MySQL事务与MVCC如何实现的隔离级别

数据库事务介绍

事务的四大特性（ACID）

1. **原子性(atomicity)：** 事务的最小工作单元，要么全成功，要么全失败。
2. **一致性(consistency)：** 事务开始和结束后，数据库的完整性不会被破坏。
3. **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）。
4. **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。

事务的隔离级别

读未提交（Read UnCommitted/RU）

又称为**脏读**，一个事务可以读取到另一个事务未提交的数据。这种隔离级别岁最不安全的一种，因为未提交的事务是存在回滚的情况。

读已提交（Read Committed/RC）

又称为**不可重复读**，一个事务因为读取到另一个事务已提交的修改数据，导致在当前事务的不同时间读取同一条数据获取的结果不一致。

举个例子，在下面的例子中就会发现SessionA在一个事务期间两次查询的数据不一样。原因就是在于当前隔离级别为 RC，SessionA的事务可以读取到SessionB提交的最新数据。

发生时间	SessionA	SessionB
1	begin;	
2	select * from user where id=1;(张三)	
3		update user set name='李四' where id=1;(默认隐式提交事务)
4	select * from user where id=1;(李四)	
5		update user set name='王二' where id=1;(默认隐式提交事务)
6	select * from user where id=1;(王二)	

可重复读（Repeatable Read/RR）

又称为**幻读**，一个事物读可以读取到其他事务提交的数据，但是在RR隔离级别下，当前读取此条数据只可读取一次，在当前事务中，不论读取多少次，数据任然是第一次读取的值，不会因为在第一次读取之后，其他事务再修改提交此数据而产生改变。因此也成为幻读，因为读出来的数据并不一定就是最新的数据。

举个例子：在SessionA中第一次读取数据时，后续其他事务修改提交数据，不会再影响到SessionA读取的数据值。此为**可重复读**。

发生时间	SessionA	SessionB
1	begin;	
2	select * from user where id=1;(张三)	
3		update user set name='李四' where id=1; (默认隐式提交事务)
4	select * from user where id=1;(张三)	
5		update user set name='王二' where id=1;(默认隐式提交事务)
6	select * from user where id=1;(张三)	

串行化（Serializable）

所有的数据库的读或者写操作都为串行执行，当前隔离级别下只支持单个请求同时执行，所有的操作都需要队列执行。所以种隔离级别下所有的数据是最稳定的，但是性能也是最差的。数据库的锁实现就是这种隔离级别的更小粒度版本。

发生时间	SessionA	SessionB
1	begin;	
2		begin;
3		update user set name='李四' where id=1;
4	select * from user where id=1;(等待、wait)	
5		commit；
6	select * from user where id=1;(李四)	

事务和MVCC原理

不同事务同时操作同一条数据产生的问题

示例：

发生时间	SessionA	SessionB
1	begin;	
2		begin;
3		查询余额 = 1000元
4	查询余额 = 1000元	
5		存入金额 100元，修改余额为 1100元
6	取出现金100元，此时修改余额为900元	
8		提交事务（余额=1100）
9	提交事务（余额=900）	

发生时间	SessionA	SessionB
1	begin;	
2		begin;
3		查询余额 = 1000元
4	查询余额 = 1000元	
5		存入金额 100元，修改余额为 1100元
6	取出现金100元，此时修改余额为900元	
8		提交事务（余额=1100）
9	撤销事务（余额恢复为1000元）	

上面的两种情况就是对于一条数据，多个事务同时操作可能会产生的问题，会出现某个事务的操作被覆盖而导致数据丢失。

LBCC 解决数据丢失

LBCC，基于锁的并发控制，Lock Based Concurrency Control。

使用锁的机制，在当前事务需要对数据修改时，将当前事务加上锁，同一个时间只允许一条事务修改当前数据，其他事务必须等待锁释放之后才可以操作。

MVCC 解决数据丢失

MVCC，多版本的并发控制，Multi-Version Concurrency Control。

使用版本来控制并发情况下的数据问题，在B事务开始修改账户且事务未提交时，当A事务需要读取账户余额时，此时会读取到B事务修改操作之前的账户余额的副本数据，但是如果A事务需要修改账户余额数据就必须要等待B事务提交事务。

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)。

InnoDB的MVCC实现逻辑

InnoDB存储引擎保存的MVCC的数据

InnoDB的MVCC是通过在每行记录后面保存两个隐藏的列来实现的。**一个保存了行的事务ID（DB_TRX_ID），一个保存了行的回滚指针（DB_ROLL_PT）**。每开始一个新的事务，都会自动递增产 生一个新的事务id。事务开始时刻的会把事务id放到当前事务影响的行事务id中，当查询时需要用当前事务id和每行记录的事务id进行比较。

下面看一下在REPEATABLE READ隔离级别下，MVCC具体是如何操作的。

SELECT

InnoDB 会根据以下两个条件检查每行记录：

1. InnoDB只查找版本早于当前事务版本的数据行（也就是，行的事务编号小于或等于当前事务的事务编号），这样可以确保事务读取的行，要么是在事务开始前已经存在的，要么是事务自身插入或者修改过的。
2. 删除的行要事务ID判断，读取到事务开始之前状态的版本，只有符合上述两个条件的记录，才能返回作为查询结果。

