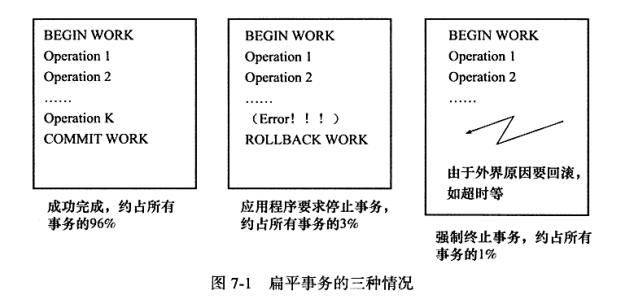
Innodb中事务完全符合ACID特性。前文《mysql之innodb原理-6-锁与隔离性》主要保证了事务隔离性，本文主要讨论原子性。

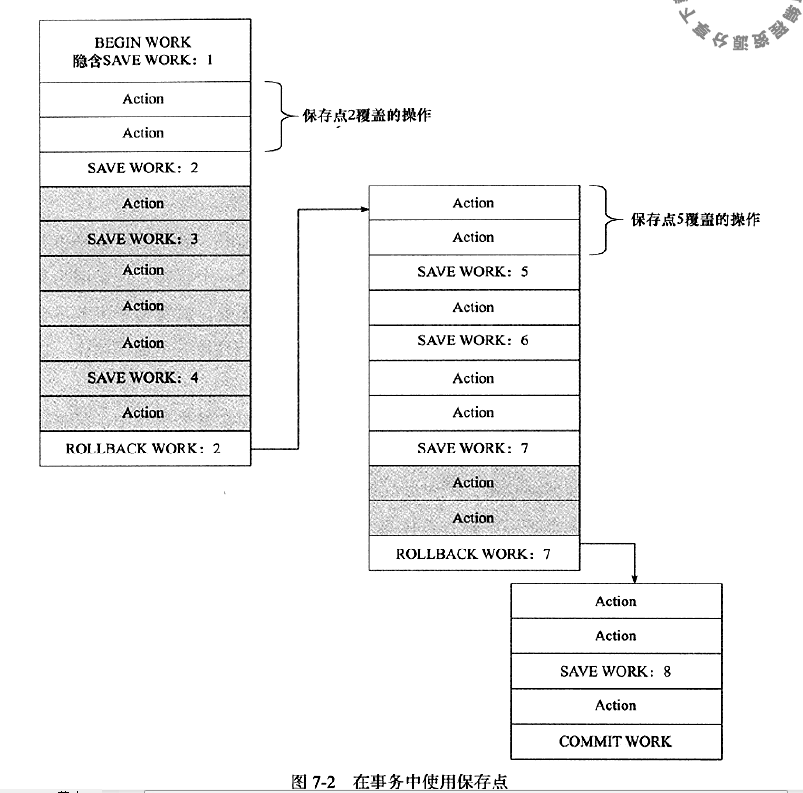
1. **事务分类**
2. **扁平事务**

扁平事务是事务中最简单的一种。从事务begin开始到commit/rollback结束，所有操作处于同一层，都是原子操作。



1. **带有保存点的扁平事务**

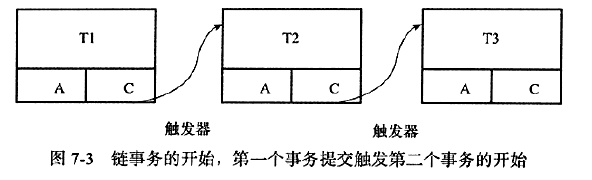
此时允许事务执行过程中回滚到同一事务较早的一个状态。保存点在事务内部是递增的，如下事务rollback 2后，新的保存点是5而不是3。



1. **链事务**

可以视为保存点模式的一个变种，带有保存点的扁平事务，一旦系统崩溃，所有的保存点都将消息。所以在恢复时事务需要从头执行。

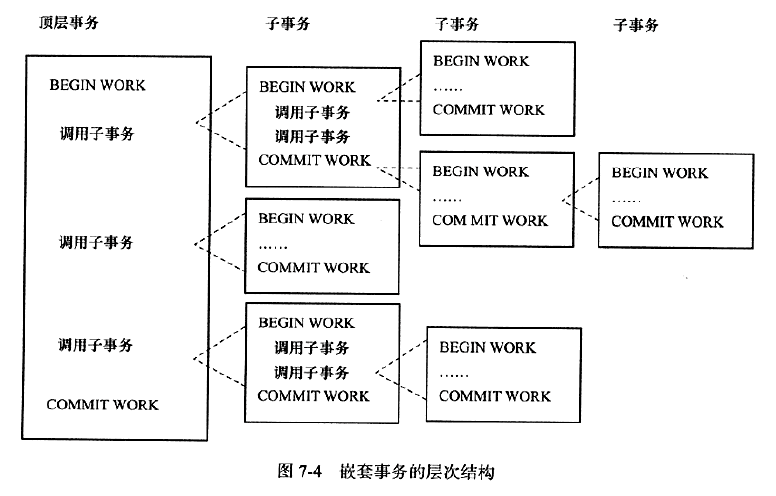
链事务的思想是，当事务提交，释放不需要的数据对象，将必要的处理上下文传递给下一个要开始的事务。注意，提交事务操作和开始下一个事务操作合并为一个院子操作，这意味着下一个事务将看到上一个事务的结果，就好像在一个事务中一样。



链事务和有保存点的扁平事务不同的是，带有保存点的扁平事务能回滚到任意一个保存点，而链事务仅仅限于当前事务，只能恢复到最近一个保存点。对于锁的处理，带有保存点的扁平事务不影响所持有的锁，而链事务在事务commit后就释放持有的锁。

1. **嵌套事务**

嵌套事务是**一个层次结构框架，由一个顶层事务**控制各个层次事务。



嵌套事务是有一棵树构成，根节点就是顶层事务。

子事务既可以提交也可以回滚，但是这些操作不是马上执行，只有等待顶层事务提交后才真正提交。树中任意一个事务回滚会导致其所有子事务回滚，故字十五仅仅保留ACI，不具有D性质。

1. **分布式事务**
2. **事务实现**

事务的隔离性是由锁（非锁定一致性读、锁定一致性读、锁）来实现，原子性、持久性则通过redo log来实现，一致性则通过undo log来实现。

Redo log对应的是bin log，用于描述页的物理修改操作，undo log则用于描述将数据所有的版本，用于在rollback时回滚到特点版本。Undo log会产生redo log，是和binlog一样是逻辑日志，而redo log是Innodb特有的用于保证持久性的物理日志。

1. **Redo log**
2. **基本概念**

Redo log用于实现事务的持久性，由2部分组成，一个是内存中的重做日志缓冲redo log buffer，易丢失；另一个是持久化的重做日志文件redo log file。

Innodb是支持事务的存储引擎，通过force log at commit机制实现事务的持久化，即当前事务commit时，必须先将事务的所有重做日志写入到redo log file进行持久化，事务的commit操作才算完成，这里的重做日志包括redo log和undo log，redo log保证持久性，undo log 用于回滚和mvcc。Redo log 基本是顺序写，数据库运行时不需要读取。而undo log则是随机读写。Undo log也记录redo log，因为undo log是存储在数据库共享表空间中undo segment，也需要在宕机后恢复。

为了保证每次日志写写入重做日志文件，每次在日志缓冲（redo log buffer）写入重做日志文件后，innodb都要执行一次fsync操作。这是因为重做日志文件此时没有使用O\_DIRECT选项（O\_DIRECT会绕过系统文件缓存，直接IO），redo log buffer先写入文件系统缓存，为了保证重做日志写入磁盘，必须进行一次fsync操作。由于fsync速度取决于磁盘性能，因此磁盘性能决定了事务提交的性能，也就是数据库的性能。

