上课时间延后21：15分钟，有部分人进不来课堂

InnoDB -> 事务

事务ACID特性

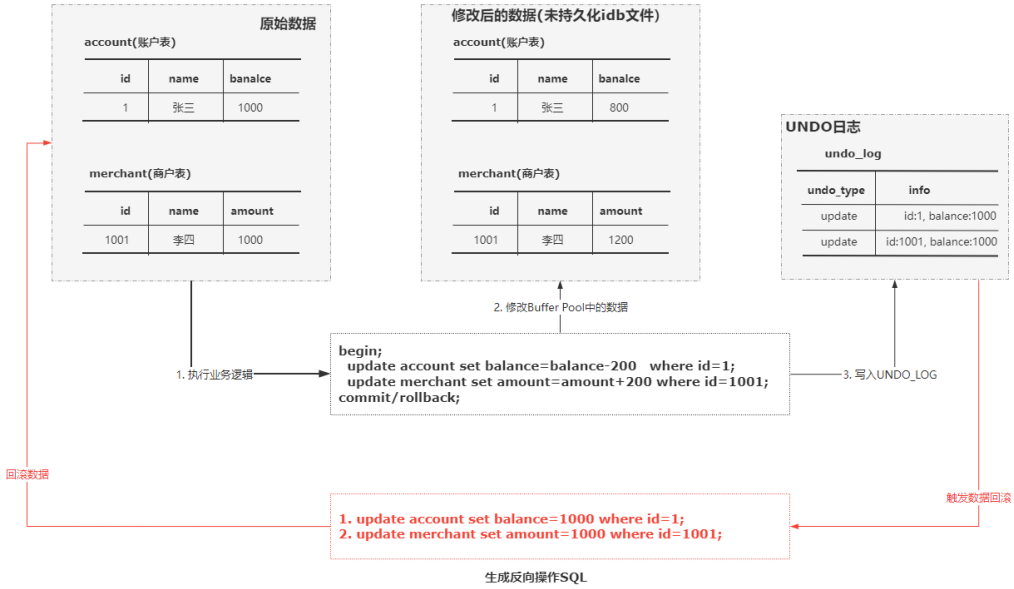
1. A：原子性
2. I： 隔离性？？
3. D：持久性
4. C: 一致性， 逻辑一致性

如何保证原子性

> 什么是原子性

```
begin;
    update account set balance=balance-100 where userid=1;
    update merchant set amount=amount+100 where mid=10001;
commit;
```

> UNDO_LOG保证原子性



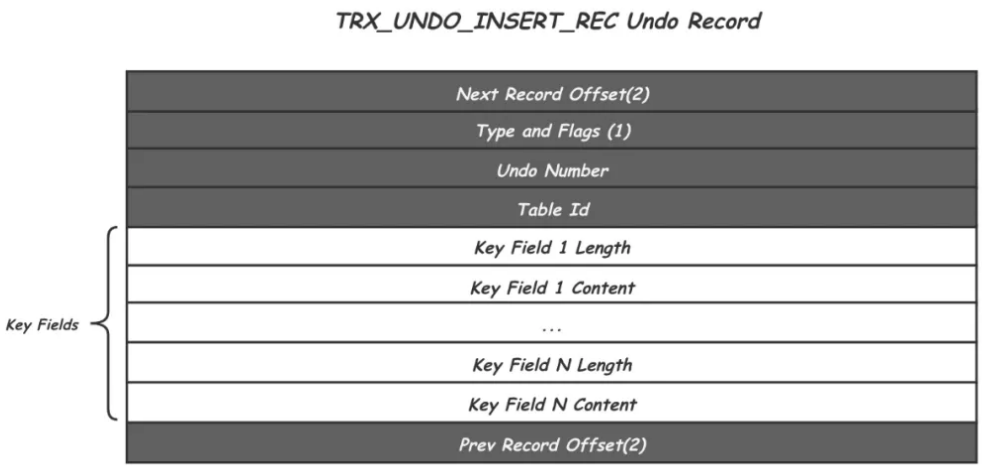
UNDO_LOG的存储结构

两种日志类型：

- insert undo log insert操作产生，事务提交之后就删除
- update undo log ， update、delete产生的，事务提交之后不一定会删除。MVCC（）

