MYSQL MVCC 多版本并发控制底层原理及实现机制

作者：liqitian3344
原文链接：https://ld246.com/article/1595557001619
来源网站：链滴

什么是MVCC

Multiversion Concurrency Control:多版本并发控制，提供并发访问数据库时，对事务内读取的到内存做处理，用来避免写操作堵塞读操作的并发问题（提高并发读写性能）。

MVCC主要适用于Mysql的RC(读已提交),RR(可重复读)隔离级别

问题（痛点）

A正在读数据库中某些内容，而B正在给这些内容做修改（A,B为两个单独的事务），A可能看到一个一致的数据，在B没有提交前，如何让A能够一直读到的数据都是一致的或读到的数据一直是最新的呢?

每个用户连接数据库时，看到的都是某一特定时刻的数据库快照，在B的事务没有提交之前，A始终读的是某一特定时刻的数据库快照，不会读到B事务中的数据修改情况，直到B事务提交，才会读取B的改内容。

一个支持MVCC的数据库，在更新某些数据时，并非使用新数据覆盖旧数据，而是标记旧数据是过时，同时在其他地方新增一个数据版本。因此，同一份数据有多个版本存储，但只有一个是最新的。

MVCC提供了 时间一致性的 处理思路，在MVCC下读事务时，通常使用一个时间戳或者事务ID来确访问哪个状态的数据库及哪些版本的数据。读事务跟写事务彼此是隔离开来的，彼此之间不会影响。设同一份数据，既有读事务访问，又有写事务操作，实际上，写事务会新建一个新的数据版本，而读务访问的是旧的数据版本，直到写事务提交，读事务才会访问到这个新的数据版本。

InnoDB的MVCC实现机制

- <u>每条记录都会保存两个隐藏列，事务id(trx_id)和回滚指针(roll_point)。
- 每次操作都会生成一条undo log日志，回滚指针指向前一条记录。 </u>

版本链

`roll_pointer`每次对记录修改的时候，都会把老版本写入undo log中。新纪录中`roll_pointer`就是了一个指针，它指向这条记录的上一个版本的位置，通过它来获得上一个版本的记录信息。

测试表结构如下:

