InnoDB磁盘文件

InnoDB的主要的磁盘文件主要分为三大块：**一是系统表空间，二是用户表空间，三是redo日志文件和归档文件。**二进制文件(binlog)等文件是MySQL Server层维护的文件，所以未列入InnoDB的磁盘文件中。

系统表空间和用户表空间

InnoDB系统表空间包含InnoDB数据字典(元数据以及相关对象)并且double write buffer,change buffer,undo logs的存储区域。系统表空间也默认包含任何用户在系统表空间创建的表数据和索引数据。系统表空间是一个共享的表空间因为它是被多个表共享的。

系统表空间是由一个或者多个数据文件组成。**默认情况下,1个初始大小为10MB，名为ibdata1的系统数据文件在MySQL的data目录下被创建。**用户可以使用 `innodb_data_file_path` 对数据文件的大小和数量进行配置。

`innodb_data_file_path` 的格式如下：

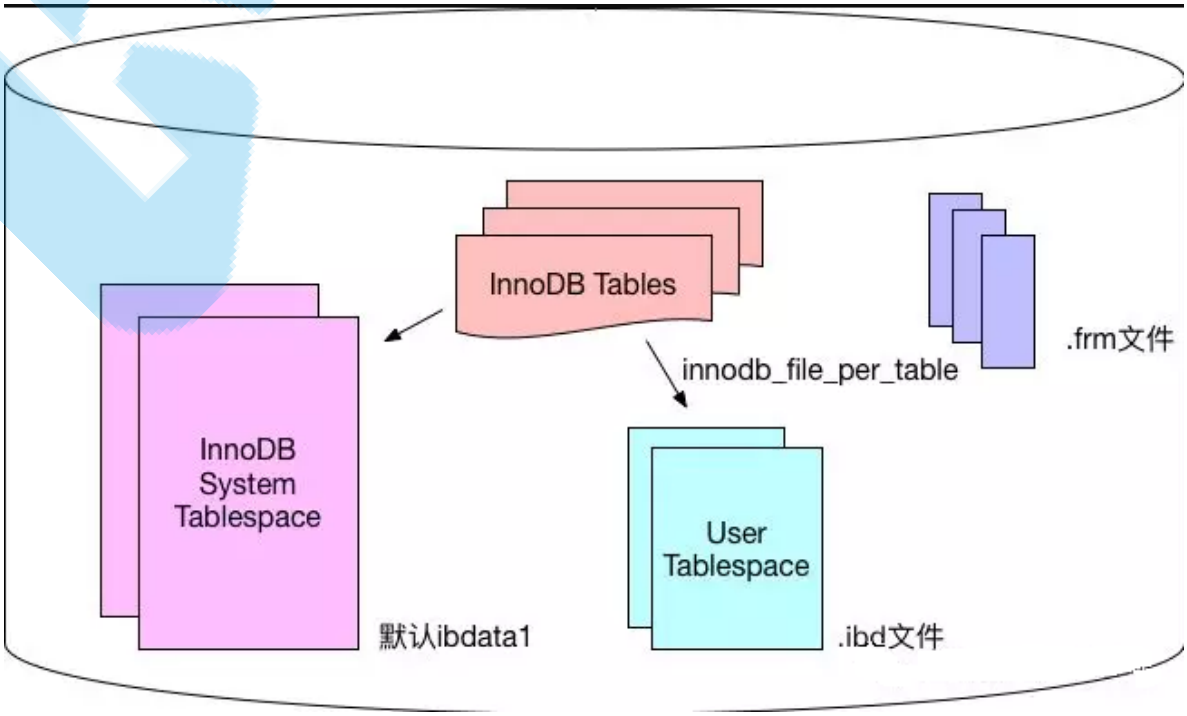
```
innodb_data_file_path=datafile1[,datafile2]...
```

用户可以通过多个文件组成一个表空间，同时制定文件的属性：

```
innodb_data_file_path = /db/ibdata1:1000M;/dr2/db/ibdata2:1000M:autoextend
```

设置**innodb_data_file_path**参数之后，所有基于InnoDB存储引擎的表的数据都会记录到该系统表空间中，如果设置了参数**innodb_file_per_table**，则用户可以将每个基于InnoDB存储引擎的表**产生一个独立的用户表空间。用户表空间的命名规则为：表名.ibd。**通过这种方式，用户不用将所有数据都存放于默认的系统表空间中，但是用户表空间只存储该表的数据、索引和插入缓冲BITMAP等信息，其余信息还是存放在默认的系统表空间中。

下图显示InnoDB存储引擎对于文件的存储方式，其中frm文件是表结构定义文件，记录每个表的表结构定义。



系统表空间（共享表空间）

1、数据字典(data dictionary)：记录数据库相关信息

2、doublewrite write buffer：解决部分写失败（页断裂）

3、insert buffer：内存insert buffer数据，周期写入共享表空间，防止意外宕机

4、回滚段(rollback segments)

5、undo空间：undo页

用户表空间

1、每个表的数据和索引都会存在自己的表空间中

2、undo空间：undo页 （需要设置）

3、doublewrite write buffer：解决部分写失败（页断裂）

重做日志文件和归档文件

默认情况下，在InnoDB存储引擎的数据目录下会有两个名为**ib_logfile0**和**ib_logfile1**的文件，这就是InnoDB的**重做日志文件(redo log file)**，它**记录了对于InnoDB存储引擎的事务日志。**

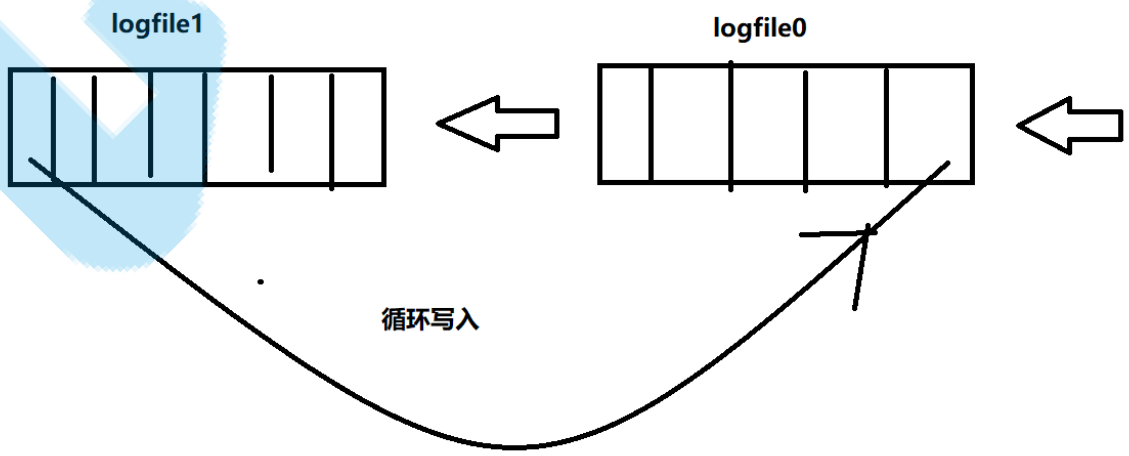
当InnoDB的数据存储文件发生错误时，重做日志文件就能派上用场。InnoDB存储引擎可以使用重做日志文件将数据恢复为正确状态，以此来保证数据的正确性和完整性。

每个InnoDB存储引擎至少有1个重做日志文件组(group)，每个文件组下至少有2个重做日志文件，如默认的ib_logfile0和ib_logfile1。

为了得到更高的可靠性，用户可以设置多个镜像日志组，将不同的文件组放在不同的磁盘上，以此来提高重做日志的高可用性。

在日志组中每个重做日志文件的大小一致，并以【循环写入】的方式运行。InnoDB存储引擎先写入重做日志文件1，当文件被写满时，会切换到重做日志文件2，再当重做日志文件2也被写满时，再切换到重做日志文件1。

当bufferpool数据落盘时，RedoLogfile中得内容 被相应得清除。

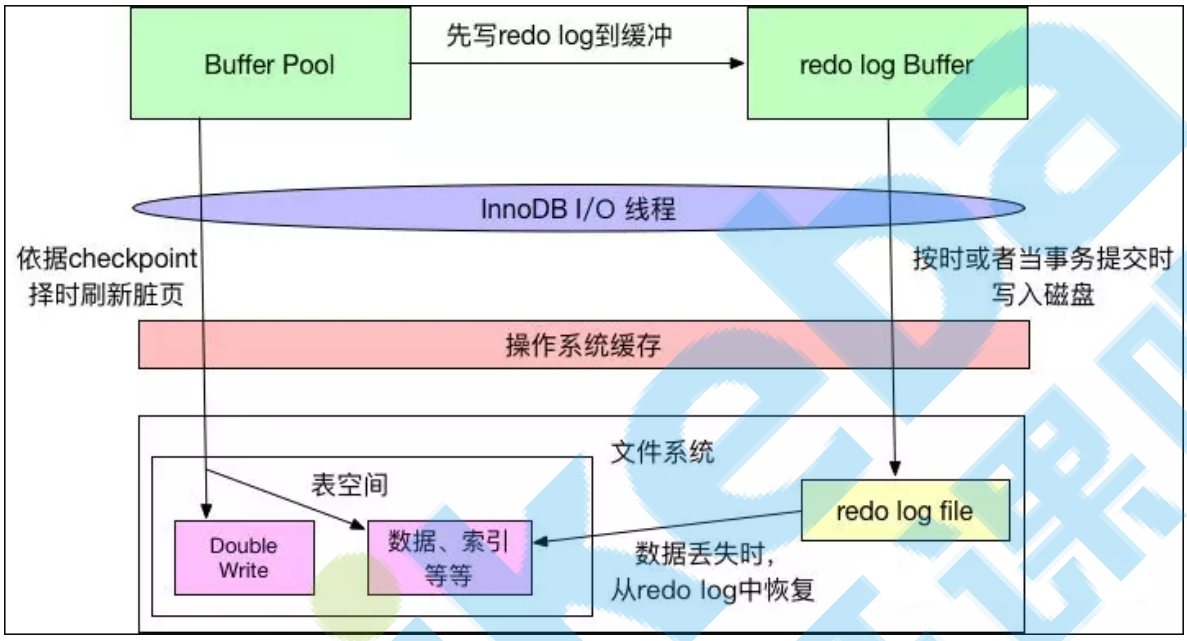


用户可以使用innodb_log_file_size来设置重做日志文件的大小，这对InnoDB存储引擎的性能有着非常大的影响。

如果重做日志文件设置的太大，数据丢失时，恢复时可能需要很长的时间；另一方面，如果设置的太小，重做日志文件太小会导致依据checkpoint的检查需要频繁刷新脏页到磁盘中，导致性能的抖动。

