First Node Page Number`和`First Node Offset`的组合就是指向链表头节点的指针。
- `Last Node Page Number`和`Last Node Offset`的组合就是指向链表尾节点的指针。

整个`List Base Node`占用`16`个字节的存储空间。

所以使用`List Base Node`和`List Node`这两个结构组成的链表的示意图就是这样：

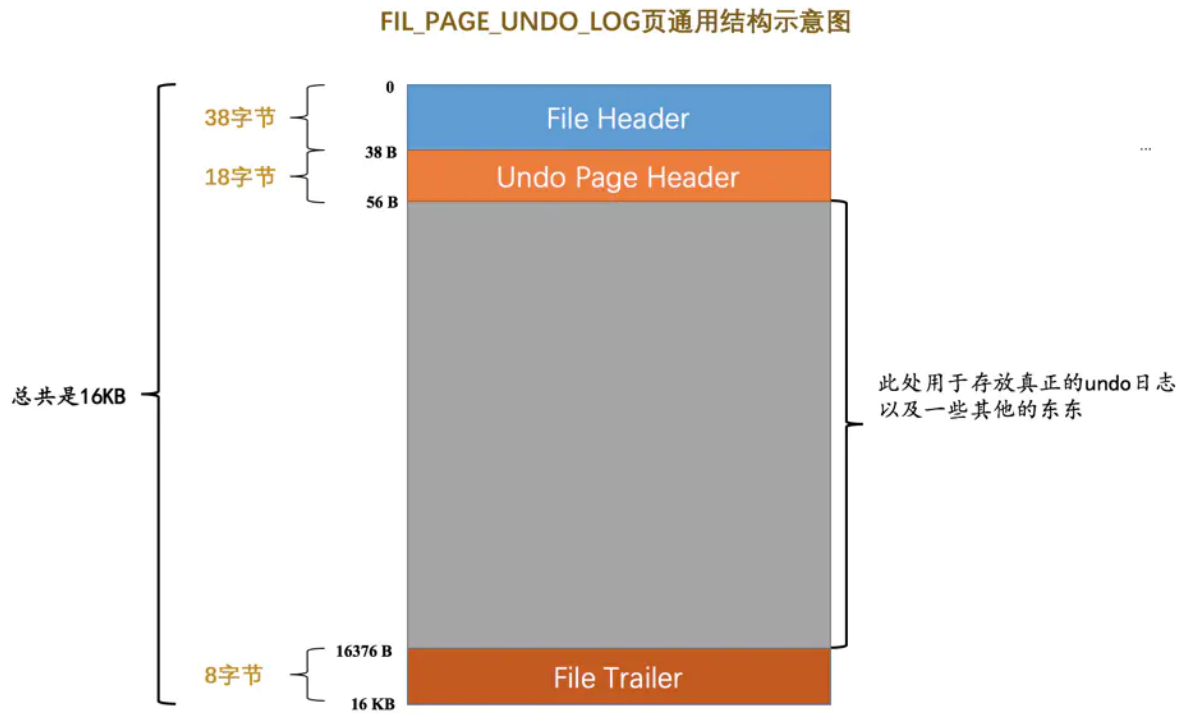


小贴士： 上述链表结构我们在前边的文章中频频提到，尤其是在表空间那一章重点描述过，不过我不敢奢求大家都记住了，所以在这里又强调一遍，希望大家不要嫌我烦，我只是怕大家忘了学习后续内容吃力而已～

FIL_PAGE_UNDO_LOG页面

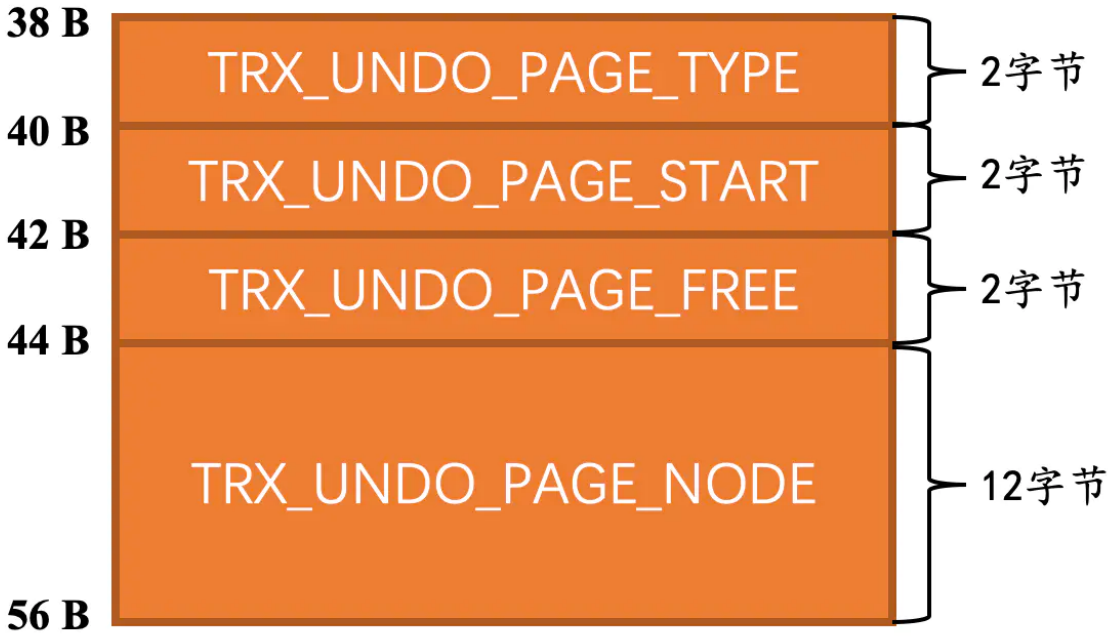
我们前边唠叨表空间的时候说过，表空间其实是由许许多多的页面构成的，页面默认大小为`16KB`。这些页面有不同的类型，比如类型为`FIL_PAGE_INDEX`的页面用于存储聚簇索引以及二级索引，类型为`FIL_PAGE_TYPE_FSP_HDR`的页面用于存储表空间头部信息的，还有其他各种类型的页

面，其中有一种称之为FIL_PAGE_UNDO_LOG类型的页面是专门用来存储undo日志的，这种类型的页面的通用结构如下图所示（以默认的16KB大小为例）：



“类型为FIL_PAGE_UNDO_LOG的页”这种说法太绕口，以后我们就简称为Undo页面了哈。上图中的File Header和File Trailer是各种页面都有的通用结构，我们前边唠叨过很多遍了，这里就不赘述了（忘记了的可以到讲述数据页结构或者表空间的章节中查看）。Undo Page Header是Undo页面所特有的，我们来看一下它的结构：

Undo Page Header结构示意图



其中各个属性的意思如下：

- `TRX_UNDO_PAGE_TYPE`：本页面准备存储什么种类的`undo日志`。

  我们前边介绍了好几种类型的`undo日志`，它们可以被分为两个大类：

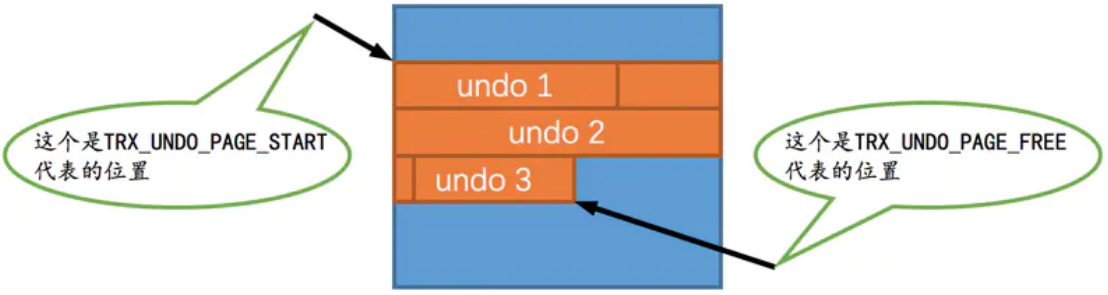
  - `TRX_UNDO_INSERT`（使用十进制`1`表示）：类型为`TRX_UNDO_INSERT_REC`的`undo日志`属于此大类，一般由`INSERT`语句产生，或者在`UPDATE`语句中有更新主键的情况也会产生此类型的`undo日志`。
  - `TRX_UNDO_UPDATE`（使用十进制`2`表示），除了类型为`TRX_UNDO_INSERT_REC`的`undo日志`，其他类型的`undo日志`都属于这个大类，比如我们前边说的`TRX_UNDO_DEL_MARK_REC`、`TRX_UNDO_UPD_EXIST_REC`啥的，一般由`DELETE`、`UPDATE`语句产生的`undo日志`属于这个大类。

  这个`TRX_UNDO_PAGE_TYPE`属性可选的值就是上边的两个，用来标记本页面用于存储哪个大类的`undo日志`，不同大类的`undo日志`不能混着存储，比如一个`Undo页面`的`TRX_UNDO_PAGE_TYPE`属性值为`TRX_UNDO_INSERT`，那么这个页面就只能存储类型为`TRX_UNDO_INSERT_REC`的`undo日志`，其他类型的`undo日志`就不能放到这个页面中了。