INSERT

InnoDB为新插入的每一行保存当前事务编号作为行版本号。

DELETE

InnoDB为删除的每一行保存当前事务编号作为行删除标识。

UPDATE

InnoDB为插入一行新记录，保存当前事务编号作为行版本号，同时保存当前事务编号到原来的行作为行删除标识。

保存这两个额外事务编号，使大多数读操作都可以不用加锁。这样设计使得读数据操作很简单，性能很好，并且也能保证只会读取到符合标准的行。不足之处是每行记录都需要额外的存储空间，需要做更多的行检查工作，以及一些额外的维护工作。

> MVCC只在REPEATABLE READ和READ COMMITIED两个隔离级别下工作。其他两个隔离级别都和MVCC不兼容，因为READ UNCOMMITIED总是读取最新的数据行，而不是符合当前事务版本的数据行。而SERIALIZABLE则会对所有读取的行都加锁。

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

undo log

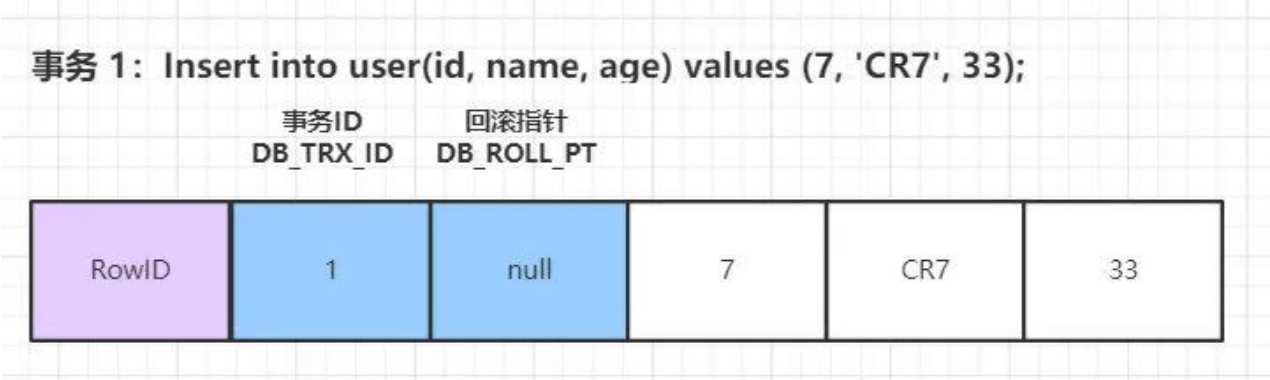
根据行为的不同，undo log分为两种：**insert undo log** 和 **update undo log**

- **insert undo log**：

insert 操作中产生的undo log，因为insert操作记录只对当前事务本身课件，对于其他事务此记录不可见，所以 insert undo log 可以在事务提交后直接删除而不需要进行purge操作。

