Part 14 MySQL事务日志

undo日志，redo日志，undo-redo日志。

1. redo日志

所有的变更都必须先更新缓冲池中的数据，然后缓冲池中的脏页会以一定的频率被刷入磁盘（checkPoint机制）。通过缓冲池来优化CPU和磁盘之间的鸿沟，这样就可以保证整体的性能不会下降太快。

1.1 为什么需要REDO日志

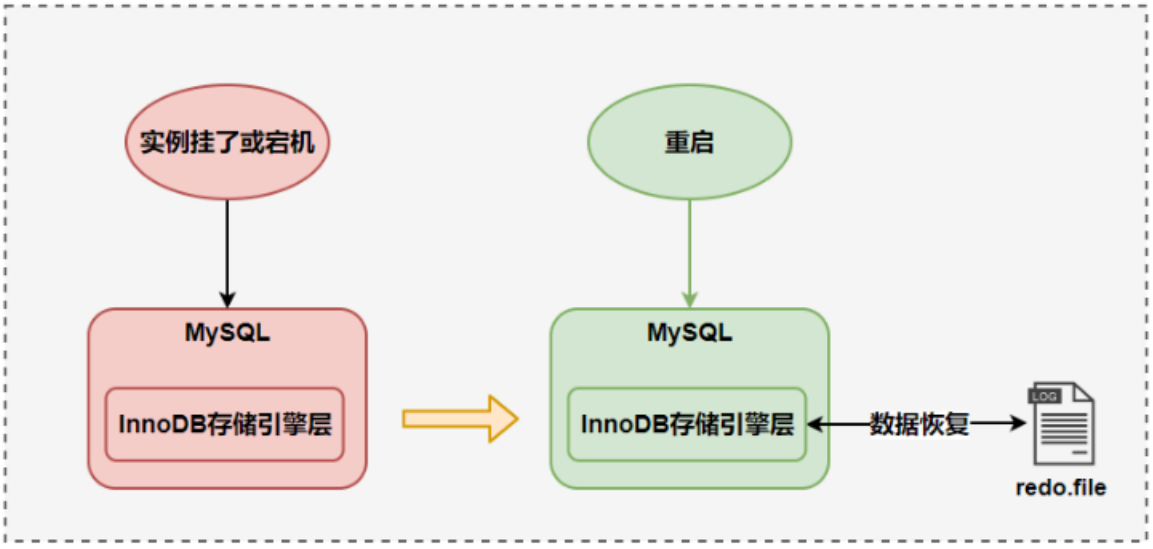
- 一方面，缓冲池可以帮助我们消除CPU和磁盘之间的鸿沟
- 另一方面，事务包含 `持久性` 的特性，就是说对于一个已经提交的事务，在事务提交后即使系统发生了崩溃，这个事务对数据库中所做的更改也不能丢失。

`一个简单的做法`：事务提交前写入磁盘，但是这个简单粗暴的做法有些问题。

- 修改量与刷新磁盘工作量严重不成比例
- 随机I/O刷新较慢

`另一个解决的思路`：事务提交后也不用立即写入磁盘，可以累计一定量或者checkpoint时写入。

InnoDB引擎的事务采用了WAL技术(`Write-Ahead Logging`)，这种技术的思想就是先写日志，再写磁盘，只有日志写入成功，才算事务提交成功，这里的日志就是redo log。

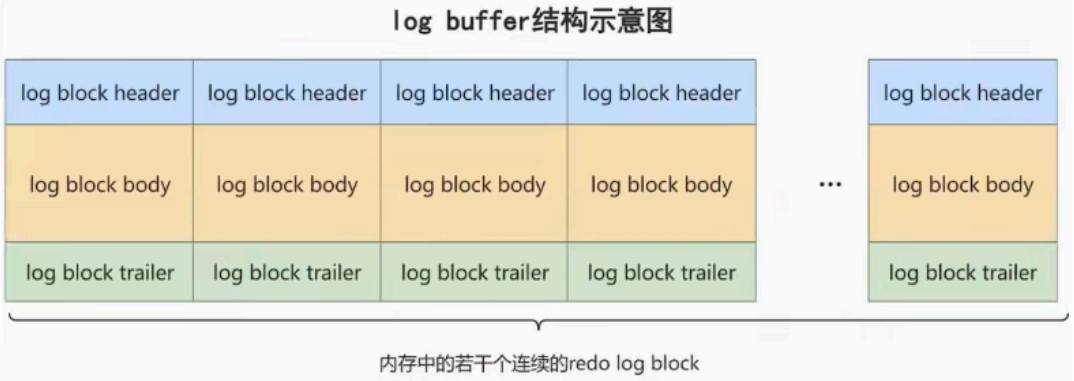


1.2 REDO日志的好处、特点

- 好处
  - redo日志降低了刷盘频率
  - 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，刷盘快。
- 特点
  - redo日志是顺序写入磁盘的
  - 事务执行过程中，redo log不断记录

1.3 redo的组成

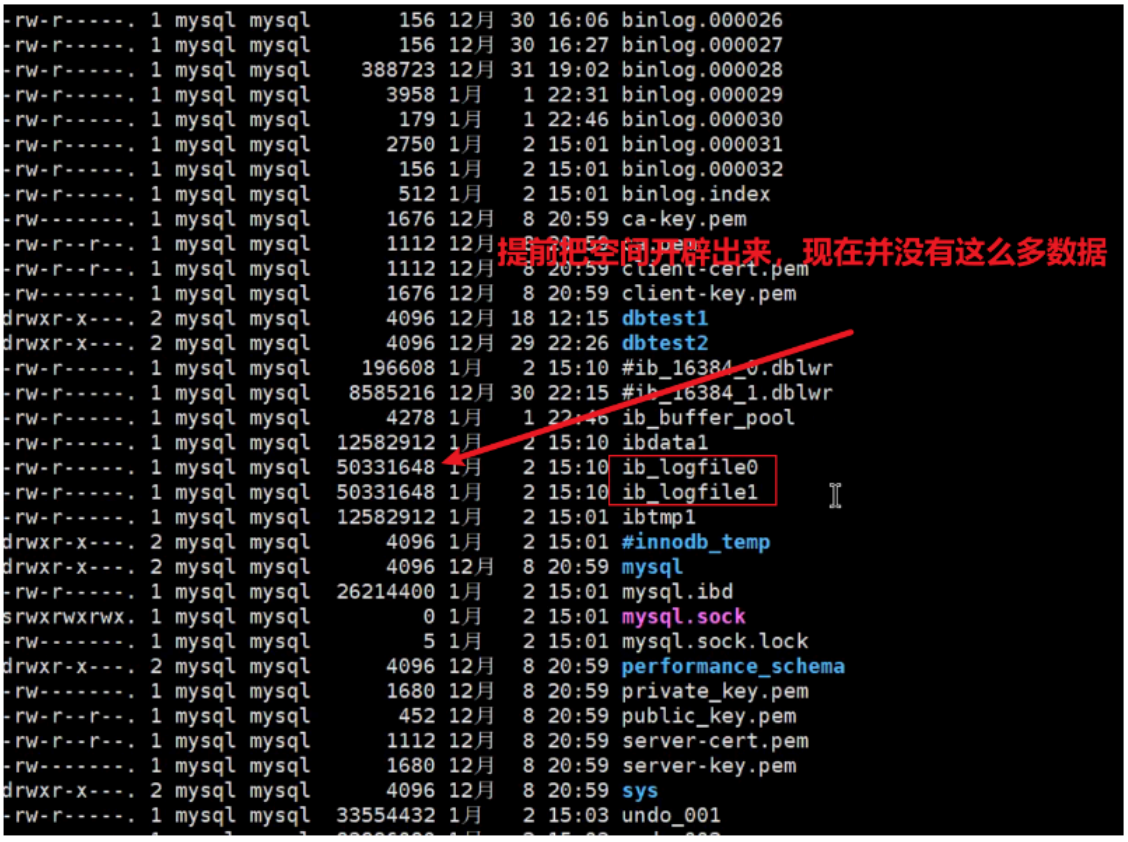
- 重做日志的缓冲 (redo log buffer)，保存在内存中，是易失的。
  - 在服务器启动时就向操作系统申请了一大片称之为redo log buffer的连续内存空间，翻译成中文就是redo日志缓冲区。这片内存空间被划分成若干个连续的redo log block。一个redo log block占用512字节大小。