Insert UNDO LOG

之所以要分开，本质上是因为Insert操作只是为事务回滚做准备，不需要在MVCC功能中承担作用，因此只需要记录对应记录的key即可。



其中：

- Type And Flags， 对应的日志类型，TRX_UNDO_INSERT_REC ,表示insert操作的undo日志类型。
- Undo Number是Undo的一个递增编号。
- Table ID用来表示是哪张表的修改。
- 下面一组Key Fields的长度不定，因为对应表的主键可能由多个field组成，这里需要记录Record完整的主键信息，回滚的时候可以通过这个信息在索引中定位到对应的Record。
- 除此之外，在Undo Record的头尾还各留了两个字节用户记录其前序和后继Undo Record的位置。

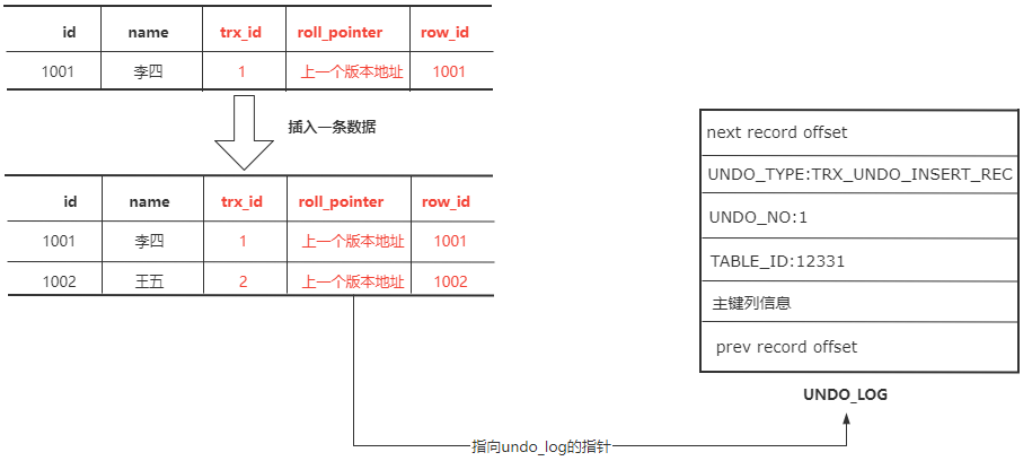
Insert 操作对应的UNDO 日志类型是TRX_UNDO_INSERT_REC，会存储上述描述的信息包括主键各列（有可能存在多个主键-联合主键）Key Fields，其中length表示主键列占用的存储空间，比如int类型占用4个字节。Content表示主键列的值。

假设我们有下面这样一个表，其中trx_id、roll_pointer、row_id是隐藏列，表中的每条记录都会存在这样三个隐藏列。

id	name	trx_id	roll_pointer	row_id
1001	李四	1	上一个版本地址	1001

当我们向这个表中插入一条记录的时候，它会向聚簇索引中插入记录，同时向二级索引中也需要插入记录（假如存在二级索引）。

同时，向UndoLog中插入一条TRX_UNDO_INSERT_REC类型的UNDOLOG，同时插入的这条记录的roll_poriter指针指向这个UNDO LOG，形成如下图所示的结构。



