事务隔离级别，MVCC

事务四大特性

- **原子性（Atomicity）**
- **一致性（Consistency）**
- **隔离性（Isolation）**
- **持久性（Durability）**

悲观并发控制

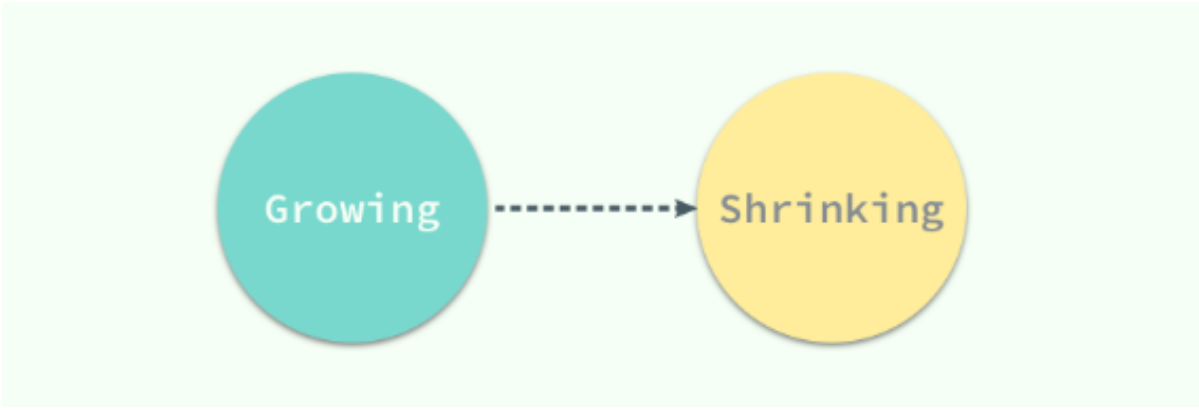
- 类似于悲观锁，在数据处理中处于独占锁状态

读写锁

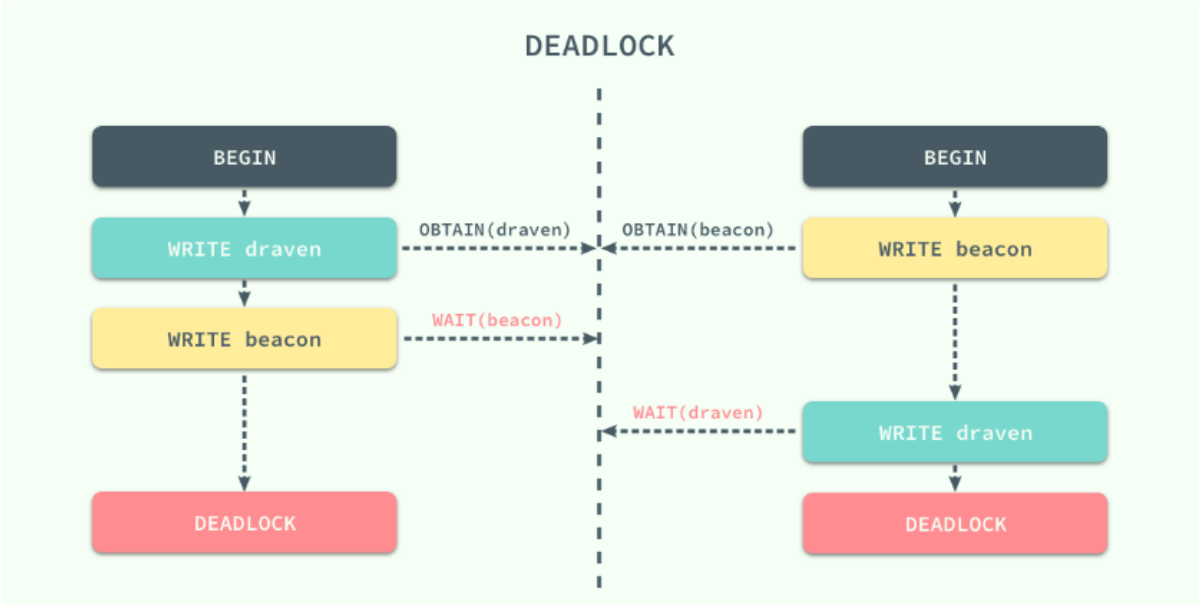
- 为了最大化数据库事务的并发能力，数据库中的锁被设计为两种模式，分别是共享锁和互斥锁。当一个事务获得共享锁之后，它只可以进行读操作，所以共享锁也叫读锁；而当一个事务获得一行数据的互斥锁时，就可以对该行数据进行读和写操作，所以互斥锁也叫写锁。