  > 小贴士： 之所以把undo日志分成两个大类，是因为类型为TRX_UNDO_INSERT_REC的undo日志在事务提交后可以直接删除掉，而其他类型的undo日志还需要为所谓的MVCC服务，不能直接删除掉，对它们的处理需要区别对待。当然，如果你看这段话迷迷糊糊的话，那就不需要再看一遍了，现在只需要知道undo日志分为2个大类就好了，更详细的东西我们后边会仔细唠叨的。

- TRX_UNDO_PAGE_START：表示在当前页面中是从什么位置开始存储undo日志的，或者说表示第一条undo日志在本页面中的起始偏移量。
- TRX_UNDO_PAGE_FREE：与上边的TRX_UNDO_PAGE_START对应，表示当前页面中存储的最后一条undo日志结束时的偏移量，或者说从这个位置开始，可以继续写入新的undo日志。

  假设现在向页面中写入了3条undo日志，那么TRX_UNDO_PAGE_START和TRX_UNDO_PAGE_FREE的示意图就是这样：



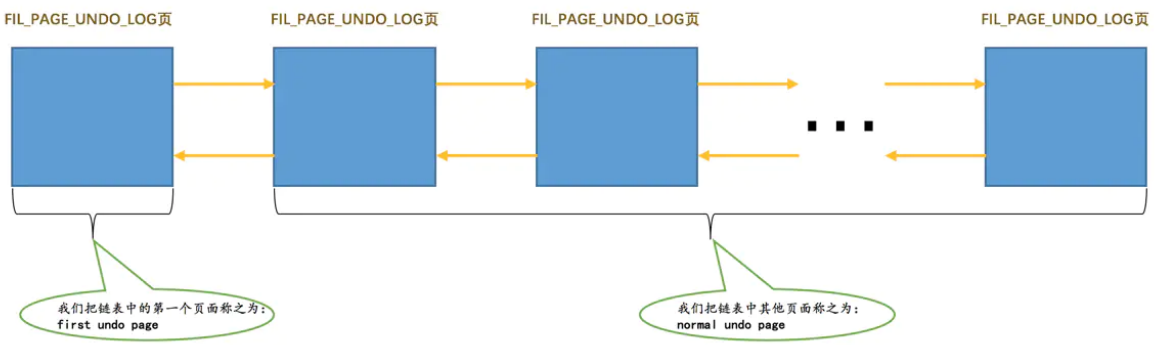
  当然，在最初一条undo日志也没写入的情况下，TRX_UNDO_PAGE_START和TRX_UNDO_PAGE_FREE的值是相同的。
- TRX_UNDO_PAGE_NODE：代表一个List Node结构（链表的普通节点，我们上边刚说的）。

  下边马上用到这个属性，稍安勿躁。

Undo页面链表

单个事务中的Undo页面链表

因为一个事务可能包含多个语句，而且一个语句可能对若干条记录进行改动，而对每条记录进行改动前，都需要记录1条或2条的undo日志，所以在一个事务执行过程中可能产生很多undo日志，这些日志可能一个页面放不下，需要放到多个页面中，这些页面就通过我们上边介绍的TRX_UNDO_PAGE_NODE属性连成了链表：

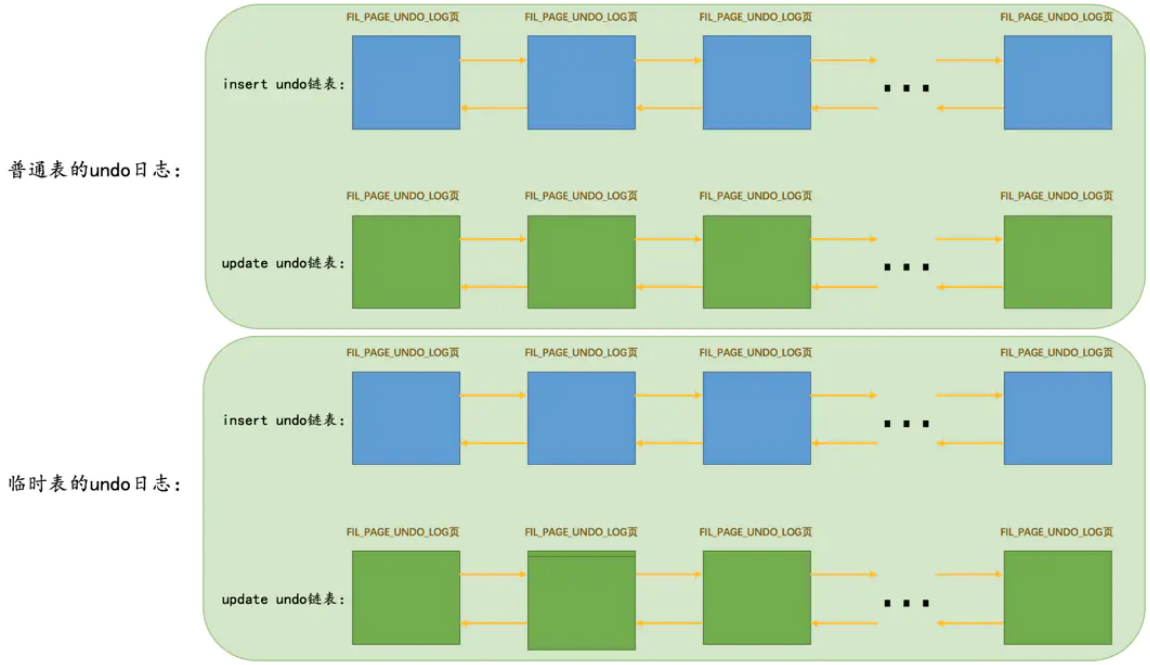


大家往上再瞅一瞅上边的图，我们特意把链表中的第一个Undo页面给标了出来，称它为first undo

page，其余的Undo页面称之为normal undo page，这是因为在first undo page中除了记录Undo Page Header之外，还会记录其他的一些管理信息，这个我们稍后再说哈。

在一个事务执行过程中，可能混着执行INSERT、DELETE、UPDATE语句，也就意味着会产生不同类型的undo日志。但是我们前边又强调过，同一个Undo页面要么只存储TRX_UNDO_INSERT大类的undo日志，要么只存储TRX_UNDO_UPDATE大类的undo日志，反正不能混着存，所以在一个事务执行过程中就可能需要2个Undo页面的链表，一个称之为insert undo链表，另一个称之为update undo链表，画个示意图就是这样：

另外，设计InnoDB的大叔规定对普通表和临时表的记录改动时产生的undo日志要分别记录（我们稍后阐释为啥这么做），所以在一个事务中最多有4个以Undo页面为节点组成的链表：



当然，并不是在事务一开始就会为这个事务分配这4个链表，具体分配策略如下：

- 刚刚开启事务时，一个Undo页面链表也不分配。
- 当事务执行过程中向普通表中插入记录或者执行更新记录主键的操作之后，就会为其分配一个普通表的insert undo链表。
- 当事务执行过程中删除或者更新了普通表中的记录之后，就会为其分配一个普通表的update

undo链表。

- 当事务执行过程中向临时表中插入记录或者执行更新记录主键的操作之后，就会为其分配一个临时表的insert undo链表。
- 当事务执行过程中删除或者更新了临时表中的记录之后，就会为其分配一个临时表的update undo链表。

总结一句就是：按需分配，啥时候需要啥时候再分配，不需要就不分配。

多个事务中的Undo页面链表

为了尽可能提高undo日志的写入效率，不同事务执行过程中产生的undo日志需要被写入到不同的Undo页面链表中。比方说现在有事务id分别为1、2的两个事务，我们分别称之为trx 1和trx 2，假设在这两个事务执行过程中：

- trx 1对普通表做了DELETE操作，对临时表做了INSERT和UPDATE操作。

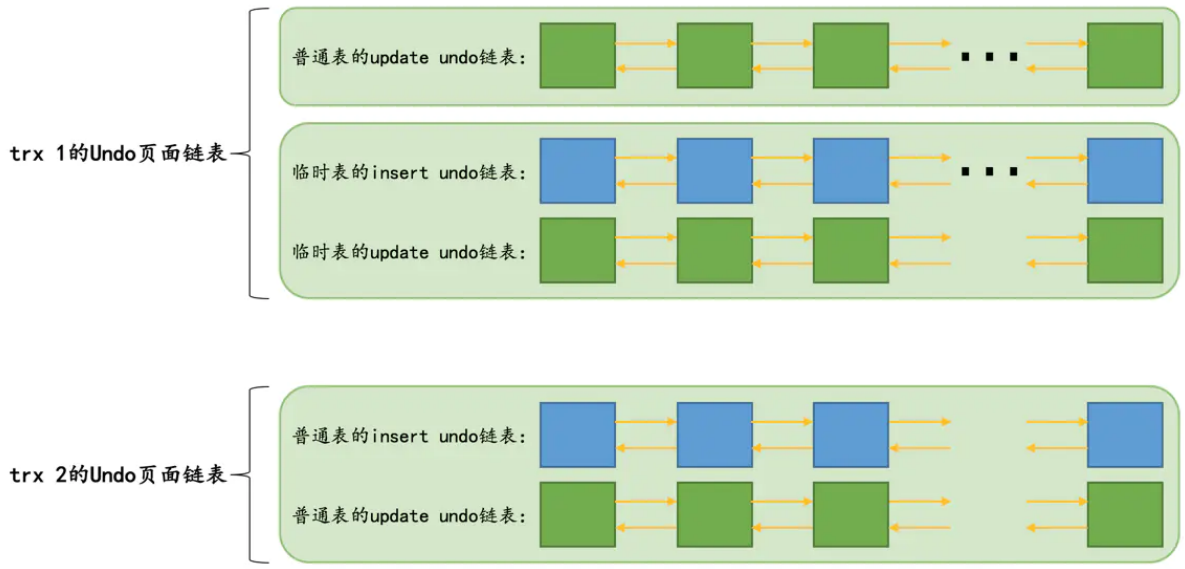
  InnoDB会为trx 1分配3个链表，分别是：

  - 针对普通表的update undo链表
  - 针对临时表的insert undo链表
  - 针对临时表的update undo链表。
- trx 2对普通表做了INSERT、UPDATE和DELETE操作，没有对临时表做改动。