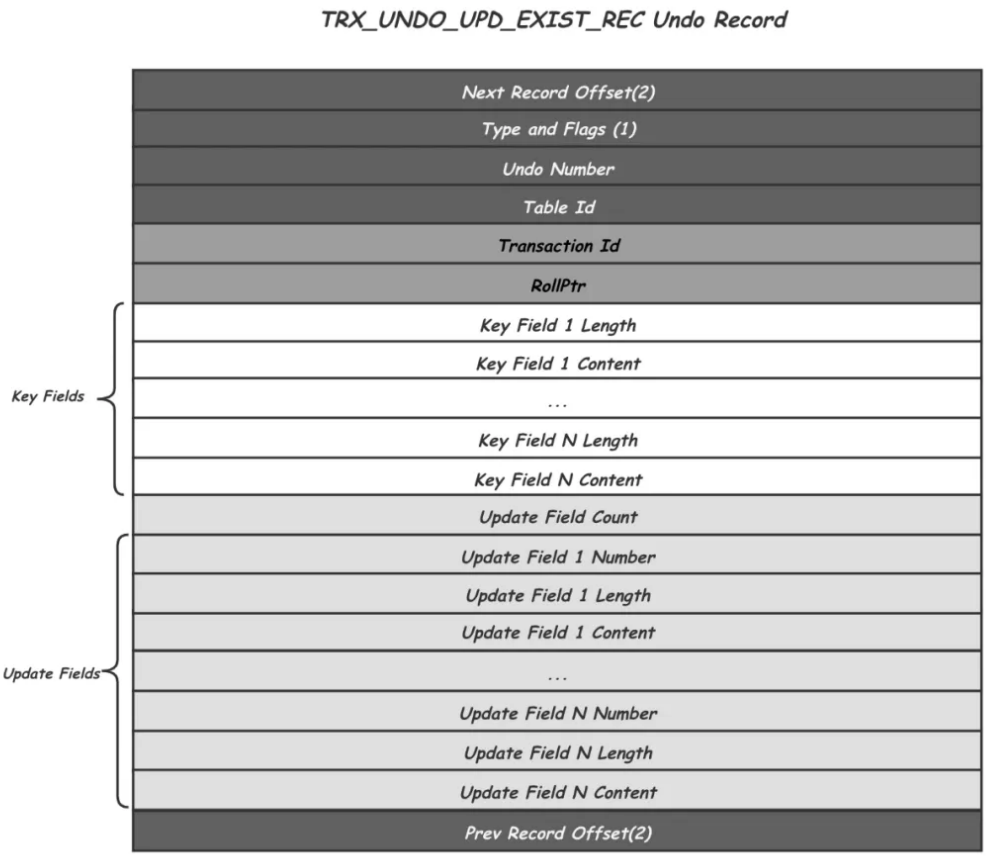
Update Undo Log

Update Undo Log中记录delete和update操作产生的 undo log。

Update Undo Log涉及到MVCC的功能，所以Update操作需要保留Record的多个历史版本，当某个Record的历史版本还在被使用的时候，这个Record不能从UndoLog中真正删除，因此需要删除的时候，只是修改对应Record的Delete Mark标记。

如果这个Record又被重新插入到UndoLog中，那么这个时候只需要修改Delete Mark标记即可，也就是把往Update Undo Log表中执行的delete和insert转变为对Delete Mark标记的Update操作。

Insert Update Log存储结构如下。



基本上和Insert Undo Log类似，都有相同的首尾信息，除此之外，Update Undo Record增加了：

- Transaction Id，记录了产生这个历史版本事务Id，用作后续MVCC中的版本可见性判断
- Rollptr，指向的是该记录的上一个版本的位置，沿着Rollptr可以找到一个Record的所有历史版本。

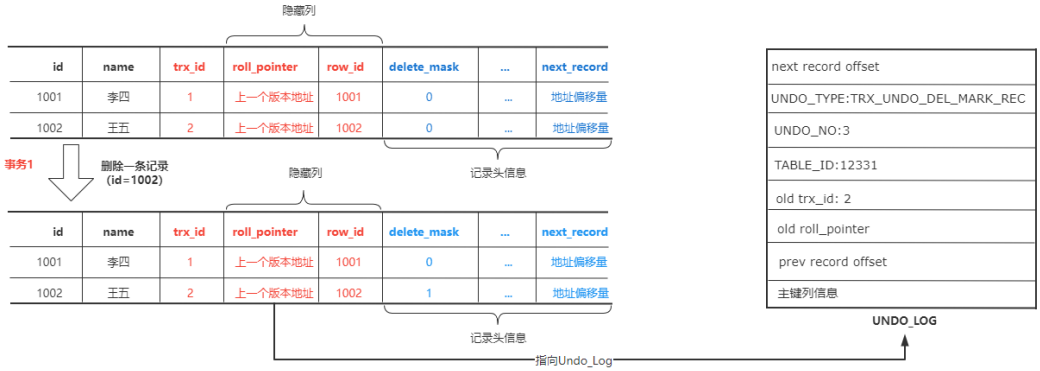
- Update Fields中记录的就是当前这个Record版本相对于其之后的一次修改的Delta信息，包括所有被修改的Field的编号，长度和历史值。

当一个事务要更新一行记录时，会把当前记录当做历史快照保存下来，多个历史快照会用两个隐藏字段trx_id和roll_pointer串起来，形成一个历史版本链，当需要事务回滚时，可以依赖这个历史版本链将记录回滚到事务开始之前的状态，从而保证了事务的原子性（一个事务对数据库的所有操作，要么全部成功，要么全部失败）。