Innodb运行手动设置非持久性情况的发生，以提高数据库性能，即事务提交时，日志不写入重做日志文件，而是等待一段时间后再执行fsync操作。由于非强制执行刷盘操作，当数据库当前及，部分日志未刷新到磁盘，导致最后一段时间的事务丢失。

重做日志有一个缓存区Innodb\_log\_buffer，Innodb\_log\_buffer的默认大小为8M(这里设置的16M),Innodb存储引擎先将重做日志写入Innodb\_log\_buffer中。然后会通过以下三种方式将innodb日志缓冲区的日志刷新到磁盘：

* Master Thread 每秒一次执行刷新Innodb\_log\_buffer到重做日志文件。
* 每个事务提交时会将重做日志刷新到重做日志文件。
* 当重做日志缓存可用空间 少于一半时，重做日志缓存被刷新到重做日志文件;

参数innodb\_flush\_log\_at\_trx\_commit用于控制redo log刷新到磁盘的策略。

* 默认为1，即事务提交即fysnc。
* 0表示提交事务不写入重做日志，这个操作只在master thread中完成，而master thread中每1s会执行一次重做日志的fsync操作。
* 2表示提交事务会写redo log file，但不执行fsync。

1. **Redo log文件**

默认情况下，innodb数据目录下会有2个名为ib\_logfile0和ib\_logfile1的文件，这就是redo log file。每个innodb至少有1个redo log file group，为了更高的可靠性，用户可以设置多个镜像日志组，将不同文件存放在不同磁盘上。

每个重做日志文件大小一样，循环写入的方式运行。即innodb先写重做文件1，到达文件最后时，会切换至重做日志文件2。

* Innodb\_log\_file\_size：每个redo log file大小，1.2版本后扩展到512GB
* Innodb\_log\_files\_in\_group：指定了日志文件组中重做文件的数量，默认为2个文件；
* Innodb\_mirroed\_log\_groups：指定了日志镜像文件组的数量，默认为1
* Innodb\_log\_group\_home\_dir：指定了日志文件所在路径

Redo log既不能太大，如果设置太大恢复可能需要很久，也不能太小，因为一个事务日志可能需要多次切换日志文件。此外，重做日志文件太小会频繁发生async checkpoint，因为循环写无法进一步存储新的日志，必须将innodb buffer pool脏页列表中的部分脏数据写回磁盘。

1. **与binlog区别**

* 产生位置不同。Redo log在Innodb存储引擎层产生，而binlog是在Mysql数据库上册产生，且不仅仅针对innbodb，任何在Mysql中的存储引擎对数据库的更改都会产生binlog
* 日志内容不同。Binlog是逻辑日志，记录对应的sql，而redo log记录物理日志，即对每个页的修改；
* 写入磁盘时间点不同。Binlog仅在事务提交后一次性写入，而innodb在事务中不断被写入，这表现为redo log不是随事务提交的顺序写入。
* Binlog不是循环使用，而是在写满后生成新的文件，redo log则是循环使用；
* 使用场景不同。Binlog作为恢复数据使用，主要用于主从复制replication，redo log作为异常宕机数据恢复使用。

所以在事务提交过程中，如果数据库宕机，此时由于事务没有提交不会记录相应的binlog，所以无法从宕机中恢复，redo log记录事务过程中的所有日志，必须有记录物理修改的redo来帮忙。

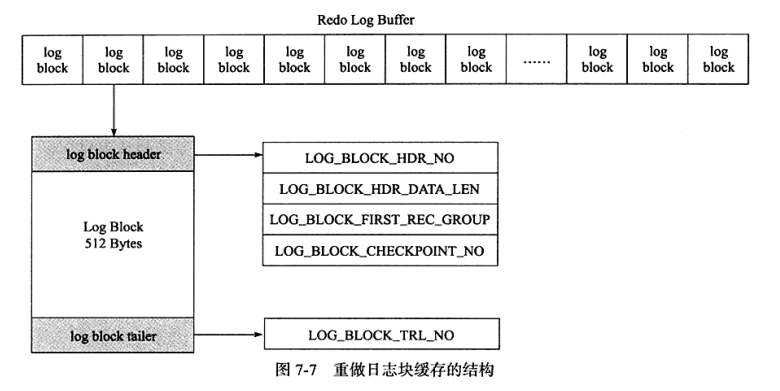
至于能不能用一个日志来合并binlog和redo log，Mysql的开发者也在考虑这个事情，就像oracle只有redo log一样。Redo 就是WAL。

需要注意的是，Hbase也只有WAL来保证宕机，也没有像redo log这样的物理日志，是因为Hbase本身并不支持像innodb这样复杂的事务体系，例如嵌套事务和回滚点。

1. **Log block和redo log格式**

Redo log是以512字节存储的，这意味着redo log buffer和redo log file都是以块方式保存，且块的大小为512字节。由于磁盘扇区也是512字节，所以redo log磁盘写入可以保证原子性，不需要double write。

重做日志块处日志本身，还包含12字节的header和8字节的tailer，故每个redo log可以存储的大小为512-20=492字节：



这492字节的重做日志本身，包括通用header和redo log body，通用header为：

* Redo\_log\_type重做日志类型
* Space：表空间Id
* Page\_no：页的偏移量

针对不同类型的日志类型，其Body也会不同。Innodb中共有51种redo log类型。

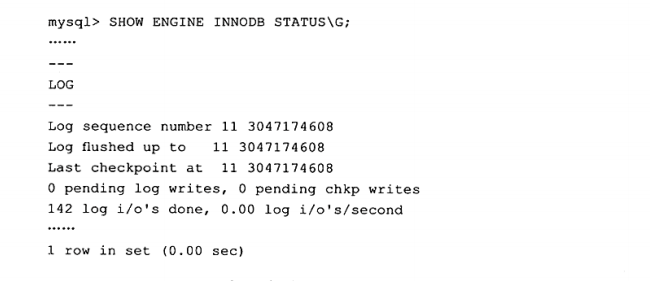
1. **LSN(log sequence Number)**

LSN代表了重做日志的字节的总量，若当前重做日志LSN为1000，有一个事务T1写入了100字节的重做日志，那么LSN变成1100。可见，LSN记录了重做日志的总量，单位为字节。

LSN不仅存在于重做日志中，还存在于每个页的头部。在页中，LSN标识该页最后刷新时LSN的大小。因为重做日志是记录了每个页的物理日志，所以每个页上必须有LSN，以方便进行恢复操作。

例如页P1的LSN是10000，数据库启动时Innnodb检测到重做日志中的LSN是13000，且事务已提交，那么就需要数据库恢复，将redo log应用到P1页中。

通过show engine innodb status查看LSN：



Log flushed up to代表刷新到重做日志文件的LSN，Last checkpoint as代表刷新到磁盘的LSN。

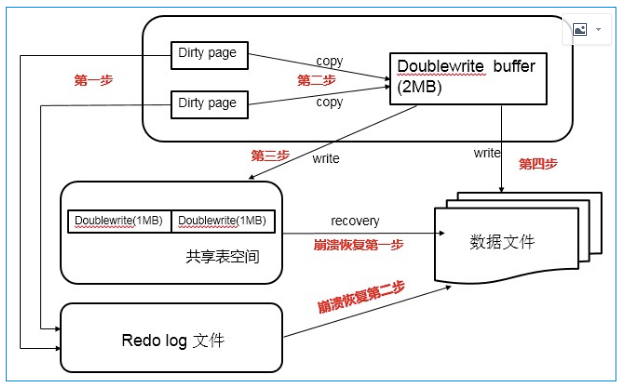
1. **Double write**

前提：

* + 数据库IO的最小单位是16K（MySQL默认，oracle是8K
  + 文件系统IO的最小单位是4K（也有1K的）
  + 磁盘IO的最小单位是512B