> purge的主要任务是将数据库中已经 mark del 的数据删除，另外也会批量回收undo pages

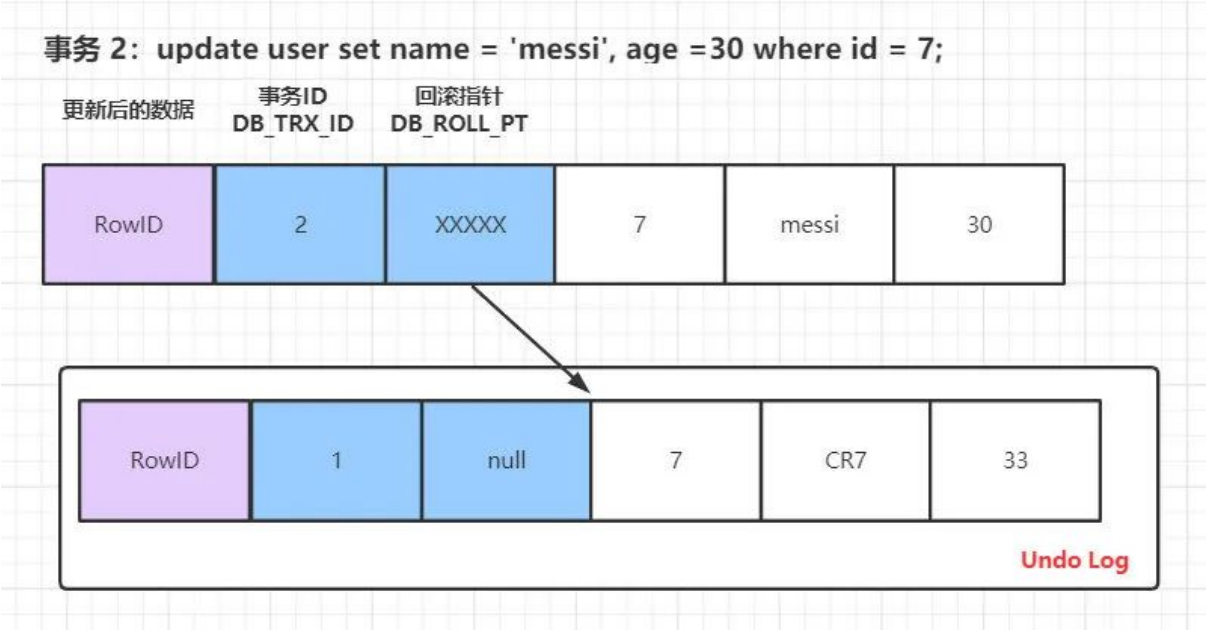
数据库 Insert时的数据初始状态：



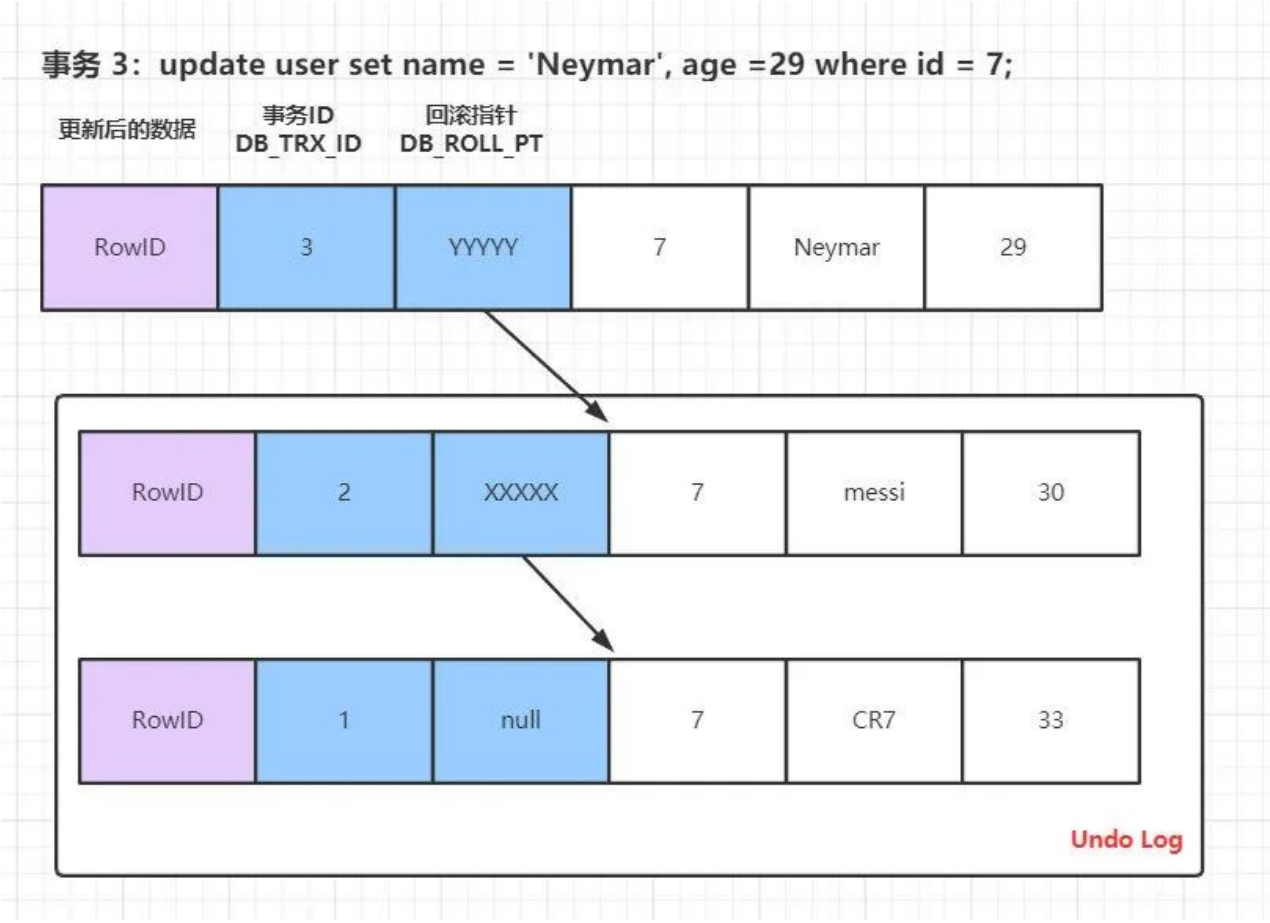
- **update undo log**：

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。

数据第一次被修改时：



当另一个事务第二次修改当前数据：



为了保证事务并发操作时，在写各自的undo log时不产生冲突，InnoDB采用回滚段的方式来维护undo log的并发写入和持久化。回滚段实际上是一种 Undo 文件组织方式。

ReadView

对于 **RU(READ UNCOMMITTED)** 隔离级别下，所有事务直接读取数据库的最新值即可，和 **SERIALIZABLE** 隔离级别，所有请求都会加锁，同步执行。所以这对这两种情况下是不需要使用到 **Read View** 的版本控制。

对于 **RC(READ COMMITTED)** 和 **RR(REPEATABLE READ)** 隔离级别的实现就是通过上面的版本控制来完成。两种隔离界别下的核心处理逻辑就是判断所有版本中哪个版本是当前事务可见的处理。针对这个问题InnoDB在设计上增加了**ReadView**的设计，**ReadView**中主要包含当前系统中还有哪些活跃的读写事务，把它们的事务id放到一个列表中，我们把这个列表命名为为**m_ids**。

对于查询时的版本链数据是否看见的判断逻辑：

- 如果被访问版本的 trx_id 属性值小于 m_ids 列表中最小的事务id，表明生成该版本的事务在生成 ReadView 前已经提交，所以该版本可以被当前事务访问。
- 如果被访问版本的 trx_id 属性值大于 m_ids 列表中最大的事务id，表明生成该版本的事务在生成 ReadView 后才生成，所以该版本不可以被当前事务访问。
- 如果被访问版本的 trx_id 属性值在 m_ids 列表中最大的事务id和最小事务id之间，那就需要判断一下 trx_id 属性值是不是在 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。