  InnoDB会为trx 2分配2个链表，分别是：

  - 针对普通表的insert undo链表
  - 针对普通表的update undo链表。

综上所述，在trx 1和trx 2执行过程中，InnoDB共需为这两个事务分配5个Undo页面链表，画个图就是这样：

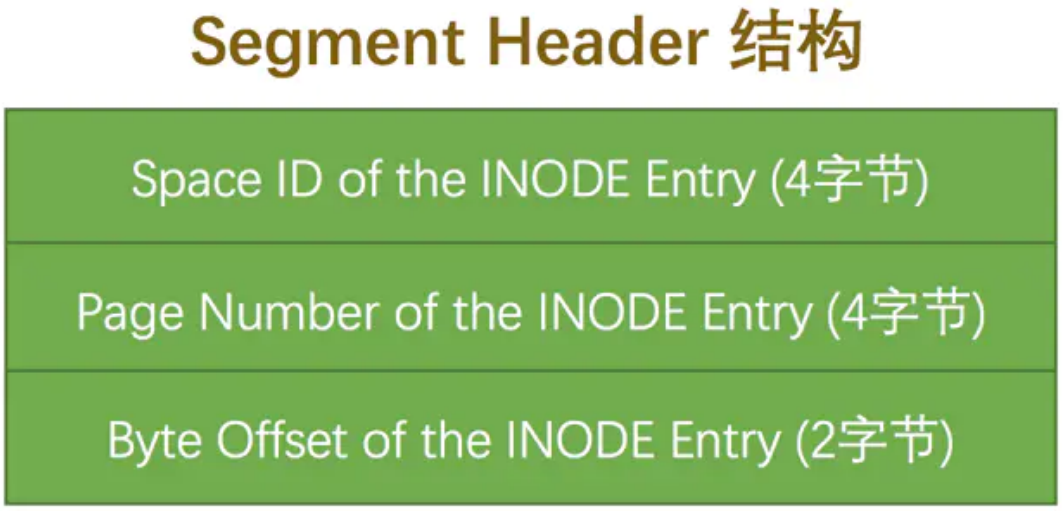


如果有更多的事务，那就意味着可能会产生更多的Undo页面链表。

undo日志具体写入过程

段（Segment）的概念

如果你有认真看过表空间那一章的话，对这个段的概念应该印象深刻，我们当时花了非常大的篇幅来唠叨这个概念。简单讲，这个段是一个逻辑上的概念，本质上是由若干个零散页面和若干个完整的区组成的。比如一个B+树索引被划分成两个段，一个叶子节点段，一个非叶子节点段，这样叶子节点就可以被尽可能的存到一起，非叶子节点被尽可能的存到一起。每一个段对应一个INODE Entry结构，这个INODE Entry结构描述了这个段的各种信息，比如段的ID，段内的各种链表基节点，零散页面的页号有哪些等信息（具体该结构中每个属性的意思大家可以到表空间那一章里再次重温一下）。我们前边也说过，为了定位一个INODE Entry，设计InnoDB的大叔设计了一个Segment Header的结构：



整个Segment Header占用10个字节大小，各个属性的意思如下：

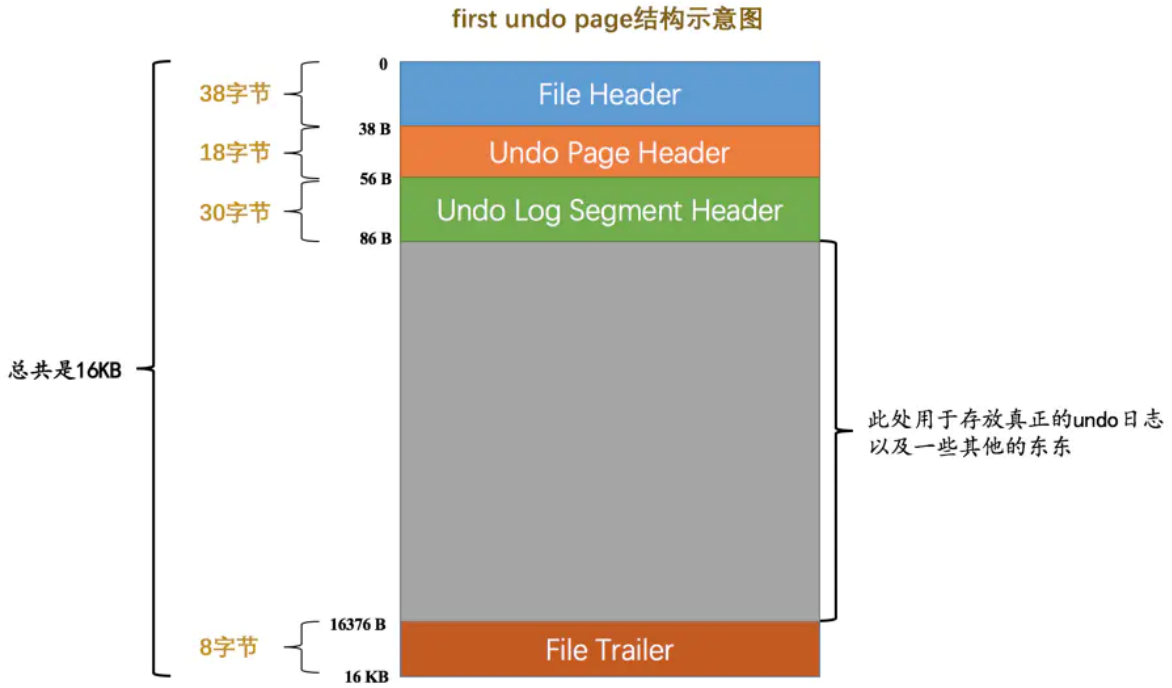
- Space ID of the INODE Entry：INODE Entry结构所在的表空间ID。
- Page Number of the INODE Entry：INODE Entry结构所在的页面页号。
- Byte Offset of the INODE Ent：INODE Entry结构在该页面中的偏移量

知道了表空间ID、页号、页内偏移量，不就可以唯一定位一个INODE Entry的地址了么~

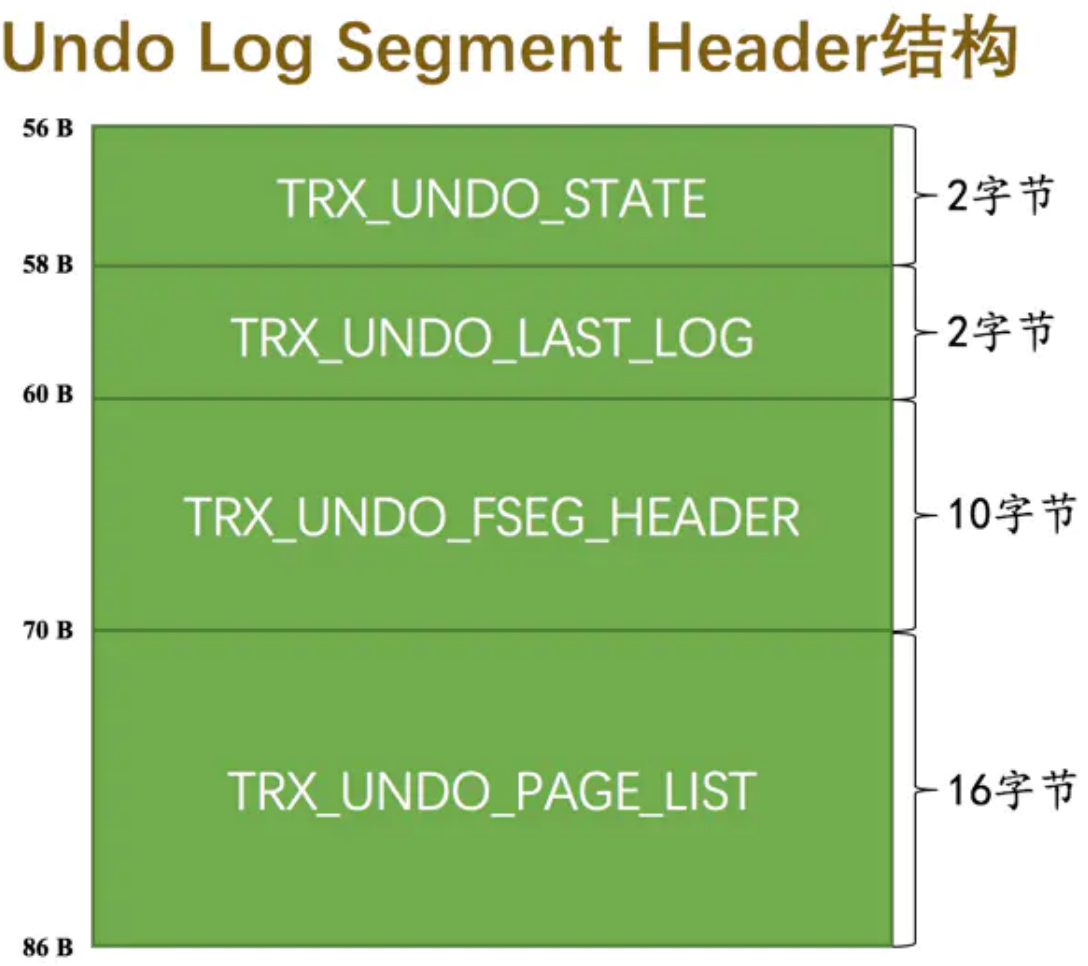
> 小贴士： 这部分关于段的各种概念我们在表空间那一章中都有详细解释，在这里重提一下只是为了唤醒大家沉睡的记忆，如果有任何不清楚的地方可以再次跳回表空间的那一章仔细读一下。