重做日志的落盘机制

InnoDB对于数据文件和日志文件的刷盘遵守WAL(Write ahead redo log) 和Force-log-at-commit两种规则，二者保证了事务的持久性。WAL要求数据的变更写入到磁盘前，首先必须将内存中的日志写入到磁盘；Force-log-at-commit要求当一 个事务提交时，所有产生的日志都必须刷新到磁盘上，如果日志刷新成功后，缓冲池中的数据刷新到磁盘前数据库发生了宕机，那么重启时，数据库可以从日志中恢复数据。



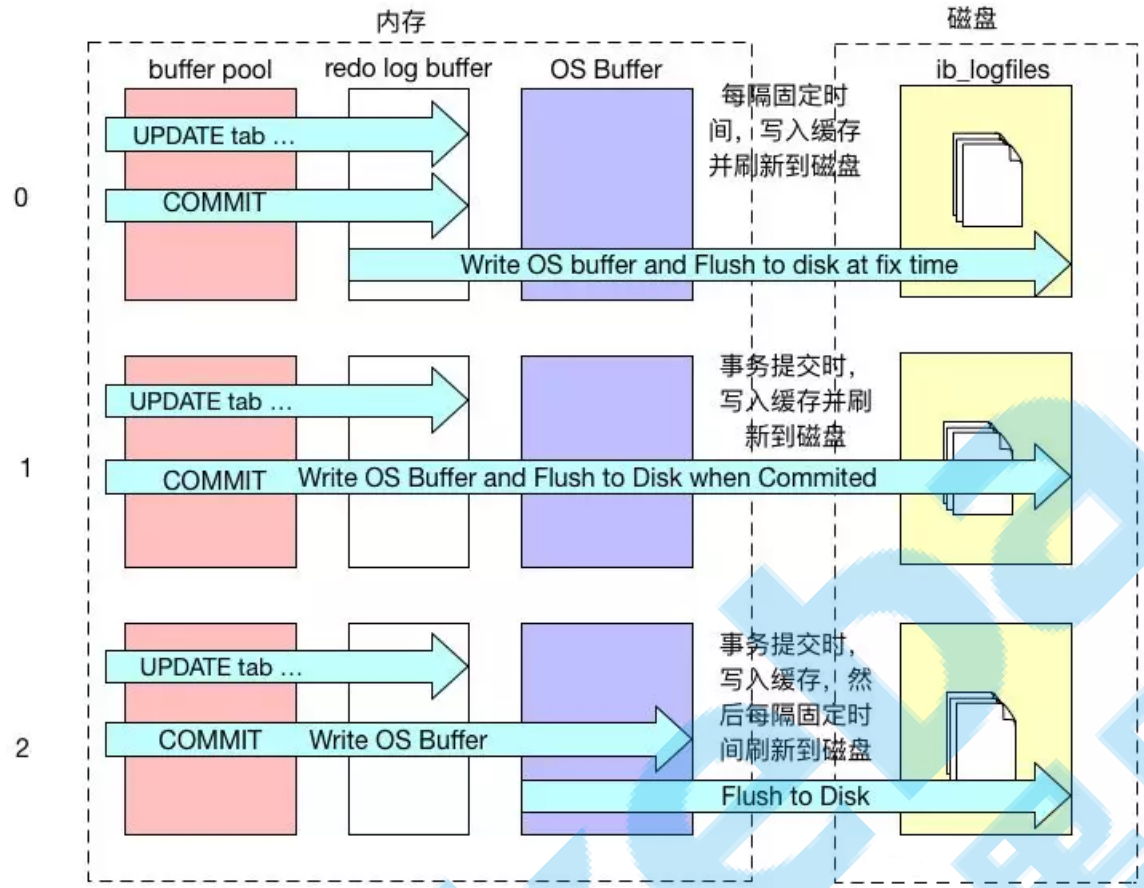
如上图所示，InnoDB在缓冲池中变更数据时，会首先将相关变更写入重做日志缓冲中，然后再按时或者当事务提交时写入磁盘，这符合Force-log-at-commit原则；当重做日志写入磁盘后，缓冲池中的变更数据才会依据checkpoint机制择时写入到磁盘中，这符合WAL原则。

在checkpoint择时机制中，就有重做日志文件写满的判断，所以，如前文所述，如果重做日志文件太小，经常被写满，就会频繁导致checkpoint将更改的数据写入磁盘，导致性能抖动。

操作系统的文件系统是带有缓存的，当InnoDB向磁盘写入数据时，有可能只是写入到了文件系统的缓存中，没有真正的“落袋为安”。

InnoDB的**innodb_flush_log_at_trx_commit**属性可以控制每次事务提交时InnoDB的行为。当属性值为0时，事务提交时，不会对重做日志进行写入操作，而是等待主线程按时写入；当属性值为1时，事务提交时，会将重做日志写入文件系统缓存，并且调用文件系统的fsync，将文件系统缓冲中的数据真正写入磁盘存储，确保不会出现数据丢失；当属性值为2时，事务提交时，也会将日志文件写入文件系统缓存，但是不会调用fsync，而是让文件系统自己去判断何时将缓存写入磁盘。

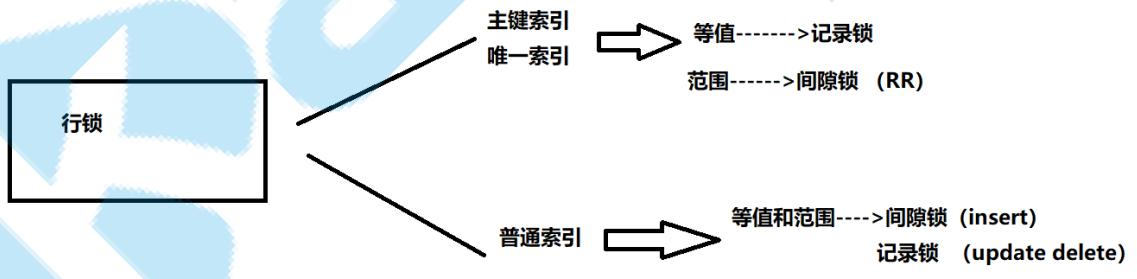
日志的刷盘机制如下图所示：

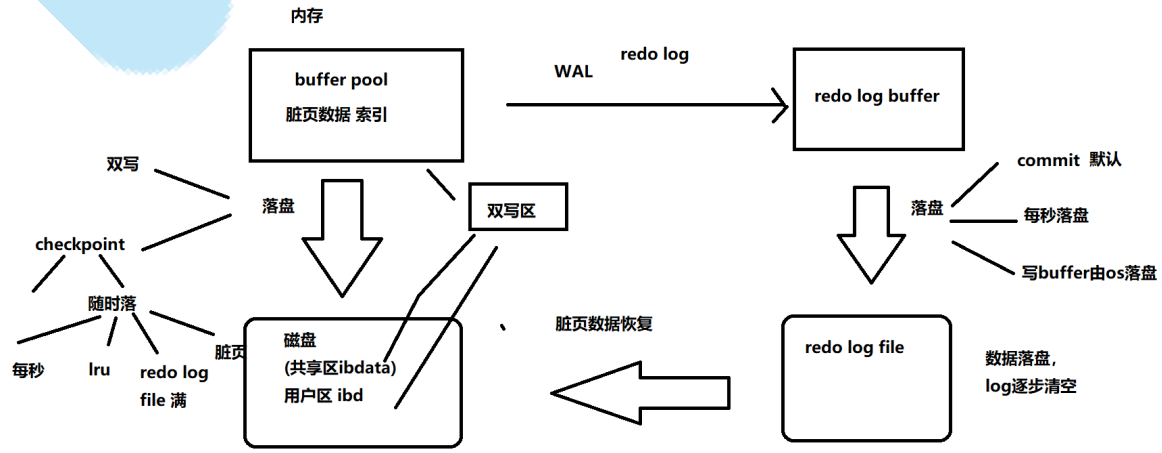


---

innodb_flush_log_at_commit是InnoDB性能调优的一个基础参数，涉及InnoDB的写入效率和数据安全。当参数值为0时，写入效率最高，但是数据安全最低；参数值为1时，写入效率最低，但是数据安全最高；参数值为2时，二者都是中等水平。一般建议将该属性值设置为1，以获得较高的数据安全性，而且也只有设置为1，才能保证事务的持久性。