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `c` varchar(11) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `n_index` (`c`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_
eneral_ci;
```

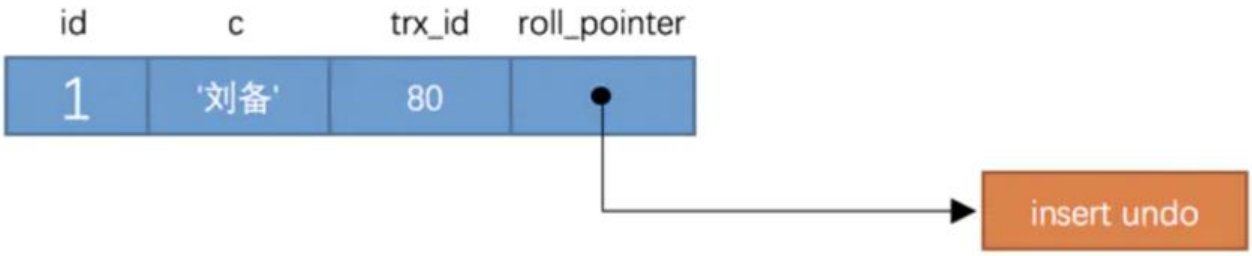
假设现在有个事务(事务id:80)进行了一个insert操作如下:

INSERT INTO t(id, c) VALUES (1, '刘备');

那么此刻的真实数据如下面这个样子：

id	c	trx_id	roll_point
1	刘备	80	上一个版本的址

示意图

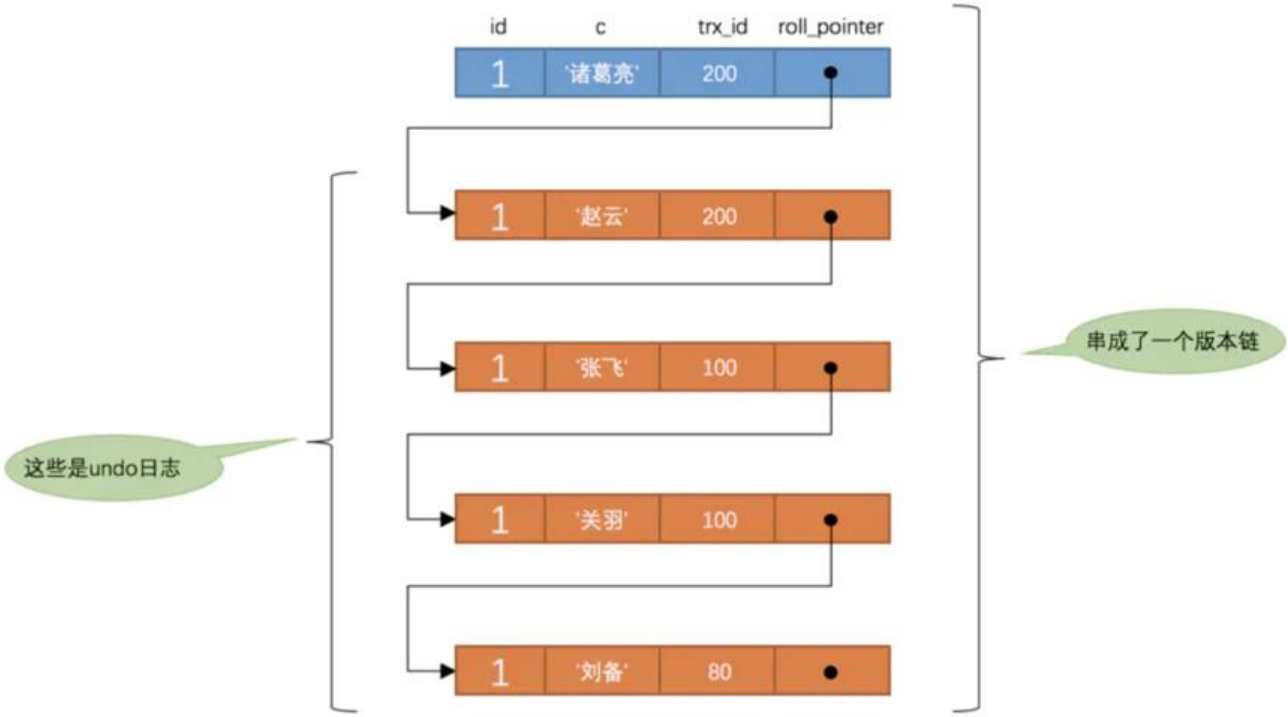


INSERT操作对应的undo日志没有该属性，因为该记录并没有更早的版本。

假设之后两个事务id分别为100、200的事务对这条记录进行UPDATE操作，操作流程如下：

发生时间编号	trx 100	trx 200
①	BEGIN;	
②		BEGIN;
③	UPDATE t SET c = '关羽' WHERE id = 1;	
④	UPDATE t SET c = '张飞' WHERE id = 1;	
	COMMIT;	
⑤		UPDATE t SET c = '赵云' WHERE id = 1;
⑥		UPDATE t SET c = '诸葛亮' WHERE id = 1;
		COMMIT;

每次对记录进行改动，都会记录一条undo日志，每条undo日志也都有一个roll_pointer属性，可以这些undo日志都连起来，串成一个链表，所以现在的情况就像下图一样：



对该记录每次更新后，都会将旧值放到一条undo日志中，就算是该记录的一个旧版本，随着更新次的增多，所有的版本都会被roll_pointer属性连接成一个链表，我们把这个链表称之为版本链，版本的头节点就是当前记录最新的值。另外，每个版本中还包含生成该版本时对应的事务id，这个信息很要，我们稍后就会用到。

ReadView

对于使用READ UNCOMMITTED隔离级别的事务来说，直接读取记录的最新版本就好了，对于使用SRIALIZABLE隔离级别的事务来说，使用加锁的方式来访问记录。对于使用READ COMMITTED和REPATABLE READ隔离级别的事务来说，就需要用到我们上边所说的版本链了，核心问题就是：需要判断一下版本链中的哪个版本是当前事务可见的。所以后面提了一个ReadView的概念，这个ReadView中主要包含当前系统中还有哪活跃的（未提交的）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和示生成该ReadView的快照读操作产生的事务id(creator_trx_id)。这样在访问某条记录时，需要按照下边的步骤判断记录的某个版本是否可见：

● 如果被访问版本的trx_id属性值与ReadView中的creator_trx_id值相同，意味着当前事务在访问它己修改过的记录，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Re

dView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rx_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，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，该版本可以被问。

如果某个版本的数据对当前事务不可见的话，那就顺着版本链找到下一个版本的数据，继续按照上边步骤判断可见性，依此类推，直到版本链中的最后一个版本，如果最后一个版本也不可见的话，那么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。

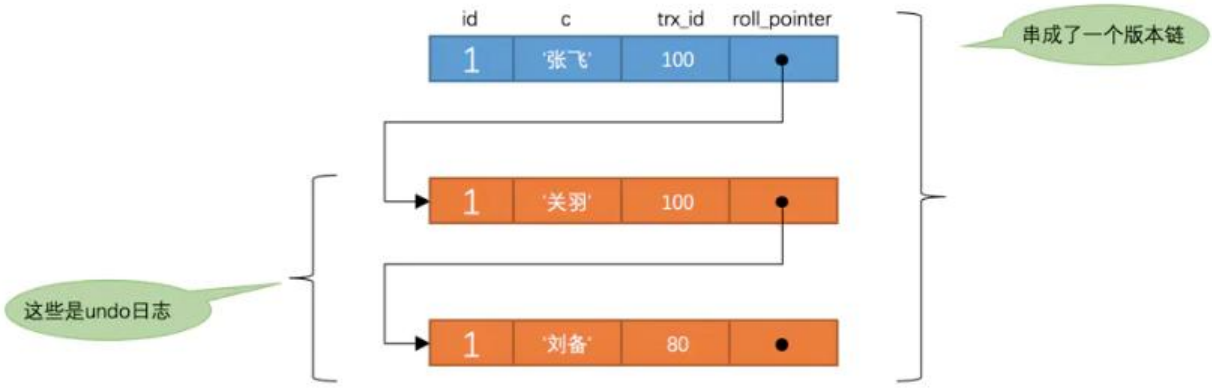
在MySQL中，READ COMMITTED和REPEATABLE READ隔离级别的的一个非常大的区别就是它们成ReadView的时机不同。

READ COMMITTED --- 每次读取数据前都生成一个ReadView

比方说现在系统里有两个id分别为100、200的事务在执行：

