## 课堂主题

MySQL事务分析 、MySQL行锁分析

## 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

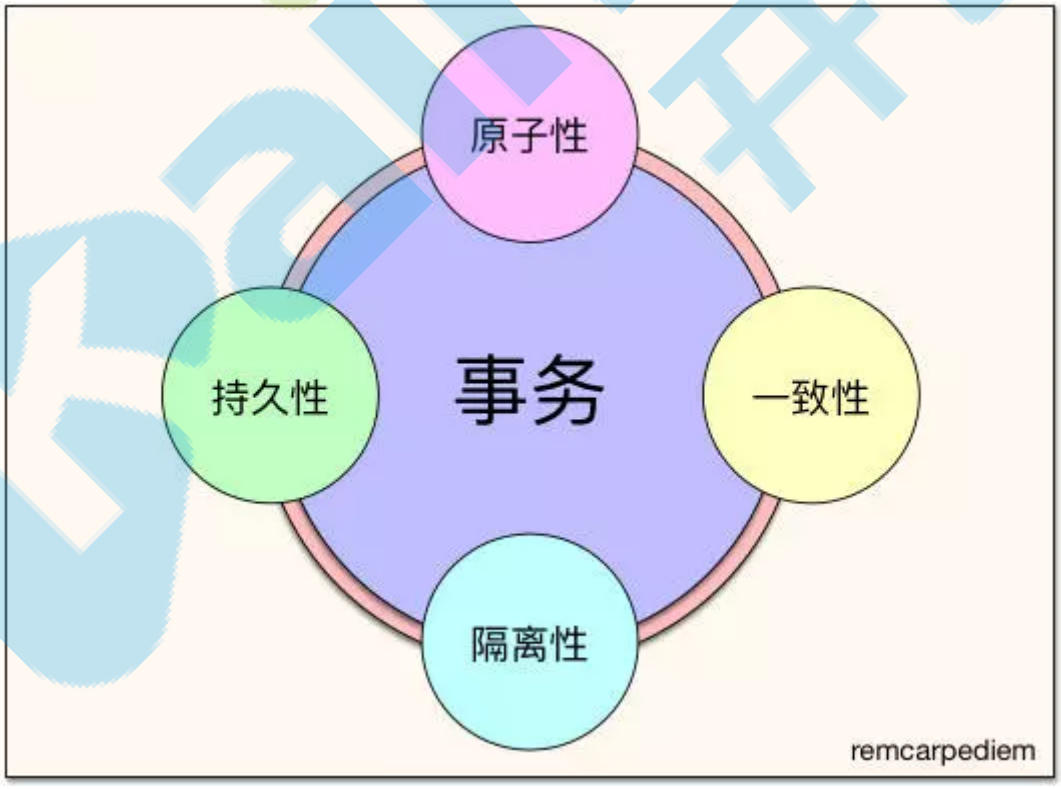
理解MVCC的概念

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

# 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 。

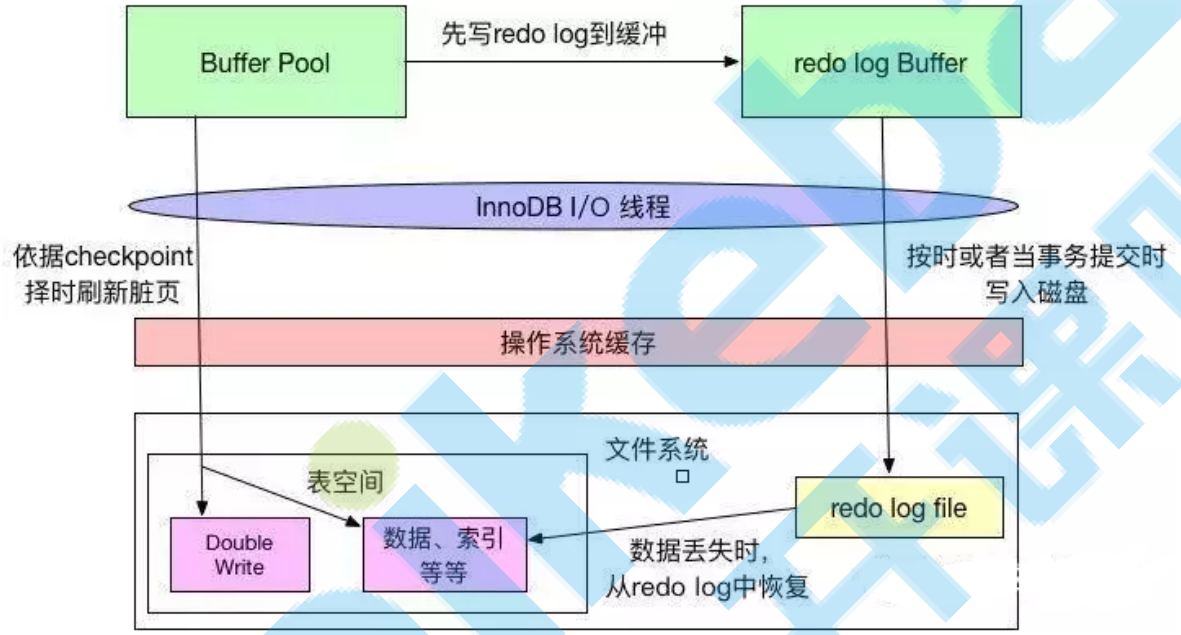
下面我们就来详细讲解一下上述示例涉及的事务的ACID特性的具体实现原理。**总结来说，事务的隔离性由多版本控制机制和锁实现，而原子性、一致性和持久性通过InnoDB的redo log、undo log和Force Log at Commit机制来实现。**

## 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

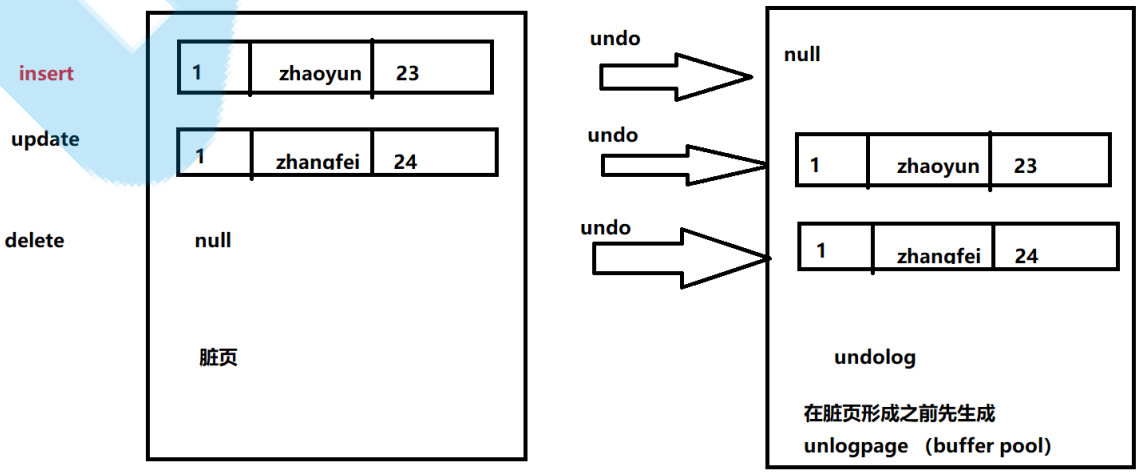
数据库日志和数据落盘机制，如下图所示：



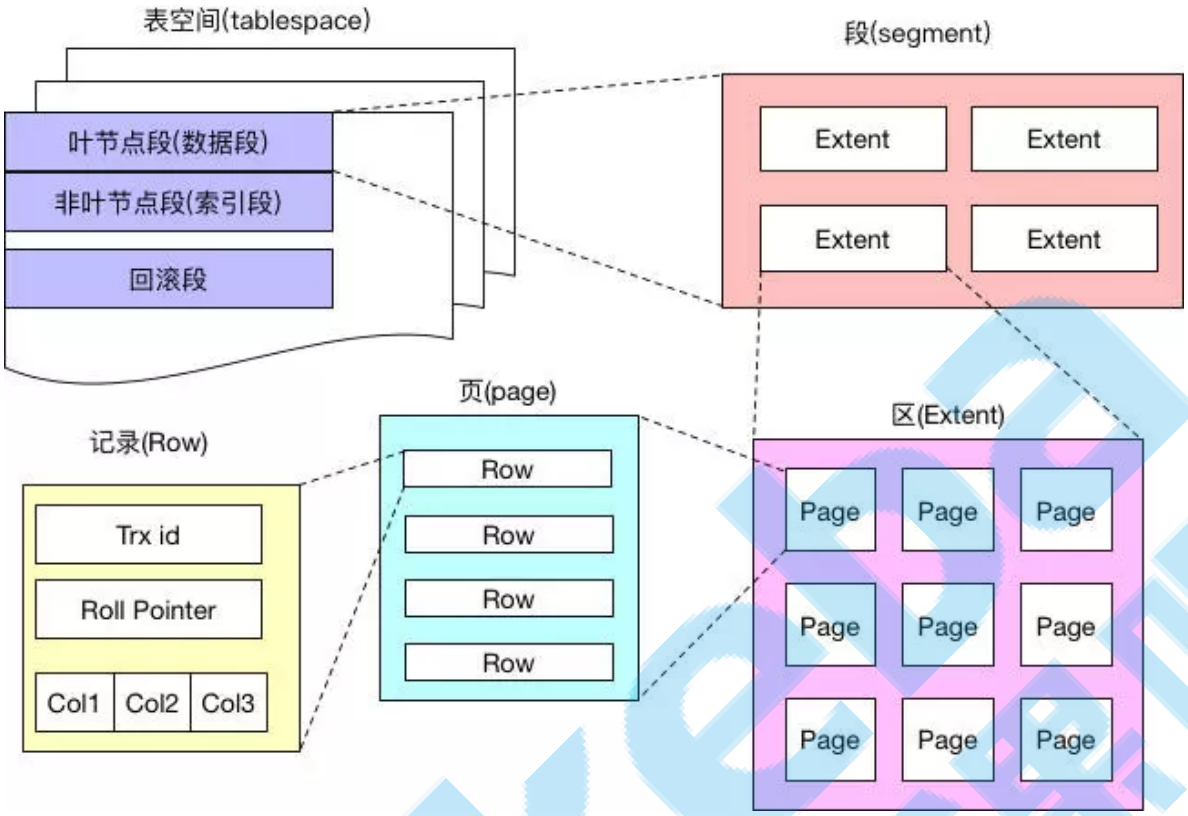

redo log写入磁盘时，必须进行一次操作系统的fsync操作，防止redo log只是写入了操作系统的磁盘缓存中。参数innodb_flush_log_at_trx_commit可以控制redo log日志刷新到磁盘的策略