复习：





课堂主题

MySQL事务分析 、MySQL行锁分析、MySQL死锁分析

课堂目标

理解RedoLog与UndoLog的关系

理解UndoLog的作用

掌握数据和回滚日志的逻辑存储结构

掌握原子性、一致性和持久性的实现机制

理解MVCC的概念

分辨当前读和快照读

理解一致性非锁定读

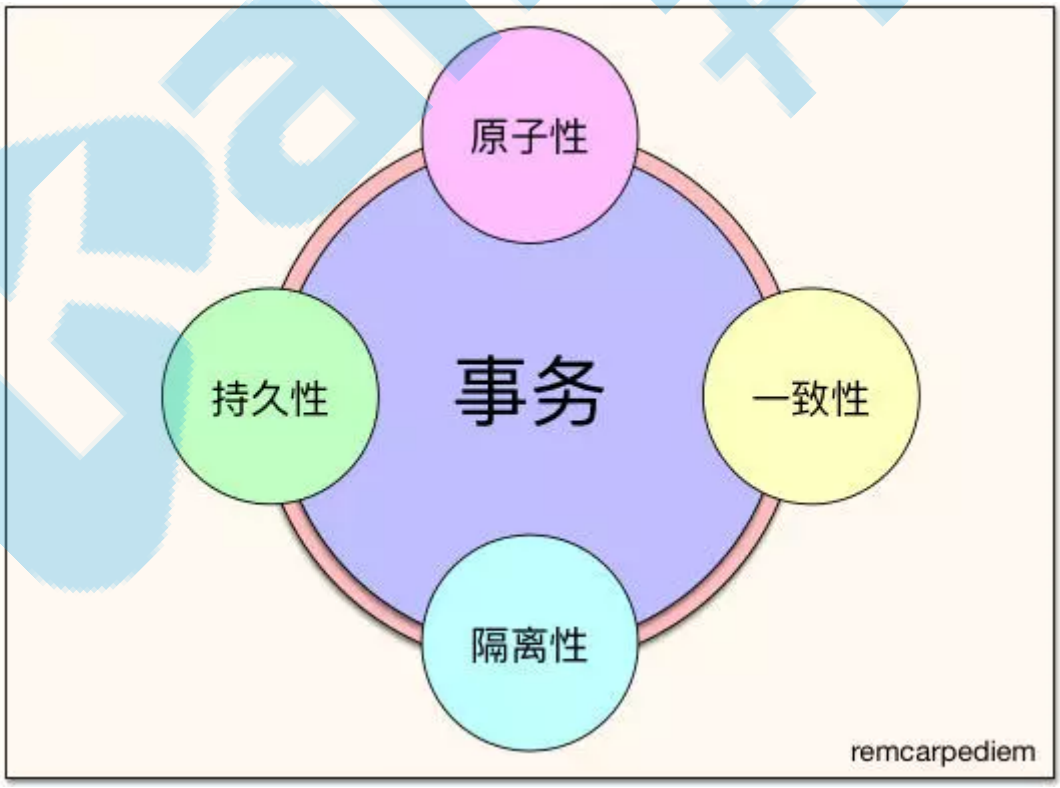
能够分析SQL语句的加锁过程

理解死锁的原理

合理的避免死锁

会查看死锁日志

InnoDB的事务分析



数据库事务具有ACID四大特性。ACID是以下4个词的缩写：

- 原子性(atomicity)：事务最小工作单元，要么全成功，要么全失败 。
- 一致性(consistency)： 事务开始和结束后，数据库的完整性不会被破坏 。

- 隔离性(isolation)：不同事务之间互不影响，四种隔离级别为RU（读未提交）、RC（读已提交）、RR（可重复读）、SERIALIZABLE （串行化）。
- 持久性(durability)：事务提交后，对数据的修改是永久性的，即使系统故障也不会丢失 。

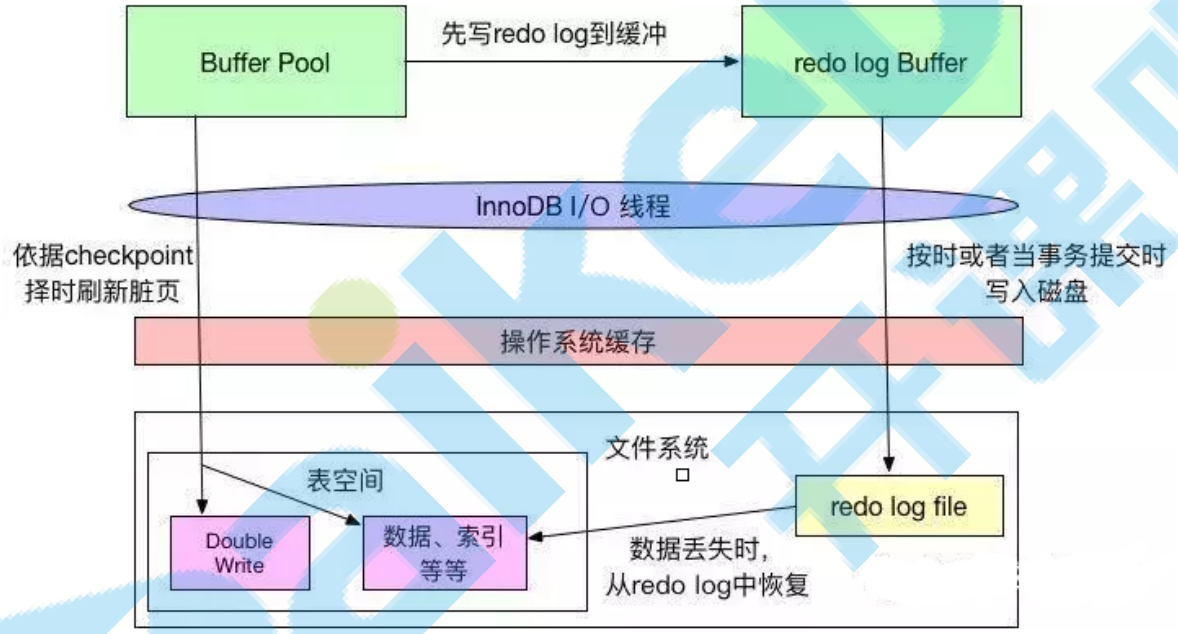
下面我们就来详细讲解一下上述示例涉及的事务的ACID特性的具体实现原理。**总结来说，事务的隔离性由多版本控制机制和锁实现，而原子性、一致性和持久性通过InnoDB的redo log、undo log和Force Log at Commit机制来实现。**

原子性，持久性和一致性

原子性，持久性和一致性主要是通过redo log、undo log和Force Log at Commit机制机制来完成的。redo log用于在崩溃时恢复数据，undo log用于对事务的影响进行撤销，也可以用于多版本控制。而Force Log at Commit机制保证事务提交后redo log日志都已经持久化。