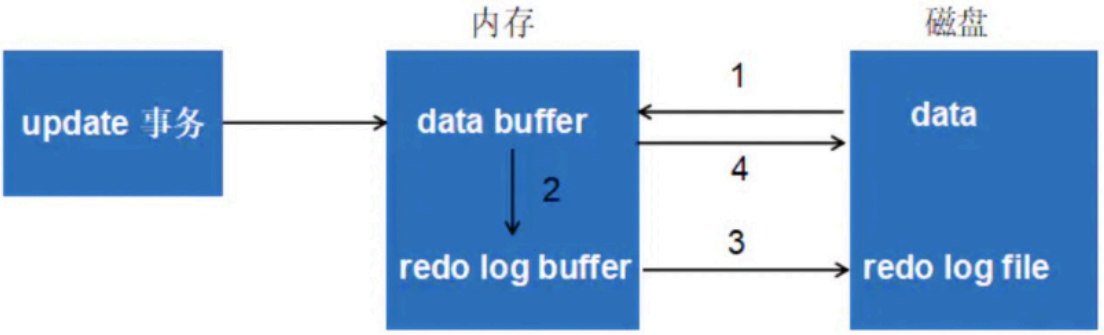
  参数设置：innodb_log_buffer_size：redo log buffer 大小，默认 16M，最大值是4096M，最小值为1M。

- 重做日志文件(redologfile)，保存在硬盘中，是持久的。

  REDO日志文件如图所示，其中的ib_logfile0和ib_logfile1即为redo log日志。



  以一个更新事务为例，redo log 流转过程，如下图所示：

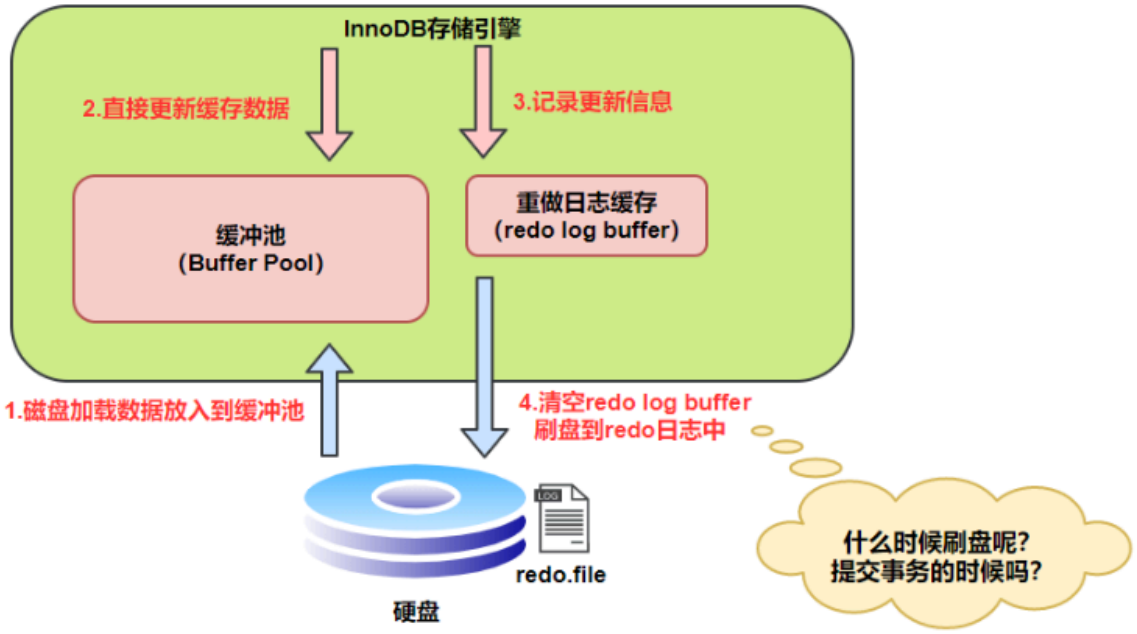


- 第1步：先将原始数据从磁盘中读入内存中来，修改数据的内存拷贝。
- 第2步：生成一条重做日志并写入redo log buffer，记录的是数据被修改后的值。
- 第3步：当事务commit时，将redo log buffer中的内容刷新到 redo log file，对 redo log file 采用追加写的方式。
- 第4步：定期将内存中修改的数据刷新到磁盘中。

> Write-Ahead Log(预先日志持久化)：在持久化一个数据页之前，先将内存中相应的日志页持久化。

1.4 redo log的刷盘策略

redo log的写入并不是直接写入磁盘的，InnoDB引擎会在写redo log的时候先写redo log buffer，之后以 `一定的频率刷` 入到真正的redo log file 中。这里的一定频率怎么看待呢？这就是我们要说的刷盘策略。

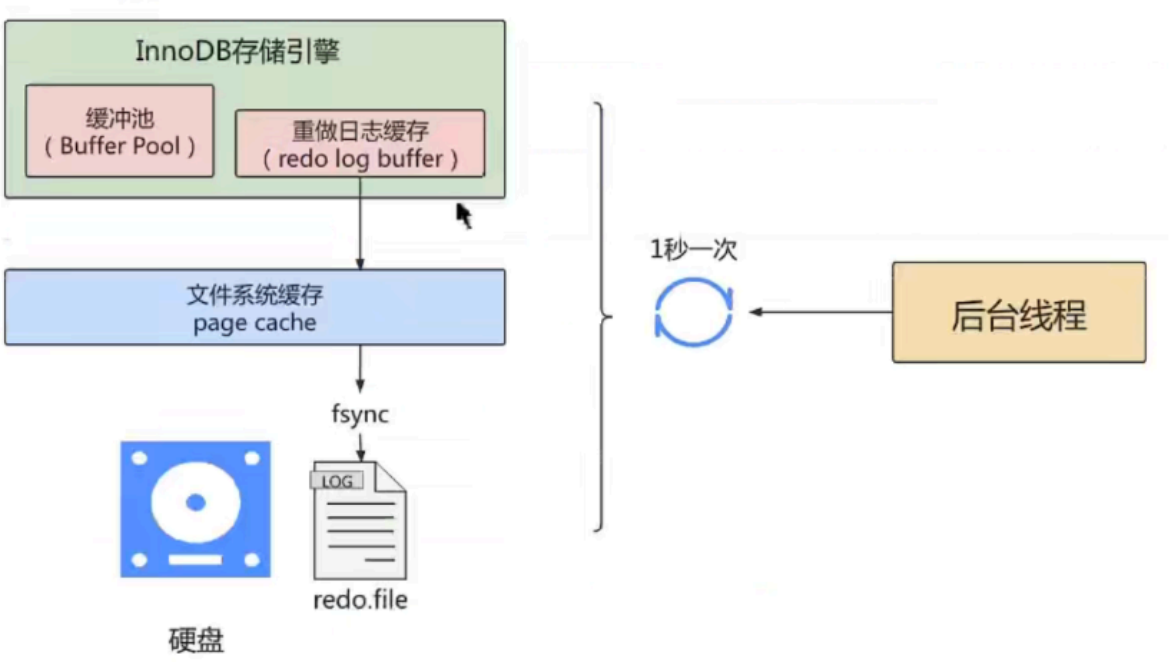


注意，redo log buffer刷盘到redo log file的过程并不是真正的刷到磁盘中去，只是刷入到 `文件系统缓存`（page cache）中去（这是现代操作系统为了提高文件写入效率做的一个优化），真正的写入会交给系统自己来决定（比如page cache足够大了）。那么对于InnoDB来说就存在一个问题，如果交给系统来同步，同样如果系统宕机，那么数据也丢失了（虽然整个系统宕机的概率还是比较小的）。