举个例子：

READ COMMITTED 隔离级别下的ReadView

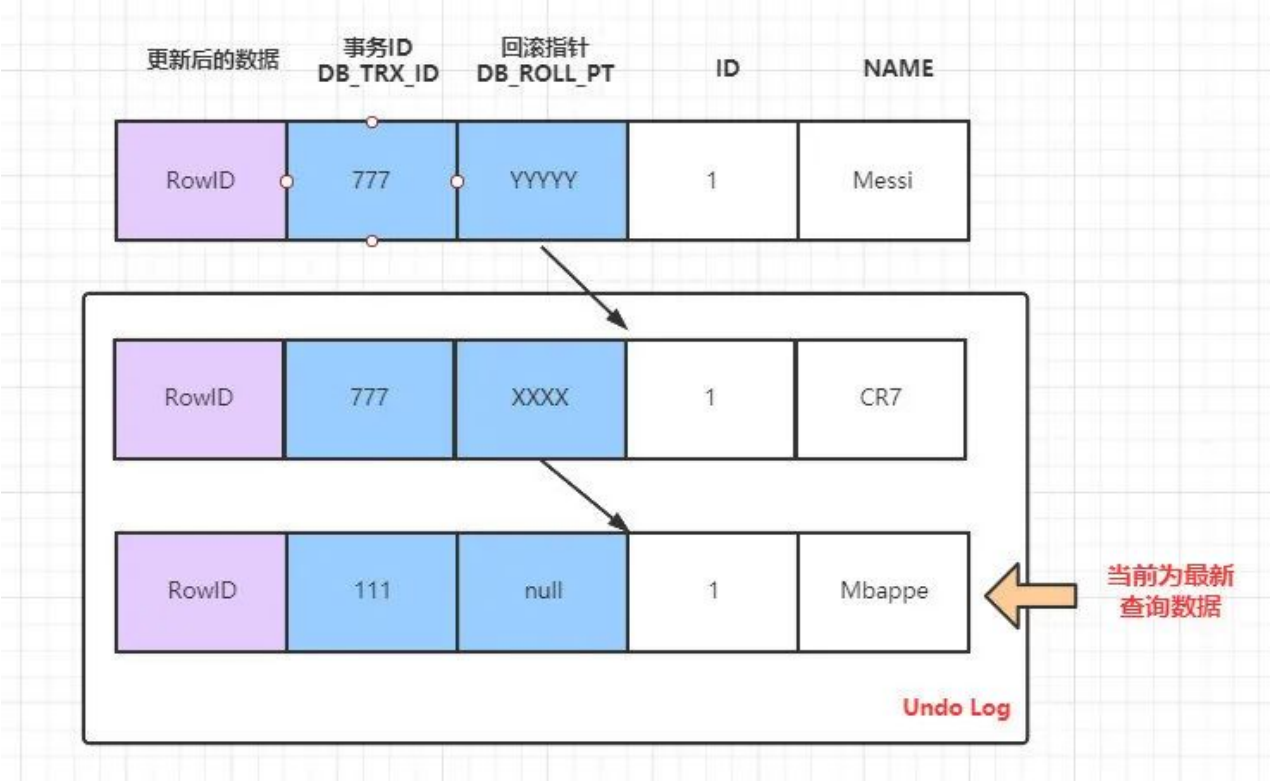
每次读取数据前都生成一个ReadView (m_ids列表)

时间	Transaction 777	Transaction 888	Trasaction 999
T1	begin;		
T2		begin;	begin；
T3	UPDATE user SET name = 'CR7' WHERE id = 1;		
T4		...	
T5	UPDATE user SET name = 'Messi' WHERE id = 1;		SELECT * FROM user where id = 1;
T6	commit;		
T7		UPDATE user SET name = 'Neymar' WHERE id = 1;	
T8			SELECT * FROM user where id = 1;
T9		UPDATE user SET name = 'Dybala' WHERE id = 1;	
T10		commit;	
T11			SELECT * FROM user where id = 1;