RedoLog

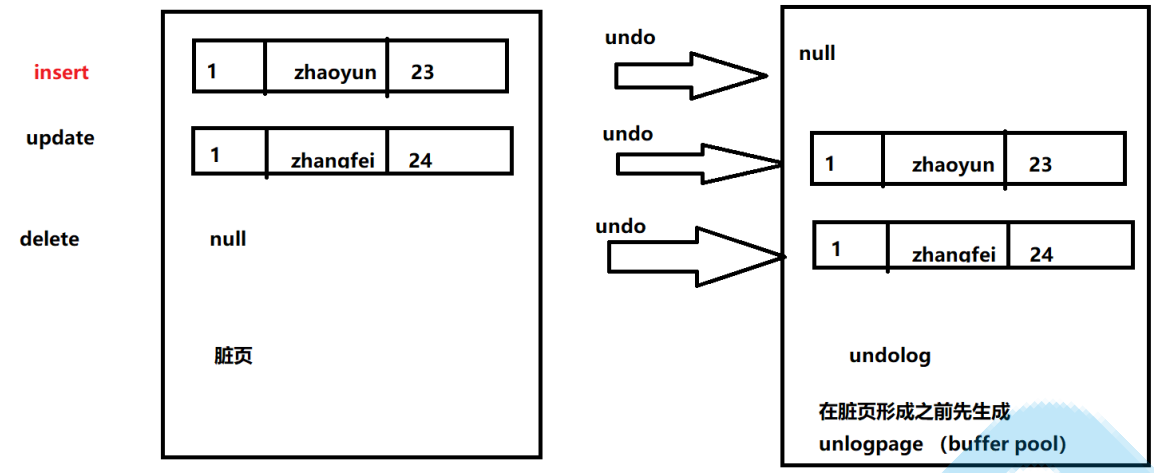
数据库日志和数据落盘机制，如下图所示：



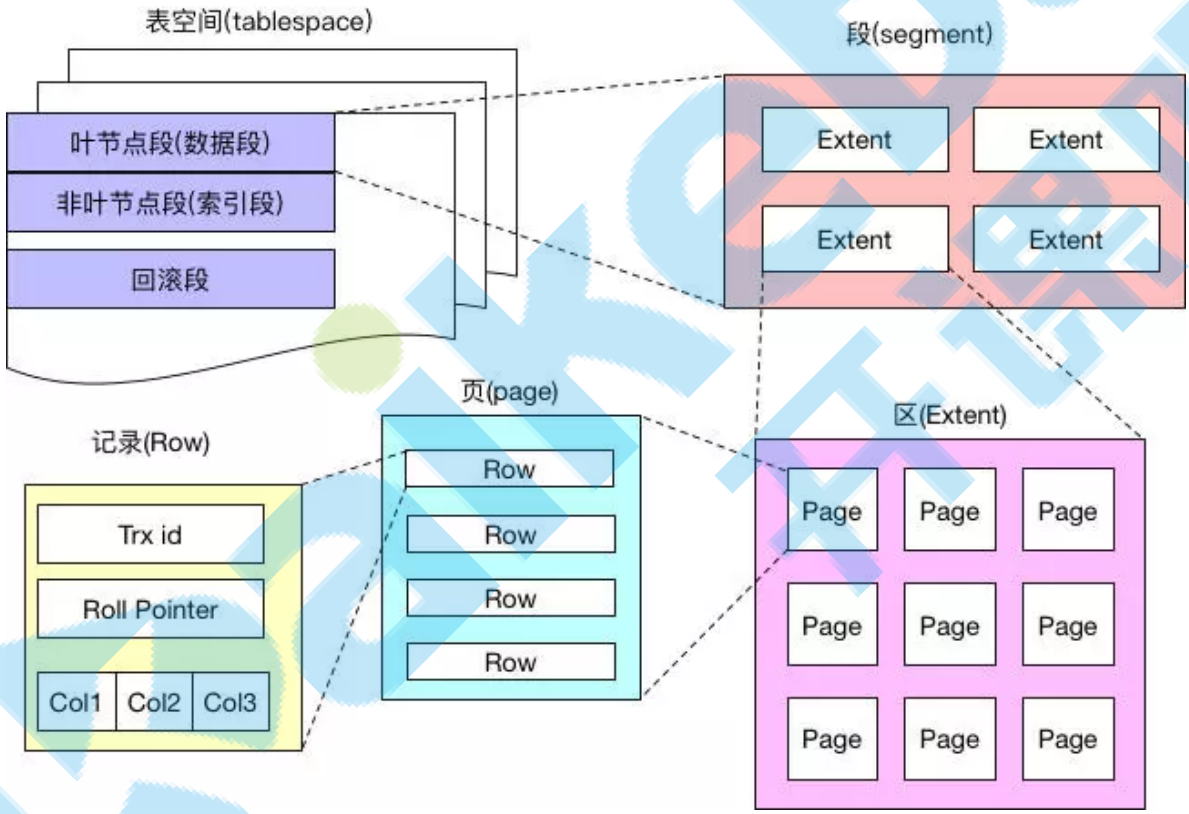
redo log写入磁盘时，必须进行一次操作系统的fsync操作，防止redo log只是写入了操作系统的磁盘缓存中。参数innodb_flush_log_at_trx_commit可以控制redo log日志刷新到磁盘的策略

UndoLog

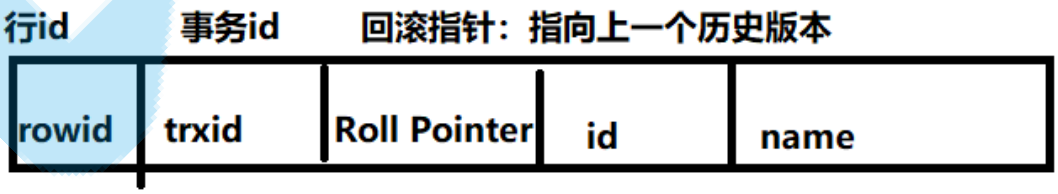
数据库崩溃重启后需要从redo log中把未落盘的脏页数据恢复出来，重新写入磁盘，保证用户的数据不丢失。当然，在崩溃恢复中还需要回滚没有提交的事务。由于回滚操作需要undo日志的支持，undo日志的完整性和可靠性需要redo日志来保证，所以崩溃恢复先做redo恢复数据，然后做undo回滚。



在事务执行的过程中，除了记录redo log，还会记录一定量的undo log。undo log记录了数据在每个操作前的状态，如果事务执行过程中需要回滚，就可以根据undo log进行回滚操作。

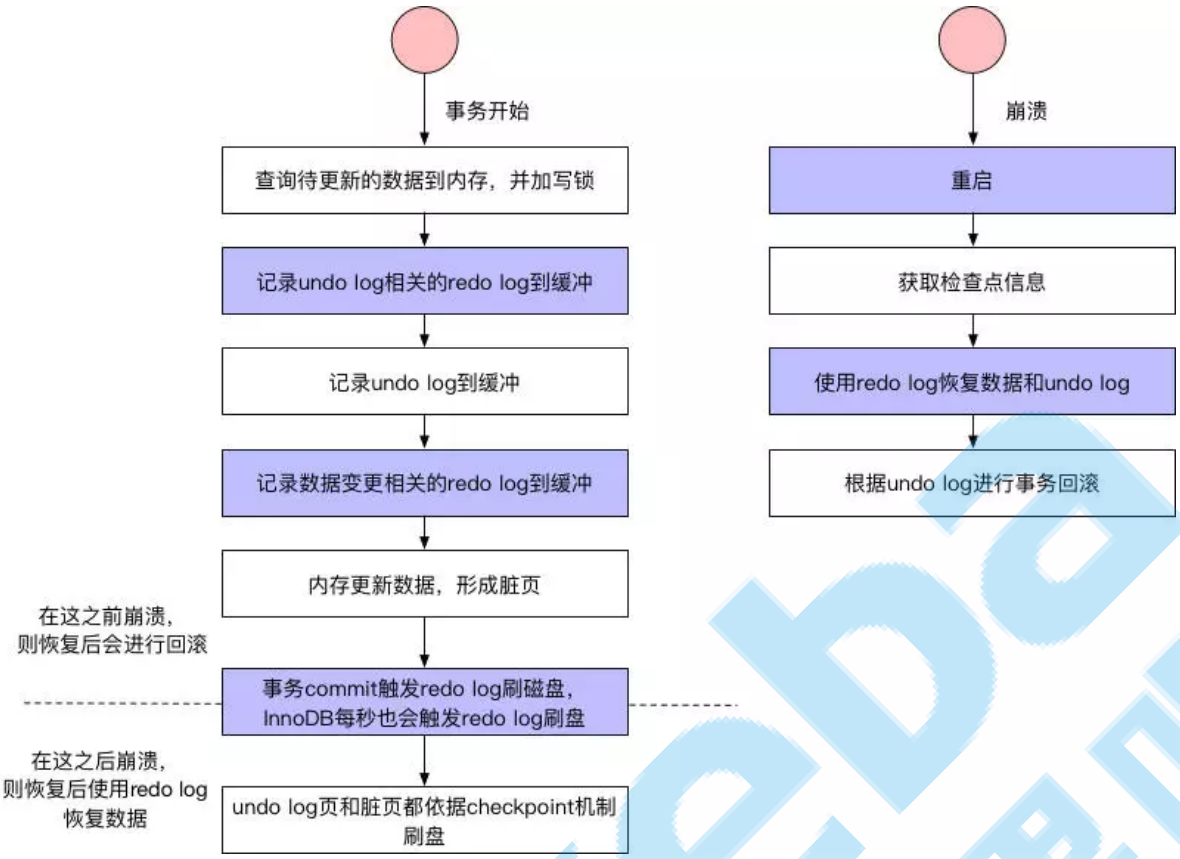


数据和回滚日志的逻辑存储结构

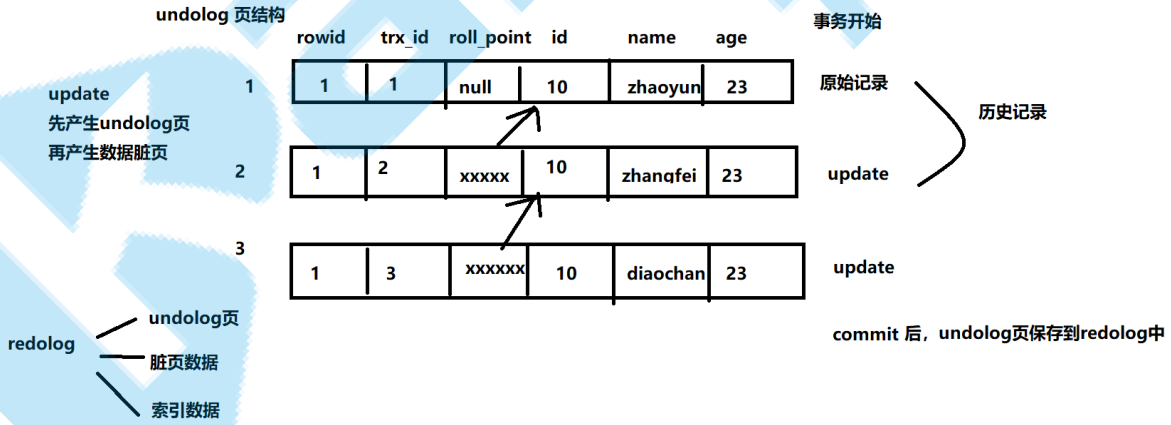


undo log的存储不同于redo log，它存放在数据库内部的一个特殊的段(segment)中，这个段称为回滚段。回滚段位于共享表空间中。undo段中的以undo page为更小的组织单位。**undo page和存储数据库数据和索引的页类似。因为redo log是物理日志，记录的是数据库页的物理修改操作。所以undo log（也看成数据库数据）的写入也会产生redo log，也就是undo log的产生会伴随着redo log的产生，这是因为undo log也需要持久性的保护。**如上图所示，表空间中有回滚段和叶节点段和非叶节点段，而三者都有对应的页结构。