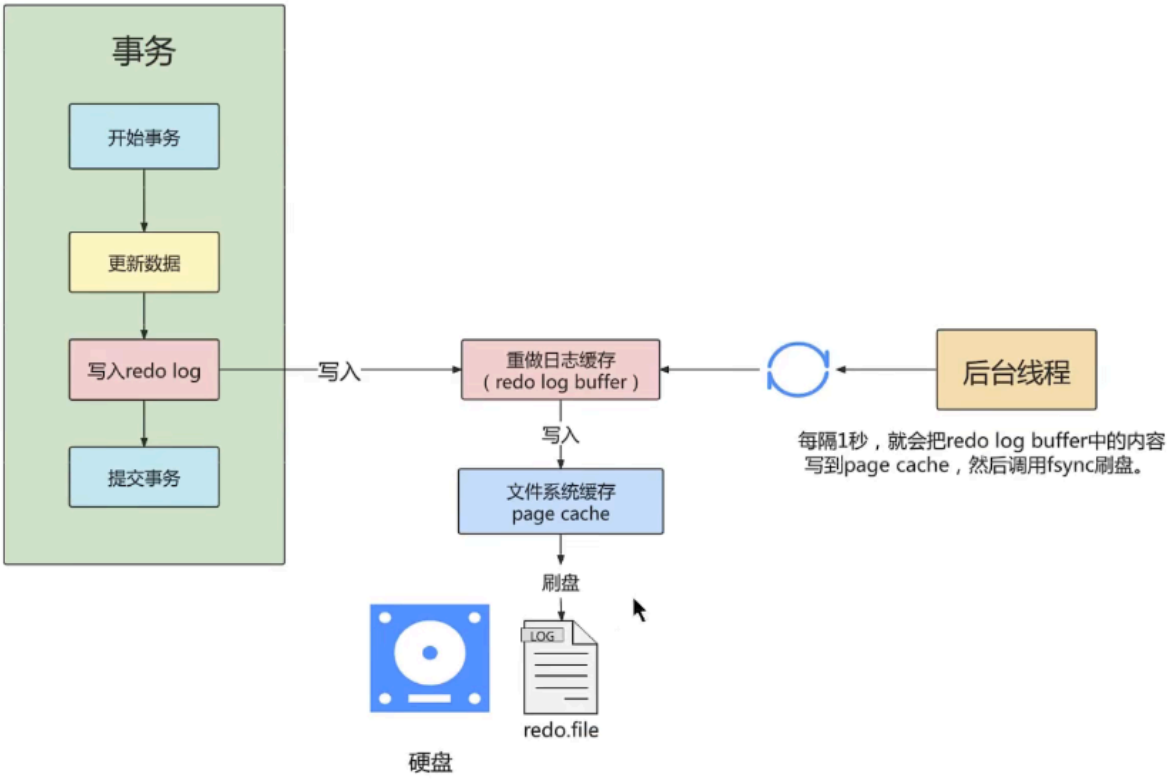
针对这种情况，InnoDB给出 `innodb_flush_log_at_trx_commit` 参数，该参数控制 commit提交事务时，如何将 redo log buffer 中的日志刷新到 redo log file 中。它支持三种策略：

- 设置为0：表示每次事务提交时不进行刷盘操作。（系统默认master thread每隔1s进行一次重做日志的同步）
- 设置为1：表示每次事务提交时都将进行同步，刷盘操作（默认值）
- 设置为2：表示每次事务提交时都只把 redo log buffer 内容写入 page cache，不进行同步。由os自己决定什么时候同步到磁盘文件。

另外，InnoDB存储引擎有一个后台线程，每隔1秒，就会把redo log buffer中的内容写到文件系统缓存(page cache)，然后调用刷盘操作。

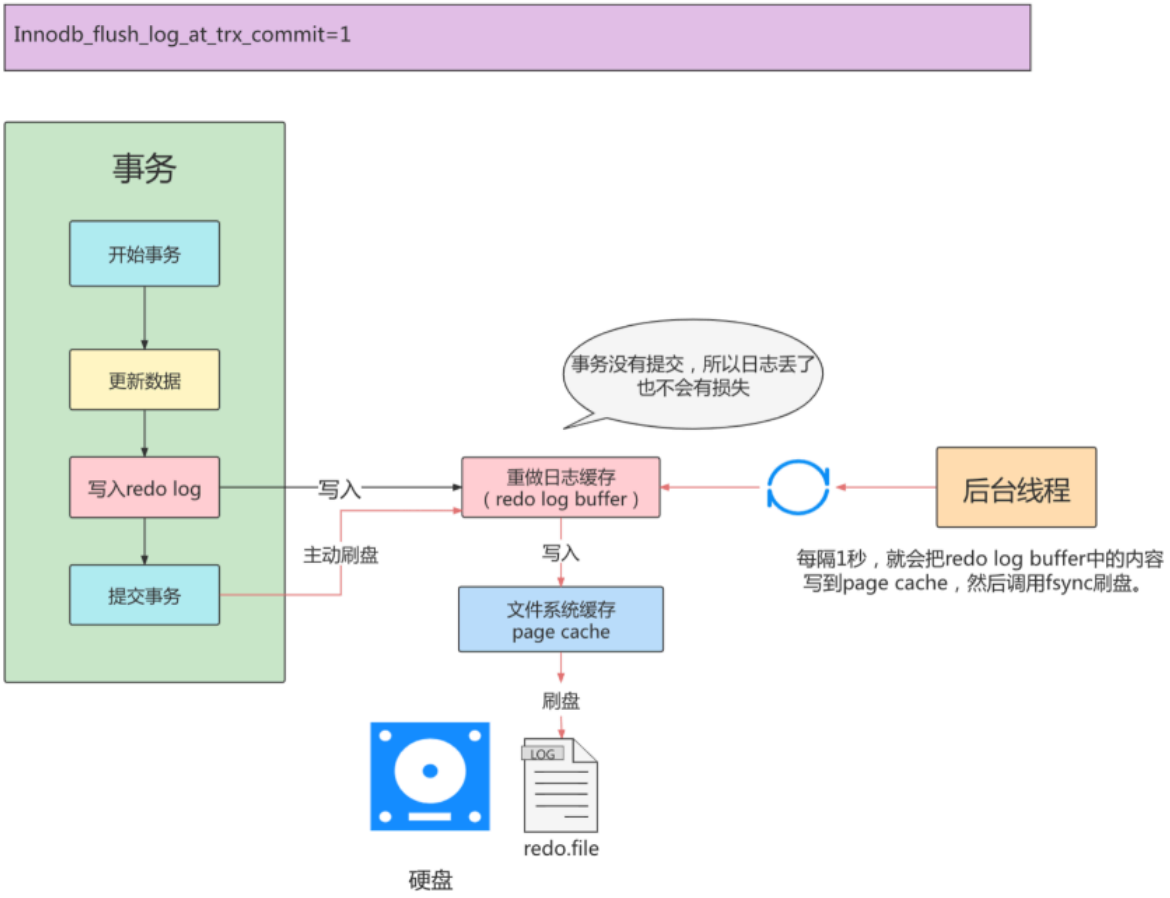


也就是说，一个没有提交事务的redo log记录，也可能会刷盘。因为在事务执行过程redo log记录是会写入redo log buffer中，这些redo log记录会被后台线程刷盘。



