MySQL事务篇

1. 一条Insert语句

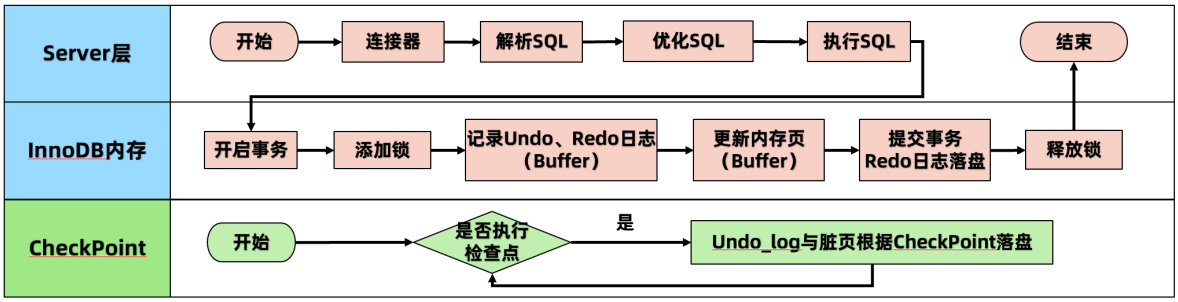
为了故事的顺利发展，我们需要创建一个表：

```
CREATE TABLE `tab_user` (
  `id` int(11) NOT NULL,
  `name` varchar(100) DEFAULT NULL,
  `age` int(11) NOT NULL,
  `address` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;

```

然后向这个表里插入一条数据：

```
Insert into tab_user(id,name,age,address) values (1,'刘备',18,'蜀国');
```

2. Insert语句执行流程



3. 事务回顾

事务指的是逻辑上的一组操作，组成这组操作的各个单元要么全都成功，要么全都失败。

事务作用：保证在一个事务中多次SQL操作要么全都成功，要么全都失败。

`MySQL` 是一个服务器 / 客户端架构的软件，对于同一个服务器来说，可以有若干个客户端与之连接，每个客户端与服务器连接上之后，就可以称之为一个会话（`Session`）。我们可以同时在不同的会话里输入各种语句，这些语句可以作为事务的一部分进行处理。不同的会话可以同时发送请求，也就是说服务器可能同时在处理多个事务，这样子就会导致不同的事务可能同时访问到相同的记录。

事务的**隔离性**在理论上是指，在某个事务对某个数据进行访问时，其他事务应该进行排队，当该事务提交之后，其他事务才可以继续访问这个数据。但是这样子的话对性能影响太大，所以才会出现各种**隔离级别**，来最大限度的提升系统并发处理事务的能力，牺牲部分**隔离性**来提升性能。

事务是数据库最为重要的机制之一，凡是使用过数据库的人，都了解数据库的事务机制，也对ACID四个基本特性如数家珍。但是聊起事务或者ACID的底层实现原理，往往言之不详，不明所以。所以接下来我们深入分析事务的原理。

由于在**MySQL中的事务是由存储引擎实现**，而且MySQL只有InnoDB支持事务。因此我们讲解**InnoDB**的事务。

3.1 事务四大特性ACID

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）：** 原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。
- **一致性（Consistency）：** 事务前后数据的完整性必须保持一致
- **隔离性（Isolation）：** 多个用户并发访问数据库时，一个用户的事务不能被其它用户的事务所干扰，多个并发事务之间数据要相互隔离。**隔离性由隔离级别保障！**
- **持久性（Durability）：** 一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

3.2 事务并发问题

1. 脏读：一个事务读到了另一个事务**未提交**的数据
2. 不可重复读：一个事务读到了另一个事务**已经提交**(update)的数据。引发事务中的多次查询结果不一致
3. 虚读 /幻读：一个事务读到了另一个事务已经**插入(insert)**的数据。导致事务中多次查询的结果不一致
4. **丢失更新的问题！**

3.3 隔离级别

- **read uncommitted** 读未提交【RU】，一个事务读到另一个事务没有提交的数据
  - 存在：3个问题（脏读、不可重复读、幻读）。
- **read committed** 读已提交【RC】，一个事务读到另一个事务已经提交的数据
  - 存在：2个问题（不可重复读、幻读）。
  - 解决：1个问题（脏读）
- **repeatable read**:可重复读【RR】，在一个事务中读到的数据始终保持一致，无论另一个事务是否提交
  - 解决：3个问题（脏读、不可重复读、幻读）
- **serializable 串行化**，同时只能执行一个事务，相当于事务中的单线程
  - 解决：3个问题（脏读、不可重复读、幻读）

安全和性能对比

- 安全性：`serializable > repeatable read > read committed > read uncommitted`
- 性能： `serializable < repeatable read < read committed < read uncommitted`