Undo Log Segment Header

设计InnoDB的大叔规定，每一个Undo页面链表都对应着一个段，称之为Undo Log Segment。也就是说链表中的页面都是从这个段里边申请的，所以他们在Undo页面链表的第一个页面，也就是上边提到的first undo page中设计了一个称之为Undo Log Segment Header的部分，这个部分中包含了该链表对应的段的segment header信息以及其他的一些关于这个段的信息，所以Undo页面链表的第一个页面其实长这样：



可以看到这个Undo链表的第一个页面比普通页面多了个Undo Log Segment Header，我们来看一下它的结构：



其中各个属性的意思如下：

- `TRX_UNDO_STATE`：本`Undo页面`链表处在什么状态。

  一个`Undo Log Segment`可能处在的状态包括：

  - `TRX_UNDO_ACTIVE`：活跃状态，也就是一个活跃的事务正在往这个段里边写入`undo日志`。
  - `TRX_UNDO_CACHED`：被缓存的状态。处在该状态的`Undo页面`链表等待着之后被其他事务重用。
  - `TRX_UNDO_TO_FREE`：对于`insert undo`链表来说，如果在它对应的事务提交之后，该链表不能被重用，那么就会处于这种状态。
  - `TRX_UNDO_TO_PURGE`：对于`update undo`链表来说，如果在它对应的事务提交之后，该链表不能被重用，那么就会处于这种状态。
  - `TRX_UNDO_PREPARED`：包含处于`PREPARE`阶段的事务产生的`undo日志`。

  > 小贴士： Undo页面链表什么时候会被重用，怎么重用我们之后会详细说的。事务的PREPARE阶段是在所谓的分布式事务中才出现的，本书中不会介绍更多关于分布式事务的事情，所以大家目前忽略这个状态就好了。

- `TRX_UNDO_LAST_LOG`：本`Undo页面`链表中最后一个`Undo Log Header`的位置。

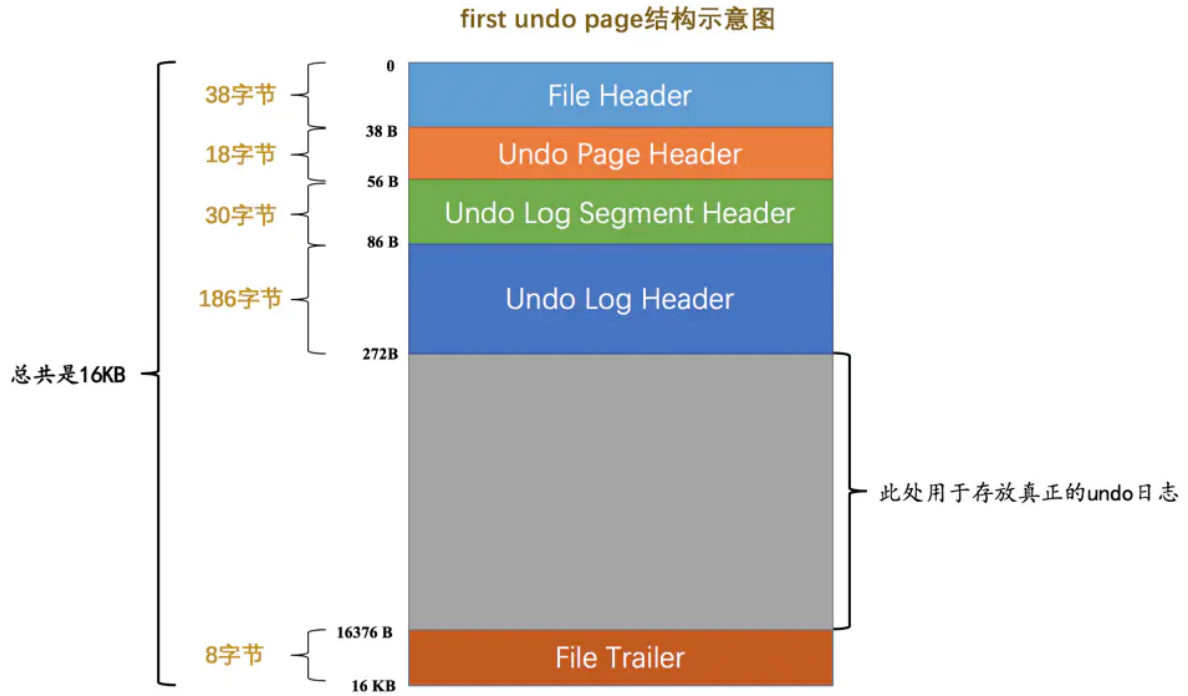
> 小贴士： 关于什么是Undo Log Header，我们稍后马上介绍哈。

- `TRX_UNDO_FSEG_HEADER`：本Undo页面链表对应的段的`Segment Header`信息（就是我们上一节介绍的那个10字节结构，通过这个信息可以找到该段对应的`INODE Entry`）。
- `TRX_UNDO_PAGE_LIST`：Undo页面链表的基节点。

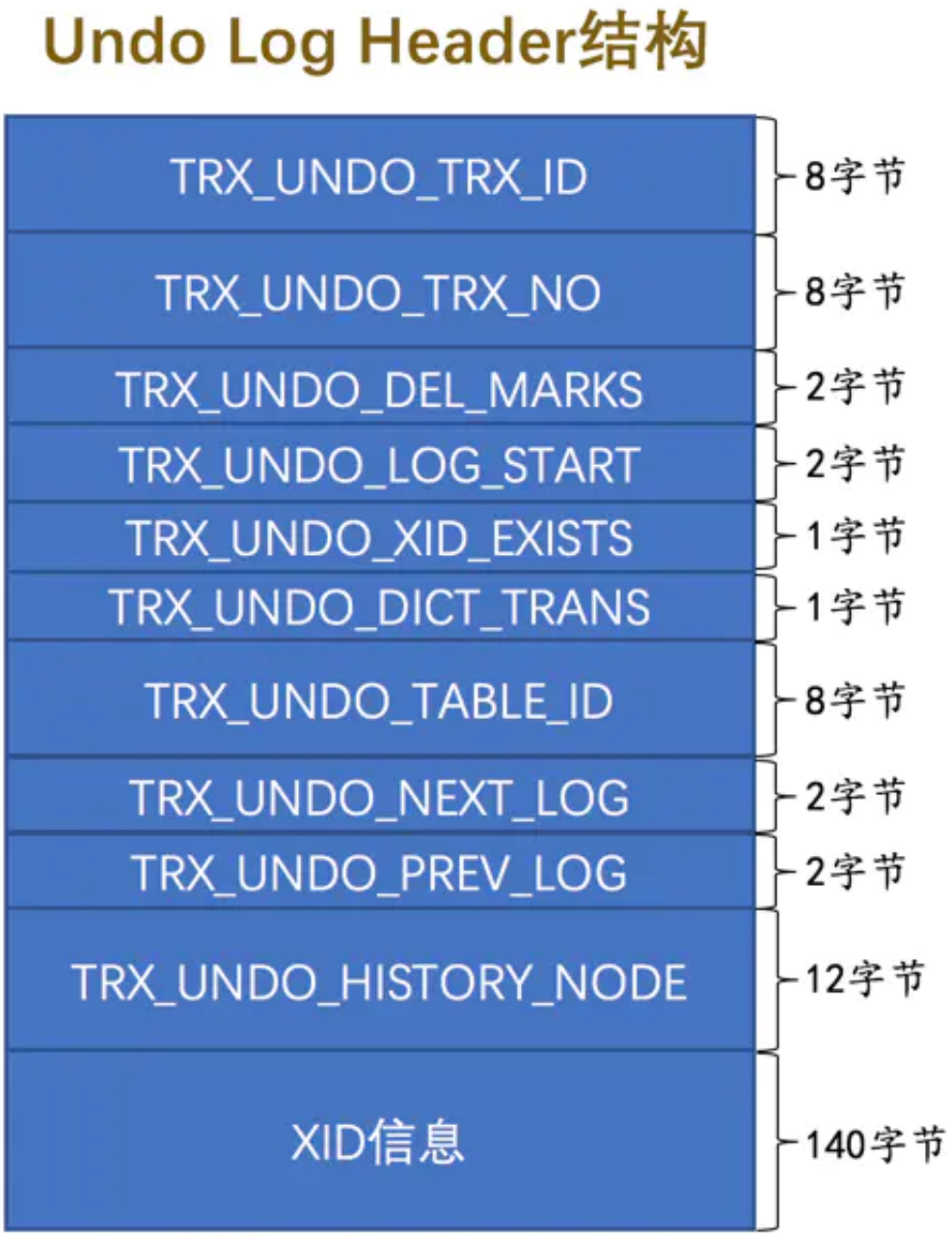
  我们上边说Undo页面的`Undo Page Header`部分有一个12字节大小的`TRX_UNDO_PAGE_NODE`属性，这个属性代表一个`List Node`结构。每一个Undo页面都包含`Undo Page Header`结构，这些页面就可以通过这个属性连成一个链表。这个`TRX_UNDO_PAGE_LIST`属性代表着这个链表的基节点，当然这个基节点只存在于Undo页面链表的第一个页面，也就是`first undo page`中。