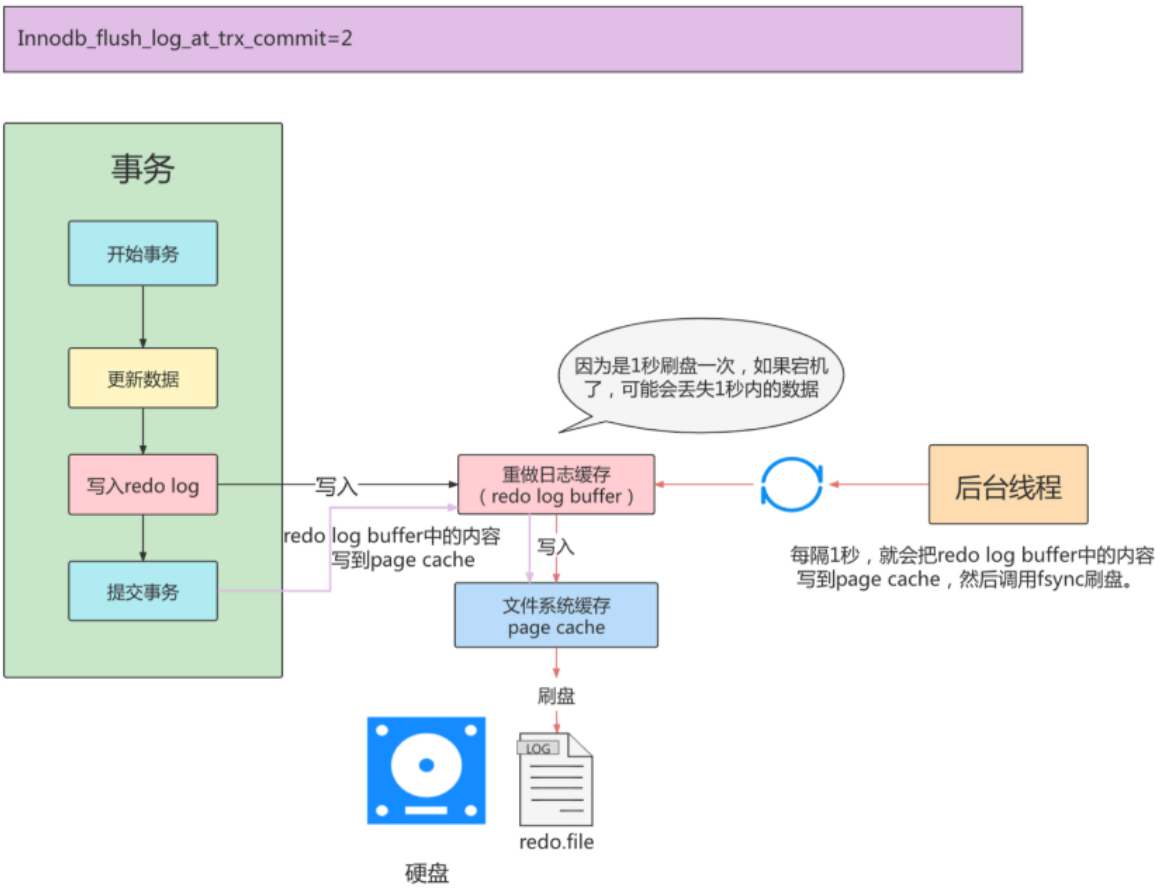
除了后台线程每秒`1次`的轮询操作，还有一种情况，当`redo log buffer`占用的空间即将达到`innodb_log_buffer_size`(这个参数默认是16M)的一半的时候，后台线程会主动刷盘。

1.5 不同刷盘策略演示



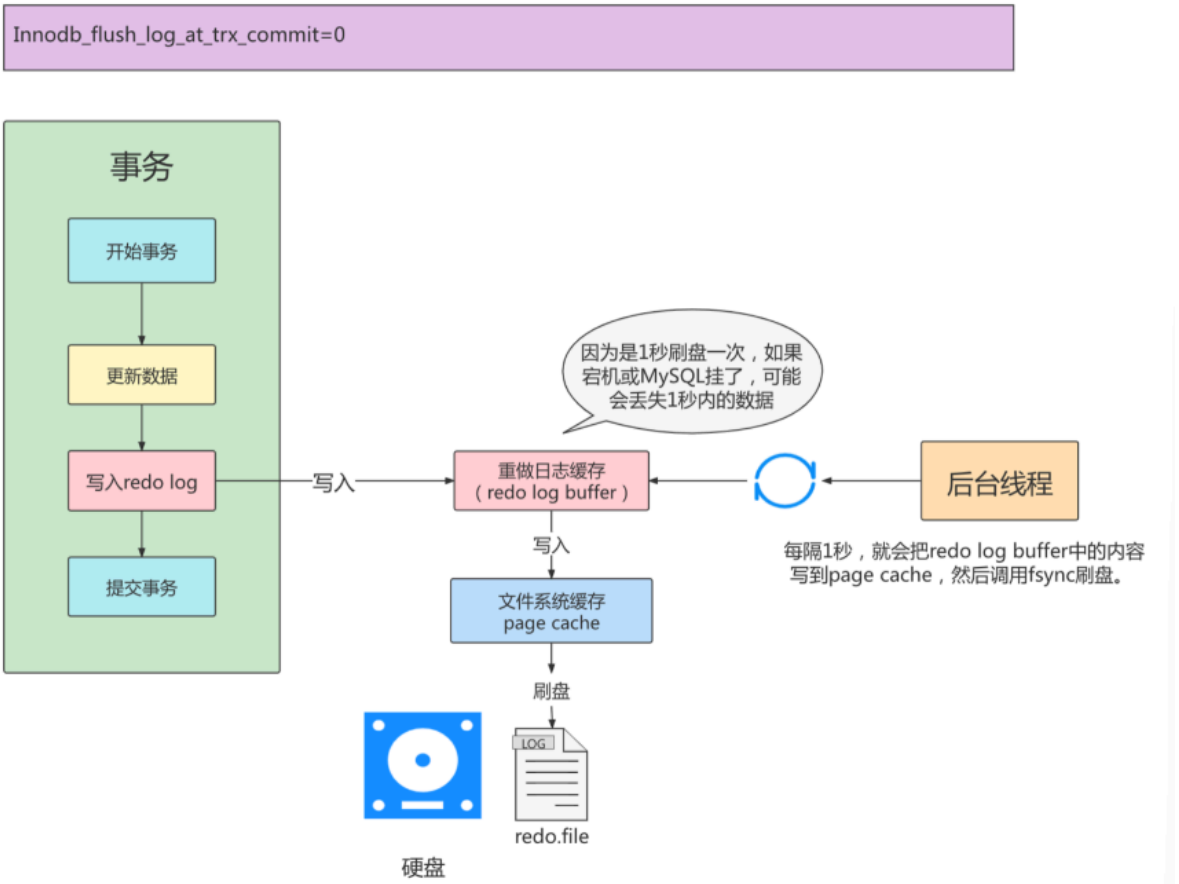
除了1秒刷盘，提交了也刷盘。效率差一些。

> 小结：innodb_flush_log_at_trx_commit=1
>
> 为1时，只要事务提交成功,redo log记录就一定在硬盘里，**不会有任何数据丢失。**
>
> 如果事务执行期间MySQL挂了或宕机，这部分日志丢了，但是事务并没有提交，所以日志丢了也不会有损失。可以保证ACID的D，数据绝对不会丢失，但是效率最差的。
>
> 建议使用默认值，虽然操作系统宕机的概率理论小于数据库宕机的概率，但是一般既然使用了事务，那么数据的安全相对来说更重要些。



除了1s 强制刷盘，page cache 由系统决定啥时候刷盘

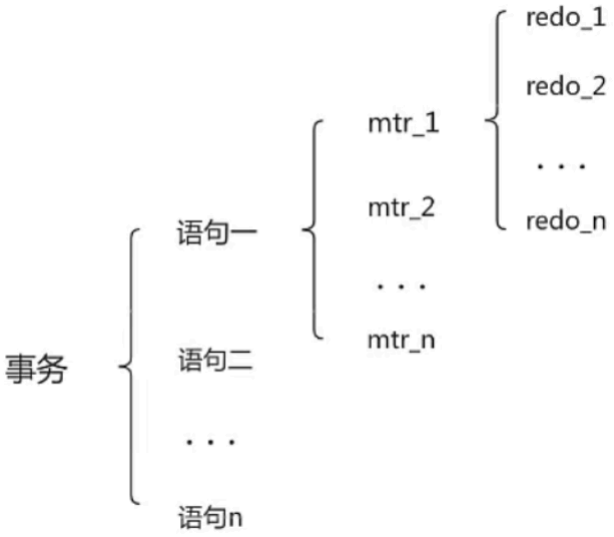
> 小结: innodb_flush_log_at_trx_commit=2
>
> 为2时，只要事务提交成功，`redo log buffer`中的内容只写入文件系统缓存（page cache）。
>
> 如果仅仅只是`MySQL`挂了不会有任何数据丢失，但是操作系统宕机可能会有`1`秒数据的丢失，这种情况下无法满足ACID中的D。但是数值2肯定是效率最高的。