两阶段锁协议（2PL）

- **2PL**是一种能够保证事务可串行化的协议，它将事务的获取锁和释放锁划分成了增长（Growing）和缩减（Shrinking）两个不同的阶段。在增长阶段，一个事务可以获得锁但是不能释放锁；而在缩减阶段事务只可以释放锁，并不能获得新的锁，如果只看 2PL 的定义，那么到这里就已经介绍完了，但是它还有两个变种：
    1. **Strict 2PL**：事务持有的**互斥**锁必须在提交后再释放；
    2. **Rigorous 2PL**：事务持有的**所有**锁必须在提交后释放；



- **2PL**引入了另一个更严重的问题：**死锁**；不同的事务等待对方已经锁定的资源会造成死锁



- 预防死锁

事务并发执行遇到的问题

- **脏写（Dirty Write）如果一个事务修改了另一个未提交事务修改过的数据，则发生了脏写**

脏写示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②		BEGIN;
③		UPDATE hero SET name ='关羽' WHERE number = 1;
④	UPDATE hero SET name ='张飞' WHERE number = 1;	
⑤	COMMIT;	
⑥		ROLLBACK;

  - 如上图，`Session A`和`Session B`各开启了一个事务，`Session B`中的事务先将`number`列为`1`的记录的`name`列更新为`'关羽'`，然后`Session A`中的事务接着又把这条`number`列为`1`的记录的`name`列更新为`张飞`。如果之后`Session B`中的事务进行了回滚，那么`Session A`中的更新也将不复存在，这种现象就称之为`脏写`。
- **脏读（Dirty Read）如果一个事务读到了另一个未提交事务修改过的数据，那就意味着发生了脏读**

脏读示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②		BEGIN;
③		UPDATE hero SET name ='关羽' WHERE number = 1;
④	SELECT * FROM hero WHERE number = 1; (如果读到列name的值为'关羽'，则意味着发生了脏读)	
⑤	COMMIT;	
⑥		ROLLBACK;

  - 如上图，`Session A`和`Session B`各开启了一个事务，`Session B`中的事务先将`number`列为`1`的记录的`name`列更新为`'关羽'`，然后`Session A`中的事务再去查询这条`number`为`1`的记录，如果du到列`name`的值为`'关羽'`，而`Session B`中的事务稍后进行了回滚，那么`Session A`中的事务相当于读到了一个不存在的数据，这种现象就称之为`脏读`。
- **不可重复读（Non-Repeatable Read）如果一个事务只能读到另一个已经提交的事务修改过的数据，并且其他事务每对该数据进行一次修改并提交后，该事务都能查询得到最新值，那就意味着发生了不可重复读**

不可重复读示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②	SELECT * FROM hero WHERE number = 1; (此时读到的列name的值为'刘备')	
③		UPDATE hero SET name ='关羽' WHERE number = 1;
④	SELECT * FROM hero WHERE number = 1; (如果读到列name的值为'关羽'，则意味着发生了不可重复读)	
⑤		UPDATE hero SET name ='张飞' WHERE number = 1;
⑥	SELECT * FROM hero WHERE number = 1; (如果读到列name的值为'张飞'，则意味着发生了不可重复读)	

  - 如上图，我们在`Session B`中提交了几个隐式事务（注意是隐式事务，意味着语句结束事务就提交了），这些事务都修改了`number`列为`1`的记录的列`name`的值，每次事务提交之后，如果`Session A`中的事务都可以查看到最新的值，这种现象也被称之为`不可重复读`。
- **幻读（Phantom）如果一个事务先根据某些条件查询出一些记录，之后另一个事务又向表中插入了符合这些条件的记录，原先的事务再次按照该条件查询时，能把另一个事务插入的记录也读出来，那就意味着发生了幻读**

幻读示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②	SELECT * FROM hero WHERE number > 0; (此时读到的列name的值为'刘备')	
③		INSERT INTO hero VALUES(2, '曹操', '魏');
④	SELECT * FROM hero WHERE number > 0; (如果读到列name的值为'刘备'、'曹操'的记录，则意味着发生了幻读)	