这里分析下上面的情况下的ReadView

时间点 T5 情况下的 SELECT 语句：

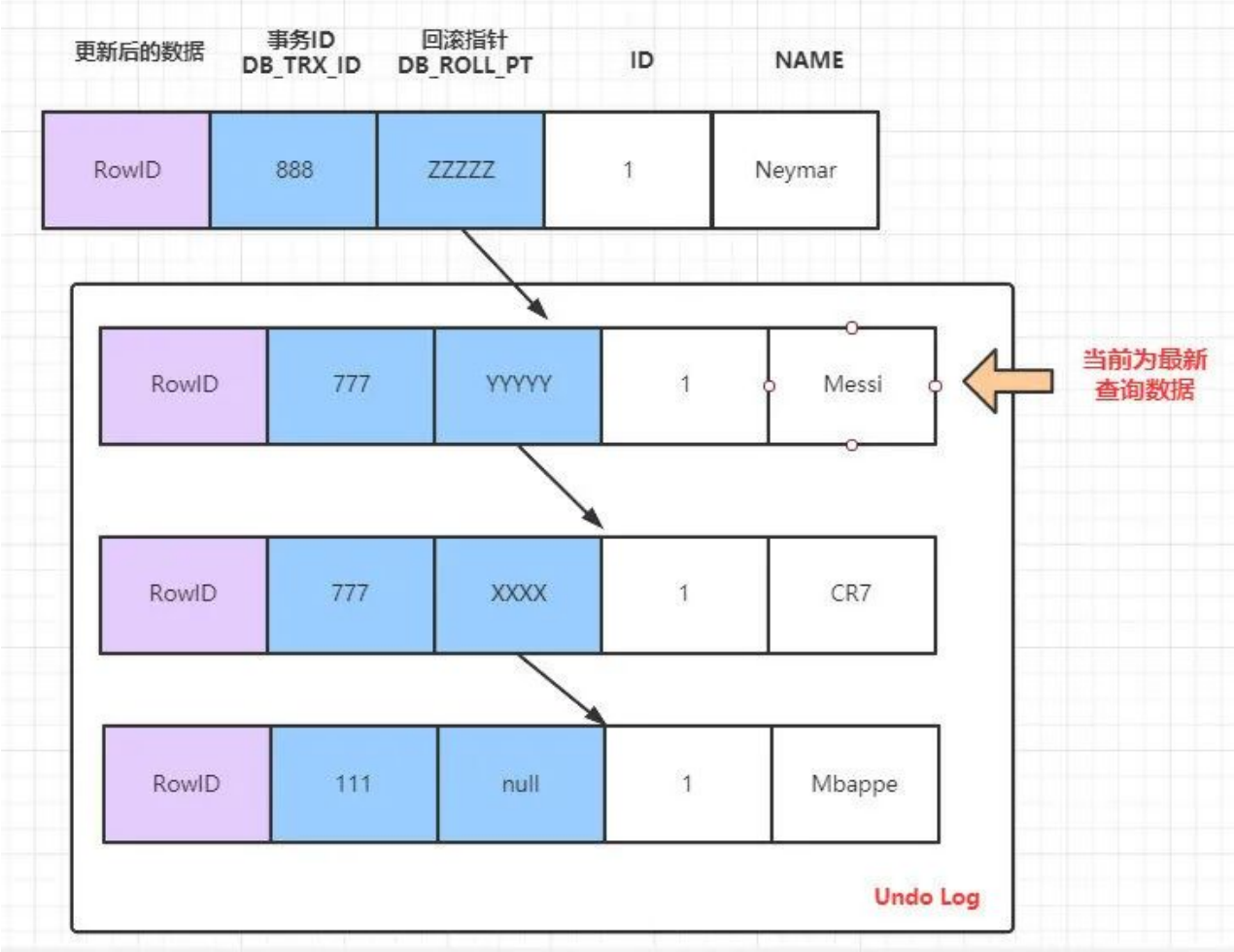
当前时间点的版本链：



> 此时 SELECT 语句执行，当前数据的版本链如上，因为当前的事务777，和事务888 都未提交，所以此时的活跃事务的ReadView的列表情况 **m_ids：[777, 888]**，因此查询语句会根据当前版本链中小于 **m_ids** 中的最大的版本数据，即查询到的是 Mbappe。

