第16章_多版本并发控制

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

1. 什么是MVCC

MVCC（Multiversion Concurrency Control），多版本并发控制。顾名思义，MVCC是通过数据行的多个版本管理来实现数据库的`并发控制`。这项技术使得在InnoDB的事务隔离级别下执行`一致性读`操作有了保证。换言之，就是为了查询一些正在被另一个事务更新的行，并且可以看到它们被更新之前的值，这样在做查询的时候就不用等待另一个事务释放锁。

MVCC没有正式的标准，在不同的DBMS中MVCC的实现方式可能是不同的，也不是普遍使用的(大家可以参考相关的DBMS文档)。这里讲解InnoDB中MVCC的实现机制（MySQL其它的存储引擎并不支持它)

2. 快照读与当前读

MVCC在MySQL InnoDB中的实现主要是为了`提高数据库并发性能`，用更好的方式去处理`读-写冲突`，做到即使有读写冲突时，也能做到`不加锁`，`非阻塞并发读`，而这个读指的就是`快照读`，而非`当前读`。当前读实际上是一种加锁的操作，是悲观锁的实现。而MVCC本质是采用`乐观锁`思想的一种方式。

2.1 快照读

快照读又叫一致性读，读取的是快照数据。**不加锁的简单的SELECT都属于快照读**，即不加锁的非阻塞读；比如这样：

```
SELECT * FROM player WHERE ...
```

之所以出现快照读的情况，是基于提高并发性能的考虑，快照读的实现是基于MVCC，它在很多情况下，避免了加锁操作，降低了开销。

既然是基于`多版本`，那么快照读**可能读到的并不一定是数据的最新版本**，而有可能是之前的历史版本。

快照读的前提是隔离级别不是串行级别，**串行级别下的快照读会退化成当前读。**

2.2 当前读

当前读读取的是记录的最新版本（最新数据，而不是历史版本的数据），读取时还要保证其他并发事务不能修改当前记录，会对读取的记录进行加锁。加锁的SELECT，或者对数据进行增删改都会进行当前读。比如：

```
SELECT * FROM student LOCK IN SHARE MODE; # 共享锁
```

```
SELECT * FROM student FOR UPDATE; # 排他锁
```

```
INSERT INTO student values ... # 排他锁
```

```
DELETE FROM student WHERE ... # 排他锁
```

```
UPDATE student SET ... # 排他锁
```

> 注意：InnoDB增删改默认加X锁，查默认不加锁

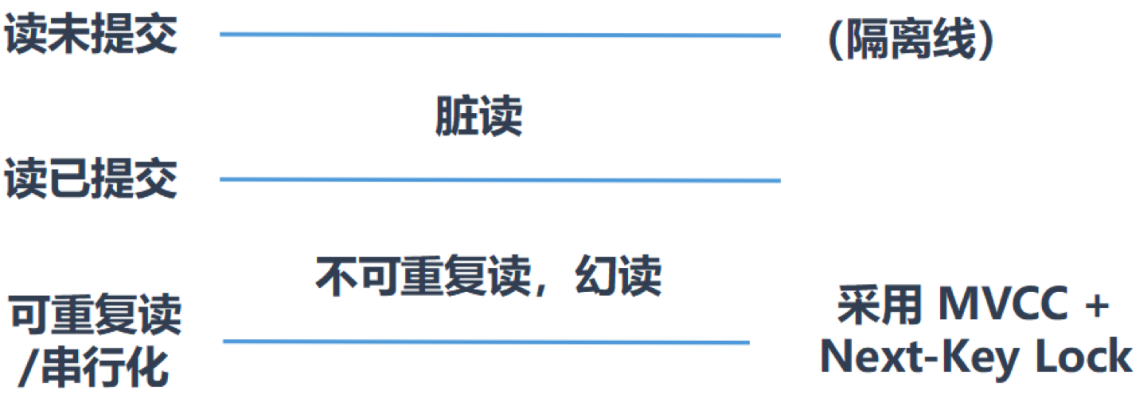
3. 复习

3.1 再谈隔离级别

我们知道事务有4个隔离级别，可能存在三种并发问题（准确来说是四种，还有一种：`脏写`）：

在MySQL中，默认的隔离级别是`可重复读`，可以解决**脏读和不可重复读**的问题，如果仅从定义的角度来看，它并不能解决幻读问题。如果想要解决幻读问题，就需要采用串行化的方式，也就是将隔离级别提升到最高，但这样一来就会大幅降低数据库的事务并发能力