  - 如上图，`Session A`中的事务先根据条件`number > 0`这个条件查询表`hero`，得到了`name`列值为`'刘备'`的记录；之后`Session B`中提交了一个隐式事务，该事务向表`hero`中插入了一条新记录；之后`Session A`中的事务再根据相同的条件`number > 0`查询表`hero`，得到的结果集中包含`Session B`中的事务新插入的那条记录，这种现象也被称之为`幻读`。
  - 如果`Session B`中是删除了一些符合`number > 0`的记录而不是插入新记录，那`Session A`中之后再根据`number > 0`的条件读取的记录变少了，这种现象算不算`幻读`呢？明确说一下，这种现象不属于`幻读`，**幻读强调的是一个事务按照某个相同条件多次读取记录时，后读取时读到了之前没有读到的记录。**
  - 那对于先前已经读到的记录，之后又读取不到这种情况，算啥呢？其实这相当于对每一条记录都发生了不可重复读的现象。幻读只是重点强调了读取到了之前读取没有获取到的记录。
- 不同隔离级别，并发事务可以发生的不同严重程度问题

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED	可能发生	可能发生	可能发生
READ COMMITTED	不可能发生	可能发生	可能发生
REPEATABLE READ	不可能发生	不可能发生	可能发生
SERIALIZABLE	不可能发生	不可能发生	不可能发生

  - **READ UNCOMMITTED**隔离级别下，可能发生**脏读**、**不可重复读**和**幻读**问题。
  - **READ COMMITTED**隔离级别下，可能发生**不可重复读**和**幻读**问题，但是不可以发生**脏读**问题。
  - **REPEATABLE READ**隔离级别下，可能发生**幻读**问题，但是不可以发**脏读**和**不可重复读**的问题。
  - **SERIALIZABLE**隔离级别下，各种问题都不可以发生。

MySQL中支持的四种隔离级别

- MySQL中默认隔离级别为**REPEATABLE READ**
- 设置隔离级别

    - 语法：**SET [GLOBAL|SESSION] TRANSACTION ISOLATION LEVEL level;**
        - 其中的`level`可选值有4个：

```
level: {
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
   | SERIALIZABLE
}
```

    - 使用**GLOBAL**关键字（在全局范围影响）
        - 只对执行完该语句之后产生的会话起作用。
        - 当前已经存在的会话无效。
    - 使用**SESSION**关键字
        - 对当前会话的所有后续的事务有效
        - 该语句可以在已经开启的事务中间执行，但不会影响当前正在执行的事务。
        - 如果在事务之间执行，则对后续的事务有效。
    - 两个关键字都不用
        - 只对当前会话中下一个即将开启的事务有效。
        - 下一个事务执行完后，后续事务将恢复到之前的隔离级别。
        - 该语句不能在已经开启的事务中间执行，会报错的。
- 如果我们在服务器启动时想改变事务的默认隔离级别，可以修改启动参数`transaction-isolation`的值，比方说我们在启动服务器时指定了`--transaction-isolation=SERIALIZABLE`，那么事务的默认隔离级别就从原来的`REPEATABLE READ`变成了`SERIALIZABLE`。
- 想要查看当前会话默认的隔离级别可以通过查看系统变量`transaction_isolation`的值来确定：

```
SHOW VARIABLES LIKE 'transaction_isolation';
```

事务隔离级别

- **读未提交 READ UNCOMMITTED**
    - 如果一个事务读到了另一个未提交事务修改过的数据，那么这种`隔离级别`就称之为`未提交读`（英文名：`READ UNCOMMITTED`）
    - Session A`和`Session B各开启了一个事务，`Session B`中的事务先将`id`为`1`的记录的列`c`更新为`'关羽'`，然后`Session A`中的事务再去查询这条`id`为`1`的记录，那么在`未提交读`的隔离级别下，查询结果就是`'关羽'`，也就是说某个事务读到了另一个未提交事务修改过的记录。但是如果`Session B`中的事务稍后进行了回滚，那么`Session A`中的事务相当于读到了一个不存在的数据，这种现象就称之为**脏读**
- **读已提交 READ COMMITTED**

- 如果一个事务只能读到另一个**已经提交的事务**修改过的数据，并且其他事务每对该数据进行一次修改并提交后，该事务都能查询得到最新值，那么这种`隔离级别`就称之为`已提交读`（英文名：`READ COMMITTED`）