时间点 T8 情况下的 SELECT 语句：

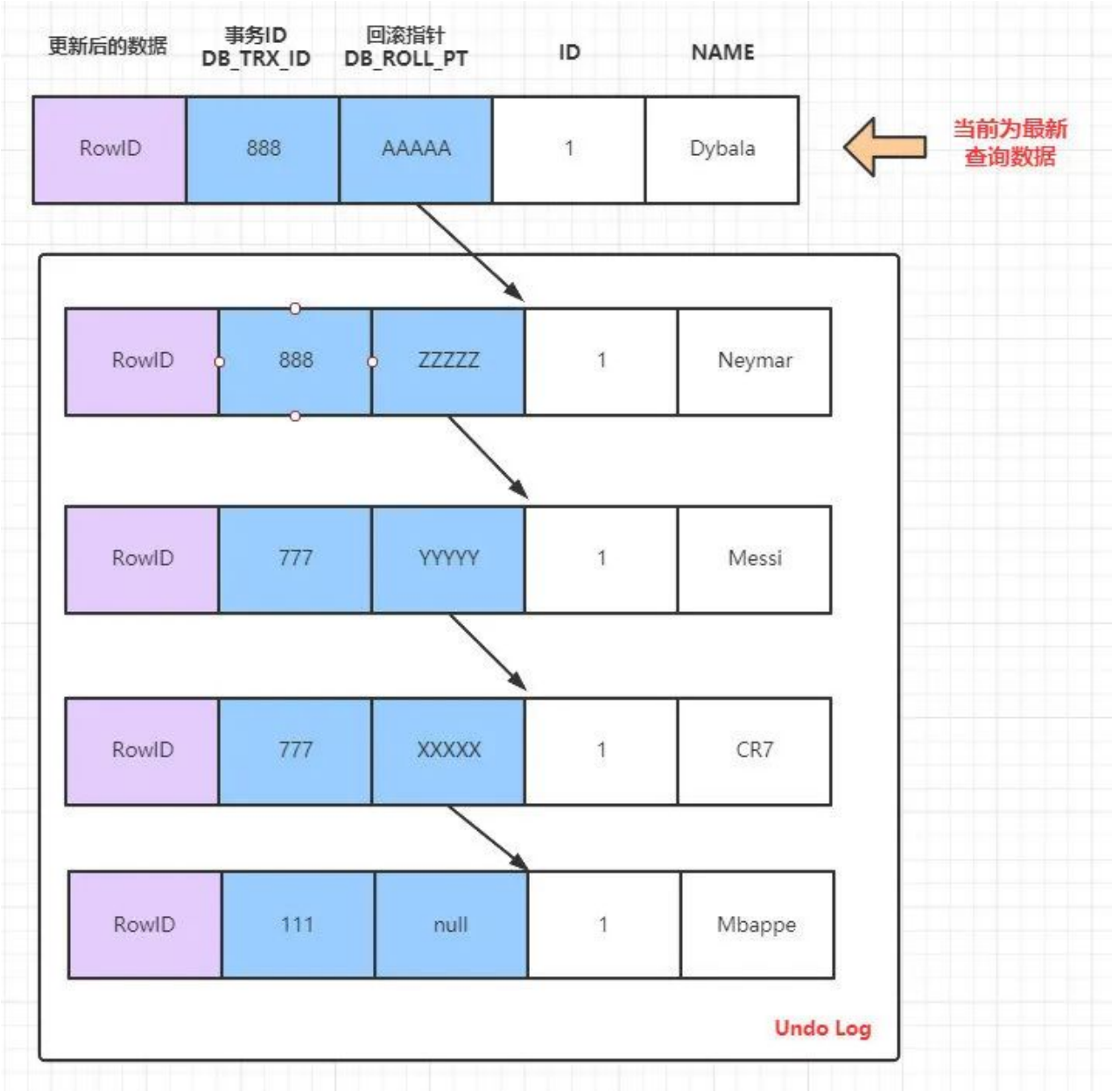
当前时间的版本链情况：



> 此时 SELECT 语句执行，当前数据的版本链如上，因为当前的事务777已经提交，和事务888 未提交，所以此时的活跃事务的ReadView的列表情况 **m_ids：[888]**，因此查询语句会根据当前版本链中小于 **m_ids** 中的最大的版本数据，即查询到的是 Messi。

时间点 T11 情况下的 SELECT 语句：

当前时间点的版本链信息：



> 此时 SELECT 语句执行，当前数据的版本链如上，因为当前的事务777和事务888 都已经提交，所以此时的活跃事务的ReadView的列表为空 ，因此查询语句会直接查询当前数据库最新数据，即查询到的是 Dybala。