## 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。**



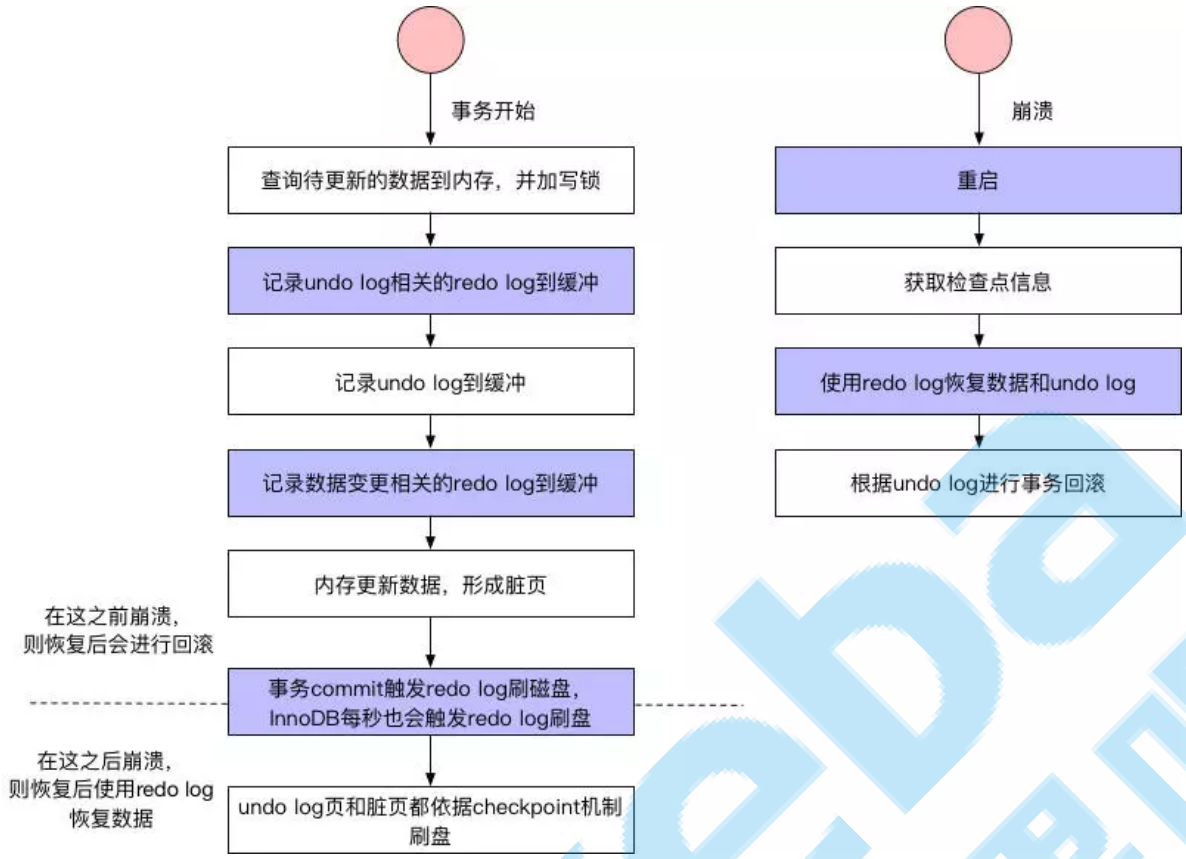
在事务执行的过程中，除了记录redo log，还会记录一定量的undo log。undo log记录了数据在每个操作前的状态，如果事务执行过程中需要回滚，就可以根据undo log进行回滚操作。



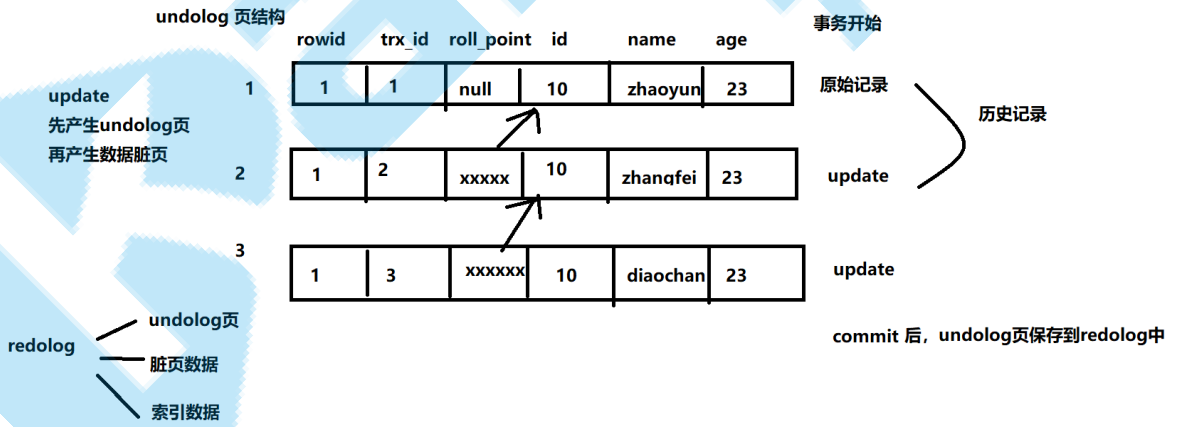

数据和回滚日志的逻辑存储结构


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**）

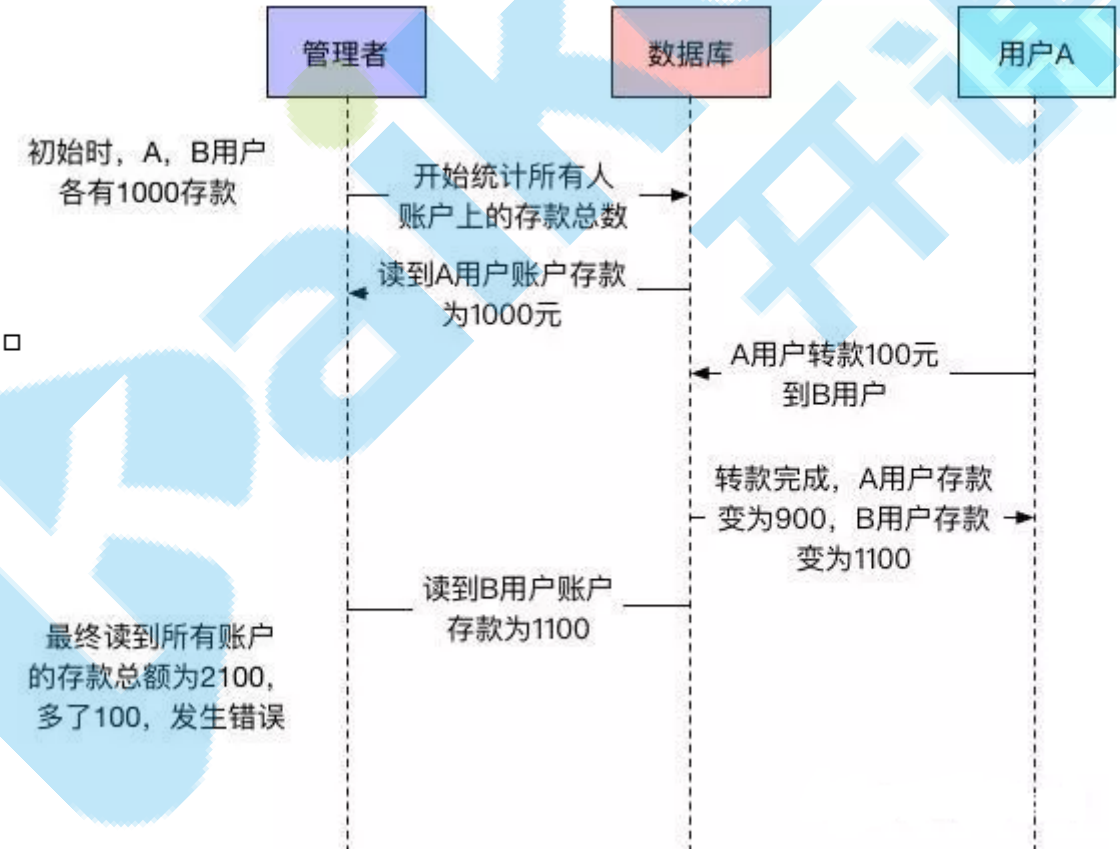
  ④ Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