1.6 写入redo log buffer 过程

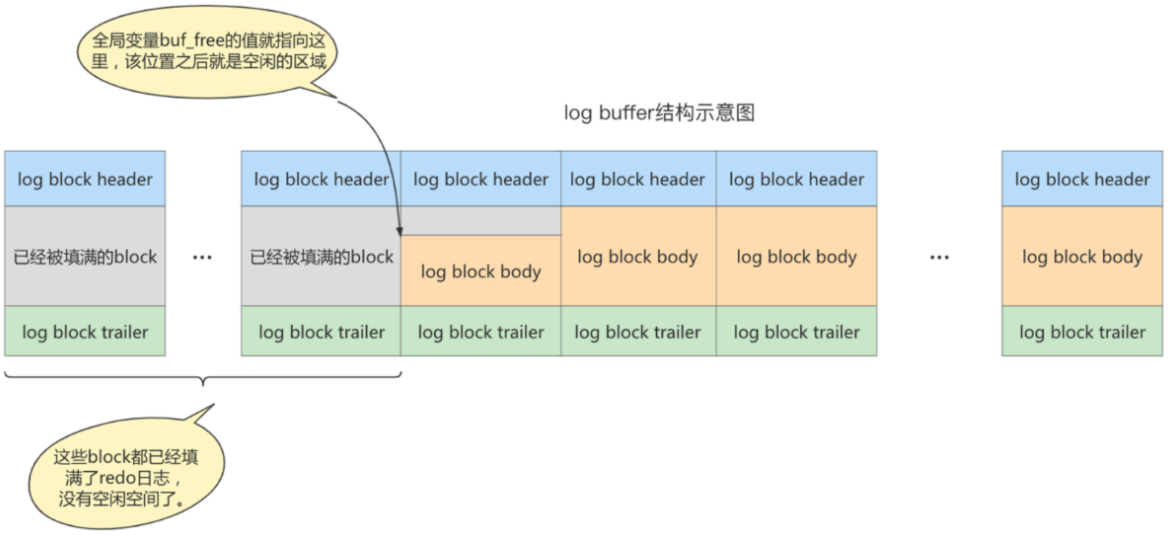
补：MySQL把对底层页面中的一次原子访问的过程称之为一个`Mini-Transaction`，简称`mtr`。一个所谓的`mtr`可以包含一组redo日志，在进行崩溃恢复时这一组`redo`日志作为一个不可分割的整体。

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。



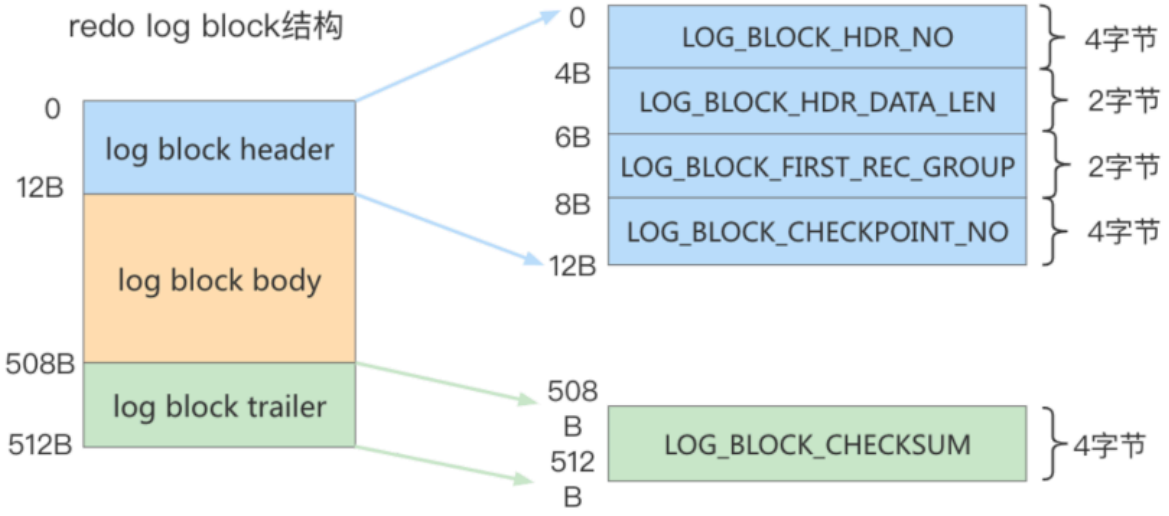
1.6.1 redo 日志写入log buffer

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`InnoDB`的设计者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。



1.6.2 redo log block的结构图

一个redo log block是由`日志头`、`日志体`、`日志尾`组成。日志头占用12字节，日志尾占用8字节，所以一个block真正能存储的数据就是512-12-8=492字节。（512字节和磁盘扇区大小保持一致）



1.7 redo log file

1.7.1 相关参数设置

- `innodb_log_group_home_dir`：指定 redo log 文件组所在的路径，默认值为`./`，表示在数据库的数据目录下。MySQL的默认数据目录（`var/lib/mysql`）下默认有两个名为`ib_logfile0`和`ib_logfile1`的文件，log buffer中的日志默认情况下就是刷新到这两个磁盘文件中。此redo日志文件位置还可以修改。

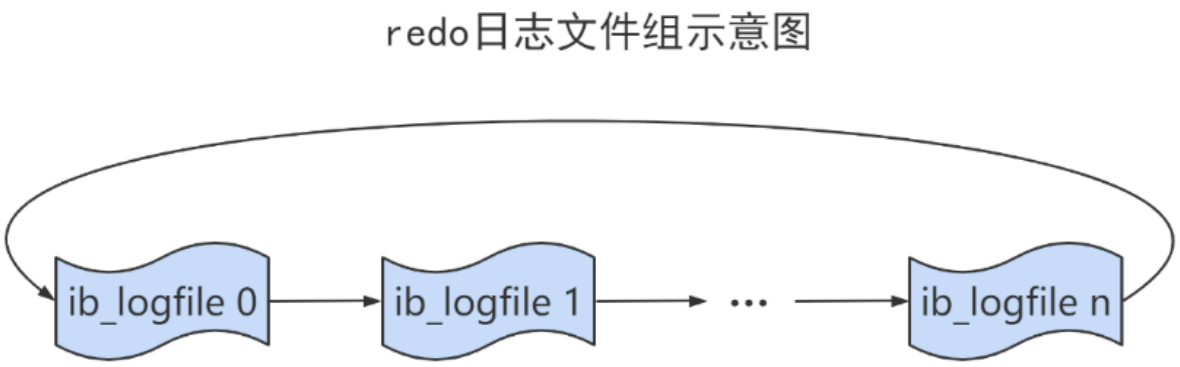
- `innodb_log_files_in_group`：指明redo log file的个数，命名方式如：ib_logfile0，iblogfile1...iblogfilen。默认2个，最大100个。
- `innodb_flush_log_at_trx_commit`：控制 redo log 刷新到磁盘的策略，默认为1。
- `innodb_log_file_size`：单个 redo log 文件设置大小，默认值为 `48M` 。最大值为512G，注意最大值指的是整个 redo log 系列文件之和，即（innodb_log_files_in_group * innodb_log_file_size ）不能大于最大值512G。

> 在数据库实例更新比较频繁的情况下，可以适当加大 redo log组数和大小。但也不推荐redo log 设置过大，在MySQL崩溃恢复时会重新执行REDO日志中的记录。

1.7.2 日志文件组

从上边的描述中可以看到，磁盘上的`redo`日志文件不只一个，而是以一个`日志文件组`的形式出现的。这些文件以`ib_logfile[数字]`（`数字`可以是0、1、2...）的形式进行命名，每个的redo日志文件大小都是一样的。