在Update Undo Log中， UNDO LOG类型有三种，下面分别来说明这三种类型的存储方法。

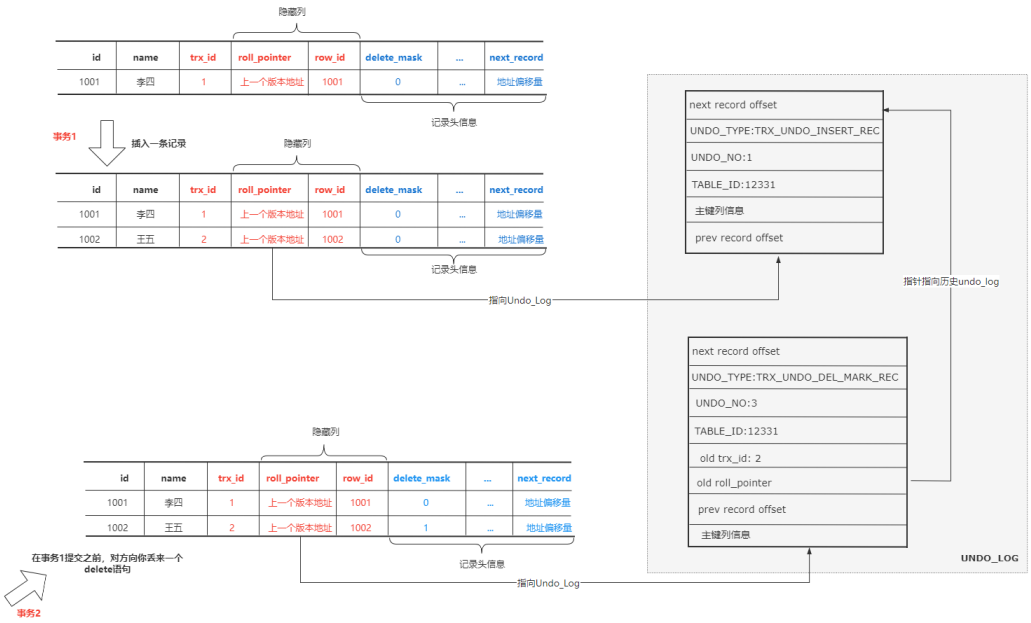
Delete操作对应的UNDO LOG

前面说过，由于MVCC需要保留Record的多个历史版本，所以当某个Record历史版本还在被使用时，那么这个Record不能真正被删除，所以在需要做删除操作时，只需要修改对应Record的Delete Mark标记即可，实现原理如下。



想象一下这样一种情况：当我们插入一条记录后，该记录的 roll_pointer 指向 TRX_UNDO_INSERT_REC 类型的 undo log（这里我们称之为 insert undo log），这时我们用 delete 删除该记录（事务未提交，执行阶段一），生成一条 TRX_UNDO_DEL_MARK_REC 类型的 undo log（这里我们称之为 delete undo log）。

此时该记录仍存在，且 **roll_pointer 指向 delete undo log，delete undo log 的 old roll_pointer 指向 insert undo log**，形成如下图所示的存储结构。