总结： 使用READ COMMITTED隔离级别的事务在每次查询开始时都会生成一个独立的 ReadView。

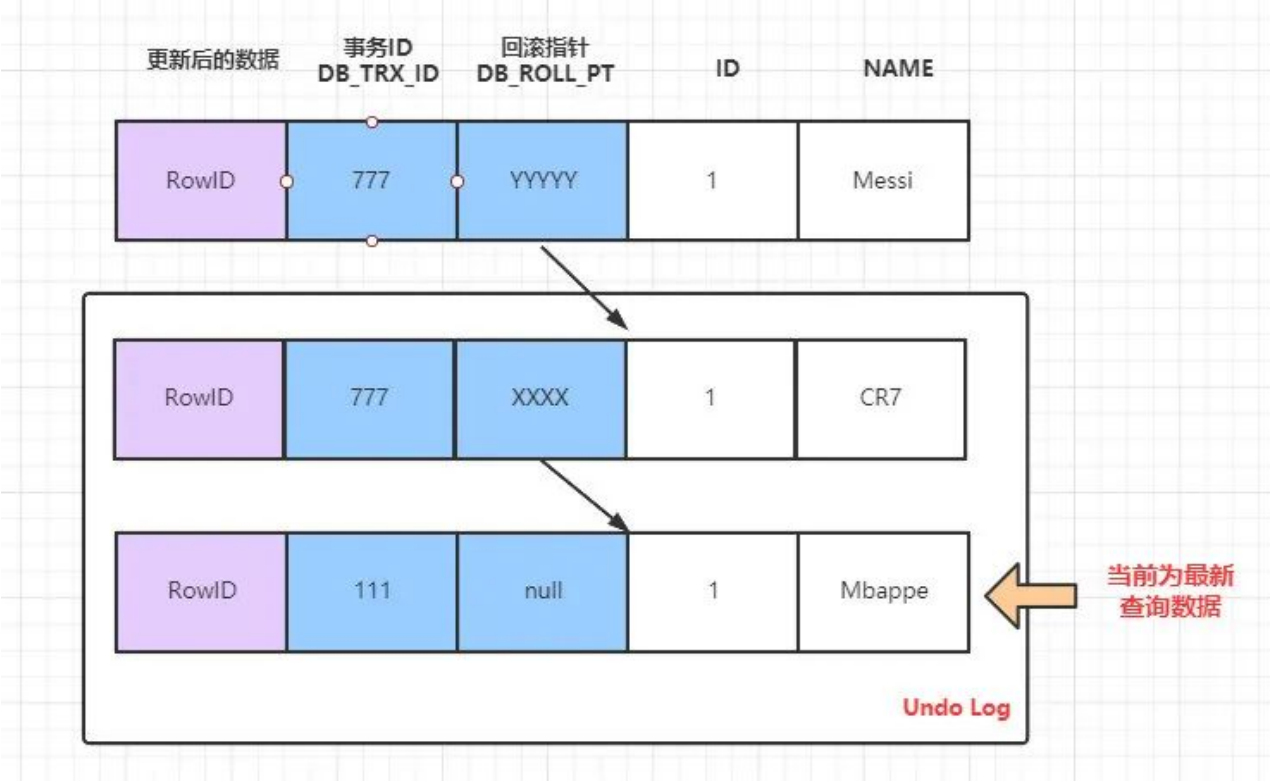
REPEATABLE READ 隔离级别下的ReadView

在事务开始后第一次读取数据时生成一个ReadView（m_ids列表）

时间	Transaction 777	Transaction 888	Trasaction 999
T1	begin;		
T2		begin;	begin；
T3	UPDATE user SET name = 'CR7' WHERE id = 1;		
T4		...	
T5	UPDATE user SET name = 'Messi' WHERE id = 1;		SELECT * FROM user where id = 1;
T6	commit;		
T7		UPDATE user SET name = 'Neymar' WHERE id = 1;	
T8			SELECT * FROM user where id = 1;
T9		UPDATE user SET name = 'Dybala' WHERE id = 1;	
T10		commit;	
T11			SELECT * FROM user where id = 1;

时间点 T5 情况下的 SELECT 语句：

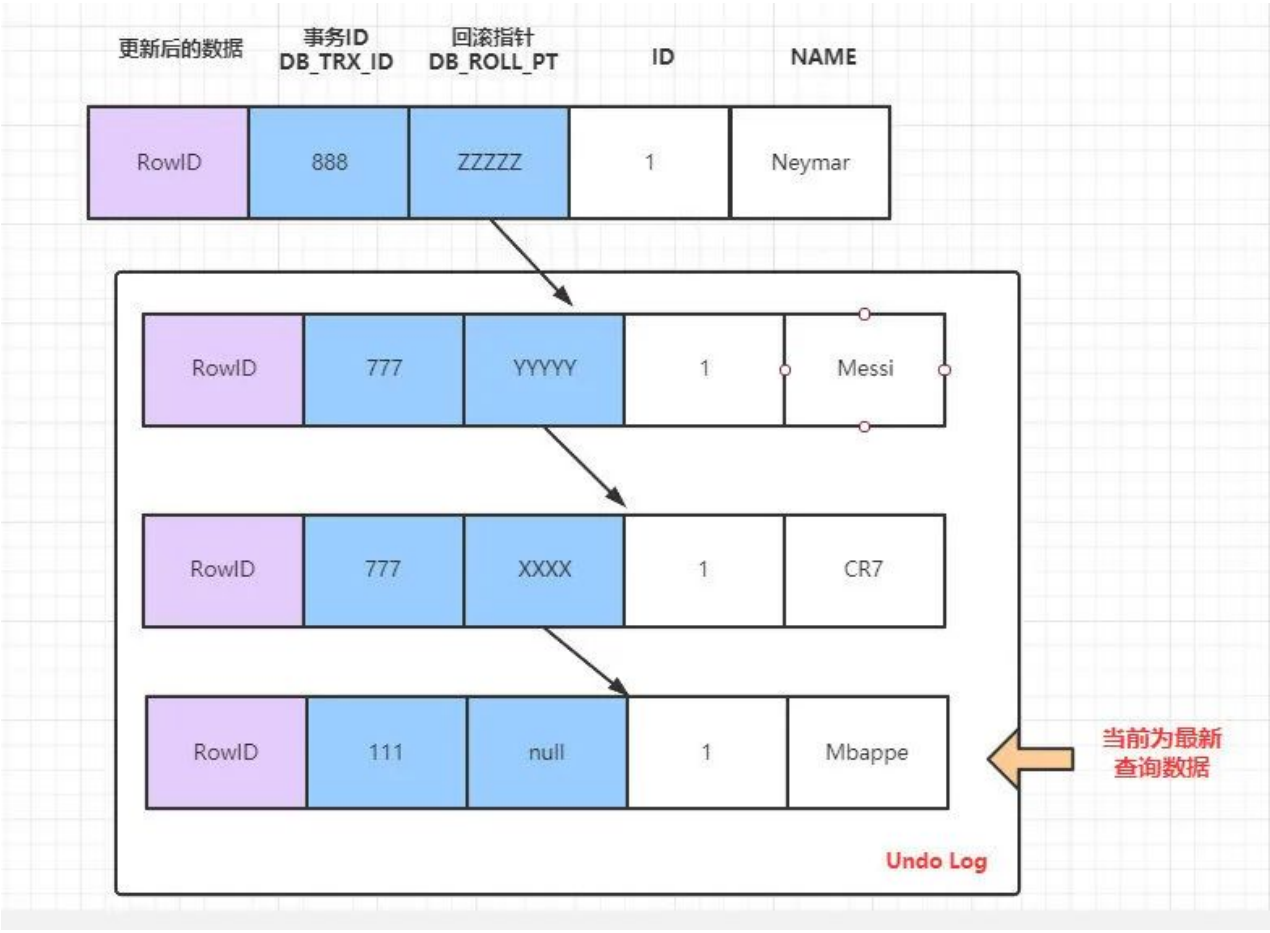
当前版本链：



> 再当前执行select语句时生成一个ReadView，此时 **m_ids** 内容是：[777,888]，所以但前根据ReadView可见版本查询到的数据为 Mbappe。