```
# Transaction 100
BEGIN;

UPDATE t SET c = '关羽' WHERE id = 1;

UPDATE t SET c = '张飞' WHERE id = 1;

# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...
```

此刻，表t中id为1的记录得到的版本链表如下所示：



假设现在有一个使用READ COMMITTED隔离级别的事务开始执行：

```
# 使用READ COMMITTED隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'
```

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_d值为100，在m_ids列表内，所以不符合可见性要求，根据roll_pointer跳到下一个版本。
- 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，所以也不符合求，继续跳到下一个版本。
- 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id100，以这个版本是符合要求的，最后返回给用户的版本就是这条列c为'刘备'的记录。

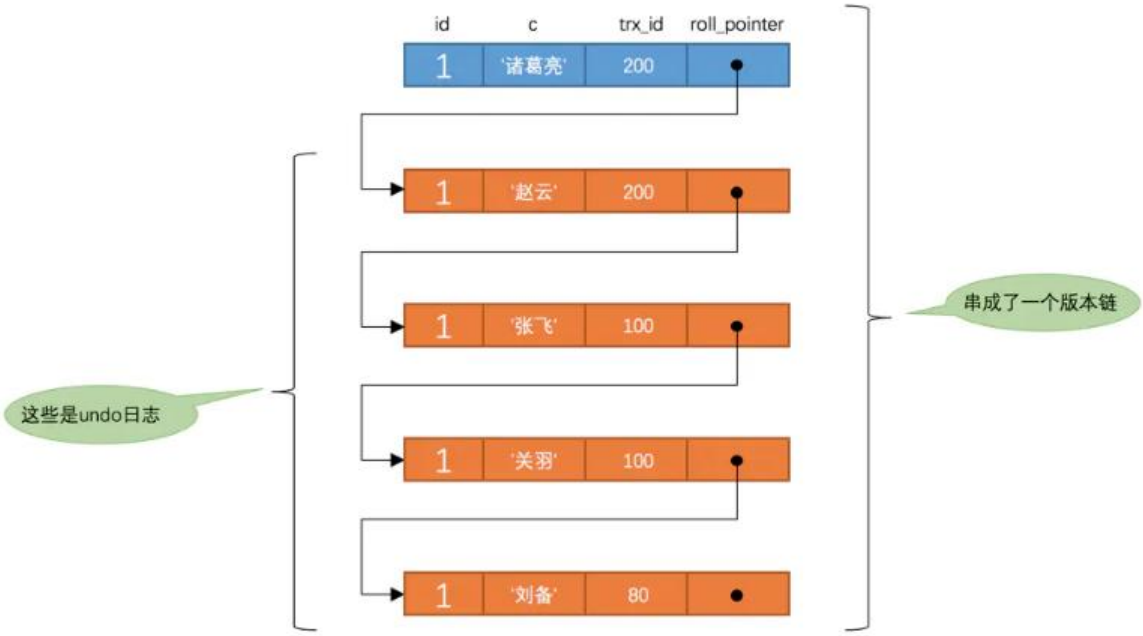
之后，我们把事务id为100的事务提交一下，就像这样：