READ COMMITTED 隔离级别示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②		BEGIN;
③		UPDATE t SET c = '关羽' WHERE id = 1;
④	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'刘备')	
⑤		COMMIT;
⑥	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'关羽')	

- 对于某个处在在`已提交读`隔离级别下的事务来说，只要其他事务修改了某个数据的值，并且之后提交了，那么该事务就会读到该数据的最新值.我们在`Session B`中提交了几个隐式事务，这些事务都修改了`id`为`1`的记录的列c的值，每次事务提交之后，`Session A`中的事务都可以查看到最新的值。这种现象也被称之为`不可重复读`。

- **可重复读 REPEATABLE READ**
  - 在一些业务场景中，一个事务只能读到另一个已经提交的事务修改过的数据，但是第一次读过某条记录后，即使其他事务修改了该记录的值并且提交，该事务之后再读该条记录时，读到的仍是第一次读到的值，而不是每次都读到不同的数据。那么这种`隔离级别`就称之为`可重复读`（英文名：`REPEATABLE READ`）

REPEATABLE READ 隔离级别示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'刘备')	
③		UPDATE t SET c = '关羽' WHERE id = 1; (隐式提交)
④	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'刘备')	
⑤		UPDATE t SET c = '张飞' WHERE id = 1; (隐式提交)
⑥	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'刘备')	

- **串行化 SERIALIZABLE**
  - 以上3种隔离级别都允许对同一条记录进行`读-读`、`读-写`、`写-读`的并发操作，如果我们不允许`读-写`、`写-读`的并发操作，可以使用`SERIALIZABLE`隔离级别

SERIALIZABLE 隔离级别示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②		BEGIN;
③		UPDATE t SET c = '关羽' WHERE id = 1;
④	SELECT * FROM t WHERE id = 1; (等待中…)	
⑤		COMMIT;
⑥	SELECT * FROM t WHERE id = 1; (此时读到的列c的值为'关羽')	

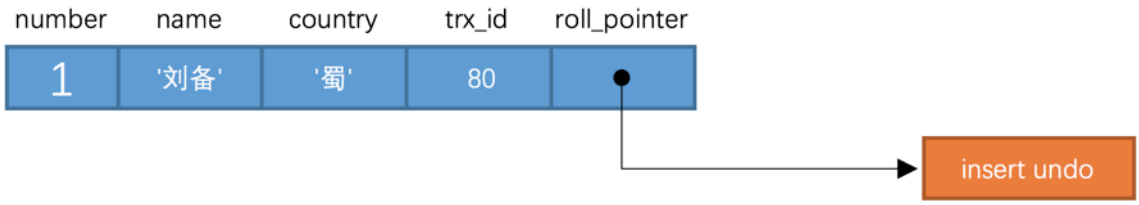
当`Session B`中的事务更新了`id`为`1`的记录后，之后`Session A`中的事务再去访问这条记录时就被卡住了，直到`Session B`中的事务提交之后，`Session A`中的事务才可以获取到查询结果。

隔离级别	脏读	不可重复读	幻影读	加锁读
未提交读	√	√	√	×
提交读	×	√	√	×
可重复读	×	×	√	×
可串行化	×	×	×	√

MVCC原理

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（`row_id`并不是必要的，我们创建的表中有主键或者非NULL的UNIQUE键时都不会包含`row_id`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，可以通过它找到该条记录修改之前的信息。
- 例如向表A中插入一条数据，则这条记录的示意图为：



- 实际上**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）。虽然真正的insert undo日志占用的存储空间被释放了，但是roll_pointer的值并不会被清除，roll_pointer属性占用7个字节，第一个比特位就标记着它指向的undo日志的类型，如果该比特位的值为1时，就代表着它指向的undo日志类型为insert undo。
- 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。另外，每个版本中还包含生成该版本时对应的`事务id`

ReadView

- 对于使用**READ UNCOMMITTED**隔离级别的事务来说，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
- 对于使用**SERIALIZABLE**隔离级别的事务来说，InnoDB规定使用**加锁**的方式来访问记录
- 对于使用**READ COMMITTED**和**REPEATABLE READ**隔离级别的事务来说，都必须保证读到已经提交了的事务修改过的记录，也就是说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。所以提出了ReadView概念。