在将redo日志写入日志文件组时，是从`ib_logfile0`开始写，如果`ib_logfile0`写满了，就接着`ib_logfile1`写。同理,`ib_logfile1`.写满了就去写`ib_logfile2`，依此类推。如果写到最后一个文件该咋办?那就重新转到`ib_logfile0`继续写



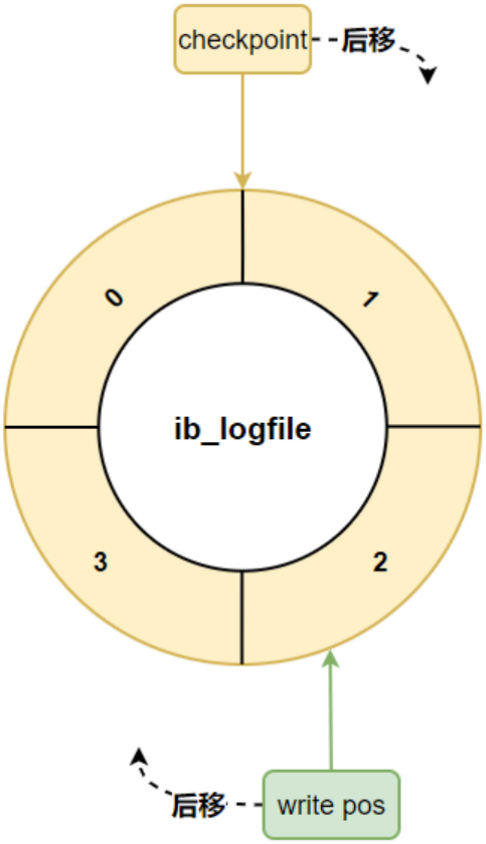
采用循环使用的方式向redo日志文件组里写数据的话，会导致后写入的redo日志覆盖掉前边写的redo日志？当然！所以InnoDB的设计者提出了checkpoint的概念。

1.7.3 checkpoint

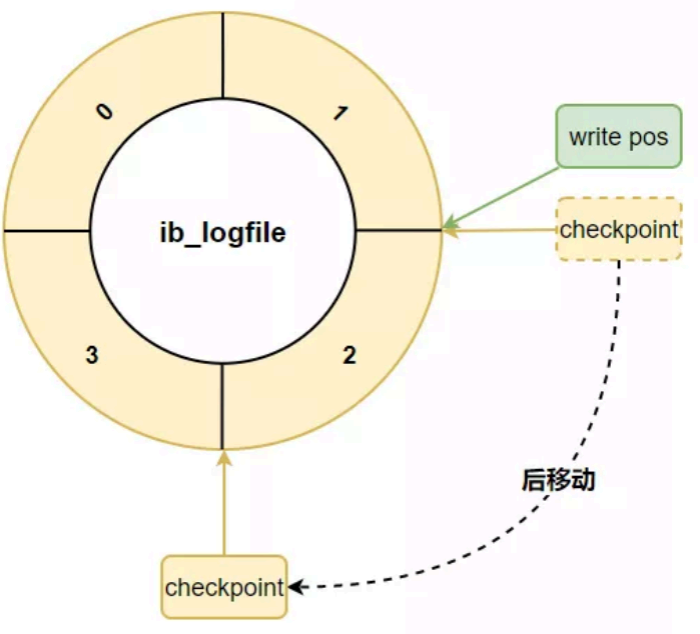
在整个日志文件组中还有两个重要的属性，分别是write pos、checkpoint

- `write pos`是当前记录的位置，一边写一边后移
- `checkpoint`是当前要擦除的位置，也是往后推移

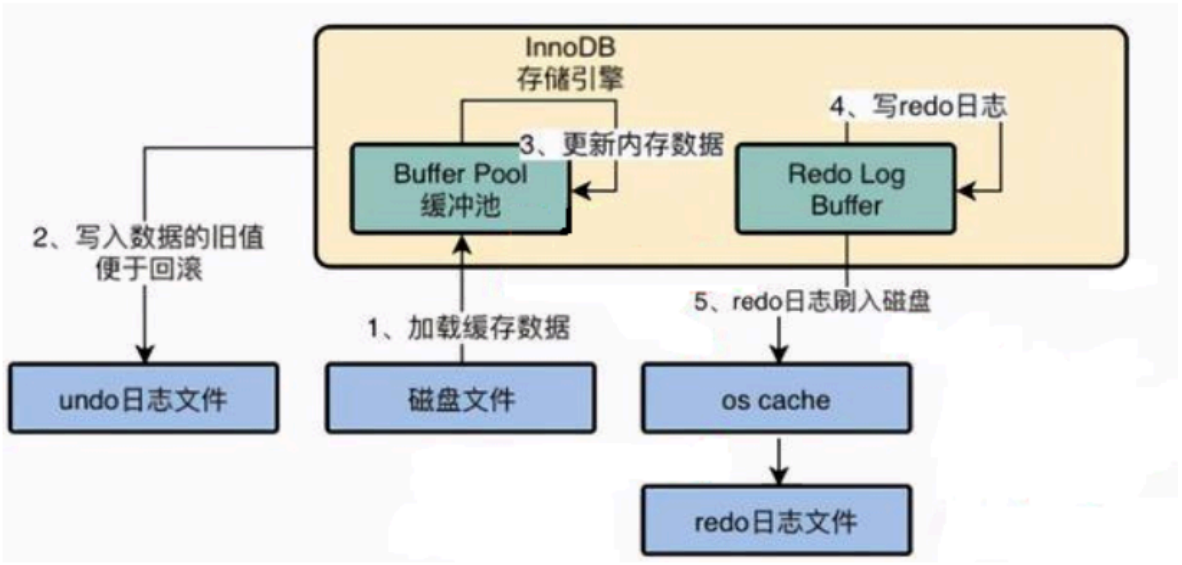
每次刷盘redo log记录到日志文件组中，write pos位置就会后移更新。每次MySQL加载日志文件组恢复数据时，会清空加载过的redo log记录，并把 checkpoint后移更新。write pos和checkpoint之间的还空着的部分可以用来写入新的redo log记录。



如果 write pos 追上 checkpoint ，表示**日志文件组**满了，这时候不能再写入新的 redo log记录，MySQL 得停下来，清空一些记录，把 checkpoint 推进一下。



1.8 redo log小结



2. Undo日志

记录下旧值，为回滚做准备，只要没提交都要回滚。

2.1 作用

- 回滚数据
- 多版本并发控制
    - undo的另一个作用是MVCC，即在InnoDB存储引擎中MVCC的实现是通过undo来完成。当用户读取一行记录时，若该记录已经被其他事务占用，当前事务可以通过undo读取之前的行版本信息，以此实现非锁定读取。

2.2 undo的存储结构

2.2.1 回滚段与undo页

InnoDB对undo log的管理采用段的方式，也就是`回滚段（rollback segment）`。每个回滚段记录了`1024`个`undo log segment`，而在每个undo log segment段中进行`undo页`的申请。

- 在`InnoDB1.1版本之前`（不包括1.1版本），只有一个rollback segment，因此支持同时在线的事务限制为`1024`。虽然对绝大多数的应用来说都已经够用。
- 从1.1版本开始InnoDB支持最大`128个rollback segment`，故其支持同时在线的事务限制提高到了 128*1024 。