MVCC可以不采用锁机制，而是通过乐观锁的方式来解决不可重复读和幻读问题！它可以在大多数情况下替代行级锁，降低系统的开销。



> 在面试的时候要按照第二幅图进行回答
>
> 如果采用加锁的方式，那么就是间隙锁解决幻读问题。

3.2 隐藏字段、Undo Log版本链

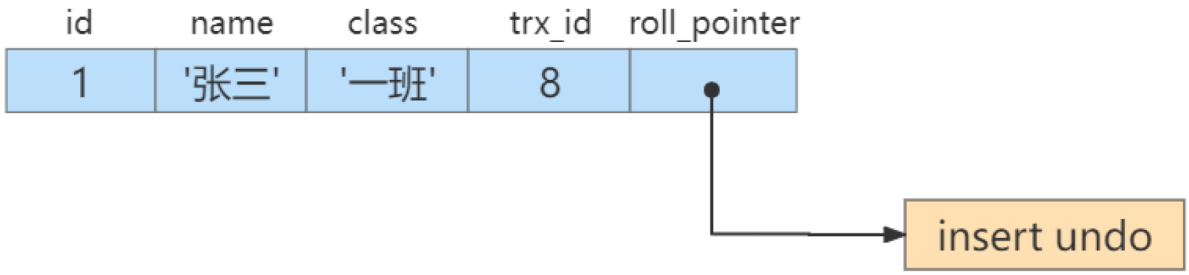
回顾一下undo日志的版本链，对于使用`InnoDB`存储引擎的表来说，它的聚簇索引记录中都包含两个必要的隐藏列。

- `trx_id`：每次一个事务对某条聚簇索引记录进行改动时，都会把该事务的`事务id`赋值给`trx_id`隐藏列。
- `roll_pointer`：每次对某条聚簇索引记录进行改动时，都会把旧的版本写入到`undo日志`中，然后这个隐藏列就相当于一个指针，可以通过它来找到该记录修改前的信息。

举例：student表数据如下

```
SELECT * FROM student ;
/*
+----+--------+--------+
| id | name   | class  |
+----+--------+--------+
|  1 | 张三   | 一班   |
+----+--------+--------+
1 row in set (0.07 sec)
*/
```

假设插入该记录的`事务id`为8，那么此刻该条记录的示意图如下所示:



> insert undo只在事务`回滚`时起作用，**当事务提交后，该类型的undo日志就没用了**，它占用的Undo Log Segment也会被系统回收（也就是该undo日志占用的Undo页面链表要么被重用，要么被释放）。

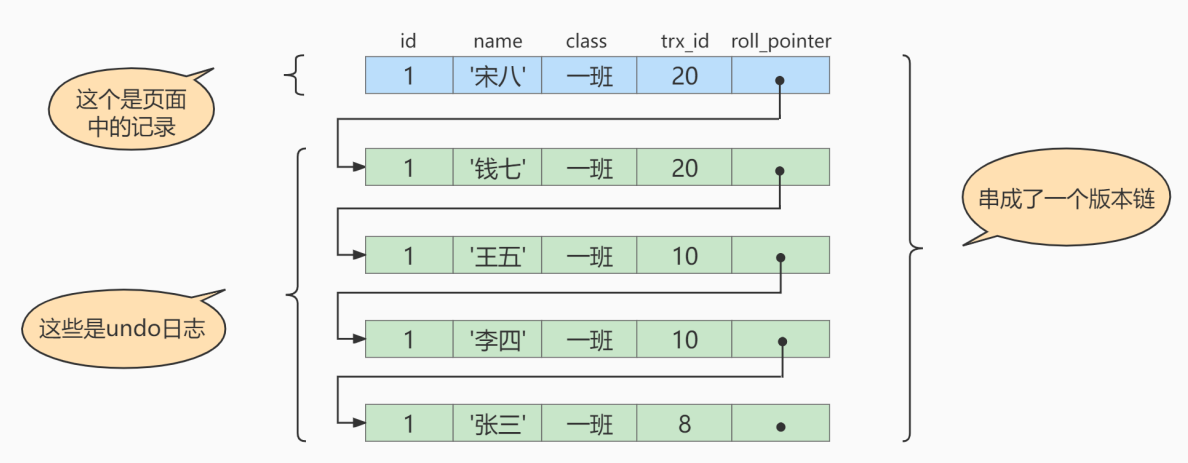
假设之后两个事务id分别为`10`、`20`的事务对这条记录进行`UPDATE`操作，操作流程如下：