常见数据库的默认隔离级别：

- MySql：`repeatable read`
- Oracle：`read committed`

4. 事务底层原理详解

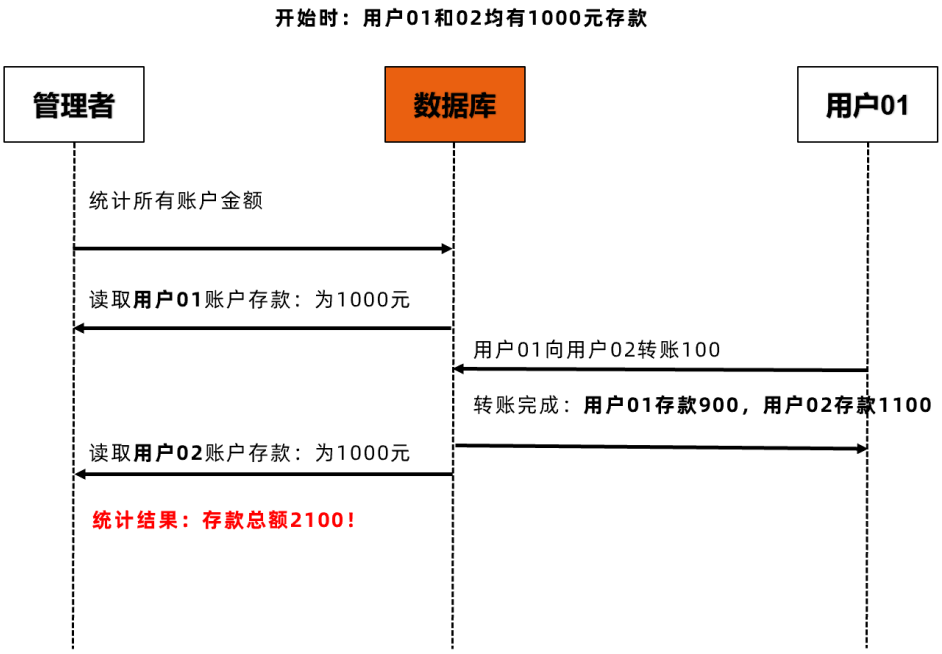
> **如何实现的隔离级别？**RC是怎么实现的？RR是怎么实现的呢？

4.1 丢失更新问题

两个事务针对同一数据进行修改操作时会丢失更新，这个现象称之为丢失更新问题

举个栗子：管理者查询所有用户的存款总额，假设除了用户01和用户01之外，其他用户的存款都为0，用户01、02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01会向用户02进行转账操作。