```
# Transaction 100
BEGIN;

UPDATE t SET c = '关羽' WHERE id = 1;

UPDATE t SET c = '张飞' WHERE id = 1;

COMMIT;
```

然后再到事务id为200的事务中更新一下表t中id为1的记录：

```
# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...

UPDATE t SET c = '赵云' WHERE id = 1;

UPDATE t SET c = '诸葛亮' WHERE id = 1;
```

此刻，表t中id为1的记录的版本链就长这样：



然后再到刚才使用READ COMMITTED隔离级别的事务中继续查找这个id为1的记录，如下：

```
# 使用READ COMMITTED隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200均未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'

# SELECT2：Transaction 100提交，Transaction 200未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'张飞'
```

这个SELECT2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]。
- 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，所以不符合可见性要求，根据roll_pointer跳到下一个版本。
- 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，所以也不符合要求，继续跳到下一个版本。
- 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id200还小，所以这个版本是符合要求的，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READ COMMITTED隔离级别的事务查询表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。总结一下就是：使用READ COMMITTED隔离级别的事务在每次查询开始时都会生成个独立的ReadView。

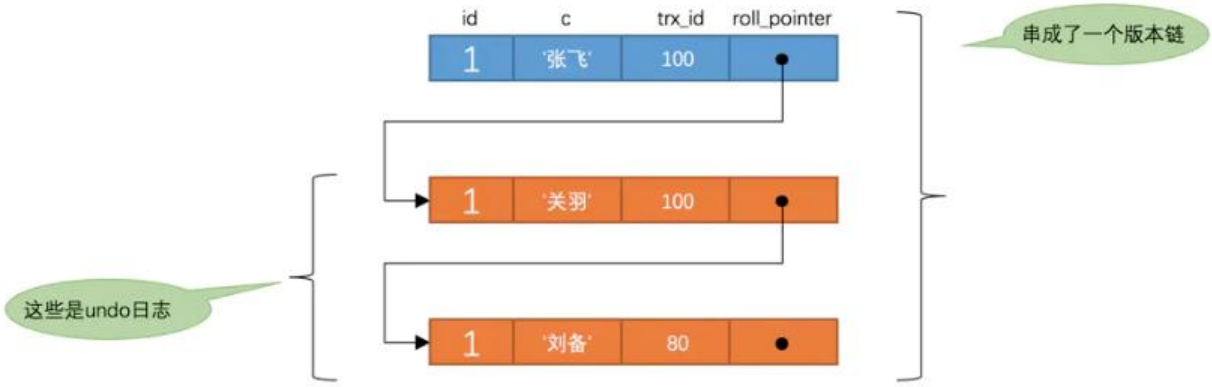
REPEATABLE READ ---在第一次读取数据时生成一个ReadView

对于使用REPEATABLE READ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。

比如现在系统里有两个id分别为100、200的事务在执行：

```
# Transaction 100
BEGIN;

UPDATE t SET c = '关羽' WHERE id = 1;

UPDATE t SET c = '张飞' WHERE id = 1;

# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...
```

此刻，表t中id为1的记录得到的版本链表如下所示：



假设现在有一个使用REPEATABLE READ隔离级别的事务开始执行：