  （**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：**

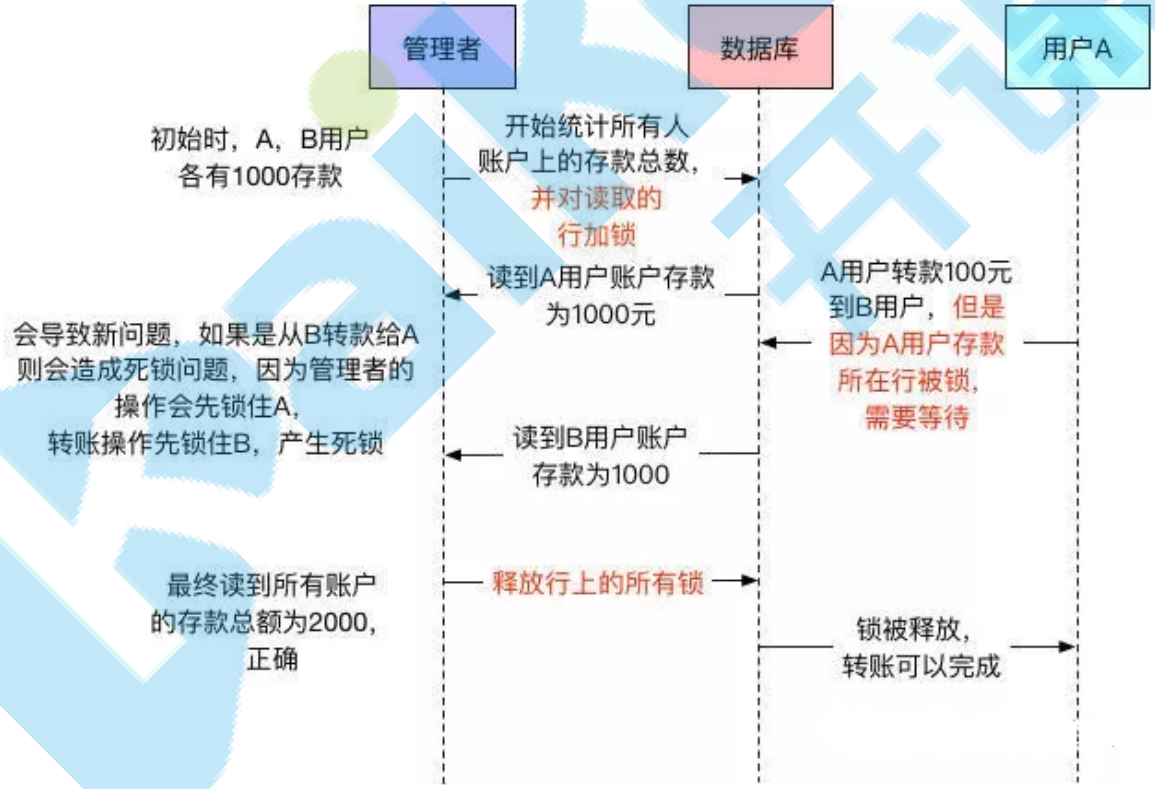



如果没有任何的并发控制机制，查询总额事务先读取了用户A的账户存款，然后转账事务改变了用户A和用户B的账户存款，最后查询总额事务继续读取了转账后的用户B的账号存款，导致最终统计的存款总额多了100元，发生错误。

```
--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：

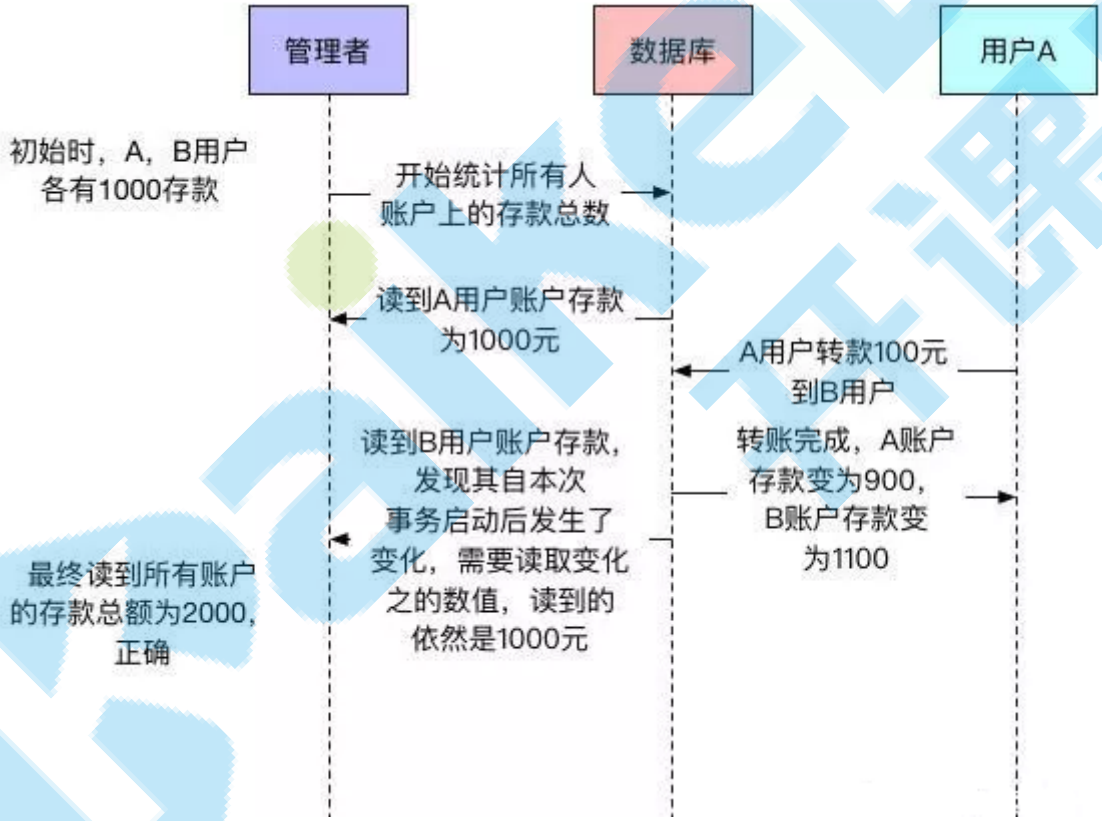



但是这时可能会引入新的问题，当转账操作是从用户B向用户A进行转账时会导致死锁。转账事务会先锁住用户B的数据，等待用户A数据上的锁，但是查询总额的事务却先锁住了用户A数据，等待用户B的数据上的锁。