Undo Log Header

一个事务在向Undo页面中写入undo日志时的方式是十分简单暴力的，就是直接往里怼，写完一条紧接着写另一条，各条undo日志之间是亲密无间的。写完一个Undo页面后，再从段里申请一个新页面，然后把这个页面插入到Undo页面链表中，继续往这个新申请的页面中写。设计InnoDB的大叔认为同一个事务向一个Undo页面链表中写入的undo日志算是一个组，比方说我们上边介绍的`trx 1`由于会分配3个Undo页面链表，也就会写入3个组的undo日志；`trx 2`由于会分配2个Undo页面链表，也就会写入2个组的undo日志。在每写入一组undo日志时，都会在这组undo日志前先记录一下关于这个组的一些属性，设计InnoDB的大叔把存储这些属性的地方称之为`Undo Log Header`。所以Undo页面链表的第一个页面在真正写入undo日志前，其实都会被填充`Undo Page Header`、`Undo Log Segment Header`、`Undo Log Header`这3个部分，如图所示：



这个`Undo Log Header`具体的结构如下：



哇唔，映入眼帘的又是一大坨属性，我们先大致看一下它们都是啥意思：

- `TRX_UNDO_TRX_ID`：生成本组`undo日志`的事务`id`。
- `TRX_UNDO_TRX_NO`：事务提交后生成的一个需要序号，使用此序号来标记事务的提交顺序（先提交的此序号小，后提交的此序号大）。
- `TRX_UNDO_DEL_MARKS`：标记本组`undo`日志中是否包含由于`Delete mark`操作产生的`undo日志`。
- `TRX_UNDO_LOG_START`：表示本组`undo`日志中第一条`undo日志`的在页面中的偏移量。
- `TRX_UNDO_XID_EXISTS`：本组`undo日志`是否包含XID信息。

  小贴士： 本书不会讲述更多关于XID是个什么东东，有兴趣的同学可以到搜索引擎或者文档中搜一搜哈。
- `TRX_UNDO_DICT_TRANS`：标记本组`undo日志`是不是由DDL语句产生的。

- TRX_UNDO_TABLE_ID：如果TRX_UNDO_DICT_TRANS为真，那么本属性表示DDL语句操作的表的table id。
- TRX_UNDO_NEXT_LOG：下一组的undo日志在页面中开始的偏移量。
- TRX_UNDO_PREV_LOG：上一组的undo日志在页面中开始的偏移量。

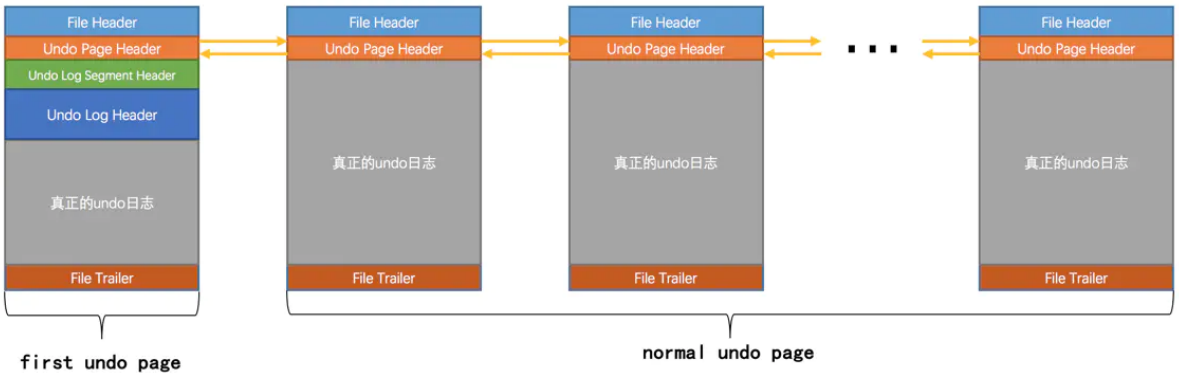
  小贴士： 一般来说一个Undo页面链表只存储一个事务执行过程中产生的一组undo日志，但是在某些情况下，可能会在一个事务提交之后，之后开启的事务重复利用这个Undo页面链表，这样就会导致一个Undo页面中可能存放多组Undo日志，TRX_UNDO_NEXT_LOG和TRX_UNDO_PREV_LOG就是用来标记下一组和上一组undo日志在页面中的偏移量的。关于什么时候重用Undo页面链表，怎么重用这个链表我们稍后会详细说明的，现在先理解TRX_UNDO_NEXT_LOG和TRX_UNDO_PREV_LOG这两个属性的意思就好了。

- TRX_UNDO_HISTORY_NODE：一个12字节的List Node结构，代表一个称之为History链表的节点。

小结

对于没有被重用的Undo页面链表来说，链表的第一个页面，也就是first undo page在真正写入undo日志前，会填充Undo Page Header、Undo Log Segment Header、Undo Log Header这3个部分，之后才开始正式写入undo日志。对于其他的页面来说，也就是normal undo page在真正写入undo日志前，只会填充Undo Page Header。链表的List Base Node存放到first undo page的Undo Log Segment Header部分，List Node信息存放到每一个Undo页面的undo Page Header部分，所以画一个Undo页面链表的示意图就是这样：

undo页面链表示意图



重用Undo页面

我们前边说为了能提高并发执行的多个事务写入undo日志的性能，设计InnoDB的大叔决定为每个事务单独分配相应的Undo页面链表（最多可能单独分配4个链表）。但是这样也造成了一些问题，比

如其实大部分事务执行过程中可能只修改了一条或几条记录，针对某个Undo页面链表只产生了非常少的undo日志，这些undo日志可能只占用一丢丢存储空间，每开启一个事务就新创建一个Undo页面链表（虽然这个链表中只有一个页面）来存储这么一丢丢undo日志岂不是太浪费了么？的确是挺浪费，于是设计InnoDB的大叔本着勤俭节约的优良传统，决定在事务提交后在某些情况下重用该事务的Undo页面链表。一个Undo页面链表是否可以被重用的条件很简单：

- 该链表中只包含一个Undo页面。

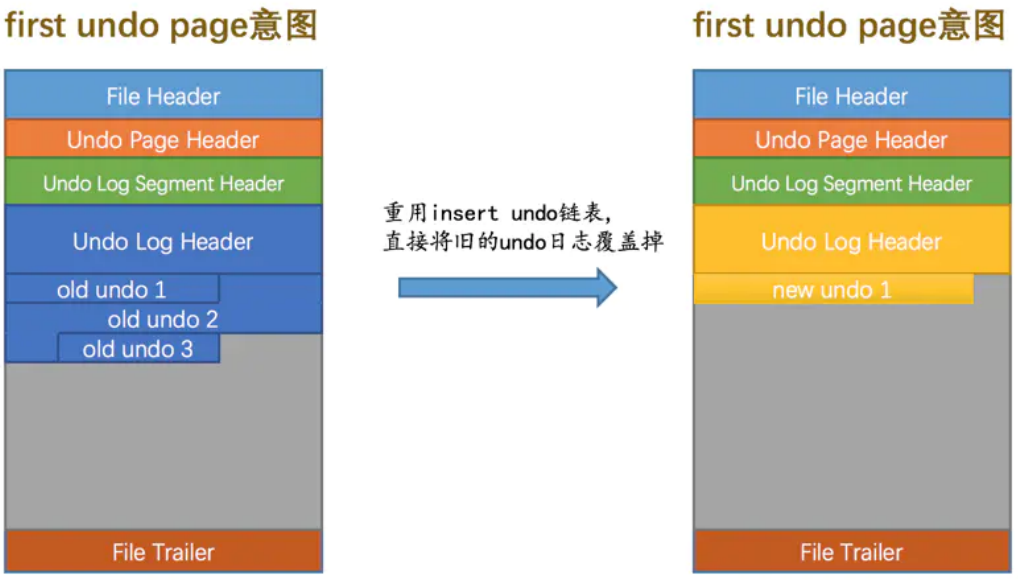
  如果一个事务执行过程中产生了非常多的undo日志，那么它可能申请非常多的页面加入到Undo页面链表中。在该事物提交后，如果将整个链表中的页面都重用，那就意味着即使新的事务并没有向该Undo页面链表中写入很多undo日志，那该链表中也得维护非常多的页面，那些用不到的页面也不能被别的事务所使用，这样就造成了另一种浪费。所以设计InnoDB的大叔们规定，只有在Undo页面链表中只包含一个Undo页面时，该链表才可以被下一个事务所重用。

- 该Undo页面已经使用的空间小于整个页面空间的3/4。

我们前边说过，Undo页面链表按照存储的undo日志所属的大类可以被分为insert undo链表和update undo链表两种，这两种链表在被重用时的策略也是不同的，我们分别看一下：