转账和查询总额操作的时序图如下：

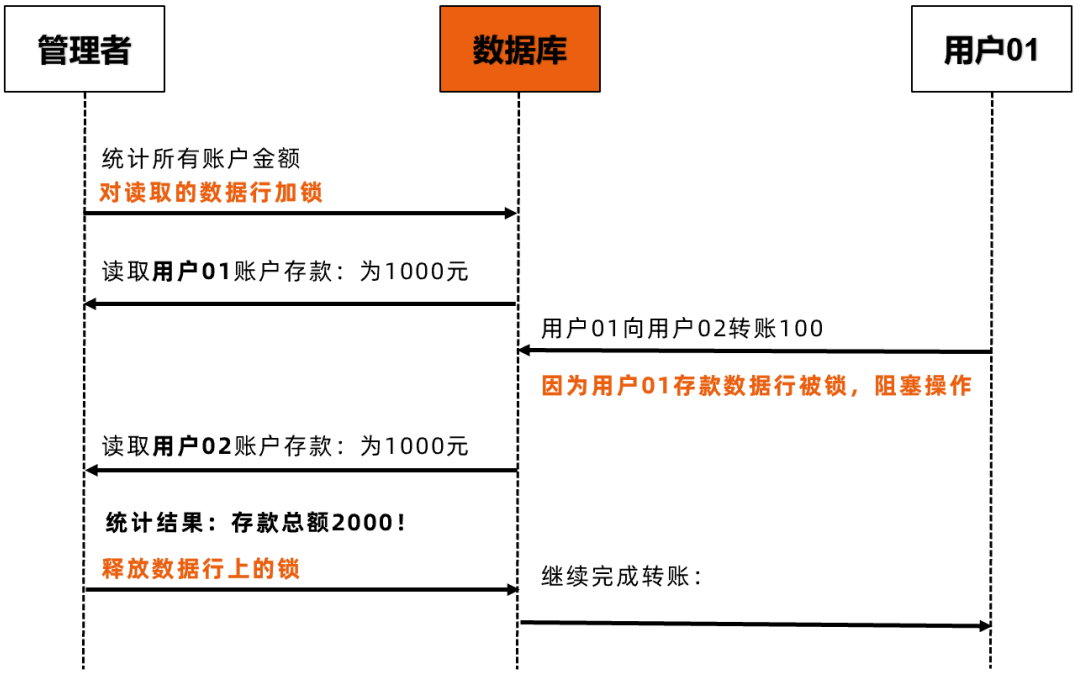


4.2 解决方案

4.2.1 解决方案一：基于锁并发控制LBCC

使用基于锁的并发控制LBCC（Lock Based Concurrency Control）可以解决上述问题。

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。



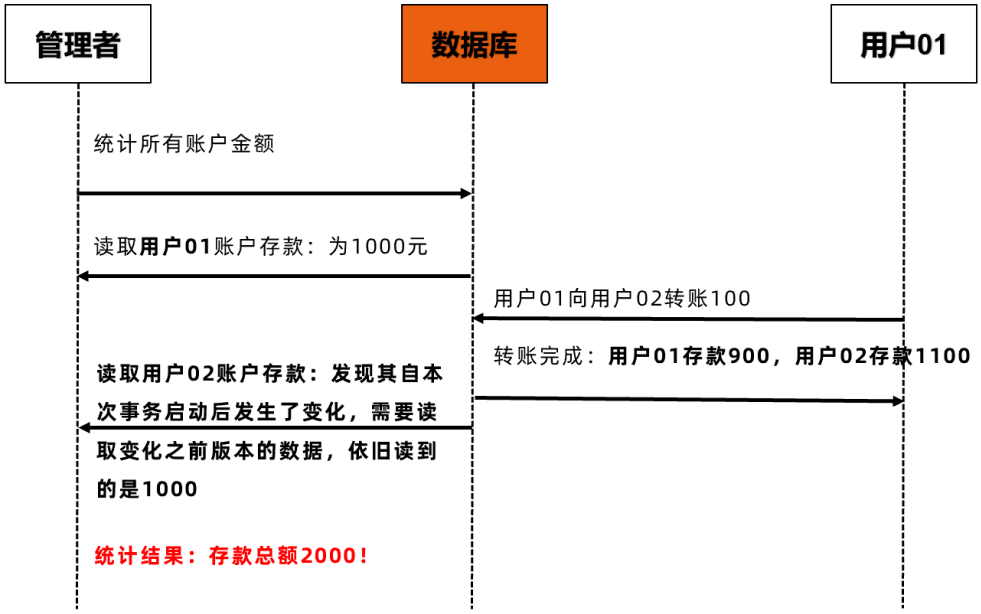
这种方案比较简单粗暴，就是一个事务去读取一条数据的时候，就上锁，不允许其他事务来操作。假如当前事务只是加**读锁**，那么其他事务就不能有**写锁**，也就是不能修改数据；而假如当前事务需要加**写锁**，那么其他事务就不能持有任何锁。总而言之，能加锁成功，就确保了除了当前事务之外，其他事务不会对当前数据产生影响，所以自然而然的，当前事务读取到的数据就只能是**最新**的，而不会是**快照**数据。

> 关于锁，会在锁篇详细讲解

4.2.2 解决方案二：基于版本并发控制MVCC

当然使用版本的并发控制MVCC（Multi Version Concurrency Control）机制也可以解决这个问题。

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副本数据【快照数据】。



MVCC使得普通的SELECT请求不加锁，读写不冲突，显著提高了数据库的并发处理能力。MVCC保障了ACID中的隔离性，究竟怎么实现？接下来看

4.3 MVCC实现原理【InnoDB】

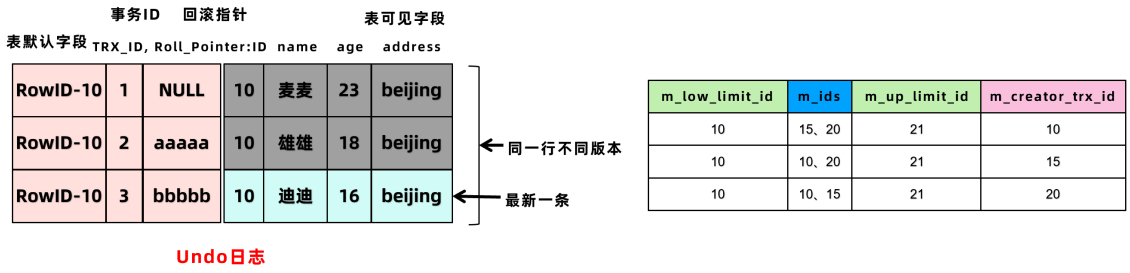
首先来看一下MVCC的定义：

> **Multiversion concurrency control (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.

MVCC全称叫多版本并发控制，是RDBMS常用的一种并发控制方法，用来对数据库数据进行并发访问，实现事务。。核心思想是**读不加锁，读写不冲突。**在读多写少的OLTP应用中，读写不冲突非常重要，极大的增加了系统的并发性能，这也是为什么几乎所有的RDBMS，都支持MVCC的原因。

MVCC 实现原理是数据快照，不同的事务访问不同版本的数据快照，从而实现事务下对数据的隔离级别。虽然说具有多个版本的数据快照，但这并不意味着必须拷贝数据，保存多份数据文件（这样会浪费存储空间），InnoDB通过事务的Undo日志巧妙地实现了多版本的数据快照。

MVCC 的实现依赖与**Undo日志** 与 **Read View** 。



InnoDB下的表有**默认字段**和**可见字段**，默认字段是实现MVCC的关键，默认字段是隐藏的列。默认字段最关键的两个列，**一个保存了行的事务ID，一个保存了行的回滚指针。**每开始新的事务，都会自动递增产生一个新的事务id。事务开始后，生成当前事务影响行的ReadView。当查询时，需要用当前查询的事务id与ReadView确定要查询的数据版本。

4.3.1 Undo日志

Redo日志记录了事务的行为，可以很好地通过其对页进行"重做"操作。但是事务有时还需要进行回滚操作，这时就需要undo。因此在对数据库进行修改时，InnoDB存储引擎不但会产生Redo，还会产生一定量的Undo。这样如果用户执行的事务或语句由于某种原因失败了，又或者用户用一条Rollback语句请求回滚，就可以利用这些undo信息将数据回滚到修改之前的样子。在多事务读取数据时，有了Undo日志可以做到读不加锁，读写不冲突。

Undo存放在数据库内部的一个特殊段（segment）中，这个段称为Undo段（undo segment）。Undo段位于系统表空间内，也可以设置为Undo表空间。

Undo日志保存了记录修改前的快照。所以，对于更新和删除操作，InnoDB并不是真正的删除原来的记录，而是设置记录的delete mark为1。因此为了解决数据Page和Undo日志膨胀问题，则需要回收机制进行清理Undo日志。