```
--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）：

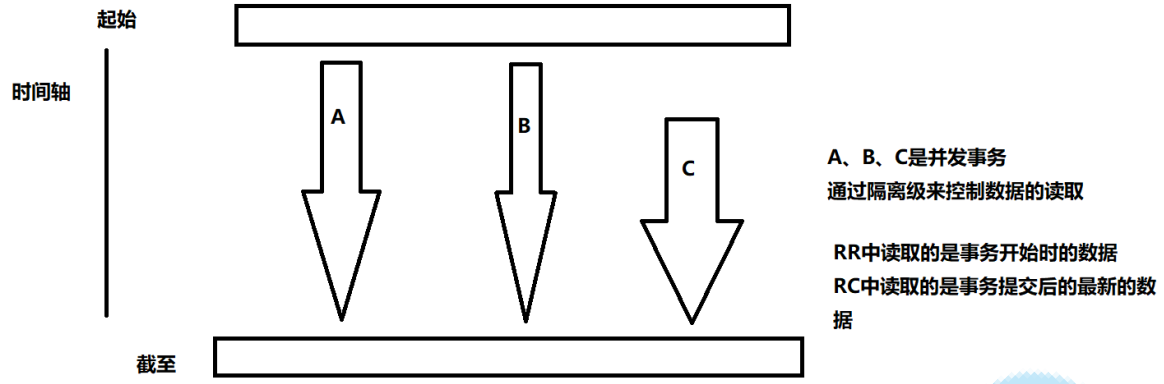



**MVCC使得数据库读不会对数据加锁，普通的SELECT请求不会加锁，提高了数据库的并发处理能力。**借助MVCC，数据库可以实现READ COMMITTED，REPEATABLE READ等隔离级别，用户可以查看当前数据的前一个或者前几个历史版本，保证了ACID中的I特性（隔离性）。

```
-- 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)。**

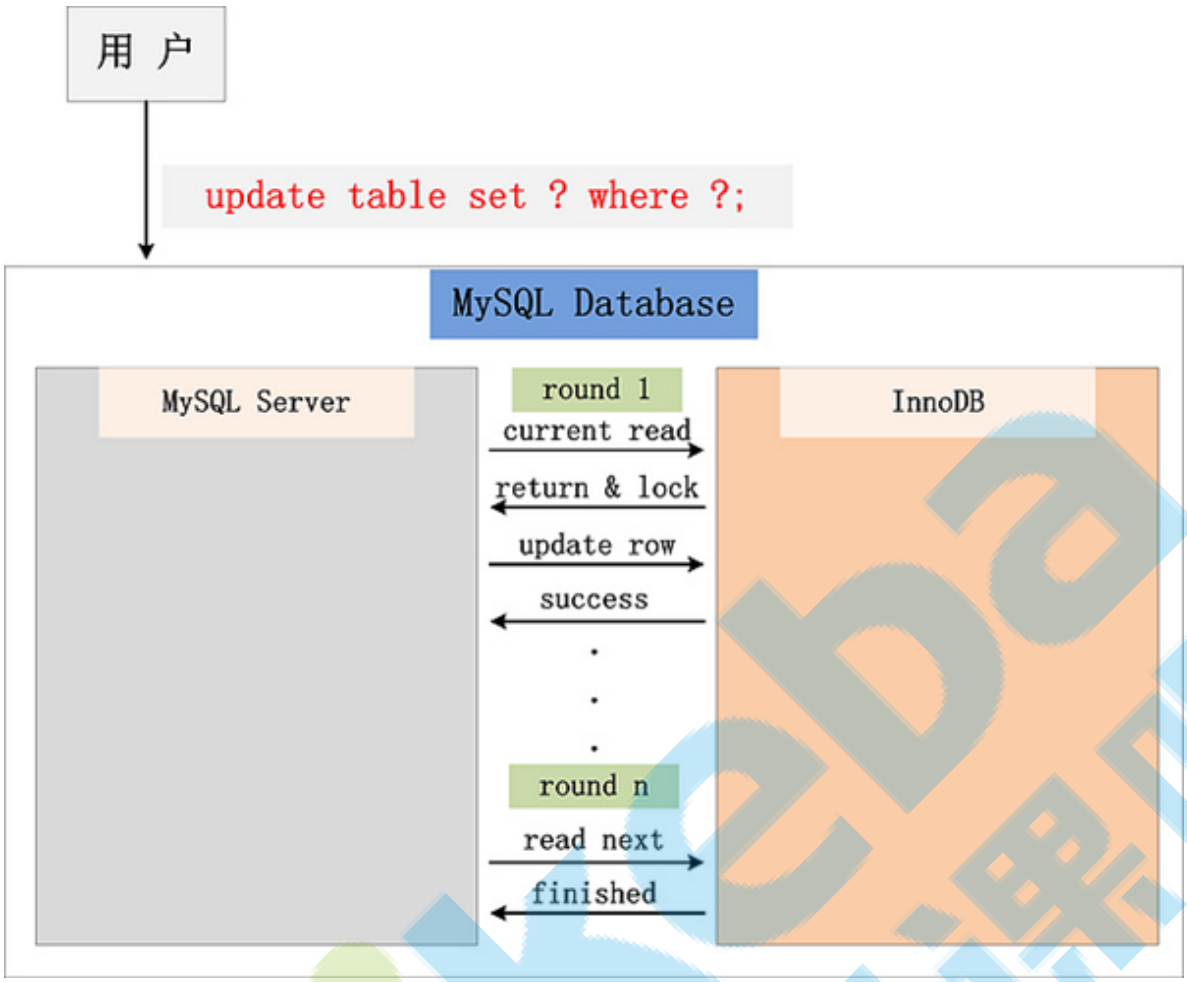
```
- 快照读，读取的是记录的可见版本 (有可能是历史版本)，不用加锁。(select)
- 当前读，读取的是记录的最新版本，并且当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。
```

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

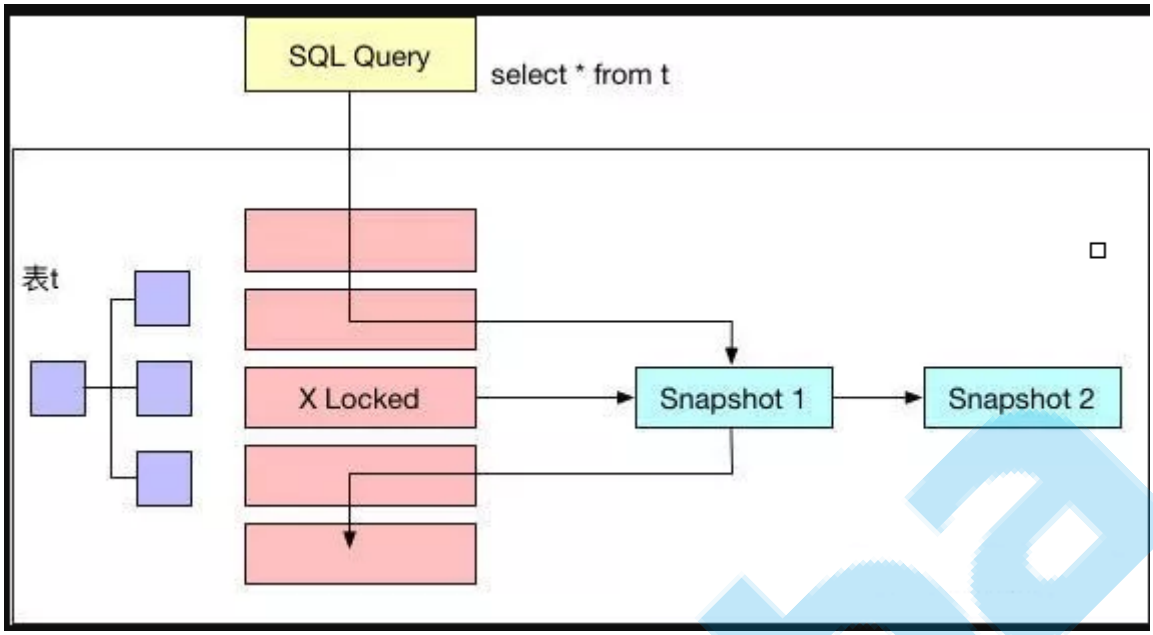
一致性非锁定读(consistent nonlocking read)是指InnoDB存储引擎通过多版本控制(MVCC)读取当前数据库中行数据的方式。

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，

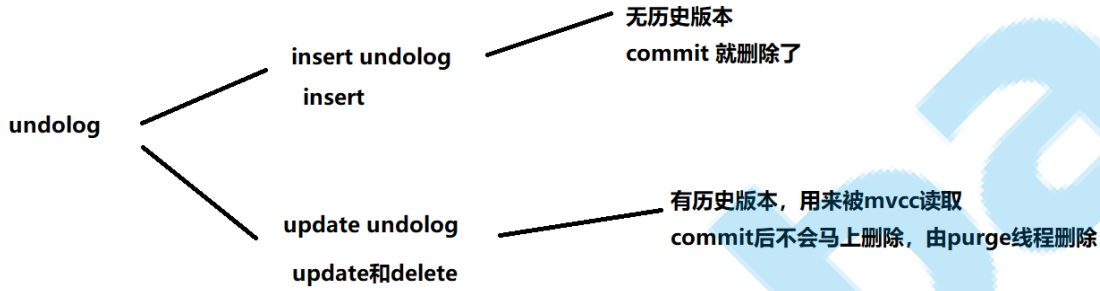
rollback 在该事务中直接删除 ， 不需要进行 purge 操作 (purge Thread)

update undo log：

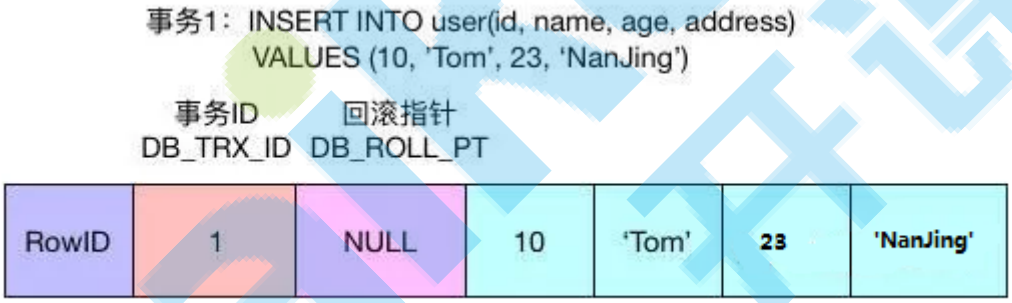
是 update 或 delete 操作中产生的 undo log，因为会对已经存在的记录产生影响，

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。

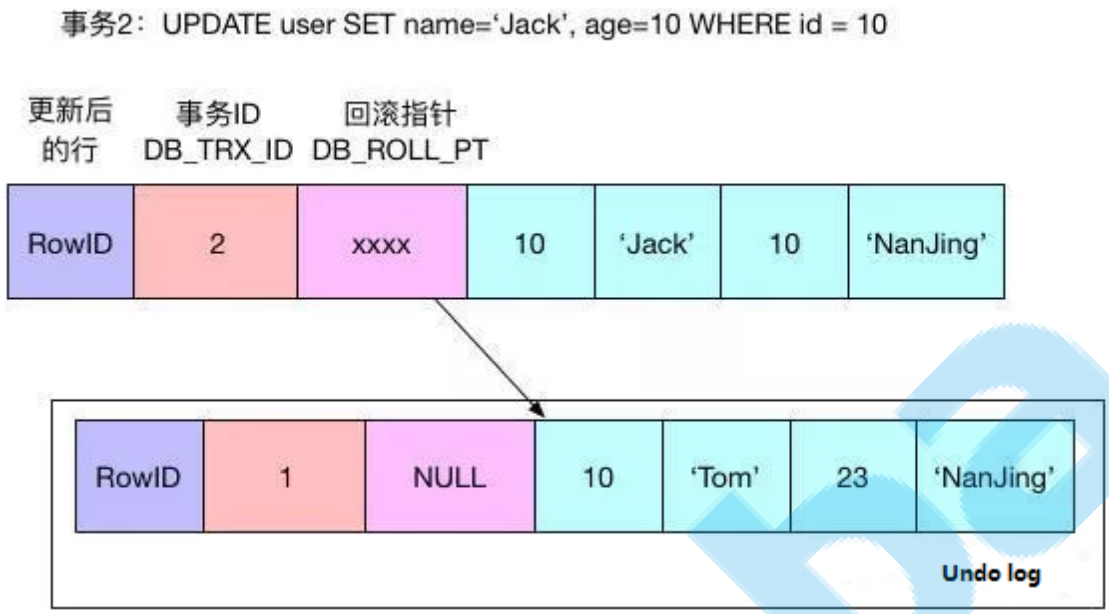



**如下图所示（初始状态）：**

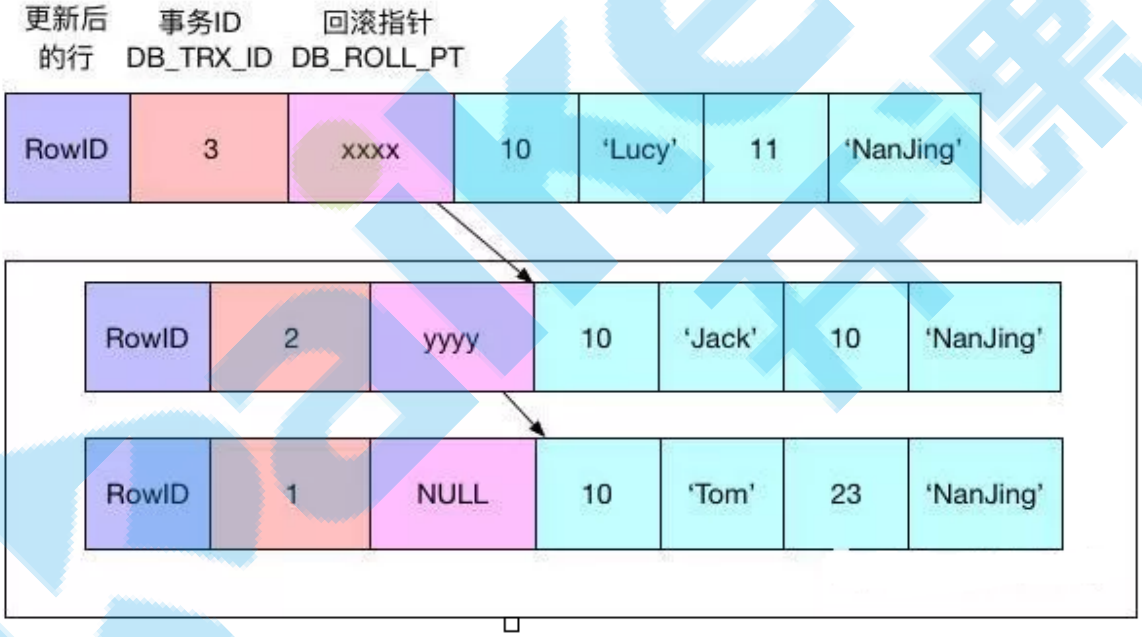



当事务2使用UPDATE语句修改该行数据时，会首先使用排他锁锁定改行，将该行当前的值复制到undo log中，然后再真正地修改当前行的值，最后填写事务ID，使用回滚指针指向undo log中修改前的行。

**如下图所示（第一次修改）：**




**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];
```