- insert undo链表

  insert undo链表中只存储类型为TRX_UNDO_INSERT_REC的undo日志，这种类型的undo日志在事务提交之后就没用了，就可以被清除掉。所以在某个事务提交后，重用这个事务的insert undo链表（这个链表中只有一个页面）时，可以直接把之前事务写入的一组undo日志覆盖掉，从头开始写入新事务的一组undo日志，如下图所示：



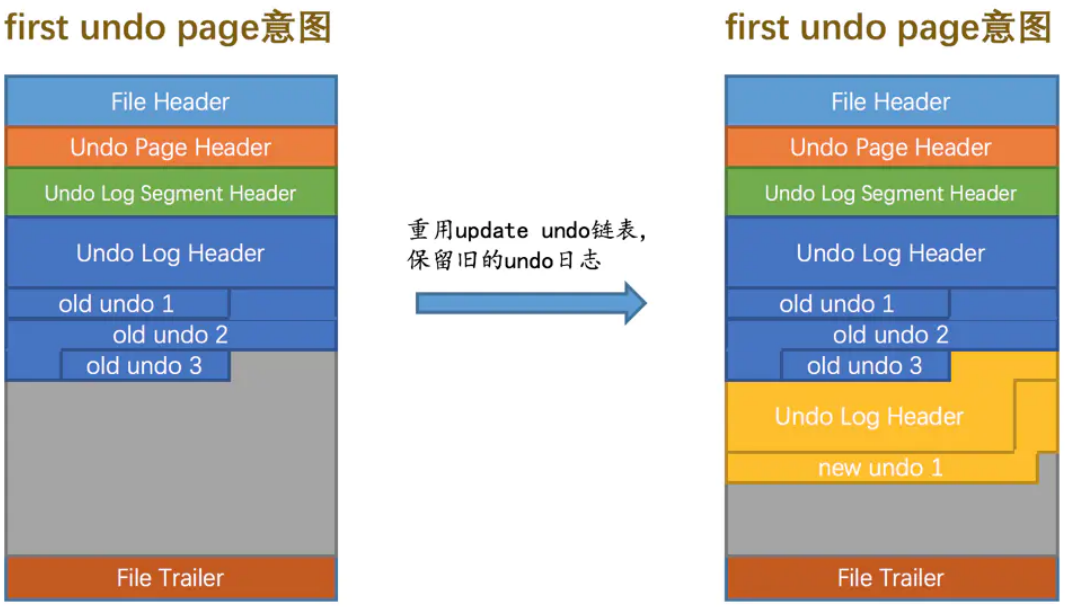
如图所示，假设有一个事务使用的insert undo链表，到事务提交时，只向insert undo链表中插入了3条undo日志，这个insert undo链表只申请了一个Undo页面。假设此刻该页面已使用的空间小于整个页面大小的3/4，那么下一个事务就可以重用这个insert undo链表（链表中只

有一个页面）。假设此时有一个新事务重用了该insert undo链表，那么可以直接把旧的一组undo日志覆盖掉，写入一组新的undo日志。

> 小贴士： 当然，在重用Undo页面链表写入新的一组undo日志时，不仅会写入新的Undo Log Header，还会适当调整Undo Page Header、Undo Log Segment Header、Undo Log Header中的一些属性，比如TRX_UNDO_PAGE_START、TRX_UNDO_PAGE_FREE等等等等，这些我们就不具体唠叨了。

- update undo链表

  在一个事务提交后，它的update undo链表中的undo日志也不能立即删除掉（这些日志用于MVCC，我们后边会说的）。所以如果之后的事务想重用update undo链表时，就不能覆盖之前事务写入的undo日志。这样就相当于在同一个Undo页面中写入了多组的undo日志，效果看起来就是这样：

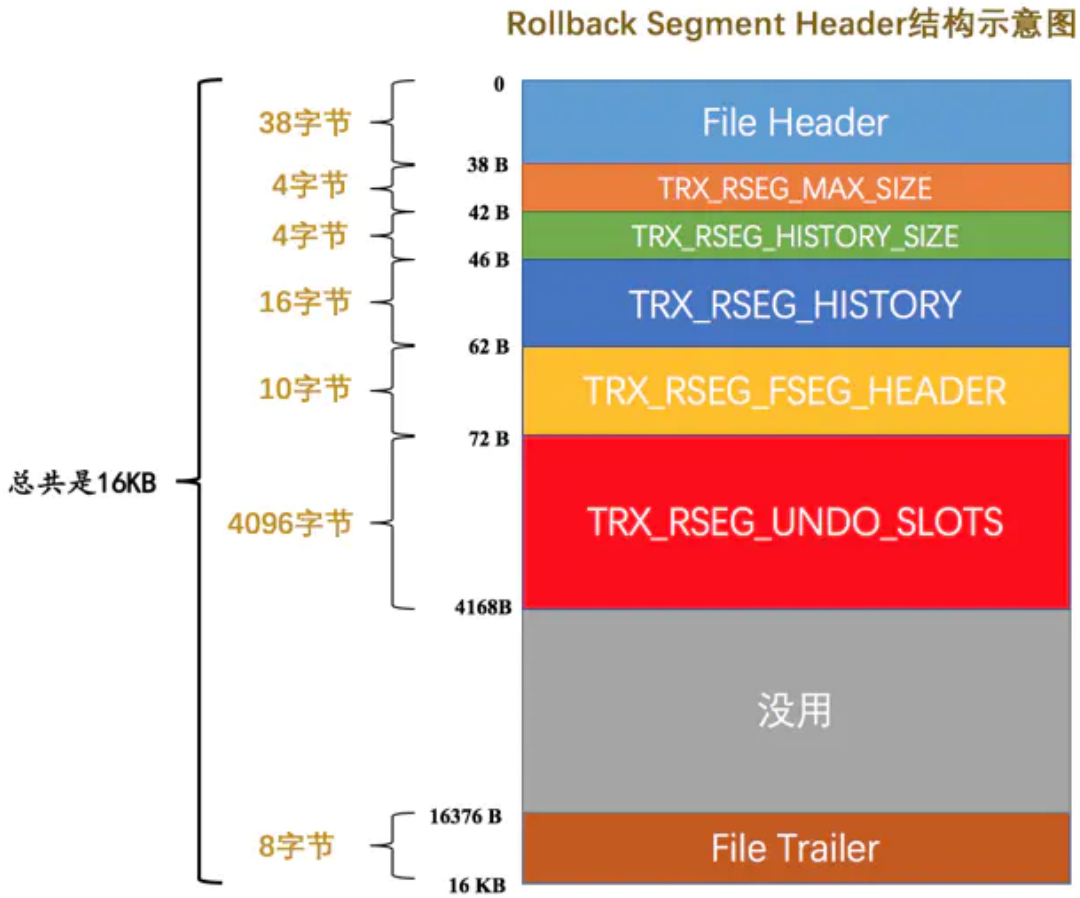


回滚段

回滚段的概念

我们现在知道一个事务在执行过程中最多可以分配4个Undo页面链表，在同一时刻不同事务拥有的Undo页面链表是不一样的，所以在同一时刻系统里其实可以有许许多多个Undo页面链表存在。为了更好的管理这些链表，设计InnoDB的大叔又设计了一个称之为Rollback Segment Header的页面，在这个页面中存放了各个Undo页面链表的frist undo page的页号，他们把这些页号称之为undo slot。我们可以这样理解，每个Undo页面链表都相当于是一个班，这个链表的first undo page就相当于这个班的班长，找到了这个班的班长，就可以找到班里的其他同学（其他同学相当于normal undo page）。有时候学校需要向这些班级传达一下精神，就需要把班长都召集在会议室，这个Rollback Segment Header就相当于是一个会议室。

我们看一下这个称之为Rollback Segment Header的页面长啥样（以默认的16KB为例）：



设计InnoDB的大叔规定，每一个Rollback Segment Header页面都对应着一个段，这个段就称为Rollback Segment，翻译过来就是回滚段。与我们之前介绍的各种段不同的是，这个Rollback Segment里其实只有一个页面（这可能是设计InnoDB的大叔们的一种洁癖，他们可能觉得为了某个目的去分配页面的话都得先申请一个段，或者他们觉得虽然目前版本的MySQL里Rollback Segment里其实只有一个页面，但可能之后的版本里会增加页面也说不定）。

了解了Rollback Segment的含义之后，我们再来看看这个称之为Rollback Segment Header的页面的各个部分的含义都是啥意思：

- TRX_RSEG_MAX_SIZE：本Rollback Segment中管理的所有Undo页面链表中的Undo页面数量之和的最大值。换句话说，本Rollback Segment中所有Undo页面链表中的Undo页面数量之和不能超过TRX_RSEG_MAX_SIZE代表的值。

  该属性的值默认为无限大，也就是我们想写多少Undo页面都可以。

  > 小贴士： 无限大其实也只是个夸张的说法，4个字节能表示最大的数也就是0xFFFFFFFF，但是我们之后会看到，0xFFFFFFFF这个数有特殊用途，所以实际上TRX_RSEG_MAX_SIZE的值为0xFFFFFFFE。

- TRX_RSEG_HISTORY_SIZE：History链表占用的页面数量。
- TRX_RSEG_HISTORY：History链表的基节点。
- TRX_RSEG_FSEG_HEADER：本Rollback Segment对应的10字节大小的Segment Header结构，通过