MySQL的数据页默认是16k，对数据页的校验也是按16k计算的。而操作系统的数据页默认是2k或者4k，IO操作是按系统页为单位就行读写的。这就可能出现一种情况，数据库对一个16k的数据页修改后，操作系统开始进行写磁盘，每次写4k，结果刚写完第一个4k，数据库挂了。这样就会导致该脏页数据不可用，redo log只记录了物理日志的修改点，而没有记录整个脏页，因此无法恢复。

数据库迫切需要一个记录所有最近脏页的地方，且这个地方是在磁盘上不会受到宕机影响，同时更新速度很快不会影响性能。这就是double write。



Double write新引入了2个新的单元，一个是内存中的double write buffer，大小为2MB，另一部分是物理磁盘上共享表空间连续的128页，大小同样为2MB。双写其实很简单，首先脏页先写入double write buffer，然后分两次每次1MB顺序写入共享表空间，并立即fsync存盘（顺序写很快，且立即存盘），最后在随机写入各个表空间（数据文件，随机写比较慢）。

这样在宕机时，首先通过LSN中从checkpoint找到需要恢复的redo log数据，然后判断所在页是否完整（基于page头和尾进行checksum），如果不完整则加载共享表空间中的页副本并加载到对应的表空间中，再执行redo log进行恢复。

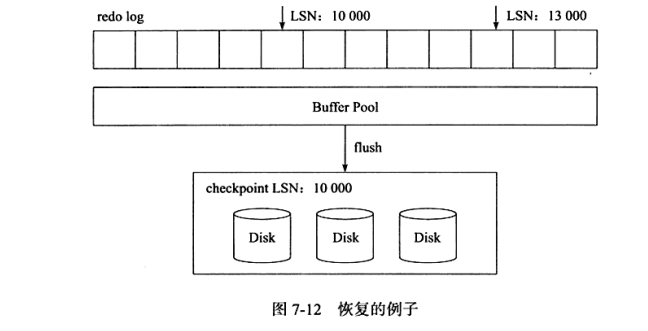
1. **宕机恢复**

当update执行的时候，会写入很多日志信息到redo日志。先写入到redo日志buffer，就是内存缓存中。更新的内容先会更新buffer里面的页。因为redo日志有写盘更新机制，基本都是一有写入buffer，就会写入磁盘。所以redo磁盘日志会时时记录LSN这一串数字。

只有当条件达到了，才会执行checkpoint，把脏页写入磁盘。当所有脏页写入磁盘后，会把checkpoint写入到redo日志。而写入磁盘的每一个页，都记录一个LSN的。

当mysql突然挂了，如何做数据一致性呢？

mysql启动之后，会检测redo日志里面的checkpoint，就知道最后写入磁盘的页的LSN都是小于等于这个checkpoint的。而redo log中大于checkpoint的LSN的页，都没及时写入到磁盘，都丢失了。此时内容需要从redo日志获取，利用redo日志进行数据一致性。例如checkpoint是10000.



利用重做日志比binlog恢复快的多，因为binlog是逻辑日志，还需要构建索引等额外操作。例如表table主键为a，辅助索引为b，执行：

Isnert into table select 1,2

对应重做日志大概为：

C:\Users\ofcard\AppData\Local\Temp\企业微信截图_15962690457446.png

物理日志的再次执行时幂等的。虽然Binlog也可以设置为row，但是insert操作在Binlog中就不是幂等，重复执行可能会插入多条重复记录。

**这样在宕机时：**

* **首先通过LSN中从checkpoint找到需要恢复的redo log数据，然后判断所在页是否完整（基于page头和尾进行checksum），如果不完整则加载共享表空间中的页副本并加载到对应的表空间中；**
* **再执行redo log进行恢复。**
* **如果开启了Binlog同步slave，则=可能出现组提交的异常情况，还需要根据Binlog的状态来进行恢复，即如果binlog存在事务提交而redo log不存在，则当前事务需要根据binlog进行提交。**

1. **undo log**
2. **基本概念**

Undo log是执行逻辑回滚操作的MVCC的snapshot。

和redo log是写在日志文件中不同的是，undo log存放在数据库内部的一个特殊段segment中，这个段称为undo segment，且存储与共享表空间ibdata1内。独立表空间中仅存放数据、索引和插入缓存bitmap。共享表空间包括undo segment，插入缓存索引页、系统事务信息、double write 等。

Undo只能将数据逻辑恢复到原来的样子，所有修改都被逻辑取消。例如，用户insert 10w条记录的事务，这个事务会导致分配一个新的段，即表空间增大，在用户只需rollback，会将插入事务回滚，但是表空间大小不会因此收缩。对于每个Insert，undo会记录一个delete；对于每个delete，undo会记录insert；对于每个update，则会记录一个相反的update。

除了回滚之外，undo的另一个作用是MVCC。当用户读取一行记录时，若该记录已被其他事务占用，当前事务可以通过undo来读取之前的版本数据（例如RR的快照读）。

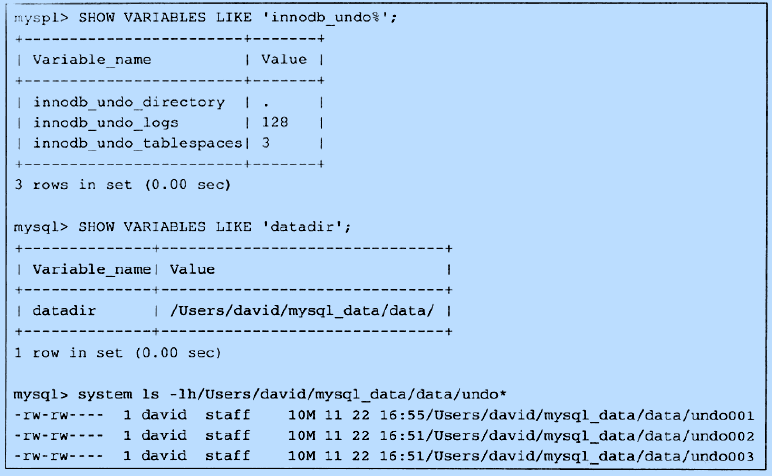
最后，undo log会产生redo log，也就是undo log的产生会产生redo log，这是因为undo log也需要持久化保护。

1. **Undo存储管理**

Undo同样采用段的形式，但是这个段和之前表空间的段有所不同，首先innodb有rollback segment，每个回滚段中记录了1024个undo log segment，而在每个undo log segment中进行undo也的申请。共享表空间偏移量为5的页记录了所有rollback segment header所在的页。

之前rollback segment只能有1个，因此支持同时在线的事务为1024。从1.1版本innodb开始支持最大128个rollback segment。同时1.2版本也可以通过以下参数进行优化：

* Innodb\_undo\_directory：设置rollback segment文件所在路径，这意味着rollback segment可以放在共享表空间以外的位置。
* Innodb\_rollback\_segments用来设置rollback segment的个数，默认128
* Innodb\_undo\_tablespaces用于设置rollback segment文件数量，这样rollback segment可以较为平均的分布在多个文件中。例如下图展示了3个文件组成的rollback segment：



**事务在undo log segment分配页并写入undo log过程中会产生redo log，**当事务提交时，innodb会做以下两件事：

1. 将undo log放入列表的header，以供后续purge：

事务提交后不能马上删除undo log及其所在页，因为可能还有其他未结束事务需要通过当前undo log来得到行之前的版本。例如T1和T2都是对ROW1快照读，但是T1和T2记录事务开始的快照读为undo log，显然只需要记录一份undo log，T1先提交修改该行数据，T2在T1提交后在自己的事务中仍然多次进行快照读，显然T1 commit后不能直接Purge自己的undo log，而是将其记录在一个链表中，是否可以最终删除undo log及其所在页由purge线程决定。