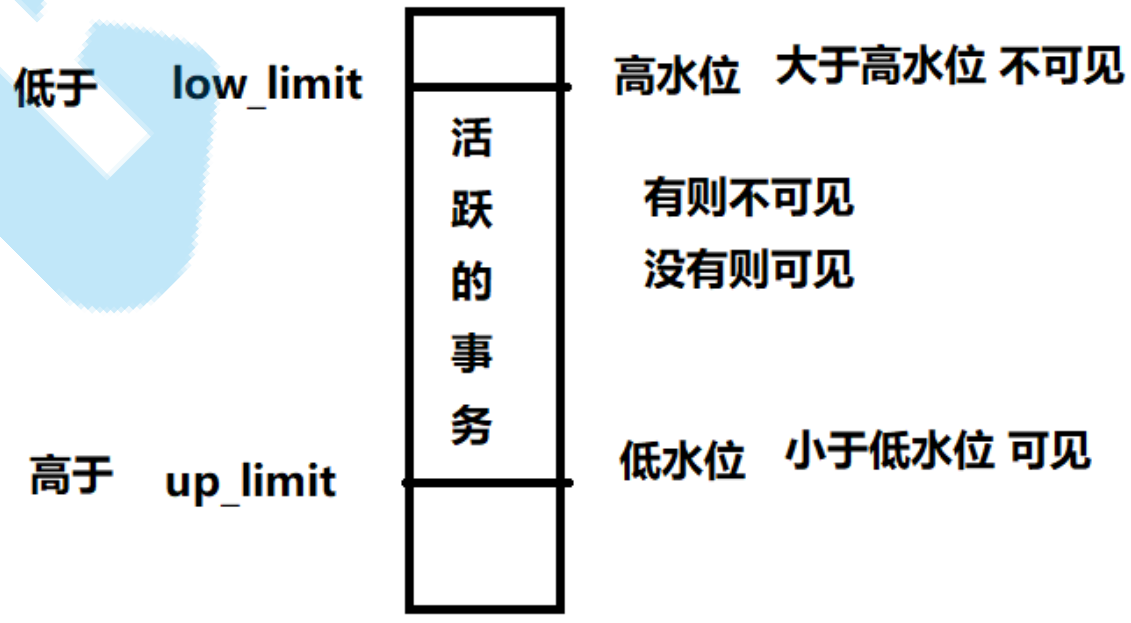
low_limit_id是"高水位"，即当时活跃事务的最大id，如果读到row的db_trx_id>=low_limit_id，说明这些id在此之前的数据都没有提交，如注释中的描述，这些数据都不可见。

```
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。

# 行锁原理分析

## 谈锁前提

```
select * from t1 where id = 10;

delete from t1 where name = 'zs';
```

前提一：id列是不是主键?

**前提二**：当前系统的隔离级别是什么?

前提三：id列如果不是主键，那么id列上有索引吗?

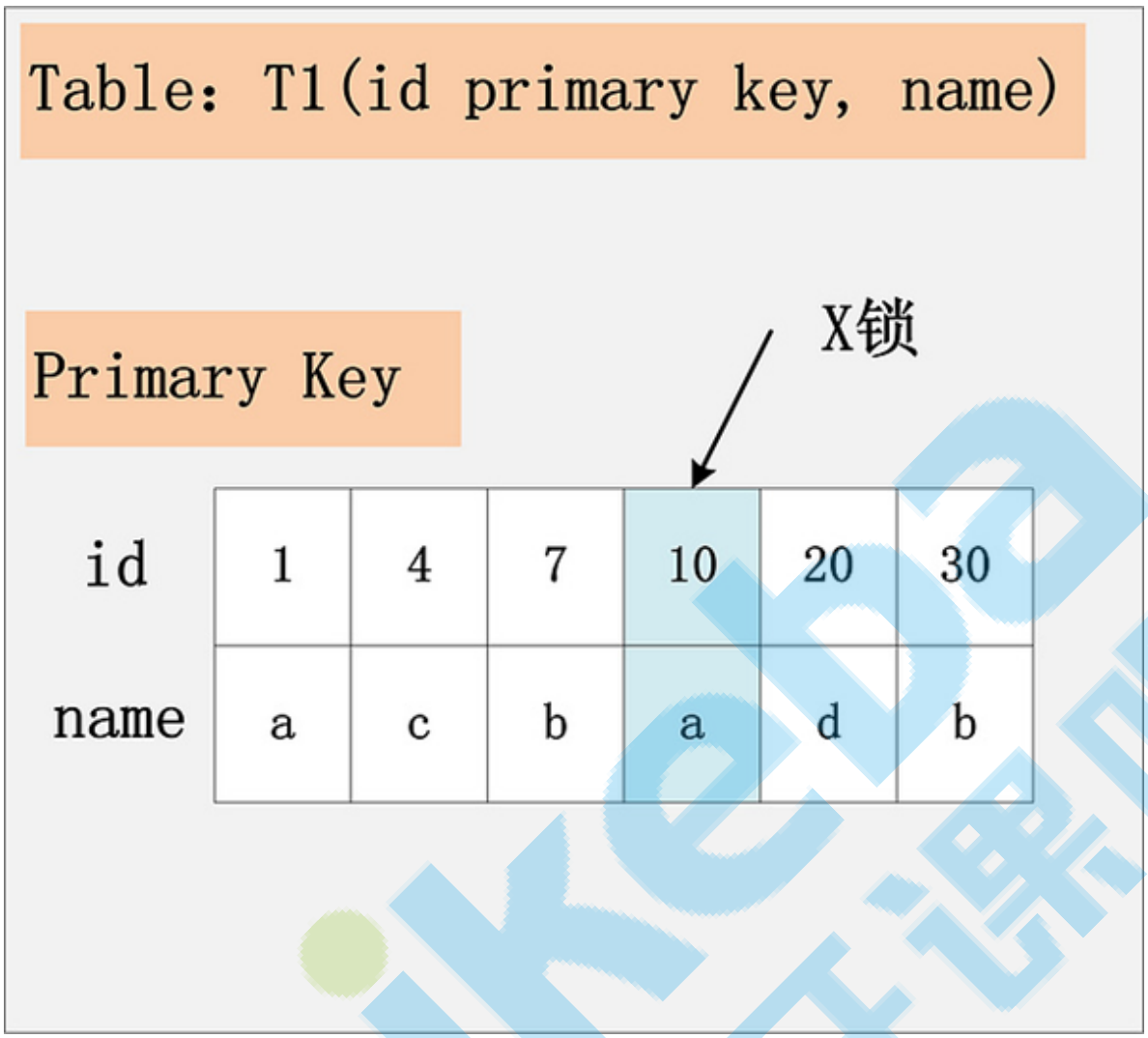
## 简单SQL的加锁分析

RC隔离级别下

### 组合一：id主键+RC

这个组合，是最简单，最容易分析的组合。id是主键，Read Committed隔离级别，给定SQL：

delete from t1 where id = 10; 只需要将主键上id = 10的记录加上X锁即可。如下图所示：




## 组合二：id唯一索引+RC

这个组合，id不是主键，而是一个Unique的二级索引键值。那么在RC隔离级别下， 需要加什么锁呢？见下图：