Update 操作对应的UNDO LOG

对于Update操作，分为两种情况

1. 更新主键的情况

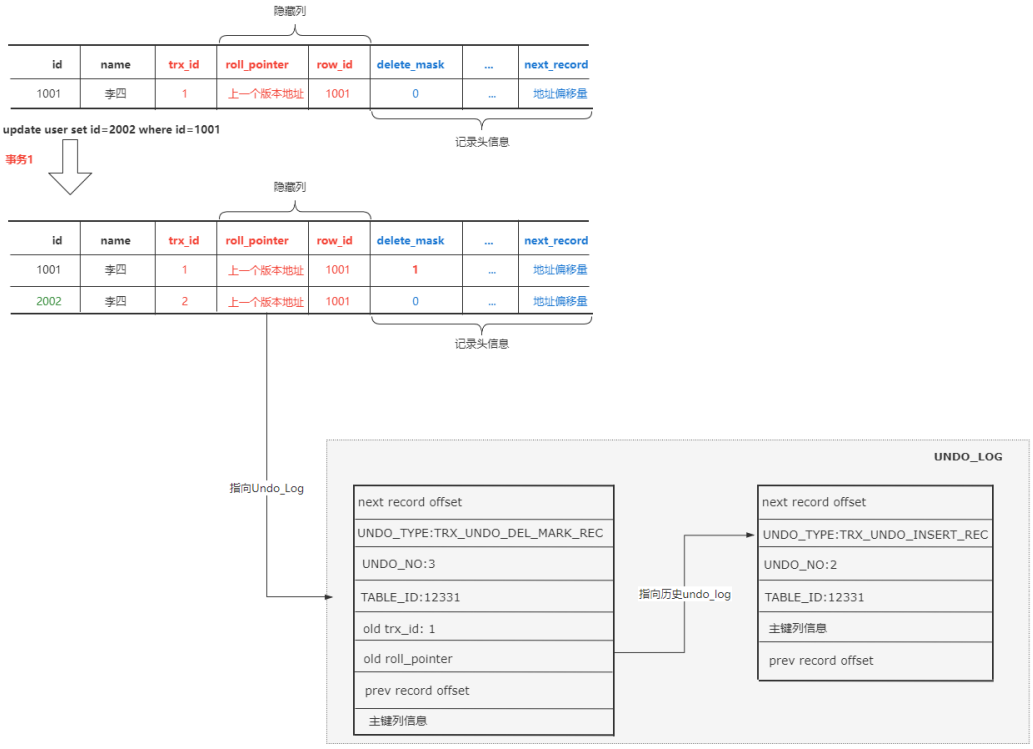
   如果涉及到主键的更新，那么数据的更新操作需要按照两个步骤来完成

   - 在Update语句所在的事务提交之前，不会直接把旧的记录删除，因为有可能其他的事务要访问到这个记录（MVCC相关内容），所以直接对旧记录的delete_mask进行更新
   - 根据更新后的各项值，创建一条新记录重新定位并插入到聚集索引中。

   而针对Update语句更新主键的情况，Undo日志的执行逻辑是：

   - 对目标记录进行delete_mask标记之前，会记录一条UNDO_TYPE为TRX_UNDO_DEL_MARK_REC的UNDO日志
   - 之后再插入新的记录时，会记录一条类型为TRX_UNDO_INSERT_REC的undo日志

   也就是说，每对一条记录的主键值做改动时，会记录两条UNDO日志。



2. 不更新主键，不更新主键的情况下，又可以分为更新的列占用的存储空间是否发生变化两种情况。
    1. 不发生变化， 表示更新的每个列，更新前后占用的存储空间不变，那么直接可以在原来的记录上修改。
    2. 发生变化， 如果被更新的字段，任意一列更新前后占用的存储空间发生了变化，那么就不能直接在原来的记录上直接修改，而是需要先在聚集索引中删除这条记录，然后再根据更新后的值创建一条新记录。

        > 注意，这里的删除，就不是像前面那样直接修改delete_mask，而是由用户线程同步执行真正的删除，因为这里有锁的保护，并不存在并发问题。

    在不更新主键的这种情况且更新后的字段占用空间不变的情况下，向Undo Log中添加的日志类型是TRX_UNDO_UPD_EXIST_REC，相比于其他类型，它会多出一些属性。
    - n_updated， 有多少个列被更新
    - 被更新的列更新前的信息
    - 如果更新的列包含索引列，则需要添加索引列的信息

如何保证持久性

持久性-REDO _LOG(参考Mysql的第一节课)

如何保证隔离性

> 当多个事务竞争的时候，会存在哪些问题呢?

- 脏读，在第二个事务还未提交之前，第一个事务执行的两次查询操作得到的数据结果不一致，这种就是事务并发情况下产生的读未提交导致脏读问题。
- 幻读，一个事务前后两次读取数据数据不一致，是由于其他事务插入数据造成的，这种情况我们把它叫做幻读。
- 不可重复读，一个事务读取到了其他事务已提交的数据导致前后两次读取数据不一致的情况，我们把它叫做读已提交下的不可重复读。

Mysql中的事务隔离级别

Mysql通过不同的事务隔离级别来解决多事务并发导致的问题。

事务隔离级别	脏读	不可重复读	幻读
未提交读（Read Uncommitted）	可能	可能	可能
已提交读（Read Committed）	不可能	可能	可能
可重复读（Repeatable Read）	不可能	不可能	对InnoDB不可能
串行化（Serializable）	不可能	不可能	不可能