1. 判断undo log所在页是否可以重用，若可以则分配给下个事务使用：

当事务提交后，先把undo log放入链表，然后判断undo 页的空间是否小于3/4，如果小于则可以重用，新的undo log记录在之前的后面。虽然不用为每个事务单独分配Undo页，但是重用也会导致相同undo 页存放不同事务的undo log，purge操作需要离散读。

我们可以通过show innodb status来查看链表中undo log数量：History list length

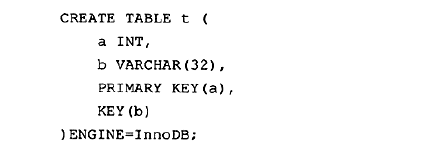
1. **Undo log格式**
2. **Insert undo log**

Insert undo log不会存在被其他事务使用的情况，在事务commit/rollback后可以直接删除。

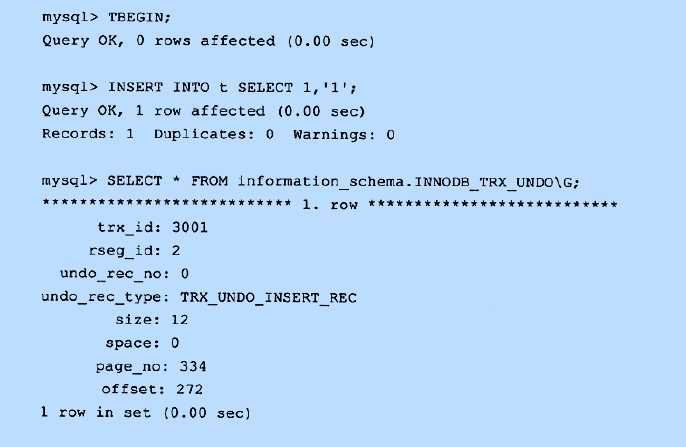
|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | Insert undo log内容 | | 含义 |
| 1 | Next（头） | | 头部next记录了下一个undo log位置，通过next可知道该undo log所占字节数 |
| 2 | Type\_cmpl | | Undo 类型，insert总是11 |
| 3 | Undo\_no | | 记录事务ID，即当前事务的第几个操作，等于undo\_rec\_no |
| 4 | Table\_id | | 对应表 |
| 5 | len1(主键长度) | col1 | 所有主键长度和值 |
| len2 | col2 |
| … | … |
| lenN | colN |
| 6 | Start（尾） | | 尾部两个字节记录的是Undo log开始的位置start，所以当前undo log大小为next – start |

我们可以通过information\_schema.INNODB\_TRX\_UNDO和INNODB\_TRX\_ROLLBACK\_SEGMENT来定位具体的undo log。

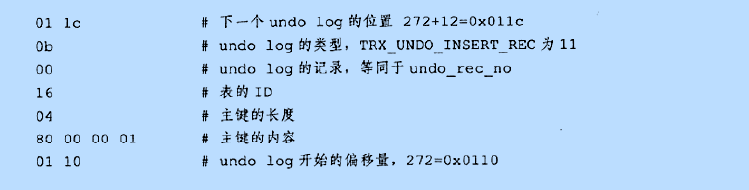
例如表：



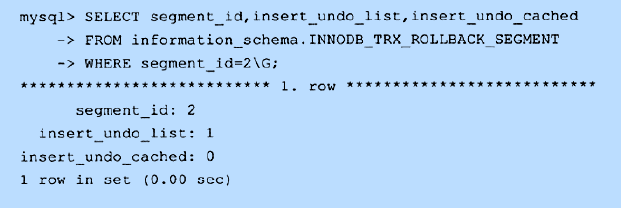
我们插入一条数据并观察INNODB\_TRX\_UNDO表， trx\_id事务ID为3001，rollback segment的ID是rseg\_id。



Space为0，page\_no为334，offset为272，若undo segment存储于共享表空间，则打开共享表空间文件ibdata1，定位到页(334,272)，读取12字节，内容如下：



此时事务没有提交，我们利用rseg\_id=2利用查看INNODB\_TRX\_ROLLBACK\_SEGMENT对应的数据，此时有一个insert\_undo\_list。



此时commit，再查询利用rseg\_id=2利用查看INNODB\_TRX\_ROLLBACK\_SEGMENT对应的数据，发现insert\_undo\_list被直接purge，该undo log页被回收insert\_undo\_cached=1

1. **update undo log**

包括update和delete，可能需要提供MVCC，所以不能在事务提交后删除，应该先放入undo log 链表，由purge线程删除。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 序号 | Insert undo log内容 | | | | 含义 |
| 1 | Next（头） | | | | 头部next记录了下一个undo log位置，通过next可知道该undo log所占字节数 |
| 2 | Type\_cmpl | | | | Undo 类型 |
| 3 | Undo\_no | | | | 事务ID |
| 4 | Table\_id | | | | 对应表 |
| 5 | len1(主键长度) | | col1 | | 所有主键和值 |
| len2 | | col2 | |
| … | | … | |
| lenN | | colN | |
| 6 | Pos1 | len1 | | Old\_col1 | Update vector，记录了修改**前后**的数据信息，col代表值。 |
| Pos2 | len2 | | Old\_col2 |
| … | … | | … |
| PosN | lenN | | Old\_colN |
|  | | | |
| Pos1 | len1 | | col1 |
| Pos2 | len2 | | col2 |
| … | … | | … |
| PosN | lenN | | colN |
| 7 | Start（尾） | | | | 尾部两个字节记录的是Undo log开始的位置start，所以当前undo log大小为next – start |

1. **MVCC**

我们知道mysql的一致性非锁定读就是利用快照版本来实现的，RR是快照读+gap lock，RC是当前读+record lock。Mysql中的MVCC就是利用undo log、read view和行数据中的4个隐藏字段实现的。

1. **行数据的4个隐藏字段**

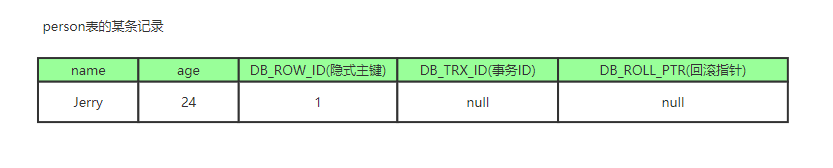
每行数据中，除了DDL定义的字段，还包含4个隐藏字段：

1. DB\_TRX\_ID ：最近修改的事务ID
2. DB\_ROLL\_PTR：回滚指针，指向该行数据对应的undo log版本（存储于rollback segment中）
3. DB\_ROW\_ID：隐含的自增ID，如果当前表没有定义主键，则作为当前表的聚簇索引。
4. DELETE\_BIT：删除标签，某条数据被删除并不是真正物理移除，而是删除flag变了



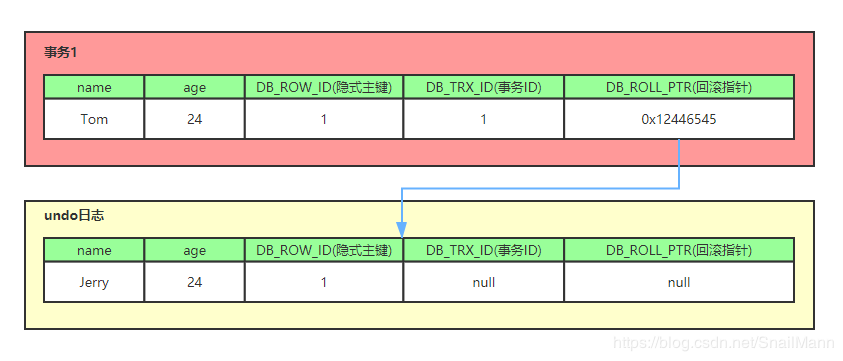
如上图，DB\_ROW\_ID是数据库默认为该行记录生成的唯一隐式主键，DB\_TRX\_ID是当前操作该记录的事务ID,而DB\_ROLL\_PTR是一个回滚指针，用于配合undo日志，指向上一个旧版本