根据行为的不同Undo日志分为两种：`Insert Undo Log`和`Update Undo Log`

1）Insert Undo日志：是在Insert操作中产生的Undo日志

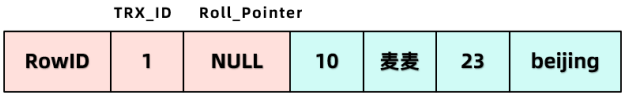
Insert 操作的记录只对事务本身可见，对于其它事务此记录是不可见的，所以 Insert Undo Log 可以在事务提交后直接删除而不需要进行回收操作。

如下图所示（初始状态）：

```
# 事务1:
Insert into tab_user(id,name,age,address) values (10,'麦麦',23,'beijing')
```

Insert into tab_user(id,name,age,address) values (10,'麦麦',23,'beijing')

事务01：初始状态，Insert操作记录只对本事务可见，对其他事务不可见，所以事务提交后直接删除Undo日志无需回收



2）Update Undo日志：是Update或Delete 操作中产生的Undo日志

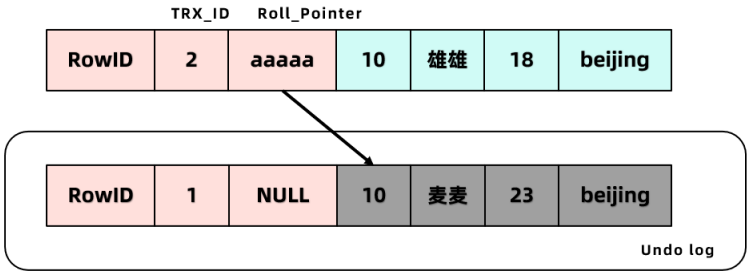
Update操作会对已经存在的行记录产生影响，为了实现MVCC多版本并发控制机制，因此Update Undo日志不能在事务提交时就删除，而是在事务提交时将日志放入指定区域，等待 Purge 线程进行最后的删除操作。

如下图所示（第一次修改）：

```
# 事务2：
update tab_user set name='雄雄',age=18 where id=10;
# 当事务2使用Update语句修改该行数据时，会首先使用写锁锁定目标行，将该行当前的值复制到Undo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中修改前的行。
```

update tab_user set name='雄雄',age=18 where id=10;

事务02：第一次修改，Update操作对已经存在行记录产生影响，为了实现MVCC，修改提交事务后，不能立即删除Update Undo日志而是会将其存入UndoLog链表中，等待Purge线程回收。

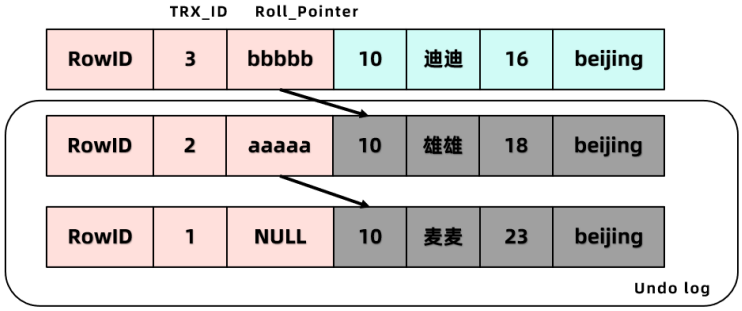


当事务3进行修改与事务2的处理过程类似，如下图所示（第二次修改）：

```
# 事务3：
update tab_user set name='迪迪',age=16 where id=10;
```

update tab_user set name='迪迪',age=16 where id=10;

事务03：第二次修改



4.3.2 ReadView

MVCC的核心问题就是：判断一下版本链中的哪个版本是当前事务可见的！

- 对于使用`RU`隔离级别的事务来说，直接读取记录的最新版本就好了，不需要Undo log。
- 对于使用`串行化`隔离级别的事务来说，使用加锁的方式来访问记录，不需要Undo log。
- 对于使用`RC`和`RR`隔离级别的事务来说，需要用到undo log的版本链。

1）什么是ReadView?

ReadView是张存储事务id的表，主要包含当前系统中有哪些活跃的读写事务，把它们的事务id放到一个列表中。结合Undo日志的默认字段【事务trx_id】来控制那个版本的Undo日志可被其他事务看见。

四个列：

- **m_ids**：表示在生成ReadView时，当前系统中**活跃的读写事务id列表**
- **m_low_limit_id**：事务id下限，表示当前系统中活跃的读写事务中最小的事务id，m_ids事务列表中的最小事务id
- **m_up_limit_id**：事务id上限，表示生成ReadView时，系统中应该分配给下一个事务的id值

- **m_creator_trx_id**：表示生成该ReadView的事务的事务id

ReadView

m_low_limit_id	m_ids	m_up_limit_id	m_creator_trx_id
10	15、20	21	10
10	10、20	21	15
10	10、15	21	20

2）ReadView怎么产生，什么时候生成?

- 开启事务之后，在第一次查询(select)时，生成ReadView
- `RC`和`RR`隔离级别的差异本质是因为MVCC中ReadView的生成时机不同，详细生成时机在案例中分析

3）如何判断可见性?