发生时间顺序	事务10	事务20
1	BEGIN;	
2		BEGIN;
3	UPDATE student SET name="李四" WHERE id=1;	
4	UPDATE student SET name="王五" WHERE id=1;	
5	COMMIT;	
6		UPDATE student SET name="钱七" WHERE id=1;

发生时间顺序	事务10	事务20
7		UPDATE student SET name="宋八" WHERE id=1;
8		COMMIT;

> 能不能在两个事务中交叉更新同一条记录呢?
>
> 不能！这不就是一个事务修改了另一个未提交事务修改过的数据，脏写。
>
> **InnoDB使用锁来保证不会有脏写情况的发生**，也就是在第一个事务更新了某条记录后，就会给这条记录加锁，另一个事务再次更新时就需要等待第一个事务提交了，把锁释放之后才可以继续更新。
>
> **InnoDB增删改默认加x锁，查默认不加锁**

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（`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），可以将这些`undo日志`都连起来，串成一个链表：



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，随着更新次数的增多，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`，版本链的头节点就是当前记录最新的值。

每个版本中还包含生成该版本时对应的`事务id`。

4. MVCC实现原理之ReadView

MVCC的实现依赖于：**隐藏字段、Undo Log、Read View。**

4.1 什么是ReadView

在MVCC机制中，**多个事务对同一个行记录进行更新会产生多个历史快照**，这些历史快照保存在Undo Log里。如果一个事务想要查询这个行记录，需要读取哪个版本的行记录呢？这时就需要用到`ReadView`了，它解决了**行的可见性**问题

ReadView就是事务A在使用MVCC机制进行快照读操作时产生的`读视图`。当事务启动时，会生成数据库系统当前的一个快照，InnoDB为每个事务构造了一个数组，用来记录并维护系统当前`活跃事务`的ID（“活跃"指的就是，**启动了但还没提交**）

> ReadView和事务是一对一的关系~ 也就是当事务中使用MVVC，且是Select时会生成一个ReadView~

4.2 设计思路

使用`READ UNCOMMITTED`隔离级别的事务，由于可以读到未提交事务修改过的记录，所以直接读取记录的最新版本就好了。

使用`SERIALIZABLE`隔离级别的事务，InnoDB规定使用加锁的方式来访问记录。

使用`READ COMMITTED`和`REPEATABLE READ`隔离级别的事务，都必须保证读到`已经提交了的`事务修改过的记录。假如另一个事务已经修改了记录但是尚未提交，是不能直接读取最新版本的记录的，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，这是ReadView要解决的主要问题。

这个ReadView中主要包含4个比较重要的内容，分别如下：

- `creator_trx_id`，创建这个ReadView的事务ID。

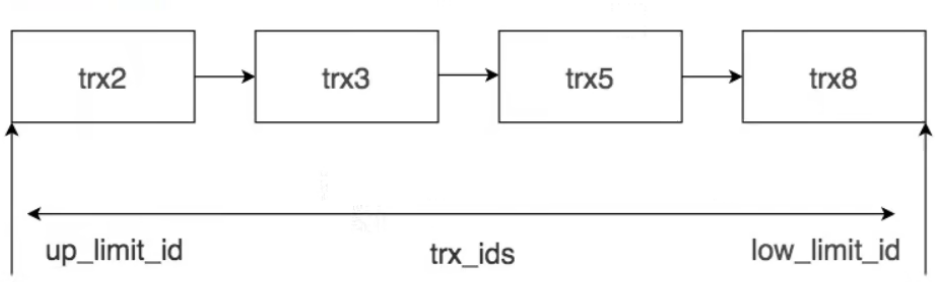
  > 说明：只有在对表中的记录做改动时（执行INSERT、DELETE、UPDATE这些语句时）才会为事务分配事务id，否则在一个只读事务中的事务id值都默认为0。

- `trx_ids`，表示在生成ReadView时当前系统中活跃的读写事务的`事务id列表`。
- `up_limit_id`，活跃的事务中最小的事务ID。
- `low_limit_id`，表示生成ReadView时系统中应该分配给下一个事务的`id`值。low_limit_id是系统最大的事务id值，这里要注意是系统中的事务id，需要区别于正在活跃的事务ID。

  > 注意：low_limit_id并不是trx_ids中的最大值，事务id是递增分配的。比如，现在有id为1，2，3这三个事务，之后id为3的事务提交了。那么一个新的读事务在生成ReadView时，trx_ids就包括1和2，up_limit_id的值就是1，low_limit_id的值就是4。