比如一个有个事务插入persion表插入了一条新记录，记录如下，name为Jerry, age为24岁，隐式主键是1，事务ID和回滚指针，我们假设为NULL：



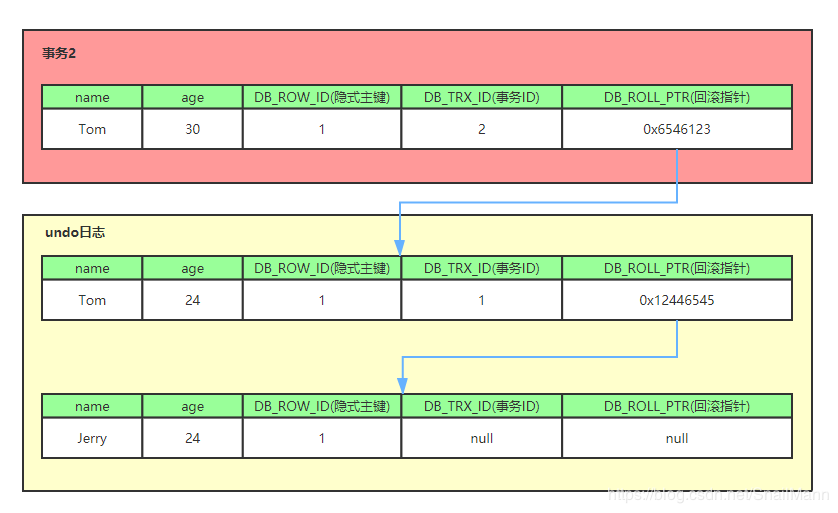
RR隔离级别下，现在来了一个事务1对该记录的name做出了修改，改为Tom：

* 事务1修改该行记录，先加入意向锁和next-key lock；
* 将该行数据拷贝到undo log page中，作为旧记录；
* 修改当前行name为Tom，修改行事务DB\_ROW\_ID为当前事务1的ID，事务ID从1自增，并修改DB\_ROLL\_PTR指向刚才写的undo log；
* 事务提交，释放锁。



又来了个事务2修改person表的同一个记录，将age修改为30岁:

* 事务2修改该行记录，先加入意向锁和next-key lock；
* 将该行数据拷贝到undo log page中，作为旧记录；
* 修改当前行age为30，修改行事务DB\_ROW\_ID为当前事务2的ID，事务ID从1自增，并修改DB\_ROLL\_PTR指向刚才写的undo log；
* 事务提交，释放锁。



从上面，我们就可以看出，不同事务或者相同事务的对同一记录的修改，会导致该记录的undo log成为一条记录版本线性表，既链表，undo log的链首就是最新的旧记录，链尾就是最早的旧记录（当然就像之前说的该undo log的节点可能是会purge线程清除掉，向图中的第一条insert undo log，其实在事务提交之后可能就被删除丢失了，不过这里为了演示，所以还放在这里）

1. **Read view**

既然行中有隐藏字段DB\_TRX\_ID和DB\_ROLL\_PTR，我们就可以构建出上文中某行数据在undo log的列表。那么那些undo log可以被read view看见呢？

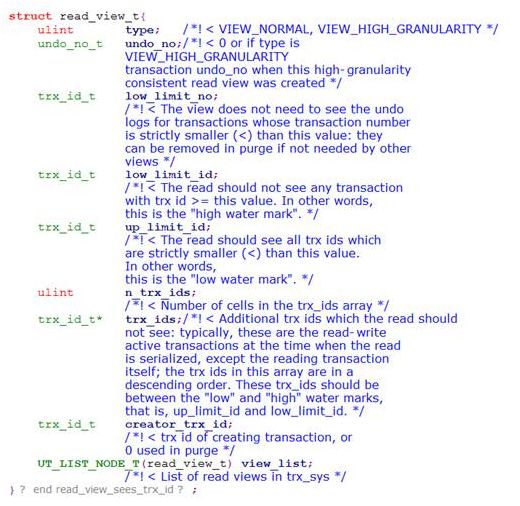
Read view遵循可见性算法f。**在事务1开启事务并不立刻创建read view，只有第一次读的时候才会创建read view时**，会记录创建readview这一刻当前活跃事务DB\_TRX\_ID集合，以及下一个未分配事务的TRX\_ID（事务ID自增）。首先读取当前行（该行数据可能已被事务2修改，但是尚未提交），我们会根据可见性算法f，判断当前行是否可见，如果不可见，则根据DB\_ROLL\_PRT根据undo 链表向下找到之前的版本，直到找到可以看见的那个undo log版本。

1. **可见性算法**

可见性算法f也很简单，即在创建read view时，会记录这一时刻当前活跃事务集合trx\_ids以及该集合中最小事务ID up\_limit\_id，以及下一个未分配事务的TRX\_ID low\_limit\_id。

可见性算法f如下：

* 当前行记录DB\_TRX\_ID >= 未分配事务的TRX\_ID low\_limit\_id，即当前行数据的修改是在read view创建之后新开的事务，此时该记录可能未提交，不可见；
* 当前DB\_TRX\_ID在快照创建时的活跃事务集合中，说明该行记录在快照开始时未提交，不可见；
* 当前DB\_TRX\_ID < 快照创建时活跃的最小事务up\_limit\_id，说明该行记录在快照开始时已经提交，可见；
* 其他情况下，即 up\_limit\_id <= 当前行记录DB\_TRX\_ID <= low\_limit\_id，且DB\_TRX\_ID不在活跃事务集合中，该行记录在快照开始时已经提交，可见；
* 本事务中的修改会刷新本事务的read view，即使未提交，可见。



1. **例子**

当事务2对某行数据执行了快照读，数据库为该行数据生成一个Read View读视图，假设当前事务ID为2，此时还有事务1和事务3在活跃中，事务4在事务2快照读前一刻提交更新了，所以Read View记录了系统当前活跃事务1，3的ID，维护在一个列表上，假设我们称为trx\_list



此时对于事务2的read view来说，up\_limit\_id = 1，low\_limit\_id = 4 + 1，trx\_list = {1,3}，

显然事务4的db\_trx\_id=4满足可见性算法的条件，即在事务2执行时，可以看到事务4提交的结果行。

需要注意的是，RR隔离级别下的READ VIEW不是事务开始时的状态，而是第一次读（后续没有update情况下）的read view。

例如当前数据：

C:\Users\ofcard\AppData\Local\Temp\企业微信截图_15964625598681.png

分别执行2个事务：

|  |  |  |
| --- | --- | --- |
| T1 | T2 | 结果及描述 |
| start TRANSACTION; |  |  |
|  | start TRANSACTION;  SELECT \* from smstest where sn = 1;  update smstest set `status`=9 where sn = 1; |  |
|  | SELECT \* from smstest where sn = 1;  commit; | 本事务可见，即使未提交，即status修改为9 |
| SELECT \* from smstest where sn = 1; |  | 第一次快照读，已经读到了T2 commit的结果，即status=9 |
| commit; |  |  |

1. **RC和RR的readview**

RC情况下每次一致性非锁定读都会重新读取read view，这样就是read commit时刻都能读取到已提交数据，即当前读；

RR只有在事务中第一次读时会创建read view，并不是事务开启时的快照！在同一事务的后续读取，除非又执行了update刷新了read view，否则会按照之前的read view，这就是快照读。

注意update会刷新read view，这是前文《mysql之innodb原理-6-锁与隔离性》中所描述的消除write skew的手段，即update操作会有gap lock，同时刷新快照。