Mysql中实现不同事务隔离级别

- MVCC，（Multi Version Concurrency Control）多版本并发控制。
- LBCC

MVCC实现

1. 一个事务只能看到第一次查询之前已经提交的事务修改以及本事务的修改
2. 一个事务不能看到当前事务第一次查询之后创建的事务，以及未提交的事务。

依赖于UNDO 版本链来完成

ReadView

用来实现一个事务对另外一个事务的可见性。

每次开启一个事务的时候，都会创建一个ReadView

ReadView的判断规则： （事务隔离性的规则）

> trx_id 表示undo中的事务id

- 如果 trx_id 等于 creator_trx_id，说明当前事务在访问它自己修改过的记录（本事务修改），所以这个版本可以被当前事务访问。
- 如果 trx_id 小于 min_trx_id，说明在Undo版本链中的这个事务在当前事务生成 ReadView 前已经提交，所以这个版本可以被当前事务访问。【当前事务在执行的时候， 这个快照已经生成了.】
- 如果 trx_id 大于或等于max_trx_id，说明在Undo版本链中的这个事务在当前事务生成 ReadView后才开启，所以这个版本不可以被当前事务访问。
- 如果 trx_id 在 min_trx_id 和 max_trx_id 之间，此时再判断一下 trx_id 是不是在 m_ids 列表中。
    - 如果在，说明创建ReadView时生成该版本的事务还是活跃的，该版本不可以被访问；
    - 如果不在，说明创建ReadView时生成该版本的事务已经被提交，该版本可以被访问。

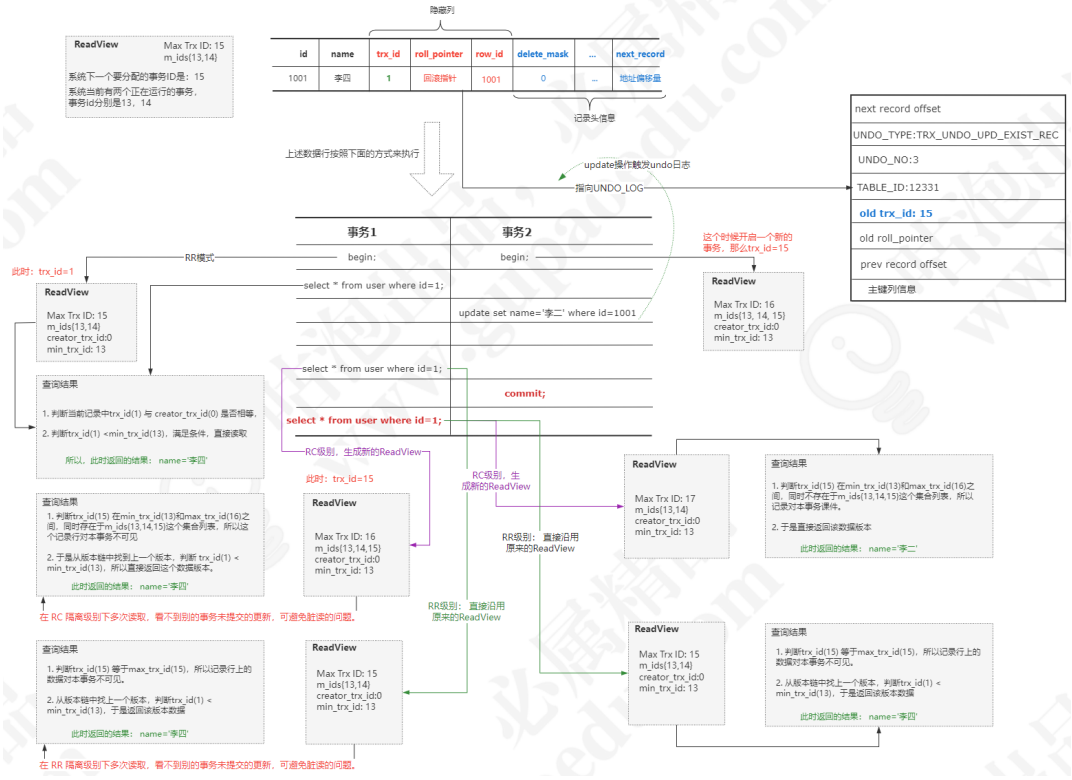
如何解决RC和RR的问题

取决于生成ReadView的时机

- RC ，每次查询前都会生成一个独立的ReadView

- RR，只是在第一次查询前生成一个ReadView，之后的查询都重复使用这个ReadView

整体实现原理如下。



LBCC

除了MVCC，Mysql还提供了LBCC（Lock Based Concurrency Control）的机制来实现事务的隔离特性。

基于锁的方式起始比较简单，就是一个事务在进行数据查询时，不允许其他事务修改。也就是说，基于锁的机制就使得数据库无法支持并发事务的读写操作，这种方案在一定程度上影响了操作数据的效率。