举例：

trx_ids为trx2、trx3、trx5和trx8的集合，系统的最大事务ID(low_limit_id)为trx8+1(如果之前没有其他的新增事务)，活跃的最小事务ID(up_limit_id)为trx2。



4.3 ReadView的规则

有了这个ReadView，这样在访问某条记录时，只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**当前事务在访问它自己修改过的记录**，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ReadView中的`up_limit_id`值，表明**生成该版本的事务在当前事务生成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属性值大于或等于ReadView中的`low_limit_id`值，表明**生成该版本的事务在当前事务生成ReadView后才开启**，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ReadView的`up_limit_id`和`low_limit_id`之间，那就需要判断一下trx_id属性值是不是在`trx_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。

4.4 MVCC整体操作流程

了解了这些概念之后，我们来看下当查询一条记录的时候，系统如何通过MVCC找到它：

1. 首先获取事务自己的版本号，也就是事务ID；
2. 获取ReadView；
3. 查询得到的数据，然后与ReadView中的事务版本号进行比较；
4. 如果不符合ReadView规则，就需要从`Undo Log`中获取历史快照；
5. 最后返回符合规则的数据。

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，继续按照上边的步骤判断可见性，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，那么就意味着该条记录对该事务完全不可见，查询结果就不包含该记录。

> InnoDB中，MVCC是通过`Undo Log + Read View`进行数据读取，Undo Log保存了历史快照，而Read View规则帮我们判断当前版本的数据是否可见。
>
> Read View和事务是一对一对应的，而且Read View也是一个动态，不断变化的~

在隔离级别为`读已提交`（Read Committed）时，一个事务中的每一次SELECT查询都会重新获取一次ReadView。

如表所示：

事务	说明
begin;	
select * from student where id>2;	获取一次Read View
...	
select * from student where id>2;	获取一次ReadView
commit;	

> 注意，此时同样的查询语句都会重新获取一次ReadView，这时如果ReadView不同，就可能产生不可重复读或者幻读的情况。

当隔离级别为可重复读的时候，就避免了不可重复读，这是因为**一个事务只在第一次SELECT的时候会获取一次ReadView**，而后面所有的SELECT都会复用这个ReadView，如下表所示：

事务	说明
begin;	
select * from user where id >2;	获取一次Read View
.....	
select * from user where id >2;	
commit;	

5. 举例说明

假设现在student表中只有一条由`事务id`为`8`的事务插入的一条记录:

```
SELECT * FROM student ;
/*
+----+--------+--------+
| id | name   | class  |
+----+--------+--------+
|  1 | 张三   | 一班    |
+----+--------+--------+
1 row in set (0.07 sec)
*/
```

MVCC只能在`READ COMMITTED`和`REPEATABLE READ`两个隔离级别下工作。接下来看一下READ COMMITTED和REPEATABLE READ所谓的生成ReadView的时机不同到底不同在哪里

> **关于不同隔离级别下Read View的事务id,可以概括如下:**
>
> - 对于RC隔离级别:
>   - 在一个事务中，每次查询会创建id为0的Read View。
>   - 一旦有修改操作,会切换到以当前事务id为creator_trx_id的新Read View。
> - 对于RR隔离级别:
>   - 在一个事务中，只有第一次的查询会创建一个Read View。
>   - 这个Read View的creator_trx_id就是当前事务的id。
>
> RR要求整个事务的查询都要一致，所以只有第一次查询才会生成一个Read View。
>
> 而RC可以在同一事务内读取不同版本的数据，所以每次修改和查询都会生成新的Read View。

5.1 READ COMMITTED隔离级别下

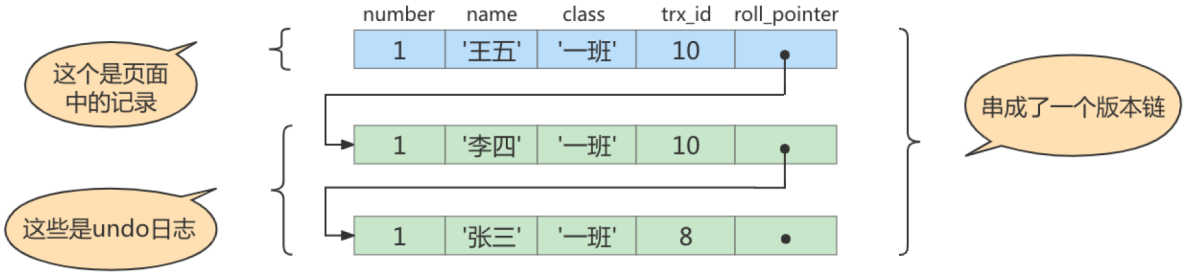
READ COMMITTED：每次读取数据前都生成一个ReadView。

现在有两个`事务id`分别为`10`、`20`的事务在执行:

```
# Transaction 10
BEGIN;
UPDATE student SET name="李四" WHERE id = 1;
UPDATE student SET name="王五" WHERE id = 1;

# Transaction 20
BEGIN;
#更新了一些别的表的记录（为了有一个trx_id，如只查询，trx_id为0）
...
```