重要概念

- ReadView中比较重要的4个概念
  - **m_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。
  - **min_trx_id**: 表示在生成`ReadView`时当前系统中活跃的读写事务中**最小的事务id**，也就是`m_ids`中的最小值。
  - **max_trx_id**: 表示生成`ReadView`时系统中应该分配给下一个事务的`id`值。
    - **max_trx_id并不是m_ids中的最大值**，事务id是递增分配的。比方说现在有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，m_ids就包括1和2，min_trx_id的值就是1，max_trx_id的值就是4。
  - **creator_trx_id**: 表示生成该`ReadView`的事务的`事务id`。
- 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。

判断记录的某个版本是否可见的步骤

- 1.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。
- 2.如果被访问版本的**trx_id**属性值小于`ReadView`中的**min_trx_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。
- 3.如果被访问版本的**trx_id**属性值大于`ReadView`中的**max_trx_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。
- 4.如果被访问版本的**trx_id**属性值在`ReadView`的**min_trx_id**和**max_trx_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。
- 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。
- **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。**

不同隔离级别的ReadView生成时机

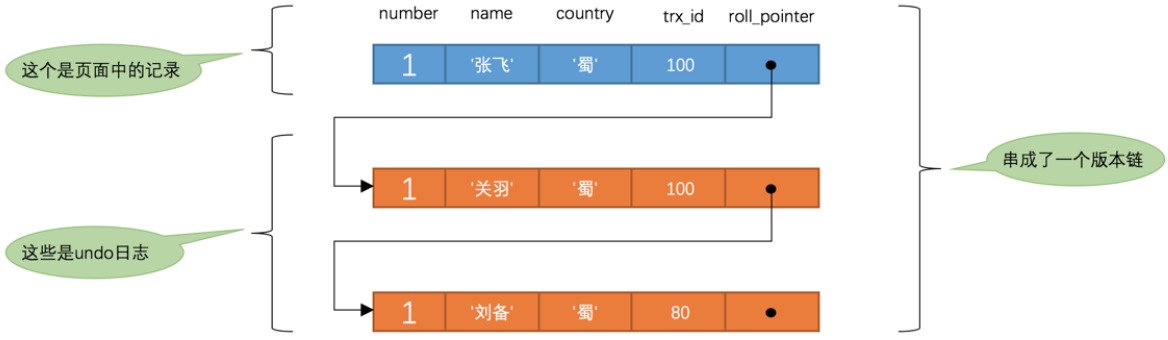
READ COMMITTED —— 每次读取数据前都生成一个ReadView

- 此时有两个事务在执行，事务ID分别为100，200

```
# Transaction 100
BEGIN;
UPDATE hero SET name = '关羽' WHERE number = 1;
UPDATE hero SET name = '张飞' WHERE number = 1;
```

```
# Transaction 200
BEGIN;
# 更新了一些别的表的记录
...
```

- 此时表中number为1的记录得到的版本链如下：



- 现在假设有一个使用**READ COMMITTED**隔离级别的事务开始执行

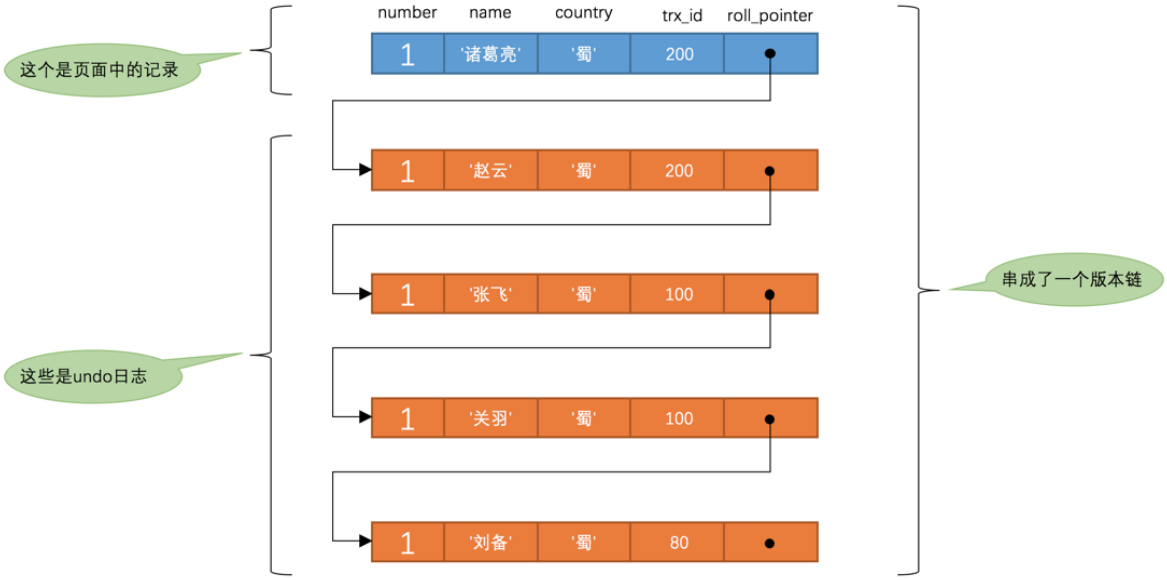
```
# 使用READ COMMITTED隔离级别的事务
BEGIN;
# SELECT1：Transaction 100、200未提交
SELECT * FROM hero WHERE number = 1; # 得到的列name的值为'刘备'
```