时间点 T8 情况下的 SELECT 语句：

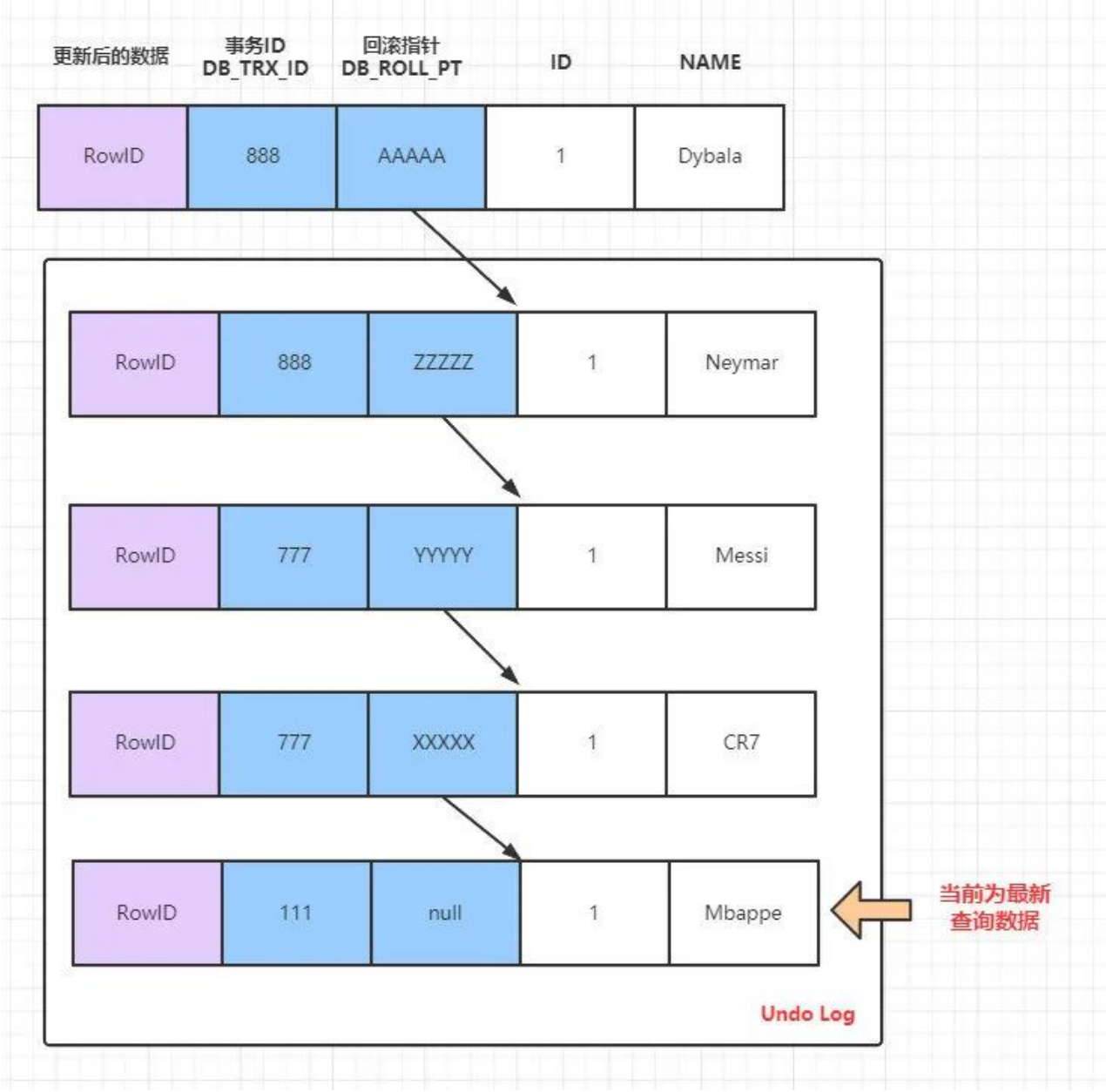
当前的版本链：



> 此时在当前的 Transaction 999 的事务里。由于T5的时间点已经生成了ReadView，所以再当前的事务中只会生成一次ReadView，所以此时依然沿用T5时的**m_ids：[777,999]**，所以此时查询数据依然是 Mbappe。

时间点 T11 情况下的 SELECT 语句：

当前的版本链：



> 此时情况跟T8完全一样。由于T5的时间点已经生成了ReadView，所以再当前的事务中只会生成一次ReadView，所以此时依然沿用T5时的**m_ids：[777,999]**，所以此时查询数据依然是 Mbappe。

MVCC总结：

所谓的MVCC（Multi-Version Concurrency Control ， 多版本并发控制）指的就是在使用 **READ COMMITTD** 、**REPEATABLE READ** 这两种隔离级别的事务在执行普通的 SEELCT 操作时访问记录的版本链的过程，这样子可以使不同事务的 `读-写` 、 `写-读` 操作并发执行，从而提升系统性能。

在 MySQL 中， READ COMMITTED 和 REPEATABLE READ 隔离级别的的一个非常大的区别就是它们生成 ReadView 的时机不同。在 READ COMMITTED 中每次查询都会生成一个实时的 ReadView，做到保证每次提交后的数据是处于当前的可见状态。而 REPEATABLE READ 中，在当前事务第一次查询时生成当前的 ReadView，并且当前的 ReadView 会一直沿用到当前事务提交，以此来保证可重复读（REPEATABLE READ）。

我是小溪，你知道的越多，你不知道的越多，我们下次见！