锁的粒度

之前我们在讲解Mysql的存储引擎时有说到，InnoDB和MyISAM支持锁的类型是不同的。

- MyISAM只支持表锁，用lock table的语法加锁

```
lock tables xxx read;
lock tables xxx write;
unlock tables ;
```

- InnoDB可以同时支持表锁和行锁，我们知道锁的粒度越小，并发性能就越高，所以InnoDB在锁的实现方面有它的优势。

锁的类型

在Mysql中，锁分为8类，共享锁/排他锁 、 意向锁、 记录锁、 间隙锁、 插入意向锁、自增锁、空间索引谓词锁、next-key lock。

> 官网的锁类型说明地址： https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html

这些锁里面，可能大家就认识共享锁/排他锁，其他都没听过，而且如果我们就按照这个锁去理解可能不是很清晰，所以我们可以把这些锁按照基本模式归类。

1. 锁的基本模式： （Shared And Exclusive Locks）行级别锁 和 （Intention Locks）表级别锁
2. 后面的三个：Record Locks、Gap Locks、Next-Key Locs 我们称为锁的算法，也就是说在什么情况下锁定什么范围。
3. 插入意向锁（Insert Intention Locks）：是一个特殊的间隙锁。
4. 自增锁(AUTO-INC Locks)：是一种特殊的表锁，用来防止自增字段重复，数据插入以后就会释放，不需要等到事务提交才释放。
5. Predicate Locks for Spatial Indexes是5.7版本里面新增的空间索引的谓词锁。

InnoDB中LBCC要解决的问题

问题1-幻读问题(InnoDB)

范围查询的时候，多次查询结果的数据行数一致

```
--transction1
select * from table where id >=1 and id<=4
//锁定2，3 [解决幻读问题]
```

```
insert into table(id,name) values(4,'mic');
```

问题二， for update 实现了排他锁(行锁)

```
user(id primary key) , name(normal)
--transaction1
select * from table where id=1 for update; //查询主键id=1 （行锁，只锁定行）
```

```
--transaction2

update table set name='111' where id=1; //阻塞
update table set name='222' where name =''; //阻塞
```

基于索引来决定的，如果where是索引，那么这个时候，直接加行锁.

问题三， 锁定整个表

```
select * from table for update; //表锁
```

```
update table set name='111' where id=1; //阻塞
```

锁的算法（行锁中的锁的算法）

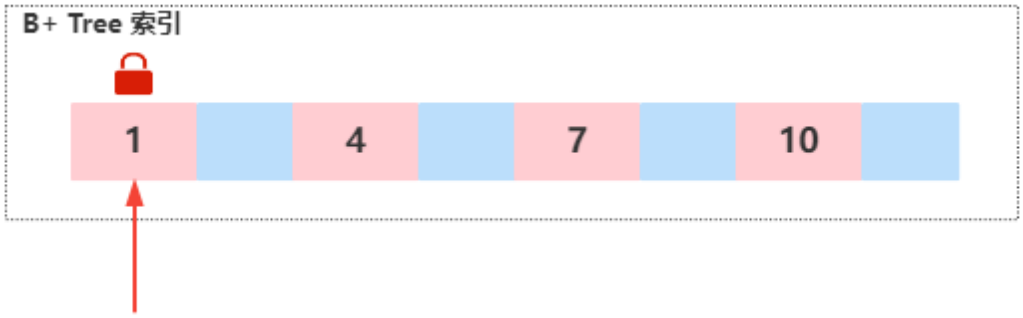
- Record Lock (行锁) [锁定的是索引]
- Gap Lock（锁定索引区间，不包括record lock）
- next Key Lock（锁定索引区间，包括record lock）

记录锁（Record Lock）

顾名思义，记录锁就是为**某行**记录加锁，它**封锁该行的索引记录，并不是真正的数据记录**：

```
-- id 列为主键列或唯一索引列
SELECT * FROM user WHERE id = 1 FOR UPDATE;
```