> 说明:事务执行过程中，只有在第一次真正修改记录时(比如使用INSERT、DELETE、UPDATE语句)，才会被分配一个单独的事务id，这个事务id是递增的。所以我们才在事务2中更新一些别的表的记录，目的是让它分配事务id。

此刻，表student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：Transaction10、20未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值为'张三'
```

这个 SELECT1 的执行过程如下：

1. 在执行`SELECT`语句时会先生成一个`ReadView`，ReadView的`trx_ids`列表的内容就是`[10, 20]`，`up_limit_id`为`10`，`low_limit_id`为`21`，`creator_trx_id`为`0`。
2. 从版本链中挑选可见的记录，从图中看出，最新版本的列name的内容是'王五'，该版本的`trx_id`值为`10`，在trx_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
3. 下一个版本的列name的内容是'李四'，该版本的trx_id值也为10，也在trx_ids列表内，所以也不符合要求，继续跳到下一个版本
4. 下一个版本的列name的内容是'张三'，该版本的trx_id值为8，小于ReadView中的up_limit_id值10，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name为'张三'的记录。

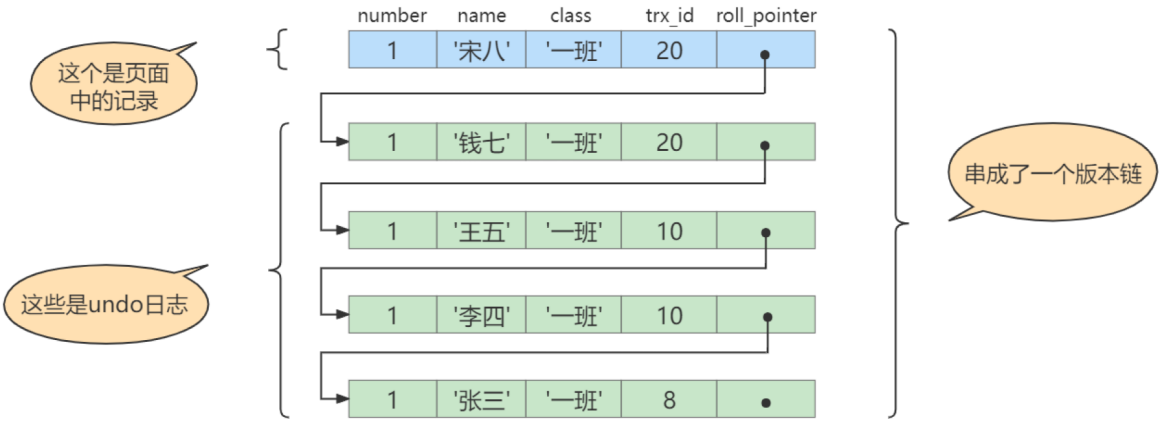
之后，我们把`事务id`为`10`的事务提交一下：

```
# Transaction 10
BEGIN;

UPDATE student SET name = "李四" WHERE id = 1;
UPDATE student SET name = "王五" WHERE id = 1;

COMMIT;
```

然后再到`事务id`为`20`的事务中更新一下表`student`中`id`为`1`的记录：

```
# Transaction 20
BEGIN;

# 更新了一些别的表的记录
...
UPDATE student SET name = "钱七" WHERE id = 1;
UPDATE student SET name = "宋八" WHERE id = 1;
```

此刻，表student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 10、20均未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值为'张三'

# SELECT2：Transaction 10提交，Transaction 20未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值为'王五'
```

这个SELECT2的执行过程如下：

1. 在执行`SELECT`语句时会又会`单独生成`一个`ReadView`，该ReadView的`trx_ids`列表的内容就是`[20]`，`up_limit_id`为`20`，`low_limit_id`为`21`，`creator_trx_id`为`0`。
2. 从版本链中挑选可见的记录，从图中看出，最新版本的列`name`的内容是`'宋八'`，该版本的`trx_id`值为`20`，在trx_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。
3. 下一个版本的列`name`的内容是`'钱七'`，该版本的`trx_id`值为`20`，也在`trx_ids`列表内，所以也不符合要求，继续跳到下一个版本
4. 下一个版本的列`name`的内容是`'王五'`，该版本的`trx_id`值为`10`，小于`ReadView`中的`up_limit_id`值`20`，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name为`'王五'`的记录。