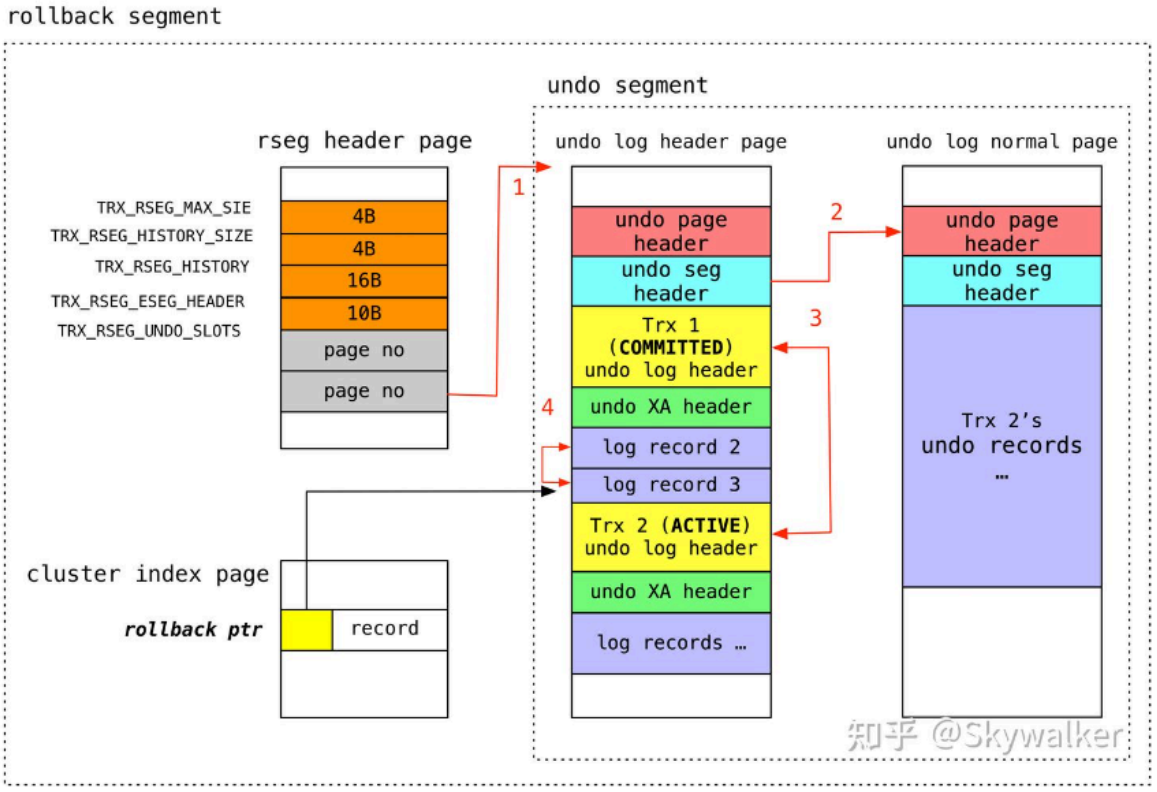
虽然InnoDB1.1版本支持了128个rollback segment，但是这些rollback segment都存储于共享表空间ibdata中。从InnoDB1.2版本开始，可通过参数对rollback segment做进一步的设置。这些参数包括：

- `innodb_undo_directory`：设置rollback segment文件所在的路径。这意味着rollback segment可以存放在共享表空间以外的位置，即可以设置为独立表空间。该参数的默认值为“”，表示当前InnoDB存储引擎的目录。
- `innodb_undo_logs`：设置rollback segment的个数，默认值为128。在InnoDB1.2版本中，该参数用来替换之前版本的参数innodb_rollback_segments。

- `innodb_undo_tablespaces`：设置构成rollback segment文件的数量，这样rollback segment可以较为平均地分布在多个文件中。设置该参数后，会在路径innodb_undo_directory看到undo为前缀的文件，该文件就代表rollback segment文件。

2.2.2 undo页的重用

当事务提交时，并不会立刻删除undo页。因为重用，所以这个undo页可能混杂着其他事务的undo log。undo log在commit后，会被放到一个`链表`中，然后判断undo页的使用空间是否`小于3/4`，如果小于3/4的话，则表示当前的undo页可以被重用，那么它就不会被回收，其他事务的undo log可以记录在当前undo页的后面。由于undo log是`离散的`，所以清理对应的磁盘空间时，效率不高。



2.2.3 回滚段与事务

1. 每个事务只会使用一个`回滚段（rollback segment）`，一个回滚段在同一时刻可能会服务于多个事务。
2. 当一个事务开始的时候，会制定一个回滚段，在事务进行的过程中，当数据被修改时，原始的数据会被复制到回滚段。
3. 在回滚段中，事务会不断填充盘区，直到事务结束或所有的空间被用完。如果当前的盘区不够用，事务会在段中请求扩展下一个盘区，如果所有已分配的盘区都被用完，事务会覆盖最初的盘区或者在回滚段允许的情况下扩展新的盘区来使用。
4. 回滚段存在于undo表空间中，在数据库中可以存在多个undo表空间，但同一时刻只能使用一个undo表空间。
5. 当事务提交时，InnoDB存储引擎会做以下两件事情：
   - 将undo log放入列表中，以供之后的purge操作

     purge: 清除，清洗
   - 判断undo log所在的页是否可以重用(低于3/4可以重用)，若可以分配给下个事务使用

2.2.4 回滚段中的数据分类

1. `未提交的回滚数据(uncommitted undo information)`：

   该数据所关联的事务并未提交，用于实现读一致性，所以该数据不能被其他事务的数据覆盖。

2. `已经提交但未过期的回滚数据(committed undo information)`：

   该数据关联的事务已经提交，但是仍受到undo retention参数的保持时间的影响。

3. `事务已经提交并过期的数据(expired undo information)`：

   事务已经提交，而且数据保存时间已经超过undo retention参数指定的时间，属于已经过期的数据。当回滚段满了之后，会优先覆盖"事务已经提交并过期的数据"。

事务提交后并不能马上删除undo log及undo log所在的页。这是因为可能还有其他事务需要通过undo log来得到行记录之前的版本。故事务提交时将undo log放入一个链表中，是否可以最终删除undo log及undo log所在页由purge线程来判断。

2.3 undo的类型

在InnoDB存储引擎中，undo log分为：

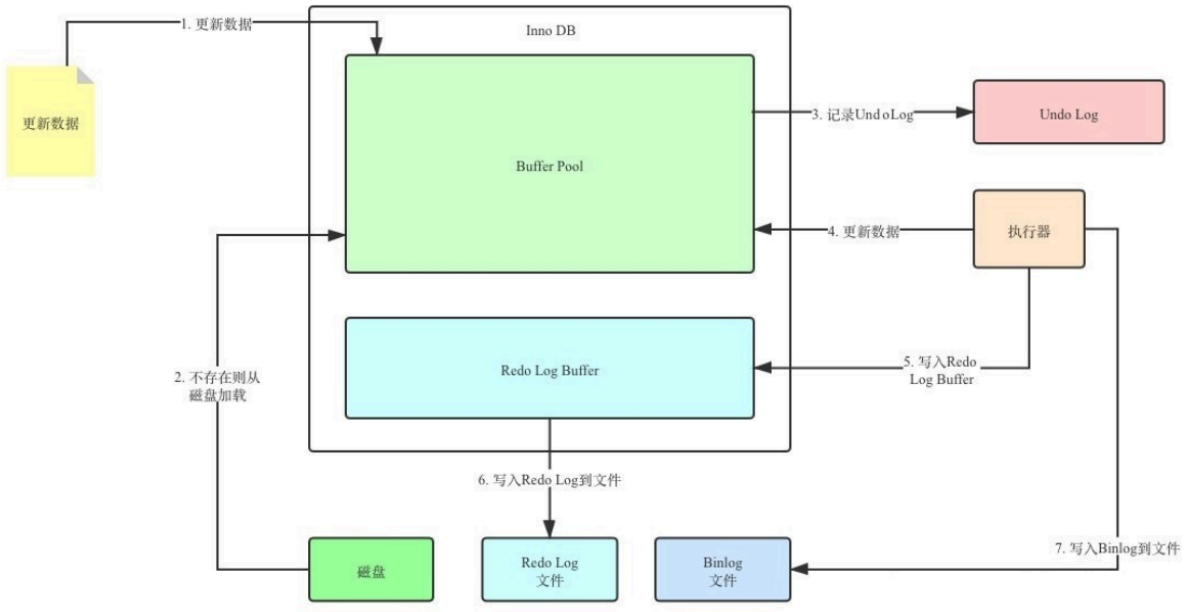
- insert undo log

  insert undo log是指在insert操作中产生的undo log。因为insert操作的记录，只对事务本身可见，对其他事务不可见(这是事务隔离性的要求)，故该undo log可以在事务提交后直接删除。不需要进行purge操作。