它可以找到本段对应的INODE Entry。

- TRX_RSEG_UNDO_SLOTS：各个Undo页面链表的first undo page的页号集合，也就是undo slot集合。

  一个页号占用4个字节，对于16KB大小的页面来说，这个TRX_RSEG_UNDO_SLOTS部分共存储了1024个undo slot，所以共需1024 × 4 = 4096个字节。

从回滚段中申请Undo页面链表

初始情况下，由于未向任何事务分配任何Undo页面链表，所以对于一个Rollback Segment Header页面来说，它的各个undo slot都被设置成了一个特殊的值：FIL_NULL（对应的十六进制就是0xFFFFFFFF），表示该undo slot不指向任何页面。

随着时间的流逝，开始有事务需要分配Undo页面链表了，就从回滚段的第一个undo slot开始，看看该undo slot的值是不是FIL_NULL：

- 如果是FIL_NULL，那么在表空间中新创建一个段（也就是Undo Log Segment），然后从段里申请一个页面作为Undo页面链表的first undo page，然后把该undo slot的值设置为刚刚申请的这个页面的页号，这样也就意味着这个undo slot被分配给了这个事务。
- 如果不是FIL_NULL，说明该undo slot已经指向了一个undo链表，也就是说这个undo slot已经被别的事务占用了，那就跳到下一个undo slot，判断该undo slot的值是不是FIL_NULL，重复上边的步骤。

一个Rollback Segment Header页面中包含1024个undo slot，如果这1024个undo slot的值都不为FIL_NULL，这就意味着这1024个undo slot都已经名花有主（被分配给了某个事务），此时由于新事务无法再获得新的Undo页面链表，就会回滚这个事务并且给用户报错：

```
Too many active concurrent transactions
```

用户看到这个错误，可以选择重新执行这个事务（可能重新执行时有别的事务提交了，该事务就可以被分配Undo页面链表了）。

当一个事务提交时，它所占用的undo slot有两种命运：

- 如果该undo slot指向的Undo页面链表符合被重用的条件（就是我们上边说的Undo页面链表只占用一个页面并且已使用空间小于整个页面的3/4）。

  该undo slot就处于被缓存的状态，设计InnoDB的大叔规定这时该Undo页面链表的TRX_UNDO_STATE属性（该属性在first undo page的Undo Log Segment Header部分）会被设置为TRX_UNDO_CACHED。

  被缓存的undo slot都会被加入到一个链表，根据对应的Undo页面链表的类型不同，也会被加入到不同的链表：

  - 如果对应的Undo页面链表是insert undo链表，则该undo slot会被加入insert undo cached链表。

    - 如果对应的Undo页面链表是update undo链表，则该undo slot会被加入update undo cached链表。

  一个回滚段就对应着上述两个cached链表，如果有新事务要分配undo slot时，先从对应的cached链表中找。如果没有被缓存的undo slot，才会到回滚段的Rollback Segment Header页面中再去找。

- 如果该undo slot指向的Undo页面链表不符合被重用的条件，那么针对该undo slot对应的Undo页面链表类型不同，也会有不同的处理：
    - 如果对应的Undo页面链表是insert undo链表，则该Undo页面链表的TRX_UNDO_STATE属性会被设置为TRX_UNDO_TO_FREE，之后该Undo页面链表对应的段会被释放掉（也就意味着段中的页面可以被挪作他用），然后把该undo slot的值设置为FIL_NULL。
    - 如果对应的Undo页面链表是update undo链表，则该Undo页面链表的TRX_UNDO_STATE属性会被设置为TRX_UNDO_TO_PRUGE，则会将该undo slot的值设置为FIL_NULL，然后将本次事务写入的一组undo日志放到所谓的History链表中（需要注意的是，这里并不会将Undo页面链表对应的段给释放掉，因为这些undo日志还有用呢~）。

  小贴士：更多关于History链表的事我们稍后再说，稍安勿躁哈。

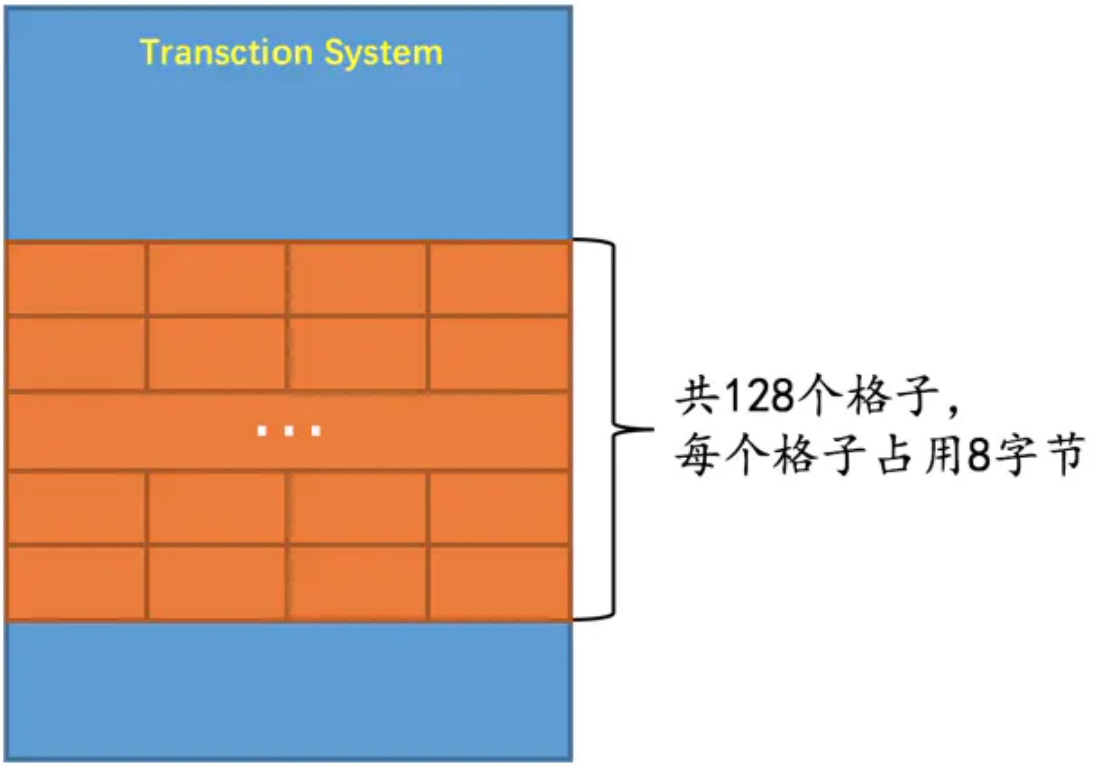
多个回滚段

我们说一个事务执行过程中最多分配4个Undo页面链表，而一个回滚段里只有1024个undo slot，很显然undo slot的数量有点少啊。我们即使假设一个读写事务执行过程中只分配1个Undo页面链表，那1024个undo slot也只能支持1024个读写事务同时执行，再多了就崩溃了。这就相当于会议室只能容下1024个班长同时开会，如果有几千人同时到会议室开会的话，那后来的那些班长就没地方坐了，只能等待前边的人开完会自己再进去开。

话说在InnoDB的早期发展阶段的确只有一个回滚段，但是设计InnoDB的大叔后来意识到了这个问题，咋解决这问题呢？会议室不够，多盖几个会议室不就得了。所以设计InnoDB的大叔一口气定义了128个回滚段，也就相当于有了128 × 1024 = 131072个undo slot。假设一个读写事务执行过程中只分配1个Undo页面链表，那么就可以同时支持131072个读写事务并发执行（这么多事务在一台机器上并发执行，还真没见过呢~）。

小贴士：只读事务并不需要分配Undo页面链表，MySQL 5.7中所有刚开启的事务默认都是只读事务，只有在事务执行过程中对记录做了某些改动时才会被升级为读写事务。

每个回滚段都对应着一个Rollback Segment Header页面，有128个回滚段，自然就要有128个Rollback Segment Header页面，这些页面的地址总得找个地方存一下吧！于是设计InnoDB的大叔在系统表空间的第5号页面的某个区域包含了128个8字节大小的格子：