以此类推，如果之后`事务id`为`20`的记录也提交了，再次在使用`READ COMMITED`隔离级别的事务查询表`student`中`id`值为`1`的记录时，得到的结果就是`'宋八'`了，具体流程我们就不分析了。

> **强调：**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

5.2 REPEATABLE READ隔离级别下

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。

比如，系统里有两个`事务id`分别为`10`、`20`的事务在执行：

```
# Transaction 10
BEGIN;
UPDATE student SET name = "李四" WHERE id = 1;
UPDATE student SET name = "王五" WHERE id = 1;

# Transaction 20
BEGIN;
#更新了一些别的表的记录
...
```

此刻，表student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：Transaction10、20未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值为'张三'
```

这个`SELECT1`的执行过程如下：

1. 在执行`SELECT`语句时会先生成一个`ReadView`，`ReadView`的`trx_ids`列表的内容就是`[10, 20]`，`up_limit_id`为`10`，`low_limit_id`为`21`，`creator_trx_id`为`0`。
2. 然后从版本链中挑选可见的记录，从图中看出，最新版本的列`name`的内容是`'王五'`，该版本的`trx_id`值为`10`，在`trx_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。
3. 下一个版本的列`name`的内容是`'李四'`，该版本的trx_id值也为10，也在trx_ids列表内，所以也不符合要求，继续跳到下一个版本。
4. 下一个版本的列name的内容是`'张三'`，该版本的`trx_id`值为`8`，小于`ReadView`中的`up_limit_id`值`10`，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`name`为`'张三'`的记录

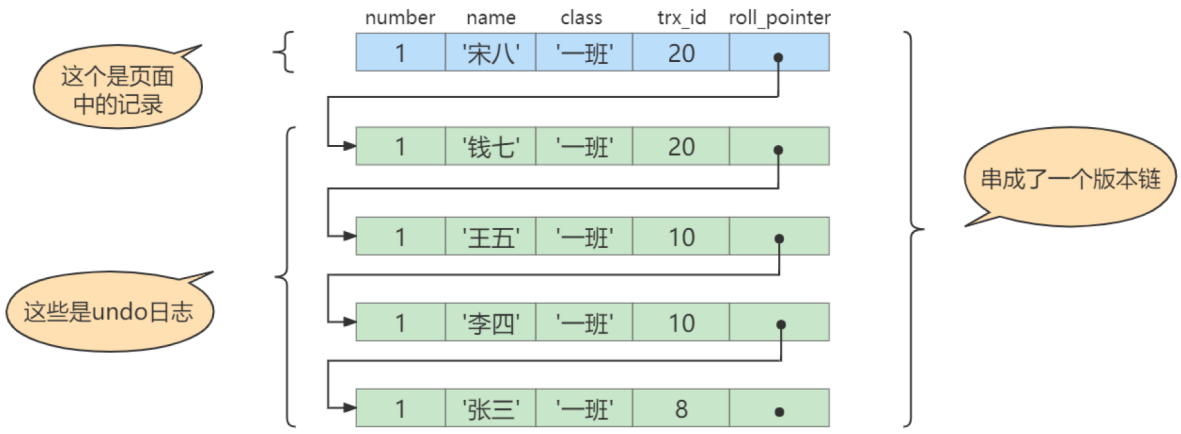
之后，我们把`事务id`为`10`的事务提交一下，就像这样：

```
# Transaction10
BEGIN;

UPDATE student SET name = "李四" WHERE id = 1;
UPDATE student SET name = "王五" WHERE id = 1;

COMMIT;
```

然后再到`事务id`为`20`的事务中更新一下表`student`中`id`为`1`的记录：

```
# Transaction20
BEGIN;

# 更新了一些别的表的记录
...
UPDATE student SET name = "钱七" WHERE id = 1;
UPDATE student SET name = "宋八" WHERE id = 1;
```

此刻，表student中`id`为`1`的记录的版本链长这样：



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 10、20均未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值为'张三'

# SELECT2：Transaction 10提交，Transaction 20未提交
SELECT * FROM student WHERE id = 1; # 得到的列name的值仍为'张三'
```