1. **RR级别下辅助索引的read view[6]**

我们知道，行数据中存在db\_trx\_id用于判断是否可见，db\_roll\_ptr用于寻找更早的版本以判断是否可见，显然我们需要通过聚簇索引拿到数据页中的整行数据才能进行可见性分析，从而构建read view。

对于辅助索引，其只关联了对应主键，那么该如何读取快照呢？

答案很简单，辅助索引查询的MVCC生成如下：

* 如果该Key的整个辅助索引记录的最大事务max\_trx\_id小于当前活跃最老事务up\_limit\_id，说明整个索引都对当前read view可见；
* 否则需要依次对索引中的每条记录回表找到聚簇索引行数据，以判断其可见性；
* 在返回前判断当前可见记录的辅助索引Key是否与回表时辅助索引key相同，如果不同则说明数据不对应，不返回。

所以我们在对**事务并发**的情况下通过**辅助索引查询**，尽管我们在select中返回字段仅有辅助索引和聚簇索引（覆盖），但是其read view的构建（无论是RR还是RC）仍然可能需要**回表查询聚簇索引**，所以事务并发情况下尽量使用聚簇索引查询，否则使用辅助索引可能回表，效率较低。

具体分析参见《mysql之innodb原理-8-辅助索引实现MVCC》

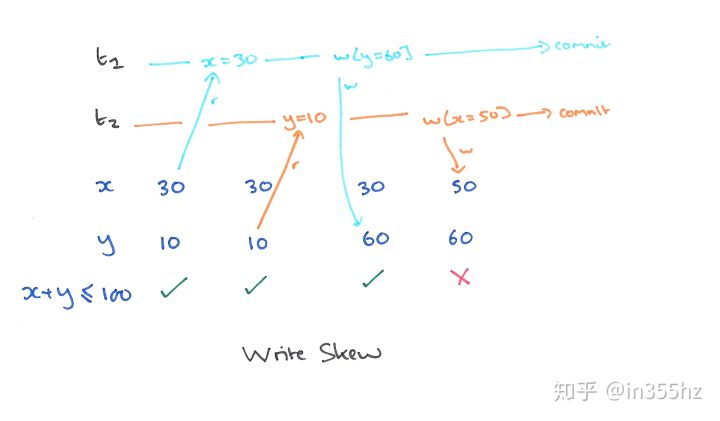
1. **RR级别下一致性非锁定读与幻读**

RR级别下既然有了MVCC，是不是就可以避免幻读了呢？

1. **普通幻读消除**

MySQL（innodb）的 RR 隔离级别实际上是 snapshot isolation，可以避免通常意义的幻读。snapshot isolation 的问题是无法处理如下的 read-write conflict（write skew）。

1. **write skew解决**



我们约定，x + y <= 100. 如果都是快照读/当前读，t1先读取x=30，然后根据快照/当前版本判断，进行y=60更新满足 x + y <= 100。

t2先读取y = 10，在t1更新y=60之后，由于t1没有提交，此时t2依然可以满足x+y<=100的情况下修改x=50。

之后t1和t2都可以成功commit。但显然违背了我们之前的约束条件。

这种现象称为write skew。由于 UPDATE 本身也是一种 read-write，如果执行 UPDATE 也会有 write skew 问题, 那对实际应用来说就太糟糕了。

Innodb为了解决RR情况下的这个问题，强行把read分为快照读和当前读（Locking read），快照读就是普通的read，当前读就是select … for update。在update或select … for update时，Innodb执行当前读，在过程中加入record lock和gap lock（next key lock），相当于变相提升到了Serializable级别，从而消除write skew。

1. **next key lock的必要性**

上文中，update或select … for update时，Innodb执行当前读，在过程中加入record lock和gap lock（next key lock）。

**Record lock就可以通过悲观锁实现目标资源的串行化，为啥recordLock还不够，还得使用next key lock呢？**

在update或select … for update之后，会更新快照进行当前读，如果只有record lock没有next key lock，在这个过程中可能会有gap内新数据插入，从而出现了原有幻读的现象。所以必须有next key lock来在更新快照后避免传统意义上的幻读出现。

1. **新的不一致读**

虽然解决了write skew，但是当前读（Locking read）和快照本身是矛盾，会出现**新的不一致读**。由于并没有Serializable的悲观锁，在两个不同的事务中仍然可以并发的执行update语句，update/select for update执行当前读之后，会更新快照，导致与之前的快照读不一致的情况。

只有Serializable才能避免这种情况。

例如：mysql正确处理的结果如下[4]：

CREATE TABLE char\_encode (

glyph CHAR(1) NOT NULL,

codepoint TINYINT(3) NOT NULL

) ENGINE=InnoDB

INSERT INTO char\_encode VALUES ('a', 97), ('b', 98);

* SESSION-1

|  |  |  |
| --- | --- | --- |
| 序号 | 执行语句 | 结果 |
| 1 | SHOW SESSION VARIABLE LIKE 'tx\_isolation' | REPEATABLE-READ |
| 2 | START TRANSACTION; |  |
| 3 | SELECT \* FROM char\_encode; | a | 97  b | 98 |

* SESSION-2

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 执行语句 | 结果 | 解释 |
|  | SHOW SESSION VARIABLE LIKE 'tx\_isolation' | REPEATABLE-READ |  |
|  | START TRANSACTION; |  |  |
|  | SELECT \* FROM char\_encode; | a | 97  b | 98 |  |
|  | UPDATE char\_encode SET codepoint = 100 WHERE glyph = 'a'; | 1 Rows affected; |  |
|  | SELECT \* FROM char\_encode; | a | 100  b | 98 | 如预期，一个事务能看到自己的本地改变 |
|  | COMMIT |  |  |

* SESSION-1

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 执行语句 | 结果 | 解释 |
|  | SELECT \* FROM char\_encode WHERE glyph = 'a'; | a | 97 | 完美，如预期，尽管session-2提交了，但是一致性非锁定读就是应该读自己事务中第一次读取时的快照 |
|  | UPDATE char\_encode SET codepoint = codepoint + 1 WHERE glyph = 'a'; | 1 Rows affected; |  |
|  | SELECT \* FROM char\_encode WHERE glyph = 'a'; | a | 101 | 刚才读才是97，更新后应该是97+1=98，为什么不是快照读了？因为update语句或select for update 会更新快照，和之前的快照不一致 |

根据[4]中的官方解释，mysql允许覆盖更新这种行为（不遵守first commit win rule)，这让它产生了上面的这种幻像（但又不是幻读）。

1. **Lost update问题——不存在这个问题**

mysql的RR级别（快照读）保证避免不可重复读，也保证了不会出现Lost update。因为在消除write skew的基础上，必然保证update 的时候要更新视图，而不是在快照基础上update。所以下面的情况不存在！

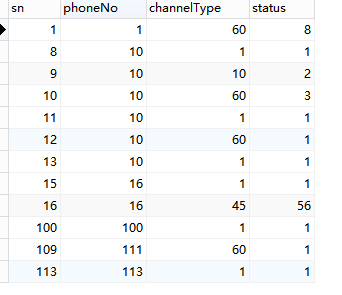
|  |  |
| --- | --- |
| tx1 | tx2 |
| START TRANSACTION; |  |
| SELECT \* FROM test WHERE id = 1;  ( say, DB\_TRX\_ID = 7 at this moment) |  |
|  | START TRANSACTION; |
|  | SELECT \* FROM test WHERE id = 1; |
|  | UPDATE test SET name="x" WHERE id = 1; |
|  | COMMIT;(say, makes DB\_TRX\_ID = 10) |
| UPDATE test SET name="y" WHERE id = 1;  COMMIT |  |

“tx1最终提交，其row version(DB\_TRX\_ID)任然是7，即之前提交的DB\_TRX\_ID = 10的版本丢失了，只能通过Seriablizable（或者select for update提供的串行化能力）来解决”。这种情况不存在！