开启事务执行第一次查询时，首先生成ReadView，然后依据Undo日志和ReadView按照判断可见性，按照下边步骤判断记录的版本链的某个版本是否可见。

循环判断规则如下：

- 如果被访问版本的`trx_id`属性值，小于ReadView中的**事务下限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，等于ReadView中的`m_creator_trx_id`，**可以**被访问。
- 如果被访问版本的`trx_id`属性值，大于等于ReadView中的**事务上限id**，在生成`ReadView`后才产生的数据，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**事务下限id**和**事务上限id**之间，那就需要判断是不是在`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。

循环判断Undo log中的版本链某一的版本是否对当前事务可见，如果循环到最后一个版本也不可见的话，那么就意味着该条记录对该事务不可见，查询结果就不包含该记录。

4.3.3 ReadView案例分析

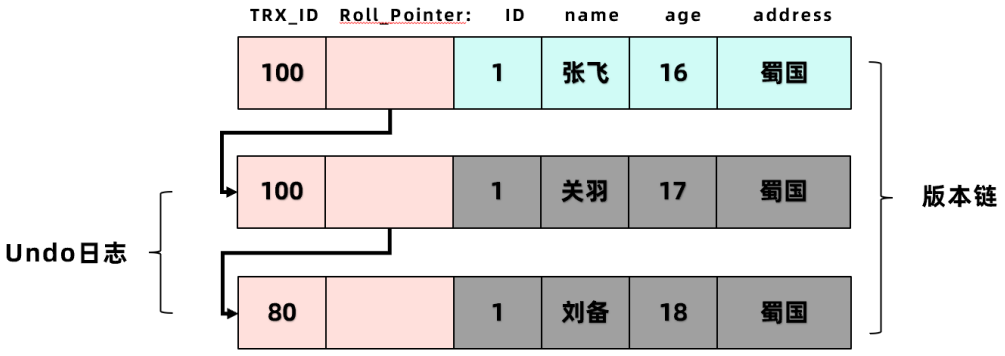
案例01-读已提交RC隔离级别下的可见性分析

每次读取数据前都生成一个ReadView，默认tab_user表中只有一条数据，数据内容是刘备。

时间	事务01【db_trx_id=100】	事务02【db_trx_id=200】	事务03【db_trx_id=300】
T1	开启事务	开启事务	开启事务
T2	更新为关羽	...	...
T3	更新为张飞	...	...
T4		更新为赵云	
T5		更新为诸葛亮	
T6			SELECT01，id=1，name为刘备
T7	提交事务01		
T8			SELECT02，id=1，name为张飞
T9		提交事务02	
T10			SELECT03，id=1，name为诸葛亮

> 注意：事务id是递增的

T3时刻，表`tab_user`中`id`为`1`的记录得到的版本链表如下所示：

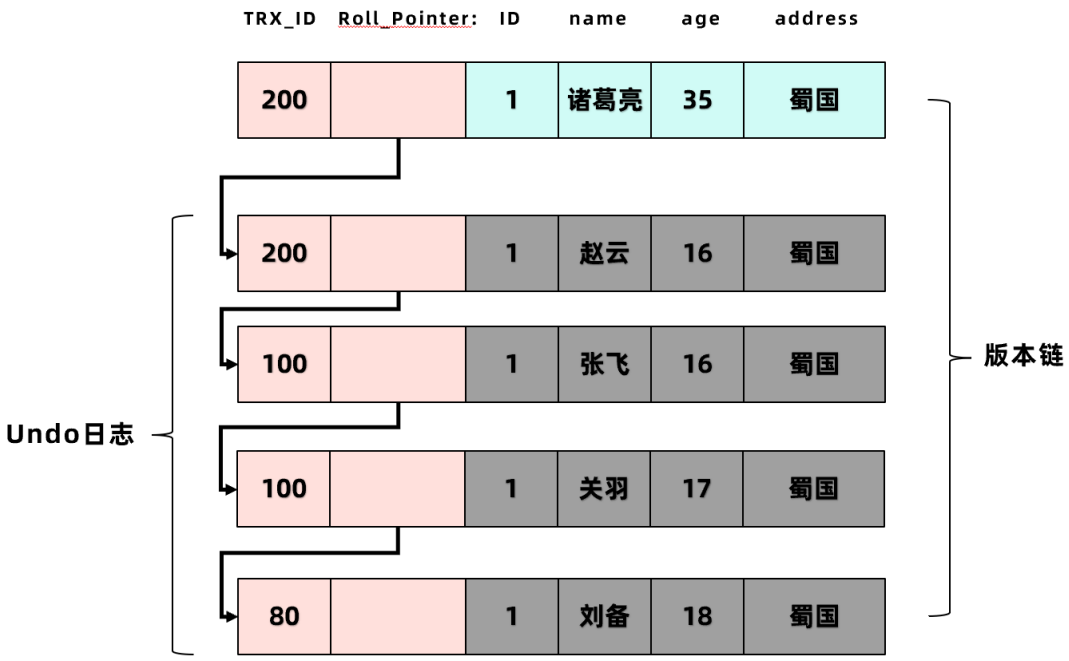


这个`SELECT01`的执行过程如下：

- 在执行`SELECT`语句时会先生成一个`ReadView`，`m_ids`列表的内容就是`[100, 200]`。
- 然后从版本链中挑选可见的记录，从图中可以看出
  - 最新版本的列`c`的内容是`'张飞'`，该版本的`trx_id`值为`100`，在`m_ids`列表内，所以不符合可见性要求，跳下一个版本。
  - 下一个版本的列`c`的内容是`'关羽'`，该版本的`trx_id`值也为`100`，也在`m_ids`列表内，所以也不符合要求，跳下一个版本。

    - 下一个版本的列`c`的内容是`'刘备'`，该版本的`trx_id`值为`80`，小于`m_ids`列表中最小的事务id`100`，**此版符合要求**
- 最后返回给用户的版本就是这条列`c`为`'刘备'`的记录。