```
# 使用REPEATABLE READ隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'
```

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，所以不符合可见性要求，根据roll_pointer跳到下一个版本。
- 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，继续跳到下一个版本。
- 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id100，以这个版本是符合要求的，最后返回给用户的版本就是这条列c为'刘备'的记录。

之后，我们把事务id为100的事务提交一下，就像这样：

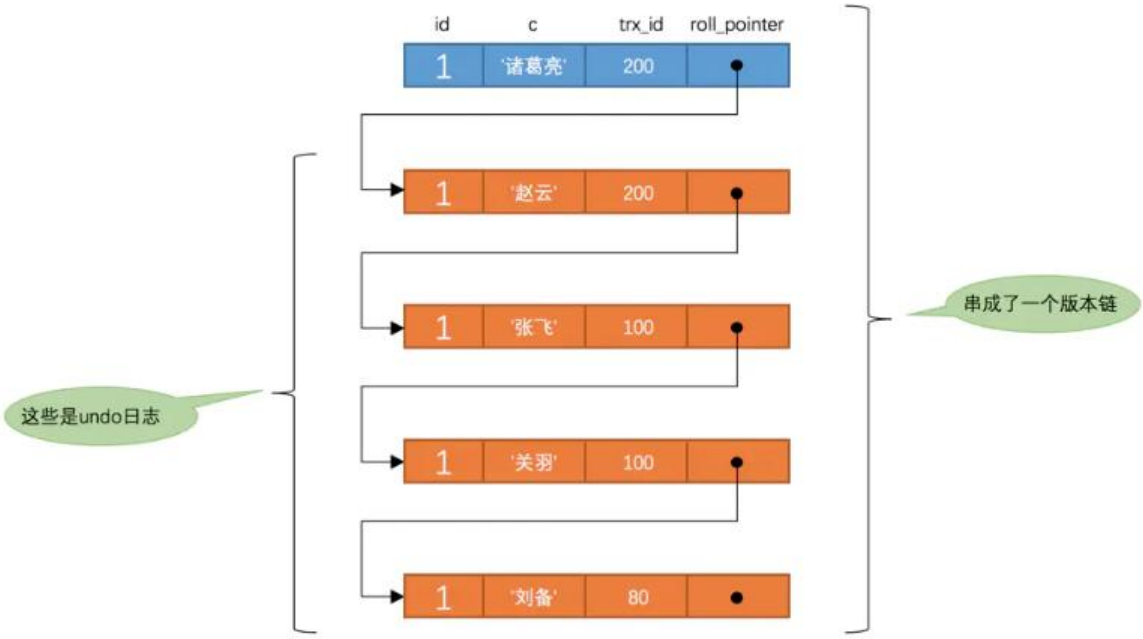
```
# Transaction 100
BEGIN;

UPDATE t SET c = '关羽' WHERE id = 1;

UPDATE t SET c = '张飞' WHERE id = 1;
```

```
COMMIT;
```

然后再到事务id为200的事务中更新一下表t中id为1的记录：

```
# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...

UPDATE t SET c = '赵云' WHERE id = 1;

UPDATE t SET c = '诸葛亮' WHERE id = 1;
```

此刻，表t中id为1的记录的版本链就长这样：



然后再到刚才使用REPEATABLE READ隔离级别的事务中继续查找这个id为1的记录，如下：

```
# 使用REPEATABLE READ隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200均未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'

# SELECT2：Transaction 100提交，Transaction 200未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值仍为'刘备'
```

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_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，所以不符合可见性要求，根据roll_pointer跳到下一个版本。
- 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，所以也不符合要

，继续跳到下一个版本。

- 下一个版本的列 c的内容是'张飞'，该版本的trx_id值为100，而m_ids列表中是包含值为100的事务d的，所以该版本也不符合要求，同理下一个列c的内容是'关羽'的版本也不符合要求。继续跳到下一版本。

- 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id100所以这个版本是符合要求的，最后返回给用户的版本就是这条列c为'刘备'的记录。

MVCC总结

MVCC（Multi-Version Concurrency Control，多版本并发控制）指的就是在使用`READ COMMITD`、`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。`READ COMMITTD、`REPEATABLE READ`这两个隔离级别的一个很大不同就是生成`ReadView`的时机不同，`READ CMMITTD`在每一次进行普通`SELECT`操作前都会生成一个`ReadView`，而`REPEATABLE READ`只在一次进行普通`SELECT`操作前生成一个`ReadView`，之后的查询操作都重复这个`ReadView`就好了。

掘金原文链接