```
delete from t1 where id = 10;
```

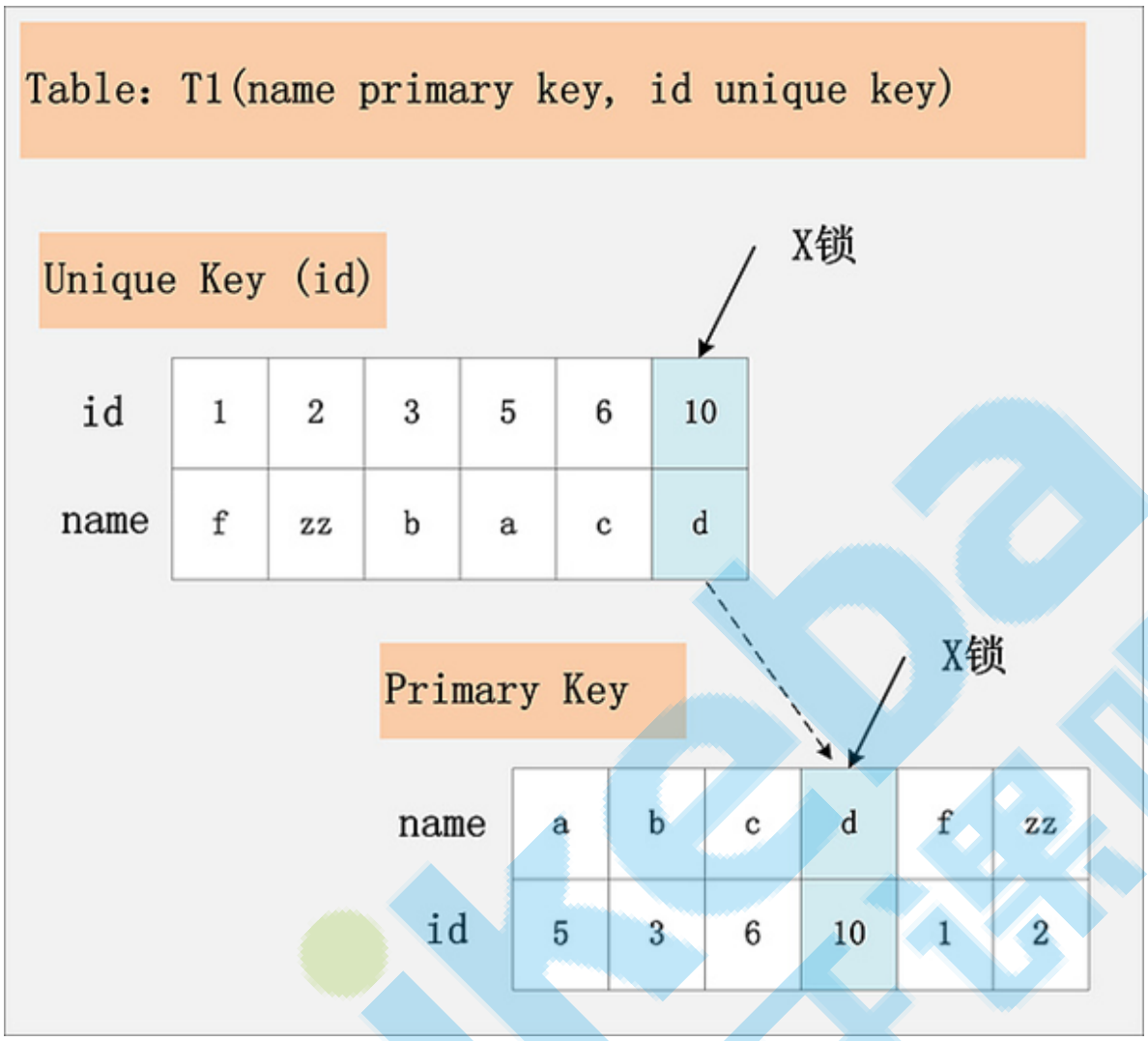
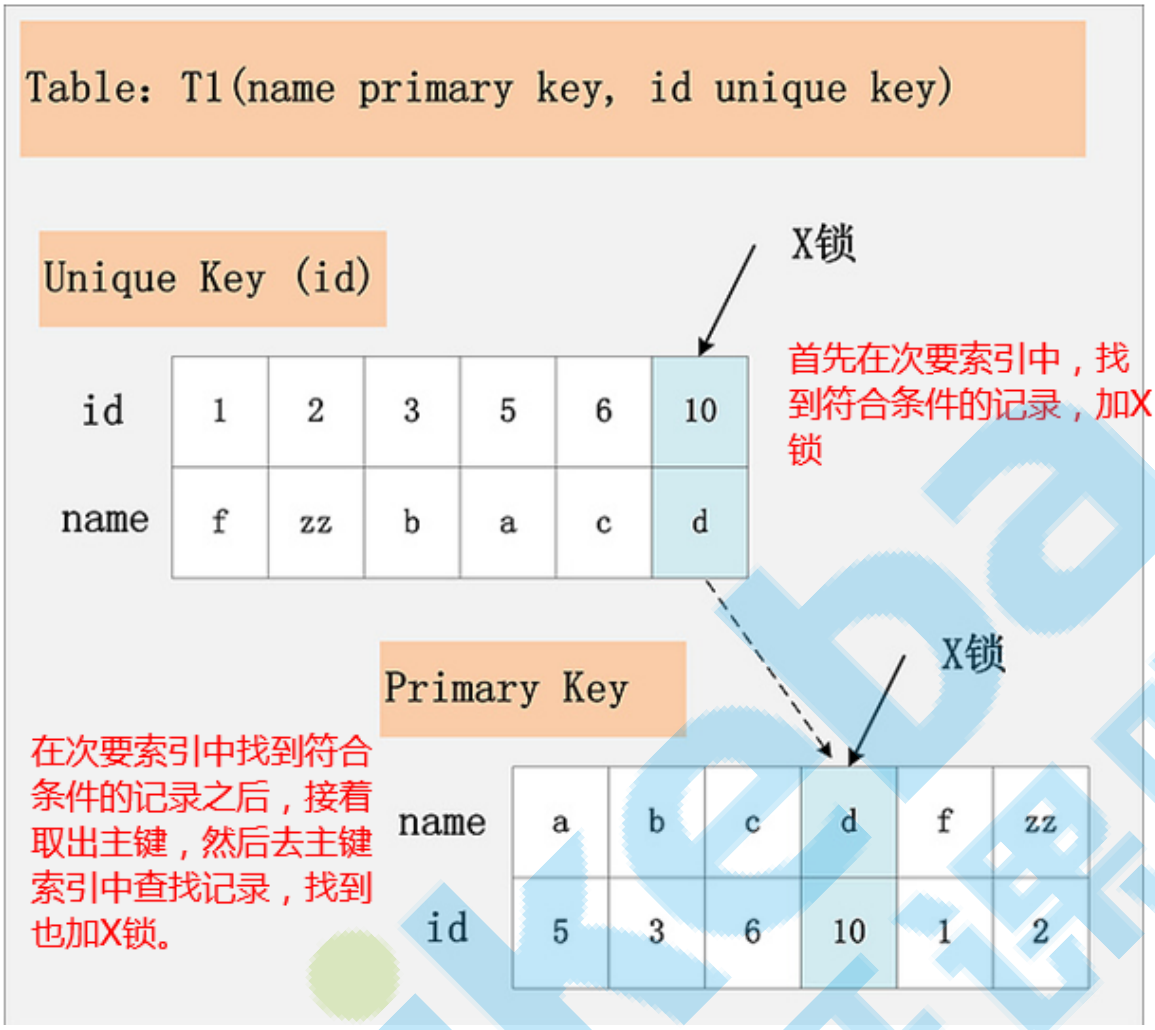
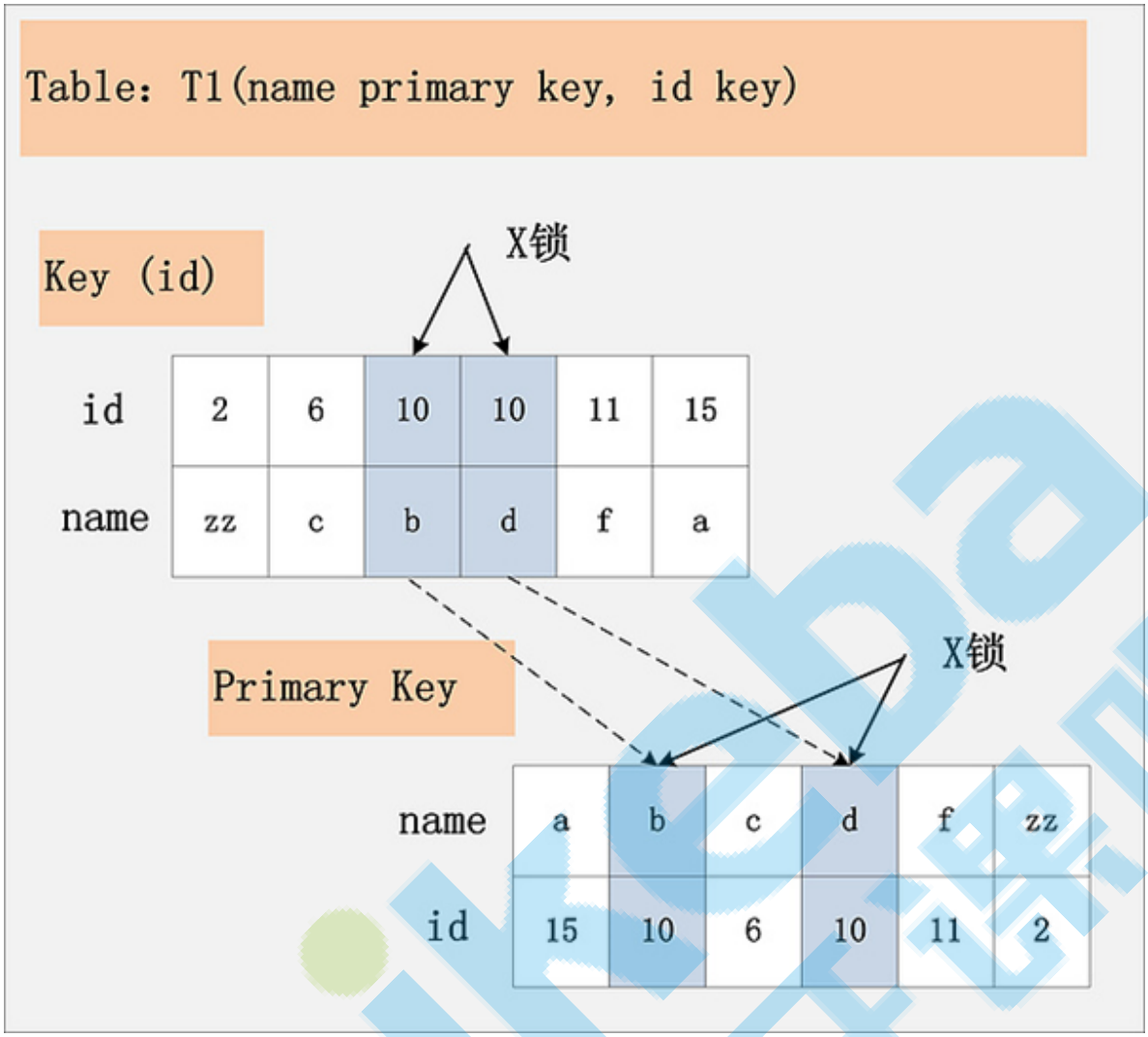


Table: T1(name primary key, id unique key)
Unique Key (id)
X锁
id
1
2
3
5
6
10
name
f
zz
b
a
c
d
Primary Key
X锁
name
a
b
c
d
f
zz
id
5
3
6
10
1
2



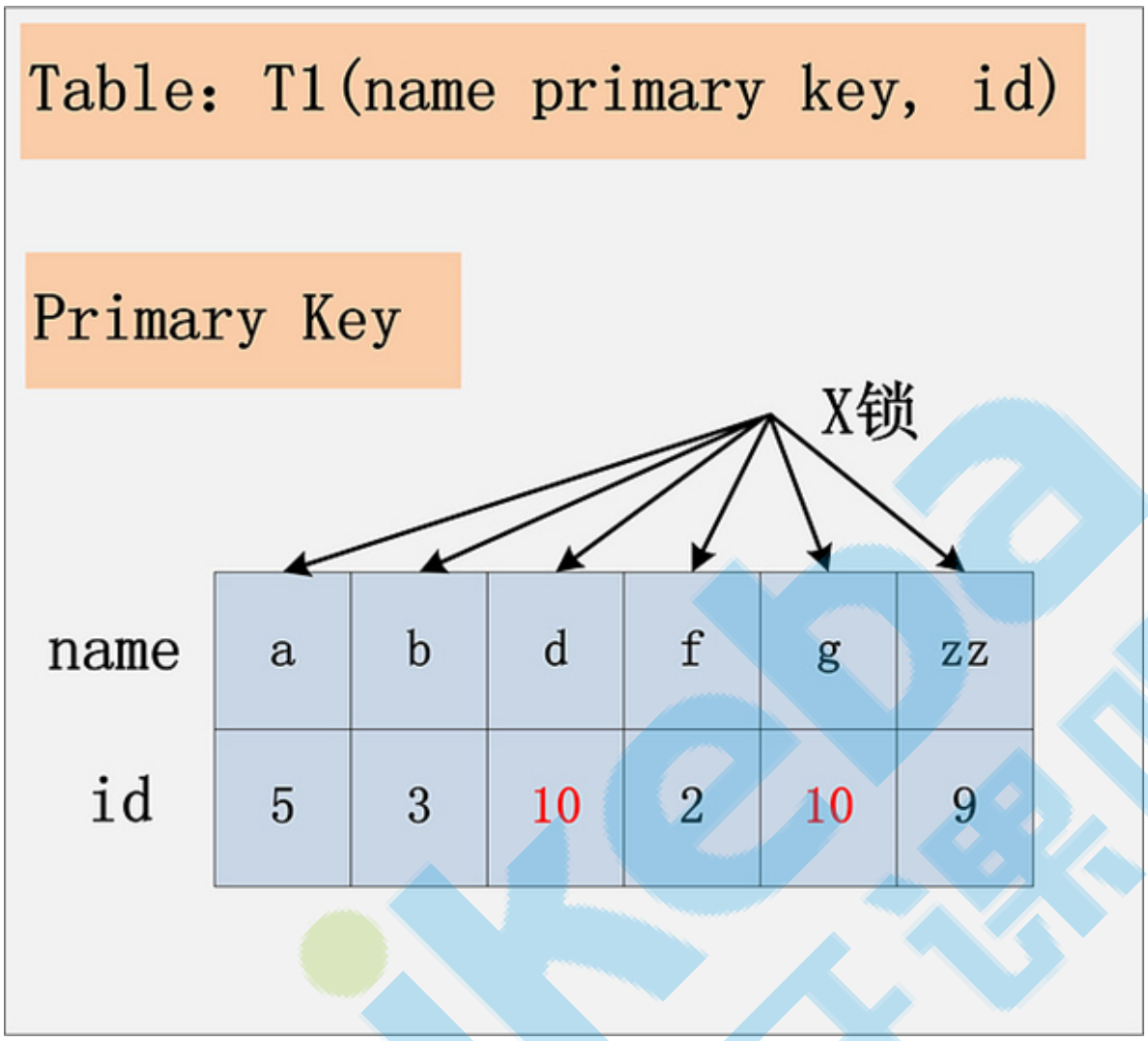