1. **隔离级别与binlog**

一般来说，如果是RC隔离级别，我们必须将binlog设置为Row。这不仅是数据记录更为详细，更重要的是binlog是在事务commit后才生产，由于RC没有next-key lock只有行锁，只能锁定具体的行而不是对满足条件的索引进行锁定，使用statement格式会生成和当前主库中不一致的binlog，在执行master-slave复制时出错。而RR隔离级别利用nextkey-lock则可以避免这个情况。具体参见后文。

例如数据如下：



1. **RR——安全使用binlog-Statement**

对于RR隔离级别，由于在tx1已经对索引phoneNo=16前后加入了next keylock且未提交，tx2此时试图将sn=1（sn也是辅助索引）的phoneNo修改为16，对索引phoneNo而言是执行了插入phoneNo=16的操作，显然在nextkey-lock锁范围内，此时tx2阻塞等待tx1释放next key lock。此时事务提交顺序tx1, tx2。对应statement类型的binlog为：

*Tx1: update smstest t set t.status = 9 where t.phoneNo='16'*

*Tx2: update smstest t set t.phoneNo='16' where t.sn='1*';

|  |  |  |
| --- | --- | --- |
| tx1 | tx2 | 解释 |
| set tx\_isolation='repeatable-read'; |  | 设置当前session事务隔离级别为RR |
| BEGIN;  update smstest t set t.status = 9 where t.phoneNo='16' |  | 对索引phoneNo=16前后加入了next keylock |
|  | set tx\_isolation='read- read';  BEGIN; |  |
| update smstest t set t.phoneNo='16' where t.sn='1'; | tx2此时试图将sn=1（sn也是辅助索引）的phoneNo修改为16，对索引phoneNo是执行了插入phoneNo=16的操作，显然在tx1的next keylock锁范围内，tx2阻塞等待tx1释放锁才能执行commit |
| commit; | 阻塞 |
| commit; |  |  |
|  |  | 此时tx2 commit才成功 |

1. **RC——数据库状态与binlog-Statement不一致**

对于RC隔离级别，情况如下：

|  |  |  |
| --- | --- | --- |
| tx1 | tx2 | 解释 |
| set tx\_isolation='read-committed'; |  | 设置当前session事务隔离级别为RC |
| BEGIN;  update smstest t set t.status = 9 where t.phoneNo='16' |  | 对索引phoneNo=16只加入了行锁 |
|  | set tx\_isolation='read-committed';  BEGIN;  update smstest t set t.phoneNo='16' where t.sn='1'; | tx2此时试图将sn=1（sn也是辅助索引）的phoneNo修改为16，对索引phoneNo是执行了插入phoneNo=16的操作，由于只有行锁直接成功 |
| commit; | 成功 |
| commit; |  |  |

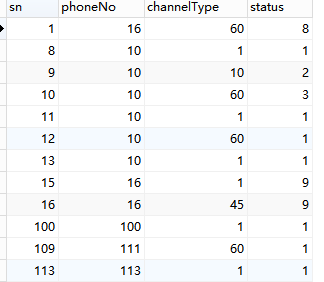
由于是RC隔离级别，只有行锁，tx2会直接执行成功，此时产生的statement 的binlog如下：

*Tx2: update smstest t set t.phoneNo='16' where t.sn='1*';

*Tx1: update smstest t set t.status = 9 where t.phoneNo='16'*

显然RC隔离级别下，数据库中数据和binlog不是等价的。

master数据库中实际数据如下：



而按照上面的binlog，在slave中所有*phoneNo=16*的status都等于9.

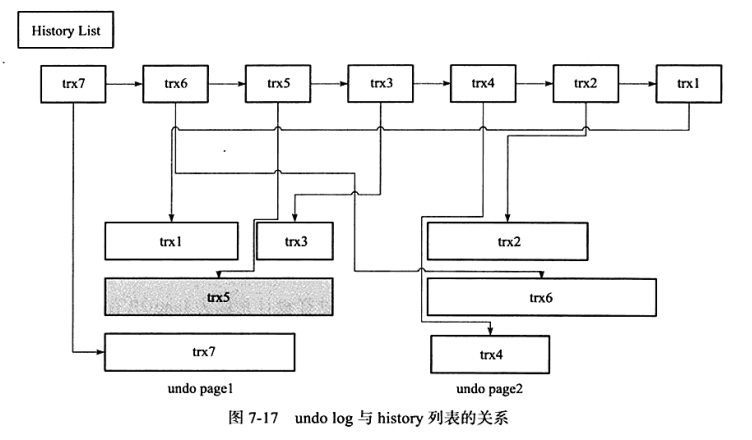
所以RC情况下binlog-format只能是row，不能是statement。

1. **Purge**

为了实现MVCC机制，更新或删除操作都只是设置了老undo log的delete\_bit，并不是真正直接将这些老记录删除。

但为了节省磁盘空间，innodb有专门的Purge线程来清理不再使用的undo log。为了不影响MVCC被其他事物使用，purge线程也有自己的read view，这个read view是系统中活跃的最老的事务的read view。如果某个undo log的DB\_TRX\_ID相对于purge线程的read view可见（见可见性算法f），说明没有任何其他事物引用它，那么该undo log可以被安全清除。

同时由于undo log可以随机写在undo log page，为了能够尽可能消除随机读写，innodb还根据事务提交的顺序对undo log进行链接，这就是undo history列表。在实际Purge过程中，首先从history 列表中找到第一个需要被清理的记录trx1，清理后还会在trx1所在undo log页中继续寻找被清理的记录，例如trx3，接着trx5，但是发现trx5不能清理，此时再去history中查找。



参数innodb\_purge\_batch\_size设置了每次purge操作需要清理的Undo log page的数量，默认300。innodb\_purge\_batch\_size设置越大，每次回收的undo log page就越多，但是purge占用的cpu和Io也会过多，是数据库性能下降。

当innodb压力太大，不能高效进行purge时，通过innodb\_max\_purge\_lag=1可以设置延缓DML操作。

1. **Group commit**

**事物提交时，innodb会在存储引擎日志记录commit的单独redo log，且binlog里会记录对应二阶段提交的全局事务xid。**

由于磁盘fsync性能有限，所以为了提高效率，事务提交时都会按照2个阶段操作：

* 修改内存中事务的信息，并将日志写入日志缓冲；
* 调用fsync来确保日志缓冲写入磁盘。

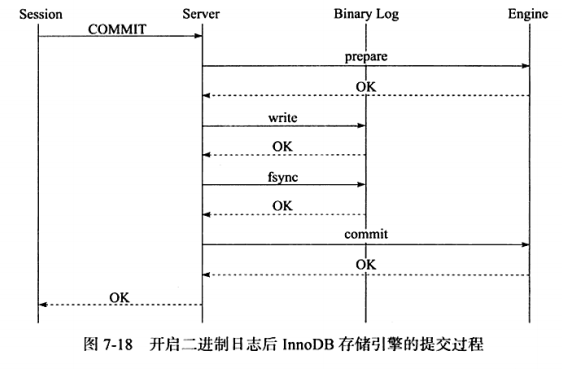
所以在阶段1，我们会暂留部分时间，等待更多数据写入日志缓冲，而不是立即使用fsync，这就是组提交。

对于无binlog时的提交，此时只需要考虑redo log的提交即可，redo log中也记录了commit和rollback的记录。

对于有binlog的提交，还需要保证数据库上层binlog和Innodb层事务日志redo log的提交顺序一直，Mysql内部使用了prepare\_commit\_mutex锁。