T5时刻，表`tab_user`中`id`为`1`的记录的版本链就长这样：



这个`SELECT02`的执行过程如下：

- 在执行`SELECT`语句时会先生成一个`ReadView`，`ReadView`的`m_ids`列表的内容就是`[200]`
    - 事务id为`100`的那个事务已经提交了，所以生成快照时就没有它了
- 然后从版本链中挑选可见的记录，从图中可以看出
    - 最新版本的列`c`的内容是`'诸葛亮'`，该版本的`trx_id`值为`200`，在`m_ids`列表内，不符合可见性要求，跳下一个版本
    - 下一个版本的列`c`的内容是`'赵云'`，该版本的`trx_id`值为`200`，也在`m_ids`列表内，不符合要求，跳下一个版本
    - 下一个版本的列`c`的内容是`'张飞'`，该版本的`trx_id`值为`100`，比`m_ids`列表中最小的事务id`200`还要小，**此版符合要求**
- 最后返回给用户的版本就是这条列`c`为`'张飞'`的记录。

以此类推，如果之后事务id为`200`的记录也提交了，再此在使用`RC`隔离级别的事务中查询表`t`中`id`值为`1`的记录时，得到的结果就是`'诸葛亮'`了，具体流程我们就不分析了。

总结：使用RC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。

```sql
CREATE TABLE `tab_user` (
  `id` int(11) NOT NULL,
  `name` varchar(100) DEFAULT NULL,
  `age` int(11) NOT NULL,
  `address` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
Insert into tab_user(id,name,age,address) values (1,'刘备',18,'蜀国');
```

案例代码如下：

```sql
# 事务01
-- 查询事务隔离级别：
select @@tx_isolation;
-- 设置数据库的隔离级别
set session transaction isolation level read committed;
SELECT * FROM tab_user; # 默认是刘备
# Transaction 100
BEGIN;

UPDATE tab_user SET name = '关羽' WHERE id = 1;

UPDATE tab_user SET name = '张飞' WHERE id = 1;

COMMIT;
```

```sql
# 事务02
-- 查询事务隔离级别：
select @@tx_isolation;
-- 设置数据库的隔离级别
set session transaction isolation level read committed;

# Transaction 200
BEGIN;
# 更新了一些别的表的记录
...
UPDATE tab_user SET name = '赵云' WHERE id = 1;

UPDATE tab_user SET name = '诸葛亮' WHERE id = 1;

COMMIT;
```

```sql
# 事务03
-- 查询事务隔离级别：
select @@tx_isolation;
-- 设置数据库的隔离级别
set session transaction isolation level read committed;

BEGIN;
```

```
8
9  # SELECT01：Transaction 100、200未提交
10 SELECT * FROM tab_user WHERE id = 1; # 得到的列c的值为'刘备'
11
12 # SELECT02：Transaction 100提交，Transaction 200未提交
13 SELECT * FROM tab_user WHERE id = 1; # 得到的列c的值为'张飞'
14
15 # SELECT03：Transaction 100、200提交
16 SELECT * FROM tab_user WHERE id = 1; # 得到的列c的值为'诸葛亮'
17 COMMIT;
```

使用到的SQL小结：

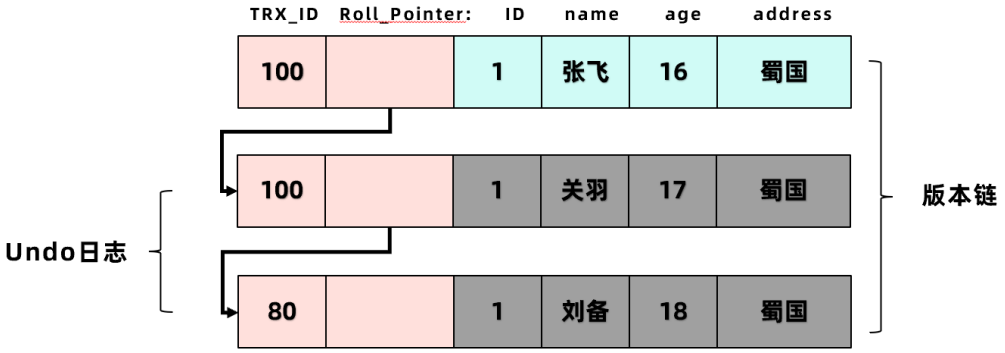
```
1  -- 开启事务：还有一种方式begin
2  start transaction
3  -- 提交事务：
4  commit
5  -- 回滚事务：
6  rollback
7  -- 查询事务隔离级别：
8  select @@tx_isolation;
9  -- 设置数据库的隔离级别
10 set session transaction isolation level read committed
11 -- 级别字符串：`read uncommitted`、`read committed`、`repeatable read【默认】`、
   `serializable`
12
13
14
15 -- 查看当前运行的事务
16 SELECT
17    a.trx_id,a.trx_state,a.trx_started,a.trx_query,
18    b.ID,b.USER,b.DB,b.COMMAND,b.TIME,b.STATE,b.INFO,
19    c.PROCESSLIST_USER,c.PROCESSLIST_HOST,c.PROCESSLIST_DB,    d.SQL_TEXT
20 FROM
21    information_schema.INNODB_TRX a
22 LEFT JOIN information_schema.PROCESSLIST b ON a.trx_mysql_thread_id = b.id
   AND b.COMMAND = 'Sleep'
23 LEFT JOIN PERFORMANCE_SCHEMA.threads c ON b.id = c.PROCESSLIST_ID
24 LEFT JOIN PERFORMANCE_SCHEMA.events_statements_current d ON d.THREAD_ID =
   c.THREAD_ID;
```