**组合三：id非唯一索引+RC**




相对于组合一、二，组合三又发生了变化，隔离级别仍旧是RC不变，但是id列上的约束又降低了，id列不再唯一，只有一个普通的索引。假设以下语句，仍旧选择id列上的索引进行过滤where条件，那么此时会持有哪些锁？同样见下图：

## 组合四：id无索引+RC

id列上没有索引，where id = 10;这个过滤条件，没法通过索引进行过滤，那么**只能走全表扫描做过滤**。对应于这个组合，SQL会加什么锁？或者是换句话说，全表扫描时，会加什么锁？这个答案也有很多：有人说会在表上加X锁；有人说会将聚簇索引上，选择出来的id = 10;的记录加上X锁。那么实际情况呢？请看下图：




### 组合五：id主键+RR

同组合一

主键等值

主键范围 产生 Gap

### 组合六：id唯一索引+RR

同组合二

### 组合七：id非唯一索引+RR

delete from t1 where id = 10;

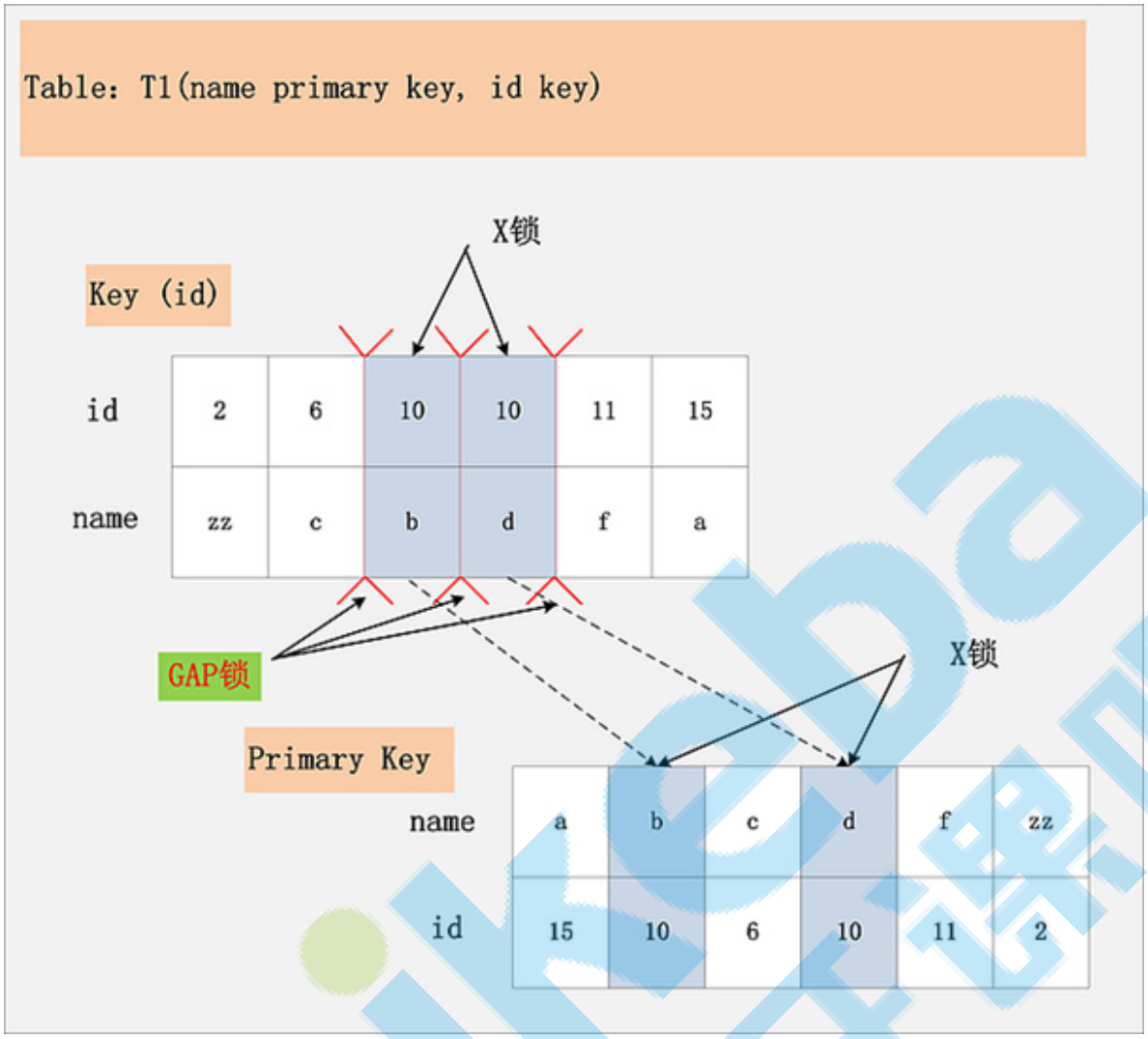
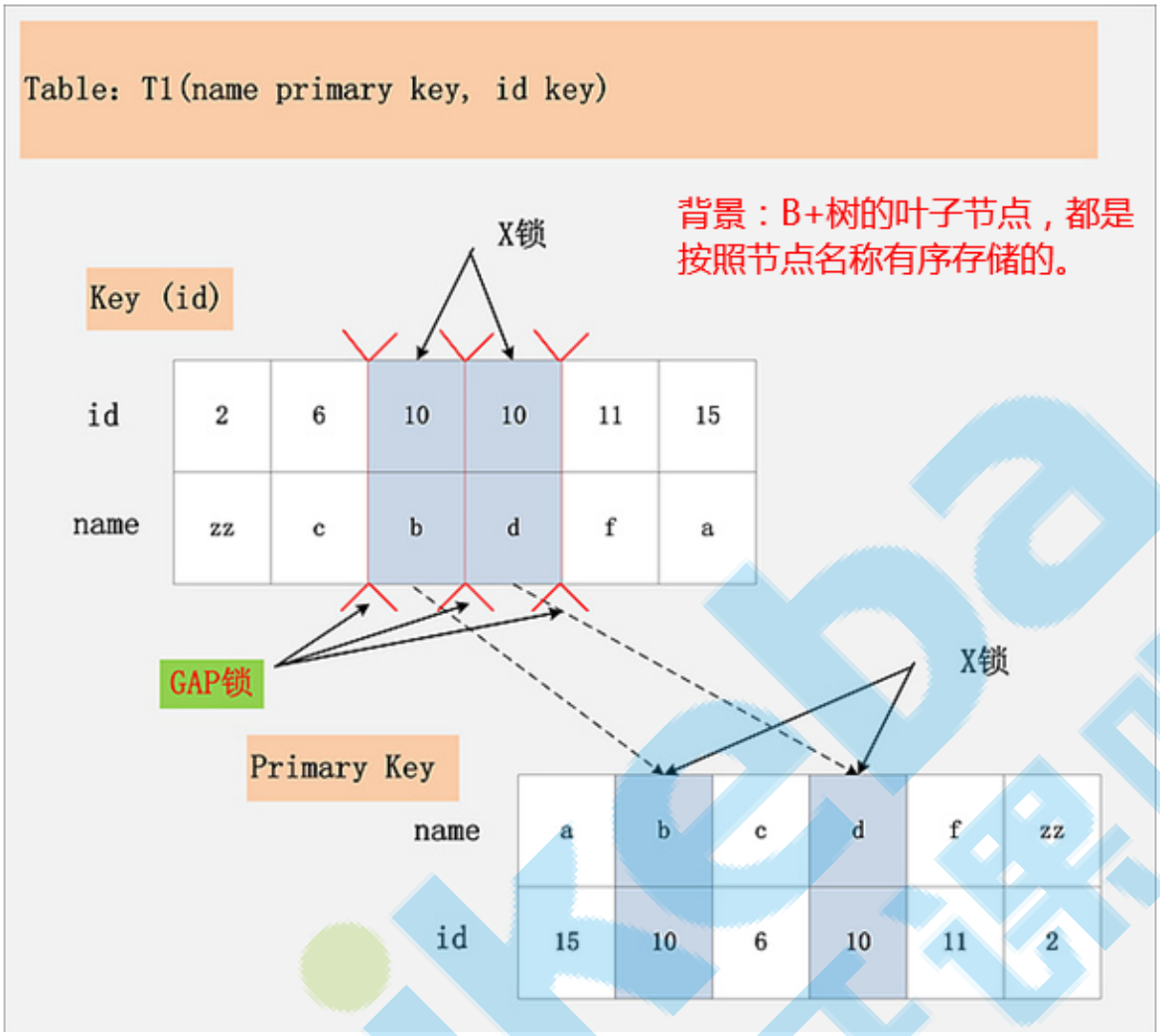
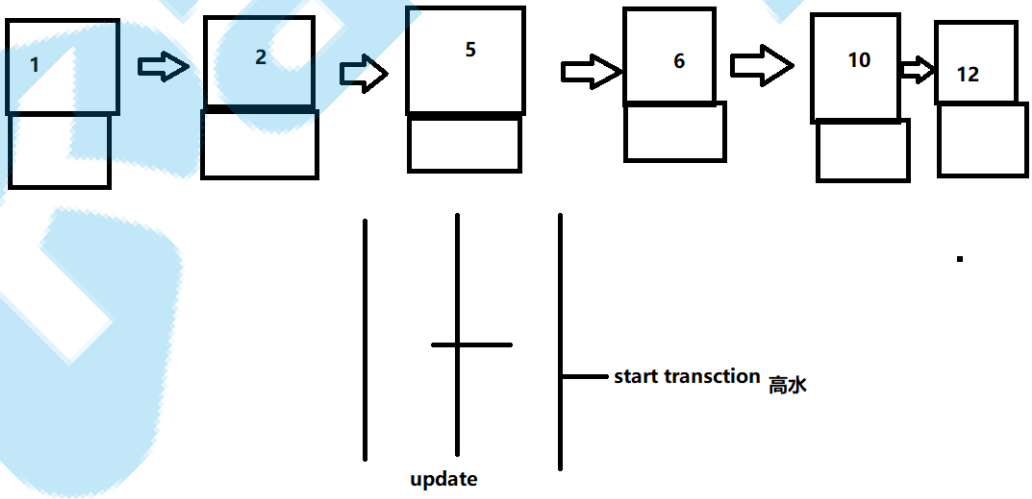


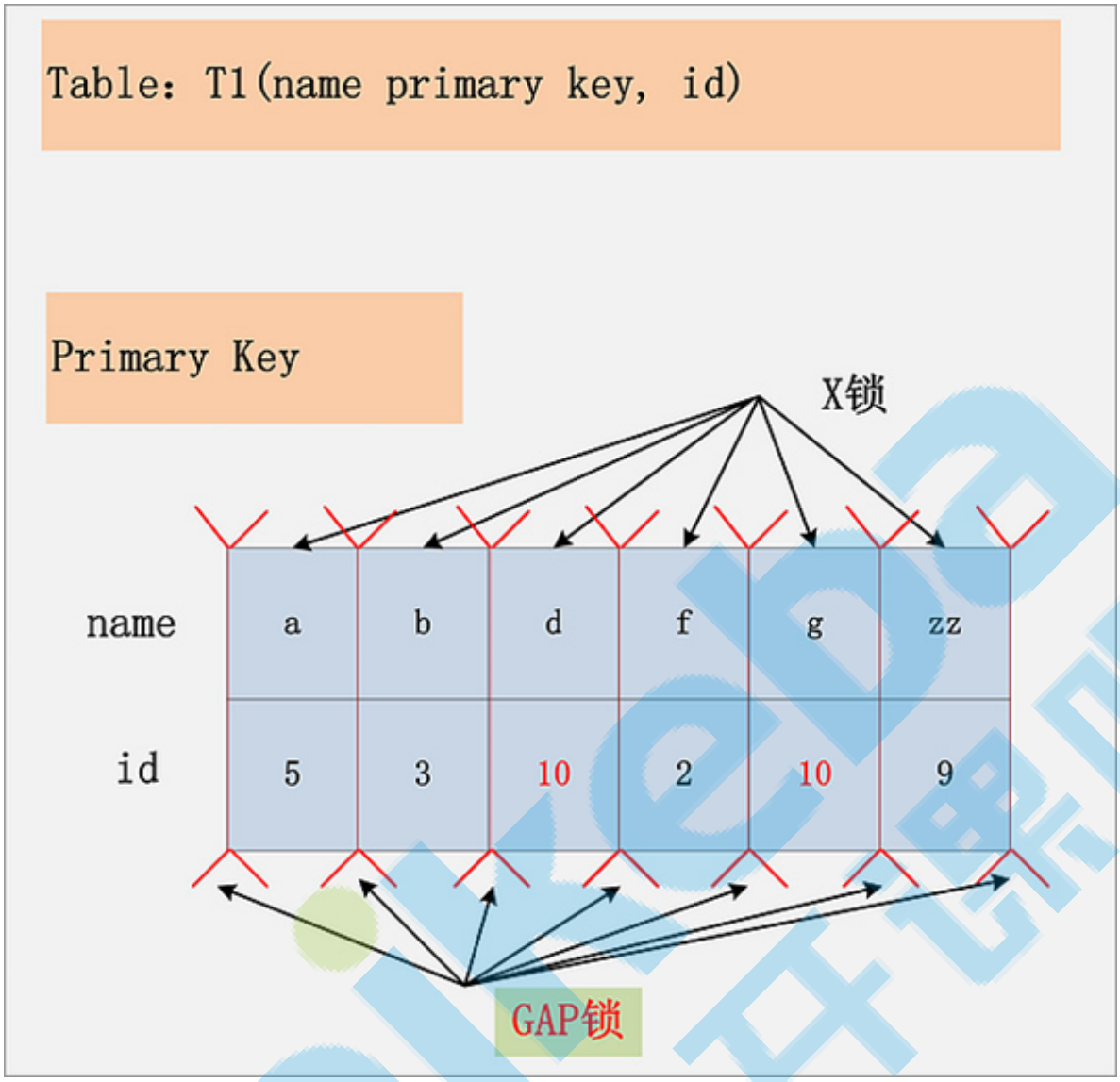
Table: T1(name primary key, id key)
X锁
Key (id)
id
2
6
10
10
11
15
name
zz
c
b
d
f
a
GAP锁
X锁
Primary Key
name
a
b
c
d
f
zz
id
15
10
6
10
11
2




不能在间隙 insert 防止幻读 （RR）




## 组合八：id无索引+RR




## 组合九：Serializable （LBCC）

只要有SQL 就锁 而且 无索引 表锁

select 是加锁的