我们再来总结一下数据库事务的整个流程，如下图所示。



事务进行过程中，每次sql语句执行，都会记录undo log和redo log，然后更新数据形成脏页，然后redo log按照时间或者空间等条件进行落盘，undo log和脏页按照checkpoint进行落盘，落盘后相应的redo log就可以删除了。此时，事务还未COMMIT，如果发生崩溃，则首先检查checkpoint记录，使用相应的redo log进行数据和undo log的恢复，然后查看undo log的状态发现事务尚未提交，然后就使用undo log进行事务回滚。事务执行COMMIT操作时，会将本事务相关的所有redo log都进行落盘，只有所有redo log落盘成功，才算COMMIT成功。然后内存中的数据脏页继续按照checkpoint进行落盘。如果此时发生了崩溃，则只使用redo log恢复数据。



隔离性

事务并发问题

在事务的**并发操作**中可能会出现一些问题：

- **丢失更新**：两个事务针对同一数据都发生修改操作时，会存在丢失更新问题。
- **脏读**：一个事务读取到另一个事务未提交的数据。
- **不可重复读**：一个事务因读取到另一个事务**已提交的update或者delete数据。**导致对同一条记录读取两次以上的结果不一致。
- **幻读**：一个事务因读取到另一个事务**已提交的insert数据。**导致*对同一张表读取*两次以上的结果不一致。

事务隔离级别

- **四种隔离级别（SQL92标准）：**

  现在来看看MySQL数据库为我们提供的四种隔离级别（*由低到高*）：

  ① Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

  ② Read committed (RC，读已提交)：可避免脏读的发生。

  ③ **Repeatable read** (RR，可重复读)：可避免脏读、不可重复读的发生。

  （**注意事项：InnoDB的RR还可以解决幻读，主要原因是Next-Key（Gap）锁，只有RR才能使用Next-Key锁**）

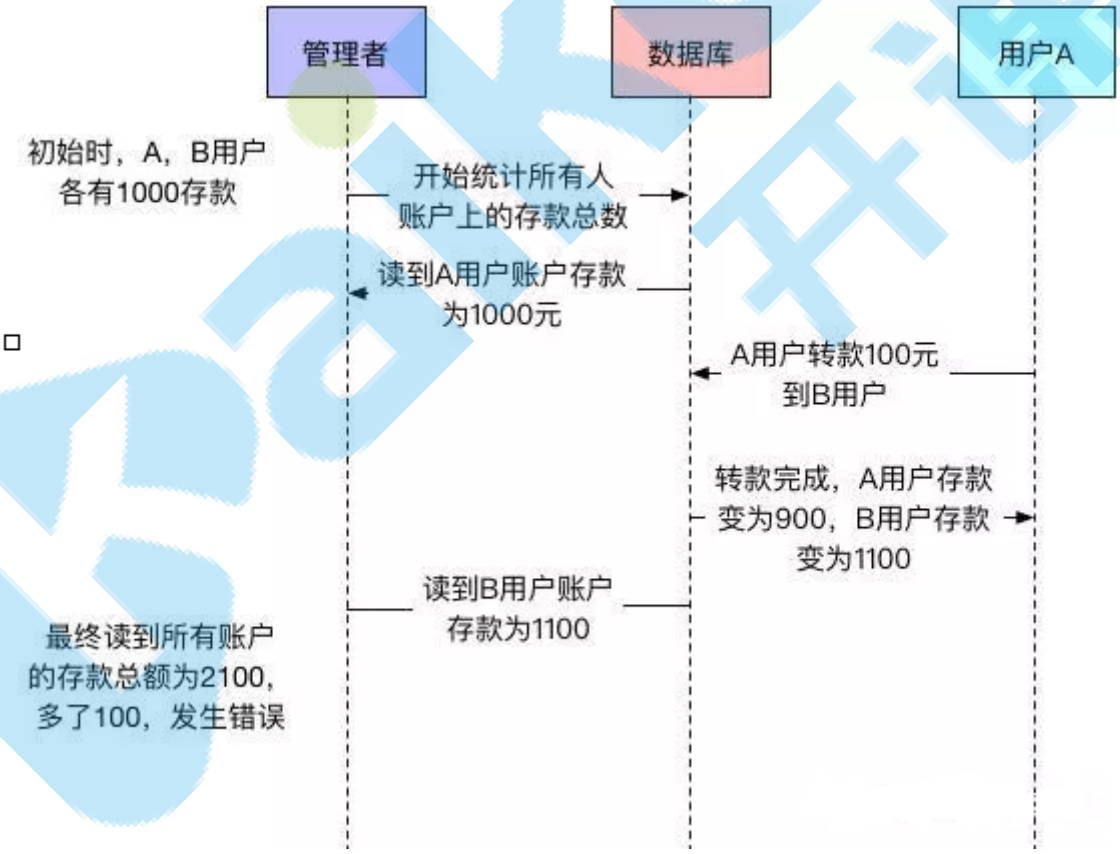
  ④ Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

  （**由MVCC降级为Locking-Base CC**）

考虑一个现实场景：

管理者要查询所有用户的存款总额，假设除了用户A和用户B之外，其他用户的存款总额都为0，A、B用户各有存款1000，所以所有用户的存款总额为2000。但是在查询过程中，用户A会向用户B进行转账操作。转账操作和查询总额操作的时序图如下图所示。

转账和查询的时序图：

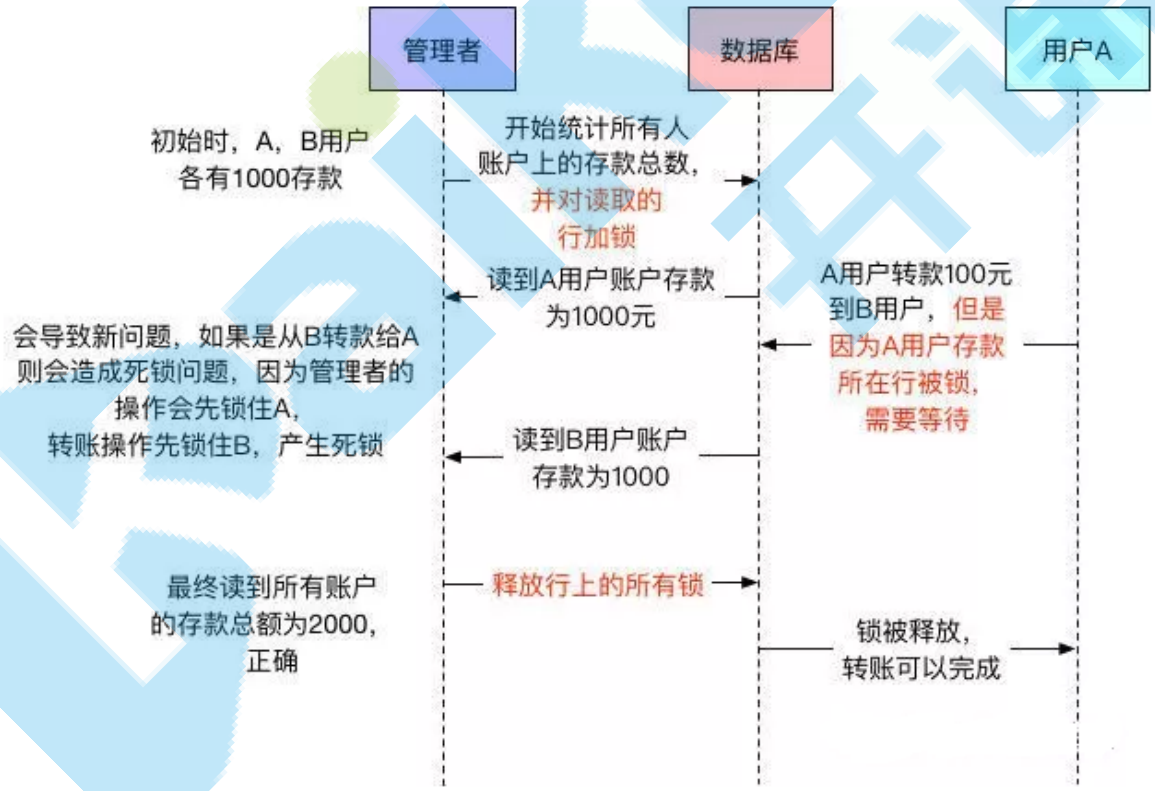


如果没有任何的并发控制机制，查询总额事务先读取了用户A的账户存款，然后转账事务改变了用户A和用户B的账户存款，最后查询总额事务继续读取了转账后的用户B的账号存款，导致最终统计的存款总额多了100元，发生错误。

```
--创建账户表并初始化数据
create table tacount(id int , aname varchar(100),acount int , primary key(id));
alter table tacount add index idx_name(aname);
insert into tacount values(1,'a',1000);
```

```
insert into tacount values(2,'b',1000);
--设置隔离级读未提交（read-uncommitted）
 mysql> set session transaction isolation level read uncommitted;
 --session 1
 mysql> start transaction ; select * from tacount where aname='a';
+----+-------+--------+
| id | aname | acount |
+----+-------+--------+
|  1 | a     |   1000 |
+----+-------+--------+
--session 2
mysql> start transaction; update tacount set acount=1100 where aname='b';
--session 1
mysql> select * from tacount where aname='b';
+----+-------+--------+
| id | aname | acount |
+----+-------+--------+
|  2 | b     |   1100 |
+----+-------+--------+
```