`SELECT2`的执行过程如下：

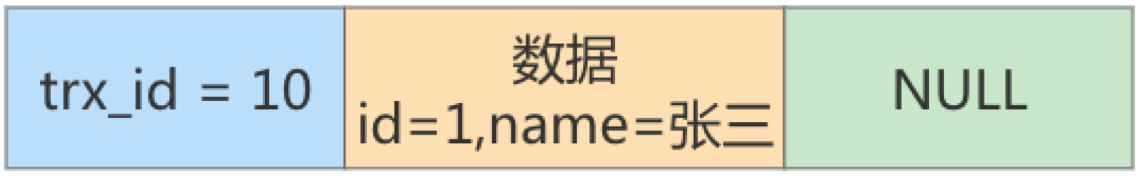
1. 因为当前事务的隔离级别为`REPEATABLE READ`，而之前在执行`SELECT1`时已经生成过`ReadView`了，所以此时直接复用之前的ReadView，之前的`ReadView`的`trx_ids`列表的内容就是`[10，20]`，`up_limit_id`为`10`，`low_limit_id`为`21`，`creator_trx_id`为`0`。
2. 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列`name`的内容是`'宋八'`，该版本的`trx_id`值为`20`，在`trx_ids`列表内，所以不符合可见性要求，根据`roll_pointer`跳到下一个版本
3. 下一个版本的列`name`的内容是`'钱七'`，该版本的`trx_id`值为`20`，也在`trx_ids`列表内，所以也不符合要求，继续跳到下一个版本
4. 下一个版本的列`name`的内容是`'王五'`，该版本的`trx_id`值为`10`，而trx_ids列表中是包含值为10的事务id的，所以该版本也不符合要求，同理下一个列`name`的内容是`'李四'`的版本也不符合要求。继续跳到下一个版本
5. 下一个版本的列`name`的内容是`'张三'`，该版本的`trx_id`值为`8`，小于`ReadView`中的`up_limit_id`值10，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`'张三'`的记录。

两次 SELECT 查询得到的结果是重复的，记录的列 c 值都是‘张三’，这就是 可重复读 的含义。如果我们之后再把 事务id 为 20 的记录提交了，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 的记得到的结果还是‘张三’，具体执行过程大家可以自己分析一下。

5.3 如何解决幻读

接下来说明InnoDB是如何解决幻读的。

假设现在表student中只有一条数据，数据内容中，主键 id=1，隐藏的 trx_id=10，它的undo log如下图所示。



假设现在有事务A和事务B并发执行，事务A 的事务id为 20，事务B 的事务id为 30。

步骤1：事务A开始第一次查询数据，查询的SQL语句如下。

```
1 select * from student where id >= 1;
```

在开始查询之前，MySQL会为事务A产生一个ReadView，此时ReadView的内容如下：trx_ids=[20,30]，up_limit_id=20，low_limit_id=31，creator_trx_id=20。

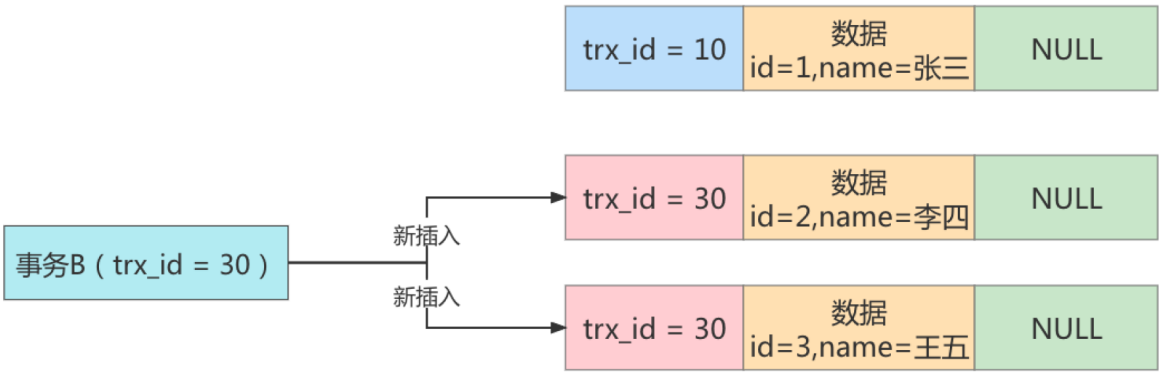
由于此时表student中只有一条数据，且符合whereid>=1条件，因此会查询出来。然后根据ReadView机制，发现该行数据的trx_id=10，小于事务A的ReadView里up_limit_id，这表示这条数据是事务A开启之前，其他事务就已经提交了的数据，因此事务A可以读取到。