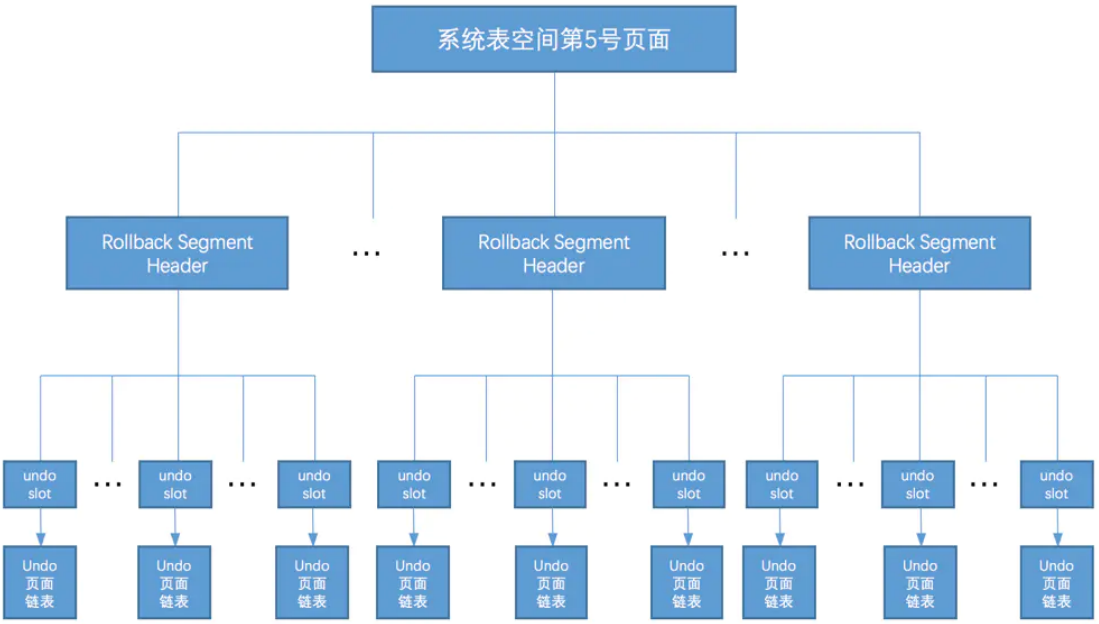
每个8字节的格子的构造就像这样：

如果所示，每个8字节的格子其实由两部分组成：

- 4字节大小的`Space ID`，代表一个表空间的ID。
- 4字节大小的`Page number`，代表一个页号。

也就是说每个8字节大小的格子相当于一个指针，指向某个表空间中的某个页面，这些页面就是`Rollback Segment Header`。这里需要注意的一点事，要定位一个`Rollback Segment Header`还需要知道对应的表空间ID，这也就意味着不同的回滚段可能分布在不同的表空间中。

所以通过上边的叙述我们可以大致清楚，在系统表空间的第`5`号页面中存储了128个`Rollback Segment Header`页面地址，每个`Rollback Segment Header`就相当于一个回滚段。在`Rollback Segment Header`页面中，又包含`1024`个`undo slot`，每个`undo slot`都对应一个`Undo`页面链表。我们画个示意图：



把图一画出来就清爽多了。

回滚段的分类

我们把这128个回滚段给编一下号，最开始的回滚段称之为第0号回滚段，之后依次递增，最后一个回滚段就称之为第127号回滚段。这128个回滚段可以被分成两大类：

- 第0号、第33～127号回滚段属于一类。其中第0号回滚段必须在系统表空间中（就是说第0号回滚段对应的Rollback Segment Header页面必须在系统表空间中），第33～127号回滚段既可以在系统表空间中，也可以在自己配置的undo表空间中，关于怎么配置我们稍后再说。

  如果一个事务在执行过程中由于对普通表的记录做了改动需要分配Undo页面链表时，必须从这一类的段中分配相应的undo slot。

- 第1～32号回滚段属于一类。这些回滚段必须在临时表空间（对应着数据目录中的ibtmp1文件）中。

  如果一个事务在执行过程中由于对临时表的记录做了改动需要分配Undo页面链表时，必须从这一类的段中分配相应的undo slot。

也就是说如果一个事务在执行过程中既对普通表的记录做了改动，又对临时表的记录做了改动，那么需要为这个记录分配2个回滚段，再分别到这两个回滚段中分配对应的undo slot。

不知道大家有没有疑惑，为啥要把针对普通表和临时表来划分不同种类的回滚段呢？这个还得从Undo页面本身说起，我们说Undo页面其实是类型为FIL_PAGE_UNDO_LOG的页面的简称，说到底它也是一个普通的页面。我们前边说过，在修改页面之前一定要先把对应的redo日志写上，这样在系统奔溃重启时才能恢复到奔溃前的状态。我们向Undo页面写入undo日志本身也是一个写页面的过程，设计InnoDB的大叔为此还设计了许多种redo日志的类型，比方说MLOG_UNDO_HDR_CREATE、MLOG_UNDO_INSERT、MLOG_UNDO_INIT等等等等，也就是说我们对Undo页面做

的任何改动都会记录相应类型的redo日志。但是对于临时表来说，因为修改临时表而产生的undo日志只需要在系统运行过程中有效，如果系统奔溃了，那么在重启时也不需要恢复这些undo日志所在的页面，所以在写针对临时表的Undo页面时，并不需要记录相应的redo日志。总结一下针对普通表和临时表划分不同种类的回滚段的原因：**在修改针对普通表的回滚段中的Undo页面时，需要记录对应的redo日志，而修改针对临时表的回滚段中的Undo页面时，不需要记录对应的redo日志。**

> 小贴士： 实际上在MySQL 5.7.21这个版本中，如果我们仅仅对普通表的记录做了改动，那么只会为该事务分配针对普通表的回滚段，不分配针对临时表的回滚段。但是如果我们仅仅对临时表的记录做了改动，那么既会为该事务分配针对普通表的回滚段，又会为其分配针对临时表的回滚段（不过分配了回滚段并不会立即分配undo slot，只有在真正需要Undo页面链表时才会去分配回滚段中的undo slot）。

为事务分配Undo页面链表详细过程

上边说了一大堆的概念，大家应该有一点点的小晕，接下来我们以事务对普通表的记录做改动为例，给大家梳理一下事务执行过程中分配Undo页面链表时的完整过程，

- 事务在执行过程中对普通表的记录首次做改动之前，首先会到系统表空间的第5号页面中分配一个回滚段（其实就是获取一个Rollback Segment Header页面的地址）。一旦某个回滚段被分配给了这个事务，那么之后该事务中再对普通表的记录做改动时，就不会重复分配了。

  使用传说中的round-robin（循环使用）方式来分配回滚段。比如当前事务分配了第0号回滚段，那么下一个事务就要分配第33号回滚段，下下个事务就要分配第34号回滚段，简单一点的说就是这些回滚段被轮着分配给不同的事务（就是这么简单粗暴，没啥好说的）。

- 在分配到回滚段后，首先看一下这个回滚段的两个cached链表有没有已经缓存了的undo slot，比如如果事务做的是INSERT操作，就去回滚段对应的insert undo cached链表中看看有没有缓存的undo slot；如果事务做的是DELETE操作，就去回滚段对应的update undo cached链表中看看有没有缓存的undo slot。如果有缓存的undo slot，那么就把这个缓存的undo slot分配给该事务。

- 如果没有缓存的undo slot可供分配，那么就要到Rollback Segment Header页面中找一个可用的undo slot分配给当前事务。

  从Rollback Segment Header页面中分配可用的undo slot的方式我们上边也说过了，就是从第0个undo slot开始，如果该undo slot的值为FIL_NULL，意味着这个undo slot是空闲的，就把这个undo slot分配给当前事务，否则查看第1个undo slot是否满足条件，依次类推，直到最后一个undo slot。如果这1024个undo slot都没有值为FIL_NULL的情况，就直接报错喽（一般不会出现这种情况）~

- 找到可用的undo slot后，如果该undo slot是从cached链表中获取的，那么它对应的Undo Log Segment已经分配了，否则的话需要重新分配一个Undo Log Segment，然后从该Undo Log Segment中申请一个页面作为Undo页面链表的first undo page。

- 然后事务就可以把undo日志写入到上边申请的Undo页面链表了！

对临时表的记录做改动的步骤和上述的一样，就不赘述了。不过需要再次强调一次，如果一个事务在执行过程中既对普通表的记录做了改动，又对临时表的记录做了改动，那么需要为这个记录分配2个回滚段。并发执行的不同事务其实也可以被分配相同的回滚段，只要分配不同的undo slot就可以了。

回滚段相关配置

配置回滚段数量

我们前边说系统中一共有128个回滚段，其实这只是默认值，我们可以通过启动参数innodb_rollback_segments来配置回滚段的数量，可配置的范围是1~128。但是这个参数并不会影响针对临时表的回滚段数量，针对临时表的回滚段数量一直是32，也就是说：

- 如果我们把innodb_rollback_segments的值设置为1，那么只会有1个针对普通表的可用回滚段，但是仍然有32个针对临时表的可用回滚段。
- - 如果我们把innodb_rollback_segments的值设置为2～33之间的数，效果和将其设置为1是一样的。
- 如果我们把innodb_rollback_segments设置为大于33的数，那么针对普通表的可用回滚段数量就是该值减去32。

配置undo表空间

默认情况下，针对普通表设立的回滚段（第0号以及第33~127号回滚段）都是被分配到系统表空间的。其中的第第0号回滚段是一直在系统表空间的，但是第33~127号回滚段可以通过配置放到自定义的undo表空间中。但是这种配置只能在系统初始化（创建数据目录时）的时候使用，一旦初始化完成，之后就不能再次更改了。我们看一下相关启动参数：

- 通过innodb_undo_directory指定undo表空间所在的目录，如果没有指定该参数，则默认undo表空间所在的目录就是数据目录。
- 通过innodb_undo_tablespaces定义undo表空间的数量。该参数的默认值为0，表明不创建任何undo表空间。

  第33~127号回滚段可以平均分布到不同的undo表空间中。

> 小贴士： 如果我们在系统初始化的时候指定了创建了undo表空间，那么系统表空间中的第0号回滚段将处于不可用状态。

比如我们在系统初始化时指定的innodb_rollback_segments为35，innodb_undo_tablespaces为2，这样就会将第33、34号回滚段分别分布到一个undo表空间中。

设立undo表空间的一个好处就是在undo表空间中的文件大到一定程度时，可以自动的将该undo表空间

截断（truncate）成一个小文件。而系统表空间的大小只能不断的增大，却不能截断。