那么，意味着id=1的这条记录会被锁住，如下图所示。



注意，id这个字段必须是唯一索引或者主键列，并且查询语句必须是精准匹配，才会加记录锁。

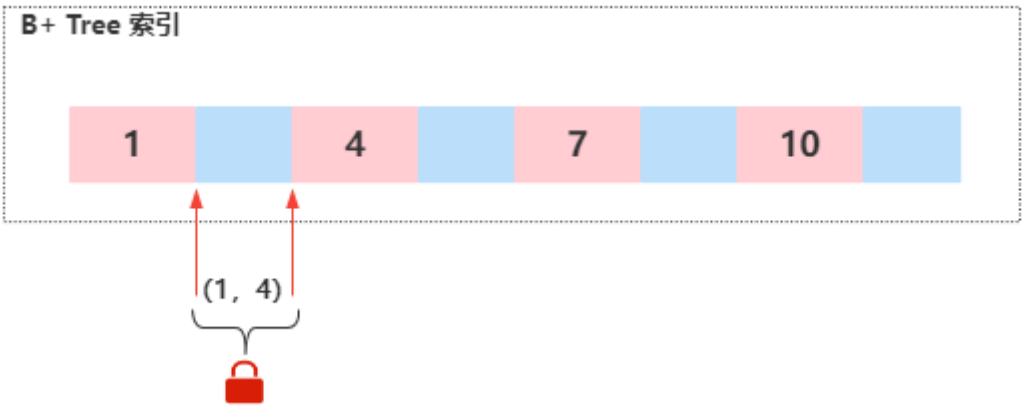
Grap Lock（间隙锁）

间隙锁顾名思义锁间隙，不锁记录。

间隙锁是基于**非唯一索引**，它锁定一段范围内的索引记录，比如下面这个查询

```
SELECT * FROM user WHERE id BETWEN 1 AND 4 FOR UPDATE;
```

那么意味着所有在(1,4)区间内的记录行都会被锁住，它是一个左右开区间的范围，意味着在这种情况下， 会锁住id为2，3的索引，但是1、4不会被锁定，如下图所示。



Next-Key Lock（临键锁）

Next-Key 可以理解为一种特殊的**间隙锁**，也可以理解为一种特殊的**算法**

每个数据行上的非唯一索引列上都会存在一把**临键锁**，当某个事务持有该数据行的**临键锁**时，会锁住一段**左开右闭区间**的数据。

为了更直观的理解临键锁，我们假设有这样一个数据表，其中id是主键，age是普通索引。

id	age	name
1	18	张三
3	21	Dylan
5	22	Mic
7	35	Willian

那么这个表中age列的潜在临键锁有（唯一索引列（包括主键列）上不存在**临键锁**）：

- （-∞，18]、（18，21]、（21，22]、（22，35]、（35，+∞]
- 为什么要锁住下一个左开右闭的区间？就是为了解决幻读的问题。

那么当事务1执行下面这个命令时：

```
-- 根据非唯一索引列 UPDATE 某条记录
UPDATE user SET name = '王五' WHERE age > 18;
-- 或根据非唯一索引列 锁住某条记录
SELECT * FROM user WHERE age > 18 FOR UPDATE;
```

不管执行上面的哪个SQL，当前事务都会锁定（18，21]这个区间。

之后如果有另外一个事务2来执行下面这个命令会被阻塞，因为插入的数据age=19，正好在锁定的区间。

```
insert into user values(10 , 19 , 'tony');
```

通过上述案例发现好像间隙锁和临键锁差不多，那什么情况下加临键锁，什么情况下加间隙锁呢?

当我们使用了范围查询，不仅仅命中了Record记录，还包含了Gap间隙，在这种情况下我们使用的就是临键锁，它是MySQL里面默认的行锁算法，相当于记录锁加上间隙锁。

Mysql中事务隔离级别总结

再回过头来看Mysql的事务隔离级别的实现。

事务隔离级别	脏读	不可重复读	幻读
未提交读（Read Uncommitted）	可能	可能	可能
已提交读（Read Committed）	不可能	可能	可能
可重复读（Repeatable Read）	不可能	不可能	对InnoDB不可能
串行化（Serializable）	不可能	不可能	不可能

Read Uncommited

RU隔离级别：不加锁。

Serializable

Serializable 所有的select语句都会被隐式的转化为select ... in share mode，会和update、delete互斥。

Repeatable Read

RR隔离级别下，普通的select使用快照读(snapshot read)，底层使用MVCC来实现。

加锁的select(select ... in share mode / select ... for update)以及更新操作update, delete等语句使用当前读（current read），底层使用记录锁、或者间隙锁、临键锁。

Read Commited

RC隔离级别下，普通的select都是快照读，使用MVCC实现。

加锁的select都使用记录锁，因为没有Gap Lock。

除了两种特殊情况——外键约束检查(foreign-key constraint checking)以及重复键检查(duplicate-key checking)时会使用间隙锁封锁区间。

所以RC会出现幻读的问题。