结论：事务A的第一次查询，能读取到一条数据，id=1。

步骤2：接着事务B(trx_id=30)，往表student中新插入两条数据，并提交事务。

```
1 insert into student(id,name) values(2,'李四');
2 insert into student(id,name) values(3,'王五');
```

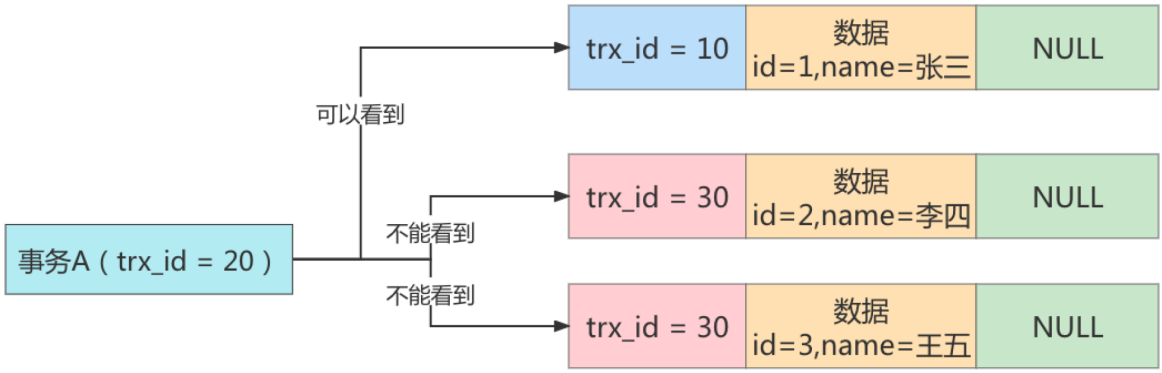
此时表student中就有三条数据了，对应的undo如下图所示：



步骤3：接着事务A开启第二次查询，根据可重复读隔离级别的规则，此时事务A并不会再重新生成ReadView。此时表student中的3条数据都满足whereid>=1的条件，因此会先查出来。然后根据ReadView机制，判断每条数据是不是都可以被事务A看到。

1. 首先id=1的这条数据，前面已经说过了，可以被事务A看到。

2. 然后是id=2的数据，它的trx_id=30，此时事务A发现，这个值处于up_limit_id和low_limit_id之间，因此还需要再判断30是否处于trx_ids数组内。由于事务A的trx_ids=[20,30]，因此在数组内，这表示id=2的这条数据是与事务A在同一时刻启动的其他事务提交的，所以这条数据不能让事务A看到。
3. 同理，id=3的这条数据，trx_id也为30，因此也不能被事务A看见。



结论：最终事务A的第二次查询，只能查询出id=1的这条数据。这和事务A的第一次查询的结果是一样的，因此没有出现幻读现象，所以说在MySQL的可重复读隔离级别下，不存在幻读问题。

6. 总结

这里介绍了`MVCC`在`READ COMMITTD`、`REPEATABLE READ`这两种隔离级别的事务在执行快照读操作时访问记录的版本链的过程。这样使不同事务的`读-写`、`写-读`操作并发执行，从而提升系统性能。

核心点在于ReadView的原理，`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 COMMITTD`在每一次进行普通SELECT操作前都会生成一个ReadView。
- `REPEATABLE READ`只在第一次进行普通SELECT操作前生成一个ReadView，之后的查询操作都重复使用这个ReadView就好了。

> 说明：之前说执行DELETE语句或者更新主键的UPDATE语句并不会立即把对应的记录完全从页面中删除，而是执行一个所谓的`delete mark`操作(标记0->1)，相当于只是对记录打上了一个删除标志位，这主要就是为MVCC服务的。另外后面回滚也可能用到这个delete mark~

通过MVCC可以解决:

- `读写之间阻塞的问题`。通过MVCC可以让读写互相不阻塞，即读不阻塞写，写不阻塞读，这样就可以提升事务并发处理能力。
- `降低了死锁的概率`。这是因为MVCC采用了乐观锁的方式，读取数据时并不需要加锁，对于写操作，也只锁定必要的行。
- `解决快照读的问题`。当查询数据库在某个时间点的快照时，只能看到这个时间点之前事务提交更新的结果，而不能看到这个时间点之后事务提交的更新结果。

> MVCC = 两个隐藏列 + Undo Log版本链 + ReadView