使用锁机制(LBCC)可以解决上述的问题。查询总额事务会对读取的行加锁，等到操作结束后再释放所有行上的锁。因为用户A的存款被锁，导致转账操作被阻塞，直到查询总额事务提交并将所有锁都释放。使用锁机制：

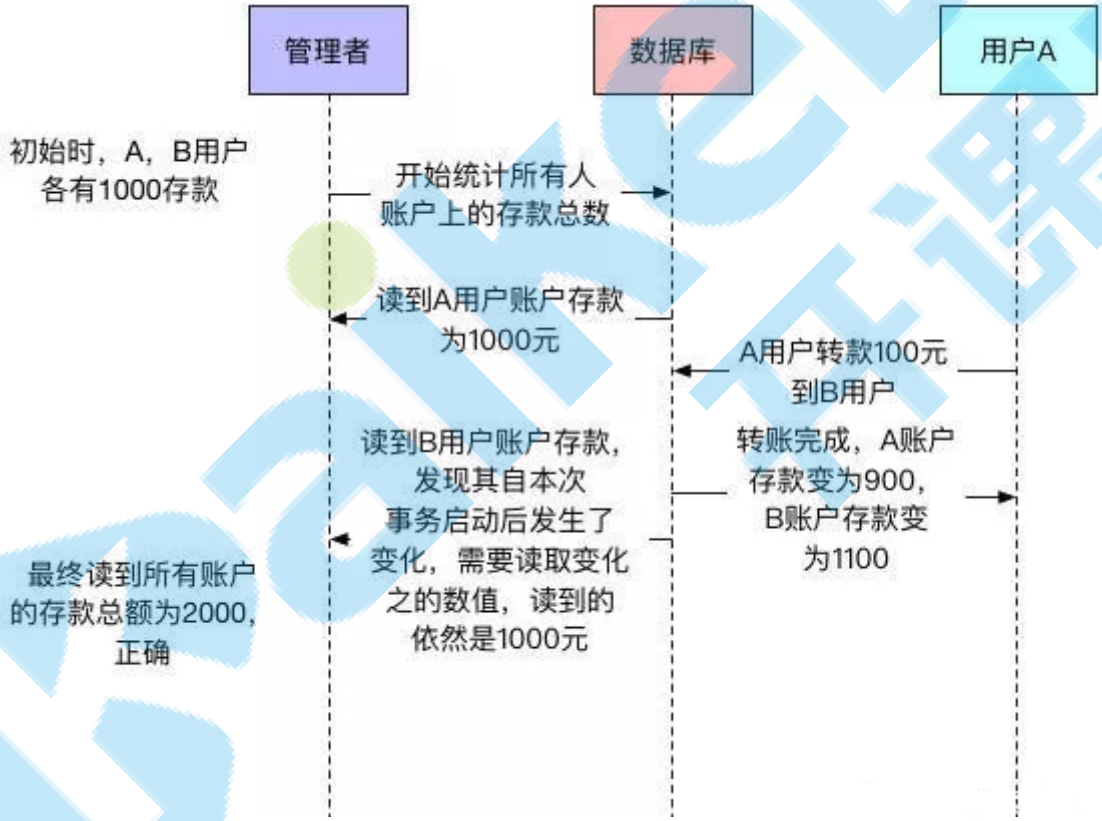


但是这时可能会引入新的问题，当转账操作是从用户B向用户A进行转账时会导致死锁。转账事务会先锁住用户B的数据，等待用户A数据上的锁，但是查询总额的事务却先锁住了用户A数据，等待用户B的数据上的锁。

```
--设置隔离级别为串行化（serializable）  死锁演示
mysql> set session transaction isolation level serializable;
--session 1
mysql> start transaction;select * from tacount where aname='a';
--session 2
mysql> start transaction ; update tacount set acount=900 where aname='b';
-- session 1
mysql> select * from tacount where aname='b';
-- session 2
mysql> update tacount set acount=1100 where aname='a';
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting
transaction
```

使用MVCC机制可以解决这个问题。查询总额事务先读取了用户A的账户存款，然后转账事务会修改用户A和用户B账户存款，查询总额事务读取用户B存款时不会读取转账事务修改后的数据，而是读取本事务开始时的数据副本(在REPEATABLE READ隔离等级下)。

使用MVCC机制（RR隔离级别下的演示情况）：

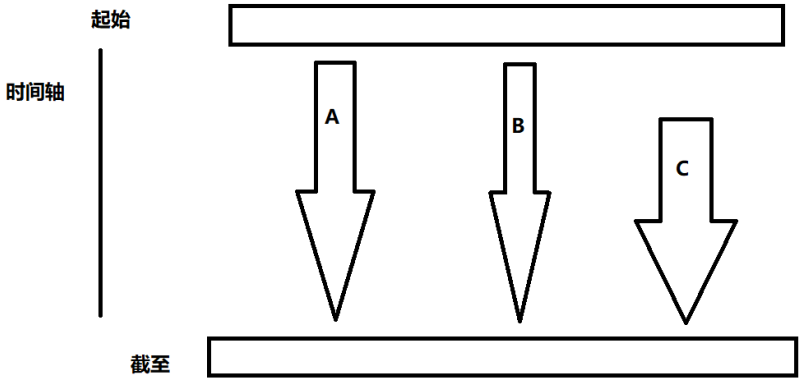


MVCC使得数据库读不会对数据加锁，普通的SELECT请求不会加锁，提高了数据库的并发处理能力。借助MVCC，数据库可以实现READ COMMITTED，REPEATABLE READ等隔离级别，用户可以查看当前数据的前一个或者前几个历史版本，保证了ACID中的I特性（隔离性）。

```
-- 显示当前隔离级别为 REPEATABLE-READ  MySQL默认隔离级别
mysql> select @@tx_isolation;
-- session 1
mysql> start transaction ; select * from tacount where aname='a';
+----+-------+--------+
| id | aname | acount |
+----+-------+--------+
|  1 | a     |   1000 |
+----+-------+--------+
-- session 2
```

```
mysql> start transaction; update tacount set acount=1100 where aname='a';
-- session 1
mysql> select * from tacount where aname='a';
+----+-------+--------+
| id | aname | acount |
+----+-------+--------+
|  1 | a     |   1000 |
+----+-------+--------+
-- session 2  提交事务
mysql> commit;
-- session 1  显示在session 1 事务开始时的数据
mysql> select * from tacount where aname='a';
+----+-------+--------+
| id | aname | acount |
+----+-------+--------+
|  1 | a     |   1000 |
+----+-------+--------+

-- 设置事务隔离级别为REPEATABLE-COMMITTED  读已提交
-- session 1
mysql> set session transaction isolation level read committed;
mysql> start transaction ; select * from tacount where aname='a';
+----+-------+--------+
| id | aname | acount |
+----+-------+--------+
|  1 | a     |   1000 |
+----+-------+--------+
-- session 2
mysql> set session transaction isolation level read committed;
mysql> start transaction; update tacount set acount=1100 where aname='a';
-- session 1
mysql> select * from tacount where aname='a';
+----+-------+--------+
| id | aname | acount |
+----+-------+--------+
|  1 | a     |   1000 |
+----+-------+--------+
-- session 2  提交事务
mysql> commit;
-- session 1   显示最新事务提交后的数据
mysql> select * from tacount where aname='a';
+----+-------+--------+
| id | aname | acount |
+----+-------+--------+
|  1 | a     |   1100 |
+----+-------+--------+
```



InnoDB的MVCC实现

我们首先来看一下wiki上对MVCC的定义：

> **Multiversion concurrency control (MCC or MVCC)**, is a concurrency control method commonly used by database management systems to provide concurrent access to the database and in programming languages to implement transactional memory.