案例02-可重复读RR隔离级别下的可见性分析

在事务开始后第一次读取数据时生成一个ReadView。对于使用`RR`隔离级别的事务来说，只会在第一次执行查询语句时生成一个`ReadView`，之后的查询就不会重复生成了。我们还是用例子看一下是什么效果。

代码与执行流程与RC案例完全相同，唯一不同的是事务隔离级别。

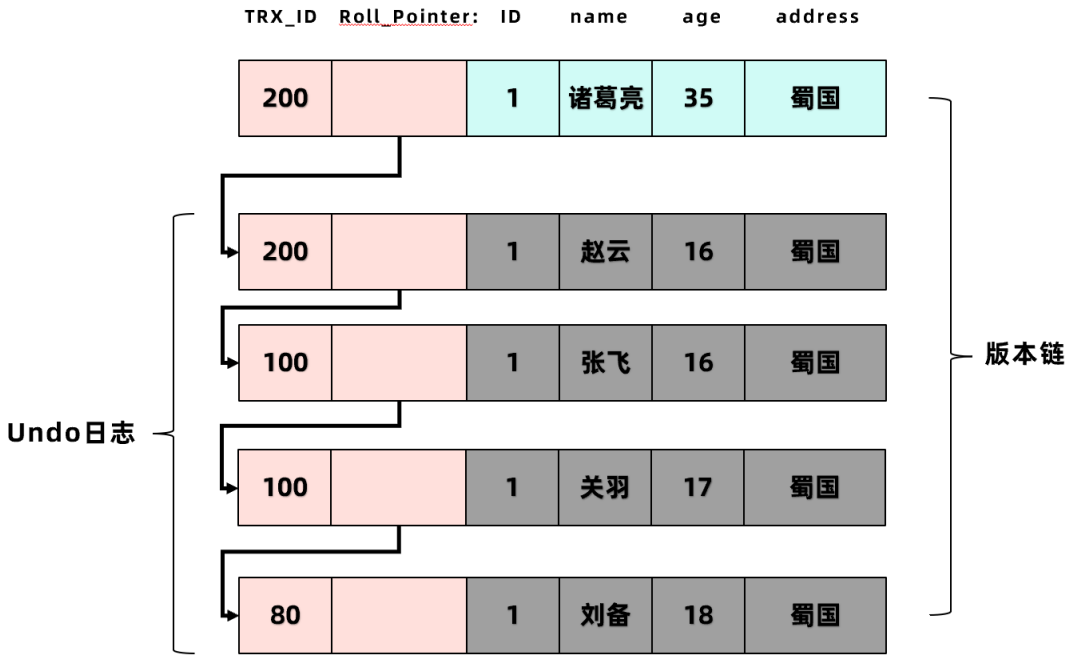
T3时刻，表`t`中`id`为`1`的记录得到的版本链表如下所示：



这个`SELECT1`的执行过程如下：

- 在执行`SELECT`语句时会先生成一个`ReadView`，`ReadView`的`m_ids`列表的内容就是`[100, 200]`。
- 然后从版本链中挑选可见的记录，从图中可以看出，
  - 最新版本的列`c`的内容是`'张飞'`，该版本的`trx_id`值为`100`，在`m_ids`列表内，不符合可见性要求，跳下一个版本。
  - 下一个版本的列`c`的内容是`'关羽'`，该版本的`trx_id`值也为`100`，也在`m_ids`列表内，不符合要求，跳下一个版本。
  - 下一个版本的列`c`的内容是`'刘备'`，该版本的`trx_id`值为`80`，小于`m_ids`列表中最小的事务id`100`，**版本符合要求**
- 最后返回给用户的版本就是这条列`c`为`'刘备'`的记录。

T5时刻，表`t`中`id`为`1`的记录的版本链就长这样：



这个`SELECT2`的执行过程如下：

- **因为之前已经生成过`ReadView`了，所以此时直接复用之前的`ReadView`，之前的`ReadView`中的`m_ids`列表就是`[100, 200]`。**
- 然后从版本链中挑选可见的记录，从图中可以看出：

    - 最新版本的列`c`的内容是`'诸葛亮'`，该版本的`trx_id`值为`200`，在`m_ids`列表内，不符合可见性要求，跳下一个版本
    - 下一个版本的列`c`的内容是`'赵云'`，该版本的`trx_id`值为`200`，也在`m_ids`列表内，不符合要求，跳下一个版本
    - 下一个版本的列`c`的内容是`'张飞'`，该版本的`trx_id`值为`100`，也在`m_ids`列表内，不符合要求，跳下一个版本
    - 下一个版本的列`c`的内容是`'关羽'`，该版本的`trx_id`值为`100`，也在`m_ids`列表内，不符合要求，跳下一个版本
    - 下一个版本的列`c`的内容是`'刘备'`，该版本的`trx_id`值为`80`，`80`小于`m_ids`列表中最小的事务id`100`，**版本符合要求**
- 最后返回给用户的版本就是这条列`c`为`'刘备'`的记录。

也就是说两次`SELECT`查询得到的结果是重复的，记录的列`c`值都是`'刘备'`，这就是`可重复读`的含义。

如果我们之后再把事务id为`200`的记录提交了，之后再到刚才使用`REPEATABLE READ`隔离级别的事务中继续查找这个id为`1`的记录，得到的结果还是`'刘备'`，具体执行过程大家可以自己分析一下。