ps:**再次强调一遍，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Transaction 200中先更新一些别的表的记录，目的是让它分配事务id。

  - 这条select1语句的执行流程是这样的：
    - 1.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]`，`min_trx_id`为`100`，`max_trx_id`为`201`，`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`'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。
    - 3.下一个版本的列`name`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。
    - 4.下一个版本的列`name`的内容是`'刘备'`，该版本的`trx_id`值为`80`，小于`ReadView`中的`min_trx_id`值`100`，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`'刘备'`的记录。
  - 然后我们commit一下事务ID为100的事务，然后再到事务ID为200的事务中更新一下表`hero`中`number`为`1`的记录：

```
# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...

UPDATE hero SET name = '赵云' WHERE number = 1;

UPDATE hero SET name = '诸葛亮' WHERE number = 1;
```

  - 此刻，表`hero`中`number`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`number`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 hero WHERE number = 1; # 得到的列name的值为'刘备'

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

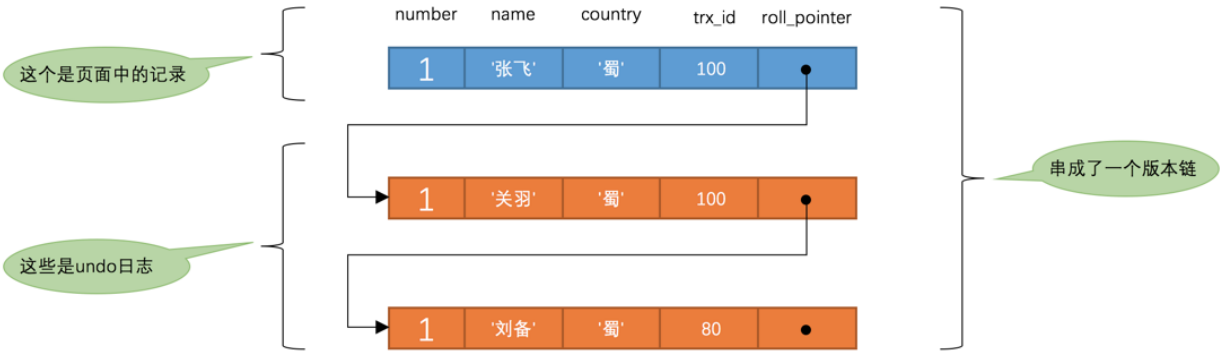
- 这个`SELECT2`的执行过程如下：
  - 1.在执行`SELECT`语句时会又会单独生成一个`ReadView`，该`ReadView`的`m_ids`列表的内容就是`[200]`（`事务id`为`100`的那个事务已经提交了，所以再次生成快照时就没有它了），`min_trx_id`为`200`，`max_trx_id`为`201`，`creator_trx_id`为`0`。
  - 2.然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列`name`的内容是`'诸葛亮'`，该版本的`trx_id`值为`200`，在`m_ids`列表内，所以不符合可见性要求，根据`roll_pointer`跳到下一个版本。
  - 3.下一个版本的列`name`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。
  - 4.下一个版本的列`name`的内容是`'张飞'`，该版本的`trx_id`值为`100`，小于`ReadView`中的`min_trx_id`值`200`，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`隔离级别的事务中查询表`hero`中`number`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一个独立的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。**
- 比方说现在系统里有两个`事务id`分别为`100`、`200`的事务在执行：

```
# Transaction 100
BEGIN;

UPDATE hero SET name = '关羽' WHERE number = 1;

UPDATE hero SET name = '张飞' WHERE number = 1;


# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...
```