- update undo log

  update undo log记录的是对delete和update操作产生的undo log。该undo log可能需要提供MVCC机制，因此**不能**在事务**提交时就进行删除。**提交时放入undo log链表，等待purge线程进行最后的删除。

2.4 undo log的生命周期



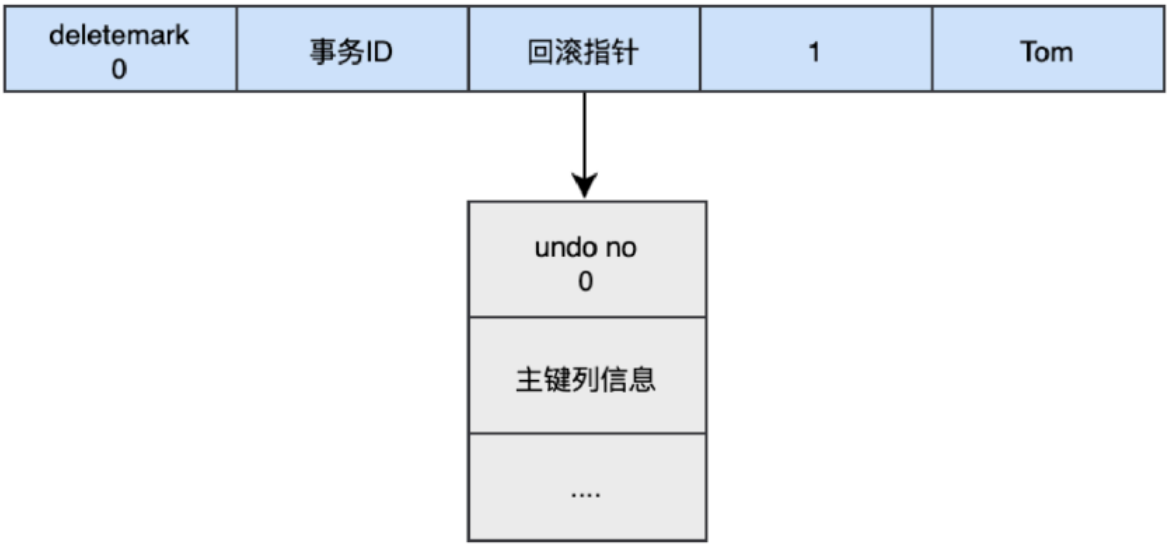
2.4.1 详细生成过程

对于InnoDB引擎来说，每个行记录除了记录本身的数据之外，还有几个隐藏的列:

- `DB_ROW_ID`: 如果没有为表显式的定义主键，并且表中也没有定义唯一索引，那么InnoDB会自动为表添加一个row_id的隐藏列作为主键。
- `DB_TRX_ID`：每个事务都会分配一个事务ID，当对某条记录发生变更时，就会将这个事务的事务ID写入trx_id中。
- `DB_ROLL_PTR`:回滚指针，本质上就是指向undo log的指针。

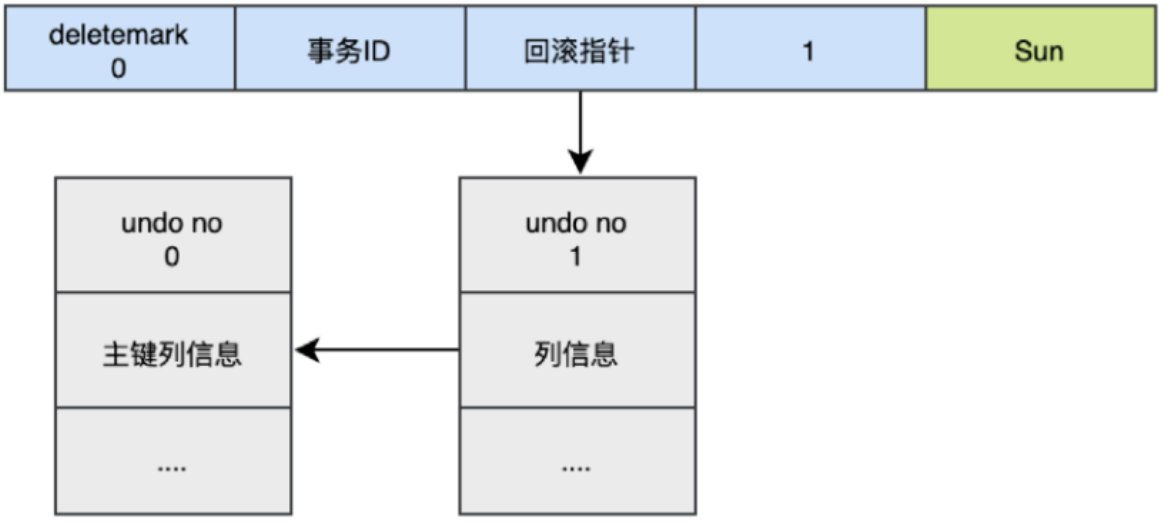
2.4.2 执行INSERT时

插入的数据都会生成一条insert undo log，并且数据的回滚指针会指向它。undo log会记录undo log的序号、插入主键的列和值...，那么在进行rollback的时候，通过主键直接把对应的数据删除即可。



2.4.3 执行UPDATE时

对上面一条update。



2.4.4 undo log的删除

- 针对于insert undo log

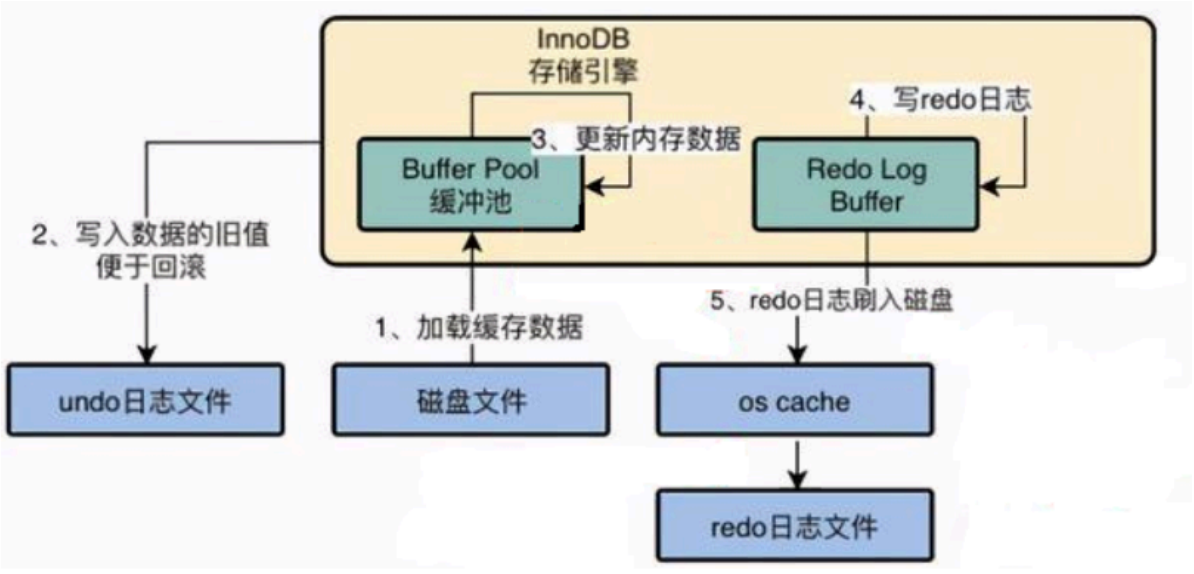
  因为insert操作的记录，只对事务本身可见，对其他事务不可见。故该undo log可以在事务提交后直接删除，不需要进行purge操作。

- 针对于update undo log

  该undo log可能需要提供MVCC机制，因此不能在事务提交时就进行删除。提交时放入undo log链表，等待purge线程进行最后的删除。

> 补充:
>
> purge线程两个主要作用是：`清理undo页`和`清除page里面带有Delete_Bit标识的数据`行。

2.5 小结



undo log是逻辑日志，对事务回滚时，只是将数据库逻辑地恢复到原来的样子。

redo log是物理日志，记录的是数据页的物理变化，undo log不是redo log的逆过程。