注意：**MVCC只在RR和RC两个隔离级别下工作。RU和串行化隔离级别不需要 MVCC，为什么?**

- 因为RU总是读取最新的数据行，本身就没有隔离性，也不解决并发潜在问题，因此不需要!
- 而SERIALIZABLE则会对所有读取的行都加锁，相当于串行执行，线程之间绝对隔离，也不需要。

4.4 MVCC下的读操作

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。刚才案例中都是快照读。
- **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。

4.4.1 当前读与快照读

快照读也就是一致性非锁定读(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。ReadView的读取操作就是快照读；

举例：

- **快照读：**简单的select操作，属于快照读，不加锁。

```
1 select * from table where ?;
```

- **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。

```
select * from table where ? lock in share mode; # 加读锁
select * from table where ? for update;# 加写锁

insert into table values (…);# 加写锁
update table set ? where ?;# 加写锁
delete from table where ?;# 加写锁
# 所有以上的语句，都属于当前读，读取记录的最新版本。并且，读取之后，还需要保证其他并发事务不能修改当前记录，对读取记录加锁。
# 其中，除了第一条语句，对读取记录加读锁外，其他的操作都加的是写锁。
```

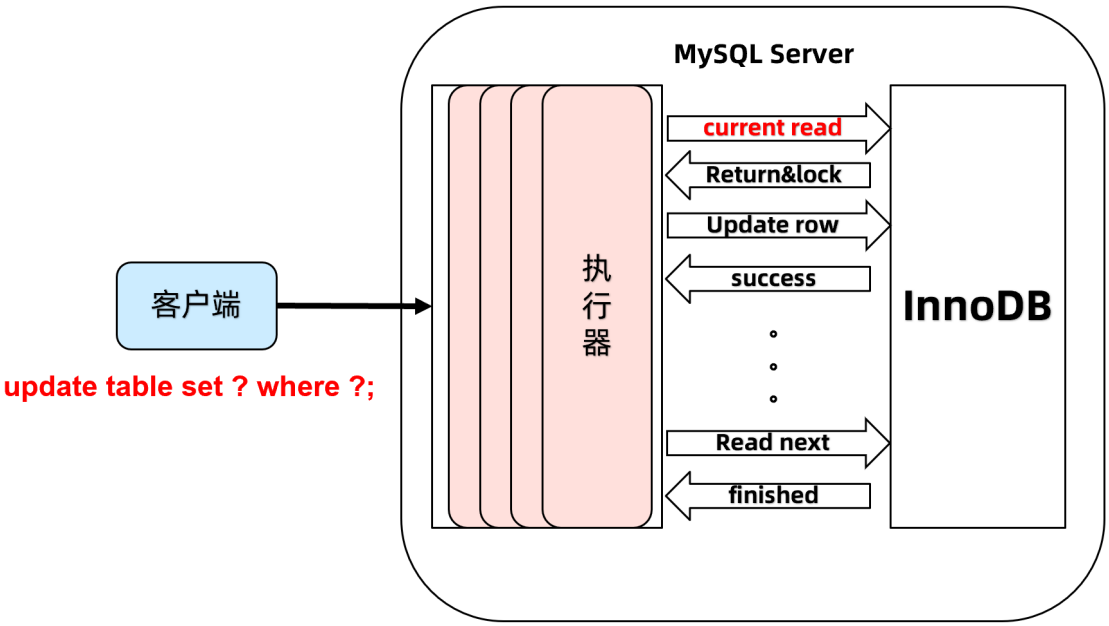
4.4.2 案例：当前读

```
BEGIN;

# SELECT1：Transaction 100、200未提交
SELECT * FROM tab_user WHERE id = 1; # 得到的列name的值为'刘备'

# SELECT2：Transaction 100提交，Transaction 200未提交
SELECT * FROM tab_user WHERE id = 1; # 得到的列name的值为'张飞'

select * from tab_user where id=1 lock in share mode; # 当前读

COMMIT;
```

注意：本案例是在之前的案例基础上进行的

4.4.3 一个CRUD的CUD操作的具体流程

```
update table set ? where ?;
```



从图中可以看到：

当Update SQL被发给MySQL后，

- 首先，**MySQL Server会根据where条件，读取第一条满足条件的记录，然后InnoDB引擎会将第一条记录返回，并加锁 (current read)。**
- 待MySQL Server收到这条加锁的记录之后，会再发起一个Update请求，更新这条记录。
- 一条记录操作完成，再读取下一条记录，直至没有满足条件的记录为止。因此，Update操作内部，就包含了一个当前读。

同理，Delete操作也一样。Insert操作会稍微有些不同，简单来说，就是Insert操作可能会触发Unique Key的冲突检查，也会进行一个当前读。

注：根据上图的交互，针对一条当前读的SQL语句，InnoDB与MySQL Server的交互，是一条一条进行的，因此，加锁也是一条一条进行的。先对一条满足条件的记录加锁，返回给MySQL Server，做一些DML操作；然后在读取下一条加锁，直至读取完毕。

4.5 小结

- MVCC指在使用RC、RR隔离级别下，使不同事务的`读-写`、`写-读`操作并发执行，提升系统性能
- MVCC核心思想是**读不加锁，读写不冲突。**
- RC、RR这两个隔离级别的一个很大不同就是生成`ReadView`的时机不同
  - RC在每一次进行普通`SELECT`操作前都会生成一个`ReadView`
  - RR在第一次进行普通`SELECT`操作前生成一个`ReadView`，之后的查询操作都重复这个`ReadView`