- 此刻，表`hero`中`number`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 hero WHERE number = 1; # 得到的列name的值为'刘备'
```

  - 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`m_ids`列表的内容就是`[100, 200]`，`min_trx_id`为`100`，`max_trx_id`为`201`，`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`'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。
    - 3.下一个版本的列`name`的内容是`'关羽'`，该版本的`trx_id`值也为`100`，也在`m_ids`

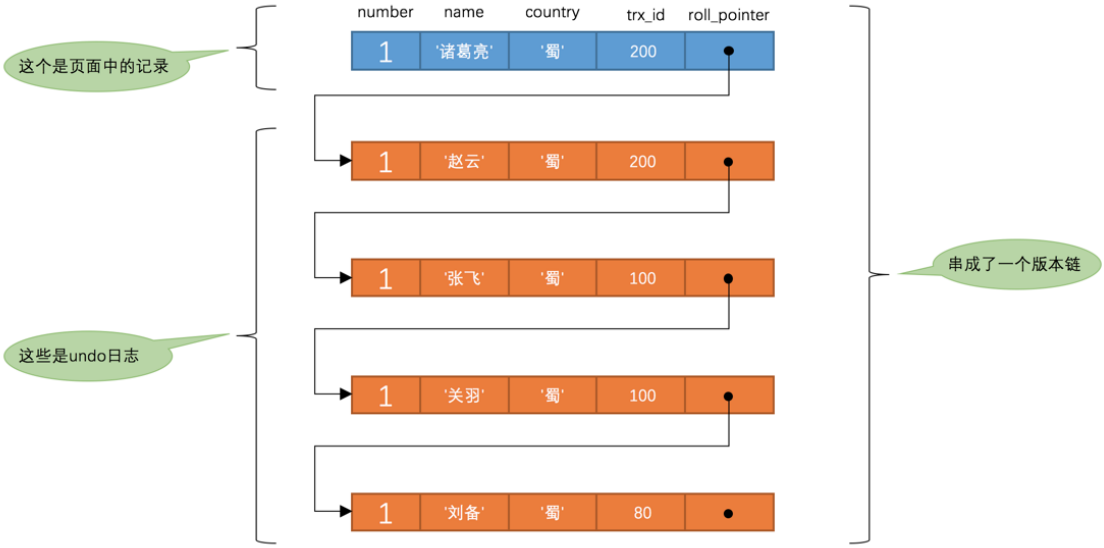
列表内，所以也不符合要求，继续跳到下一个版本。

- 4.下一个版本的列`name`的内容是`'刘备'`，该版本的`trx_id`值为`80`，小于`ReadView`中的`min_trx_id`值`100`，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`'刘备'`的记录。

- 然后我们commit一下事务ID为100的事务，然后再到`事务id`为`200`的事务中更新一下表`hero`中`number`为`1`的记录：

```
# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...

UPDATE hero SET name = '赵云' WHERE number = 1;

UPDATE hero SET name = '诸葛亮' WHERE number = 1;
```

- 此刻，表`hero`中`number`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`number`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 hero WHERE number = 1; # 得到的列name的值为'刘备'

# SELECT2：Transaction 100提交，Transaction 200未提交
SELECT * FROM hero WHERE number = 1; # 得到的列name的值仍为'刘备'
```

- 这个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`m_ids`列表的内容就是`[100, 200]`，`min_trx_id`为`100`，`max_trx_id`为`201`，`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`'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。
        - 3.下一个版本的列`name`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。
        - 4.下一个版本的列`name`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id`的，所以该版本也不符合要求，同理下一个列`name`的内容是`'关羽'`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`'刘备'`，该版本的`trx_id`值为`80`，小于`ReadView`中的`min_trx_id`值`100`，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。
    - 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，然后再到刚才使用`REPEATABLE READ`隔离级别的事务中继续查找这个`number`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。

MVCC小结

- 从上边的描述中我们可以看出来，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。`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，而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**。
- 我们之前说执行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，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，大家可以对比上边举的例子自己试想一下怎么使用。 另外，所谓的MVCC只是在我们进行普通的SEELCT查询时才生效，截止到目前我们所见的所有SELECT语句都算是普通的查询，至于啥是个不普通的查询，在锁部分会写。
- 我们说**insert undo**在事务提交之后就可以被释放掉了，而**update undo**由于还需要支持`MVCC`，不能立即删除掉。