当前读和快照读

在MVCC并发控制中，读操作可以分成两类：**快照读 (snapshot read)与当前读 (current read)。**

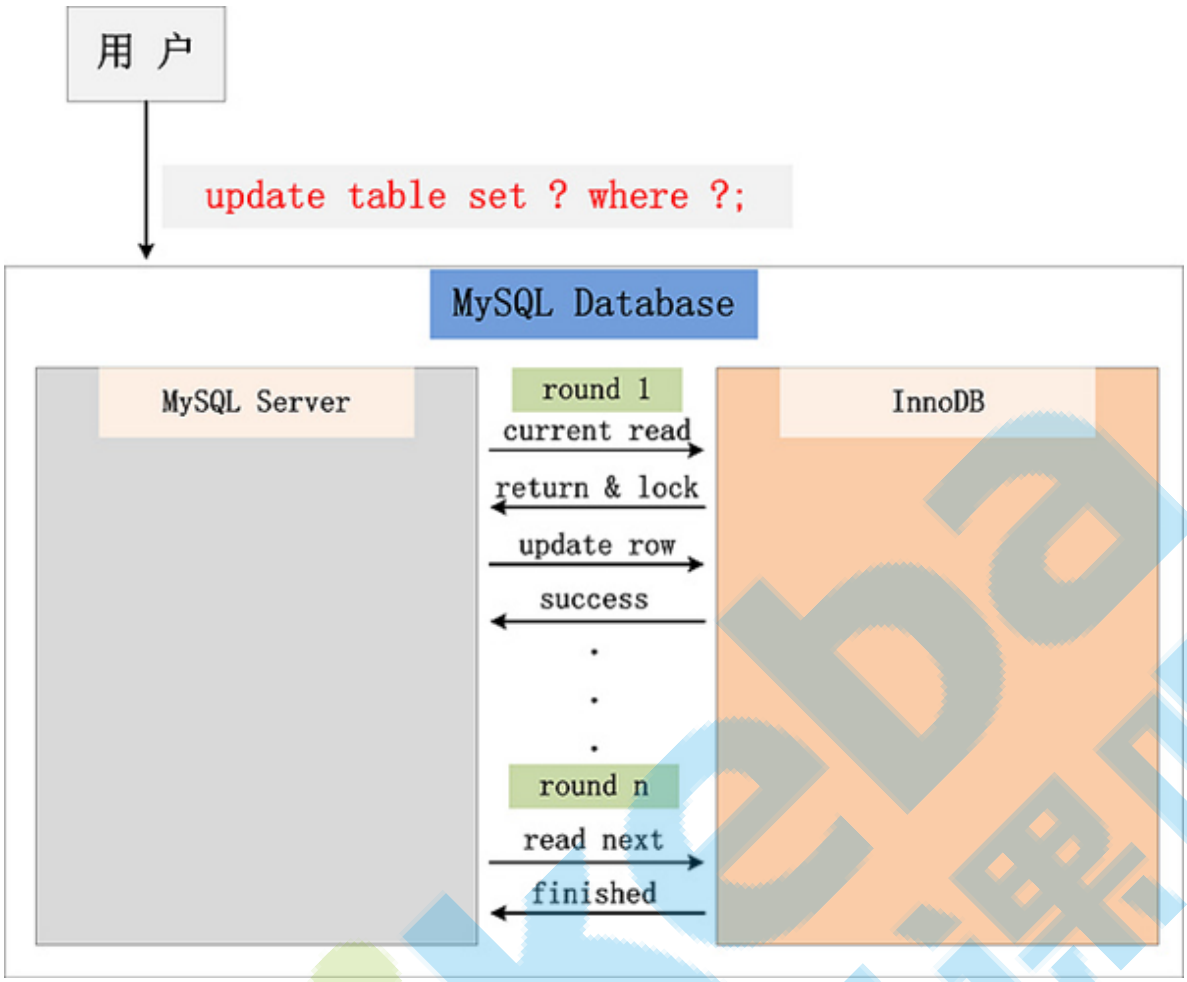
```
- 快照读，读取的是记录的可见版本 (有可能是历史版本)，不用加锁。(select)
- 当前读，读取的是记录的最新版本，并且当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。
```

在一个支持MVCC并发控制的系统中，哪些读操作是快照读？哪些操作又是当前读呢？

以MySQL InnoDB为例：

快照读：简单的select操作，属于快照读，不加锁。(当然，也有例外，下面会分析) 不加读锁 读历史版本

当前读：特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁。 加行写锁 读当前版本



一致性非锁定读

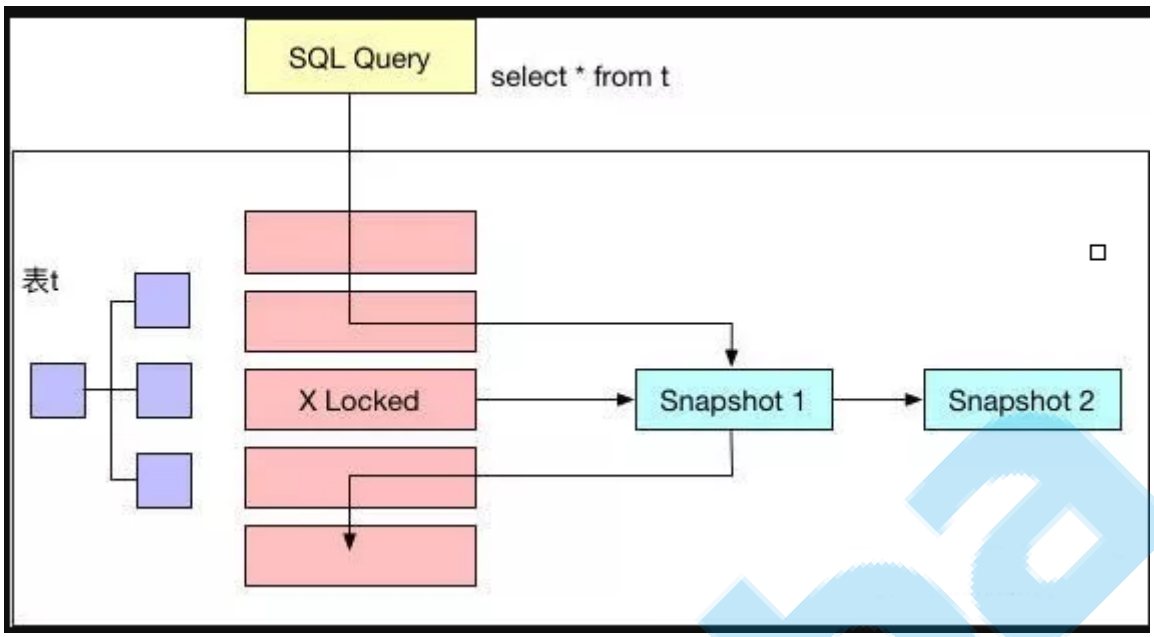
一致性非锁定读(consistent nonlocking read)是指InnoDB存储引擎通过多版本控制(MVCC)读取当前数据库中行数据的方式。

如果读取的行正在执行DELETE或UPDATE操作，这时读取操作不会因此去等待行上锁的释放。相反地，InnoDB会去读取行的一个最新可见快照。

undolog

1、回滚

2、让mvcc读历史版本



会话A和会话B示意图：

会话A	会话B
BEGIN	
SELECT * FROM test WHERE id = 1;	
	BEGIN
	UPDATE test SET id = 3 WHERE id = 1;
SELECT * FROM test WHERE id = 1;	
	COMMIT;
SELECT * FROM test WHERE id = 1;	
COMMIT	

如上图所示，当会话B提交事务后，会话A再次运行`SELECT * FROM test WHERE id = 1`的SQL语句时，两个事务隔离级别下得到的结果就不一样了。

MVCC 在mysql 中的实现依赖的是 undo log 与 read view 。

Undo Log

InnoDB行记录有三个隐藏字段：分别对应该行的rowid、事务号db_trx_id和回滚指针db_roll_ptr，其中db_trx_id表示最近修改的事务的id，db_roll_ptr指向回滚段中的undo log。

根据行为的不同，undo log分为两种：insert undo log和update undo log

insert undo log：

是在 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，

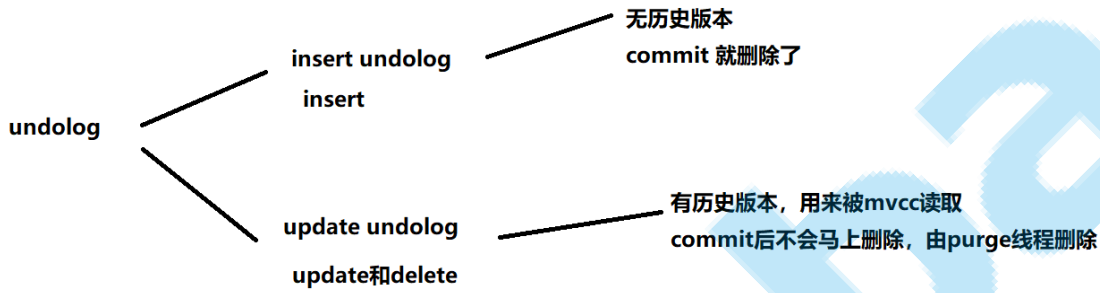
rollback 在该事务中直接删除 ， 不需要进行 purge 操作 (purge Thread)

update undo log：

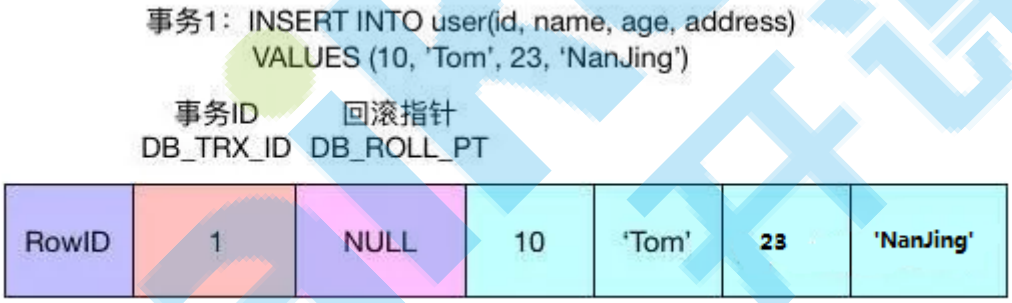
是 update 或 delete 操作中产生的 undo log，因为会对已经存在的记录产生影响，

rollback MVCC机制会找他的历史版本进行恢复

是 update 或 delete 操作中产生的 undo log，因为会对已经存在的记录产生影响，为了提供 MVCC机制，因此 update undo log 不能在事务提交时就进行删除，而是将事务提交时放到入 history list 上，等待 purge 线程进行最后的删除操作。