1. **原始情况的二阶段提交**

|  |  |
| --- | --- |
| 步骤 | 过程 |
| 1 prepare | 当事务提交时innodb存储引擎进行prepare操作，即将redo log 和undo log写入内存； |
| 2 write/fsync | Mysql上层写入binlog，其fsync由sync\_binlog控制 |
| 3 commit | innodb commit，将innodb 的redo log写入redo log file |
| 3-1 写缓冲 | Innodb 日志写重做日志缓冲 |
| 3-2 fsync | 调用fsync将redo log写入磁盘，其fsync由innodb\_flush\_log\_at\_trx\_commit控制 |



这里的二阶段提交使用了XA，首先在prepare时创建了一个分布式事务XID，且会在commit阶段（step2）写入到Binlog中。在执行本地恢复时是进行提交还是回滚，由binlog决定：

* Step1 Prepare阶段宕机，binlog中Xid\_log\_event不存在该XID，直接回滚；
* Step2 写binlog失败，此时binlog中Xid\_log\_event不存在该XID，直接回滚；
* Step3 innodb commit失败，此时binlog中Xid\_log\_event存在该XID，则根据binlog进行commit；

1. **group commit为什么要保证binlog和redo log提交的顺序一致**

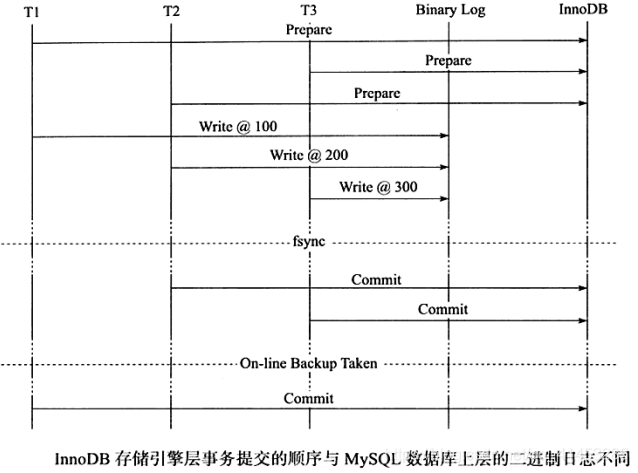
*问题：T1的数据丢不丢和T3有什么关联呢？既然T1的binlog已经刷盘，且也会同步给slave，为什么slave会丢失T1呢？难道slave接收到的binlog里还记录了存储引擎的commit状态？如果加上prepare\_commit\_mutex锁保证了顺序性会有什么改变呢？*

1. **主备日志顺序不一致**

master的事务顺序是redo log记录的顺序，而slave的事务顺序是同步了master的binlog的事务顺序，如果master中redo log和 binlog提交顺序不一致，的确可能导致master和slave数据的不一致。（不过Innodb本身修改数据会加入行锁和间隙锁，事务commit后才会释放锁，group commit肯定是互相没有锁的事务，也就不存在对相同行进行修改）。

1. **主机热备丢事务**

并发情况下如果仍然开启group commit来进行fsync，可能会导致innodb日志redo log和Mysql binlog提交顺序不一致，无法正确完成online backup replication（在线热备）。例如：



此时innodb日志提交顺序为：T2 -> T3 –> T1（在backup 前T1未提交），而binlog日志提交顺序为：T1 -> T2 - >T3，此时我们在T1 commit之前执行了online-backup taken操作（如XtraBackup备份）。

**XtraBackup备份原理：直接拷贝数据库文件和redo log，并且记录下当前二进制日志中已经提交的最后一个事务标记。在新的数据库实例上完成 recovery 操作。Binlog日志的replication是point in time的恢复而不是增量备份。而XtraBackup支持innodb存储引擎的增量备份：**

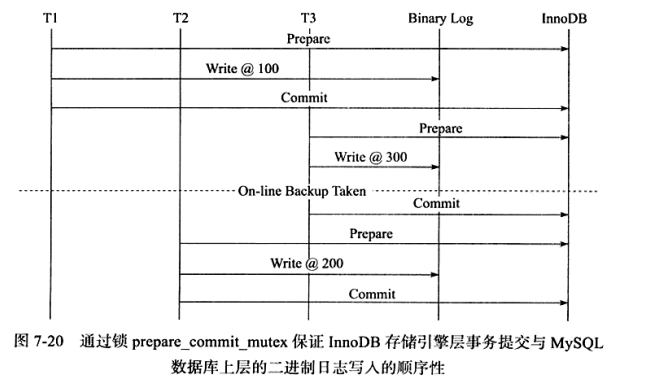
1. 首先完成一个全量备份，记录下此时检查点的LSN；
2. 在进行增量备份，比较表空间中每页的LSN是否大于上次备份的LSN，如果是则备份该页，并记录当前监测点的LSN。

此时XtraBackup会将msater的数据库日志文件直接拷贝到slave，且拷贝的日志截止为存储引擎最新提交的T3，即CHANGE MASTER TO 的日志偏移量在 T3 事务之后。然后从CHANGE MASTER TO的日志偏移量开始增量备份。如果最新提交redo日志之前任然有部分事务没有提交，我们执行热备（如xtraBackup）复制数据时这部分事务的数据就会丢失。

所以我们需要保证，任意已提交事务，之前都没有未提交事务！这样就需要保证binlog和redo log的顺序性。

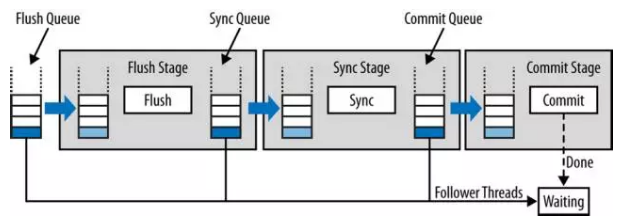
1. **prepare\_commit\_mutex锁的二阶段提交**

为了保证数据库上层日志（binlog）和Innodb层事务提交顺序一致，Mysql使用了prepare\_commit\_mutex锁，Step3中 的3-1 写缓冲不可以在其他事务执行3-2 fsync时执行，也就导致了group commit失效。



1. **三阶段提交（Binary log group commit）**

MySQL 5.6 BLGC技术出现后，在这种情况下，不但MySQL数据库上层二进制日志写入是group commit的，InnoDB存储引擎层也是group commit的。此外还移除了原先的锁prepare\_commit\_mutex，从而大大提高了数据库的整体性。其事务的提交（commit）过程分成三个阶段，Flush stage、Sync stage、Commit stage：



Binlog组提交的基本思想是，引入队列机制保证innodb commit顺序与Binlog落盘顺序一致。每个队列中都有一个Leader，用于控制整个队列中所有follower的行为。每个队列的Leader不会立即提交，而是等待一段时间将整个当前组移动到下一阶段的队列里，原leader进入新队列就不一定是leader了。

* Flush：将每个事物的二进制文件写入内存，并通知dump线程dump binlog；
* Sync：将Binlog刷新到磁盘，一次fsync完成组提交（具体是否fsync由sync\_binlog控制）；
* Commit：存储引擎事务组提交（具体是否fsync由innodb\_flush\_at\_trx\_commit控制）

1. **参考**
2. Mysql多主多从<https://www.cnblogs.com/liangshaoye/p/9810393.html>
3. Double write <https://www.cnblogs.com/nandi001/p/11662992.html>
4. Double write以及oracle <http://www.dataguru.cn/article-11087-1.html>
5. 主从校验 <https://www.cnblogs.com/nandi001/p/11727724.html>
6. Mvcc <https://blog.csdn.net/SnailMann/article/details/94724197>
7. 关于InnoDB中mvcc和覆盖索引查询的困惑？<https://www.zhihu.com/question/27674363>
8. Group commit <https://www.cnblogs.com/mao3714/p/8734838.html>
9. Group commit上次日志和引擎日志不一致原因<http://www.openskill.cn/article/588>
10. MySQL隔离级别为读提交的时候为什么会出现的数据和日志不一致，还必须把binlog格式设置为row？https://www.zhihu.com/question/344037151