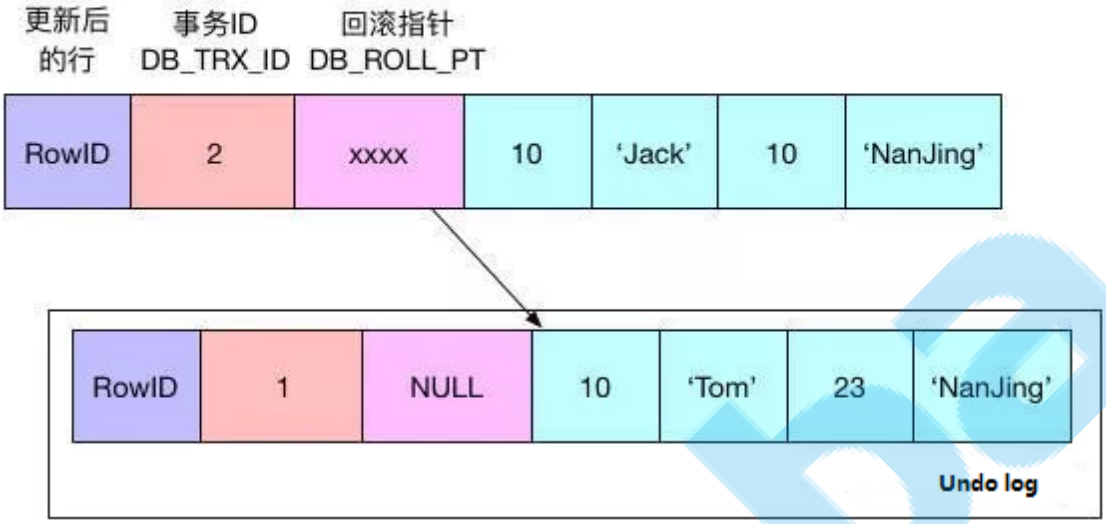
如下图所示（初始状态）：



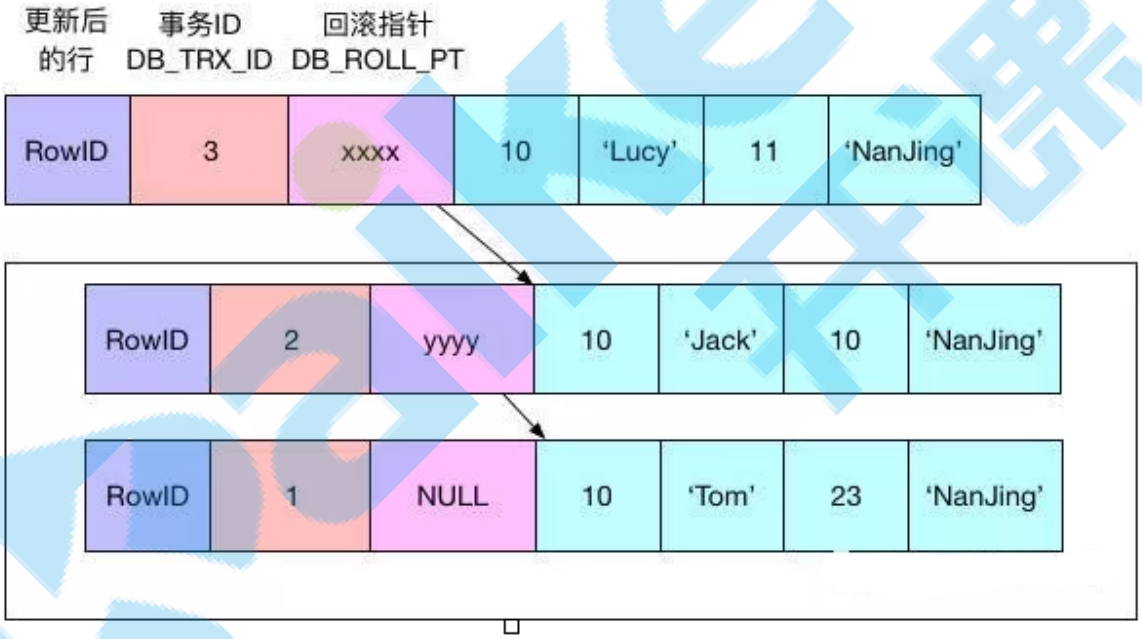
当事务2使用UPDATE语句修改该行数据时，会首先使用排他锁锁定改行，将该行当前的值复制到undo log中，然后再真正地修改当前行的值，最后填写事务ID，使用回滚指针指向undo log中修改前的行。

如下图所示（第一次修改）：

事务2：UPDATE user SET name='Jack', age=10 WHERE id = 10



当事务3进行修改与事务2的处理过程类似，如下图所示（第二次修改）：



事务链表

MySQL中的事务在开始到提交这段过程中，都会被保存到一个叫trx_sys的事务链表中，这是一个基本的链表结构：

ct-trx --> trx11 --> trx9 --> trx6 --> trx5 --> trx3;

事务链表中保存的都是还未提交的事务，事务一旦被提交，则会被从事务链表中摘除。

RR隔离级别下，在每个事务开始的时候，会将当前系统中的所有的活跃事务拷贝到一个列表中(read view)

RC隔离级别下，在每个语句开始的时候，会将当前系统中的所有的活跃事务拷贝到一个列表中(read view)

show engine innodb status ,就能够看到事务列表。

ReadView

当前事务（读）能读哪个历史版本?

Read View是事务开启时当前所有事务的一个集合，这个类中存储了当前Read View中最大事务ID及最小事务ID。

这就是当前活跃的事务列表。如下所示，

ct-trx --> trx11 --> trx9 --> trx6 --> trx5 --> trx3;

ct-trx 表示当前事务的id，对应上面的read_view数据结构如下，

```
read_view->creator_trx_id = ct-trx;
read_view->up_limit_id = trx3;    低水位
read_view->low_limit_id = trx11;   高水位
read_view->trx_ids = [trx11, trx9, trx6, trx5, trx3];
```

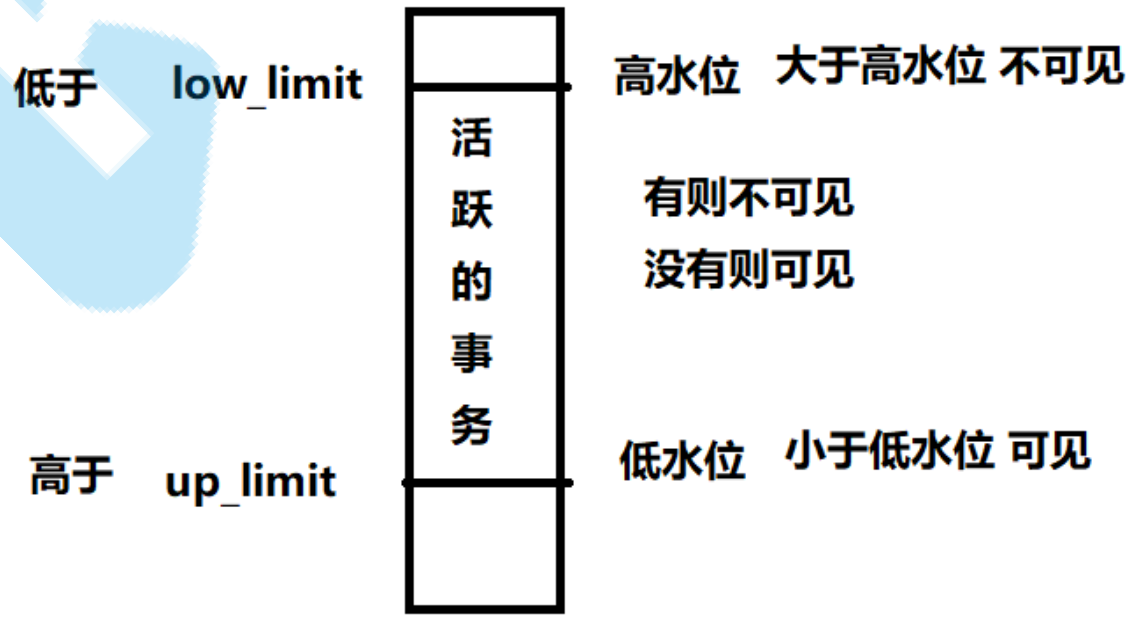
low_limit_id是"高水位"，即当时活跃事务的最大id，如果读到row的db_trx_id>=low_limit_id，说明这些id在此之前的数据都没有提交，如注释中的描述，这些数据都不可见。

```
if (trx_id >= view->low_limit_id) {
    return(FALSE);
}
注：readview 部分源码
```

up_limit_id是"低水位"，即当时活跃事务列表的最小事务id，如果row的db_trx_id<up_limit_id,说明这些数据在事务创建的id时都已经提交，如注释中的描述，这些数据均可见。

```
if (trx_id < view->up_limit_id) {
    return(TRUE);
}
```

row的db_trx_id在low_limit_id和up_limit_id之间，则查找该记录的db_trx_id是否在自己事务的read_view->trx_ids列表中，如果在则该记录的当前版本不可见，否则该记录的当前版本可见。



不同隔离级别ReadView实现方式

1. read-commited:

```
函数：ha_innobase::external_lock

   if (trx->isolation_level <= TRX_ISO_READ_COMMITTED

      && trx->global_read_view) {

      / At low transaction isolation levels we let

      each consistent read set its own snapshot /

   read_view_close_for_mysql(trx);
```

即：在每次语句执行的过程中，都关闭read_view, 重新在row_search_for_mysql函数中创建当前的一份read_view。这样就会产生不可重复读现象发生。

2. repeatable read：

在repeatable read的隔离级别下，创建事务trx结构的时候，就生成了当前的global read view。使用trx_assign_read_view函数创建，一直维持到事务结束。在事务结束这段时间内 每一次查询都不会重新重建Read View， 从而实现了可重复读。