MySQL事务篇

一、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 t (
    id INT PRIMARY KEY,
    c VARCHAR(100)
) Engine=InnoDB CHARSET=utf8;
```

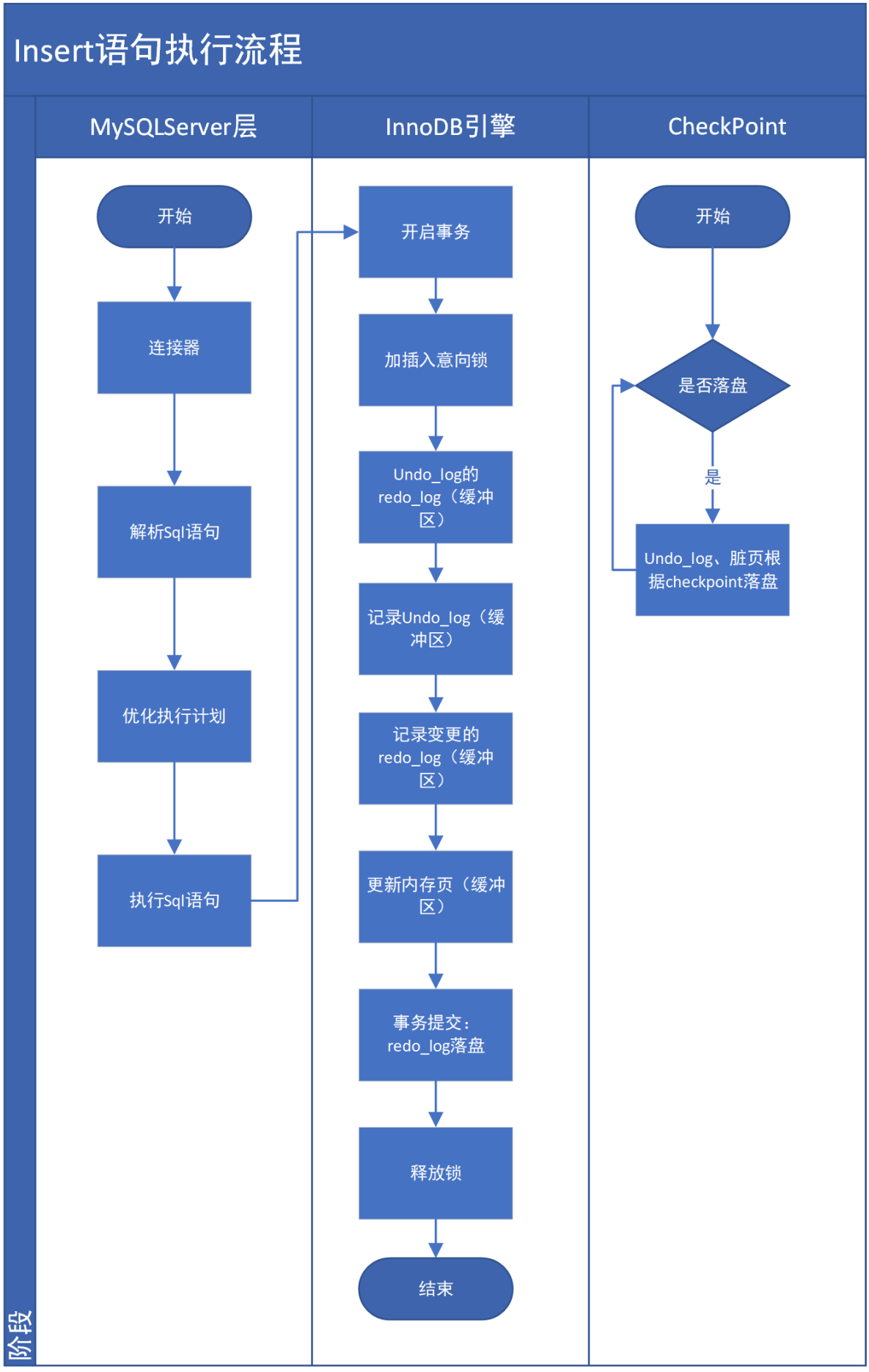
然后向这个表里插入一条数据：

```
INSERT INTO t VALUES(1, '刘备');
```

现在表里的数据就是这样的：

```
mysql> SELECT * FROM t;
+----+--------+
| id | c      |
+----+--------+
|  1 |  刘备   |
+----+--------+
1 row in set (0.01 sec)
```

二、执行流程


Insert语句执行流程
MySQLServer层
InnoDB引擎
CheckPoint
开始
连接器
解析Sql语句
优化执行计划
执行Sql语句
开启事务
加插入意向锁
Undo_log的redo_log（缓冲区）
记录Undo_log（缓冲区）
记录变更的redo_log（缓冲区）
更新内存页（缓冲区）
事务提交：redo_log落盘
释放锁
结束
开始
是否落盘
是
Undo_log、脏页根据checkpoint落盘
阶段

三、事务回顾

`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。

我们前边说过事务有一个特性称之为**隔离性**，理论上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，所以设计数据库的程序员提出了各种**隔离级别**，来最大限度的提升系统并发处理事务的能力，但是这也是以牺牲一定的**隔离性**来达到的。

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。接下来我们深入分析一下事务的原理。

由于在**MySQL中的事务是由存储引擎实现的**，而且MySQL内支持事务的存储引擎只有InnoDB。因此我们主要讲解**InnoDB**存储引擎中的事务。

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)： 事务最小工作单元，要么全成功，要么全失败 。
- 一致性(consistency)： 事务开始和结束后，数据库的完整性不会被破坏 。
- 隔离性(isolation)： 不同事务之间互不影响
- 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 。

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。引发另一个事务，在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。导致另一个事务，在事务中多次查询的结果不一致。

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）。
    - 解决：0个问题
- **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。
    - 存在：1个问题（幻读）。
    - 解决：2个问题（脏读、不可重复读）
    - MySQL默认的事务隔离级别
- **serializable 串行化**，同时只能执行一个事务，相当于事务中的单线程。
    - 存在：0个问题。
    - 解决：3个问题（脏读、不可重复读、幻读）

四、事务底层原理详解

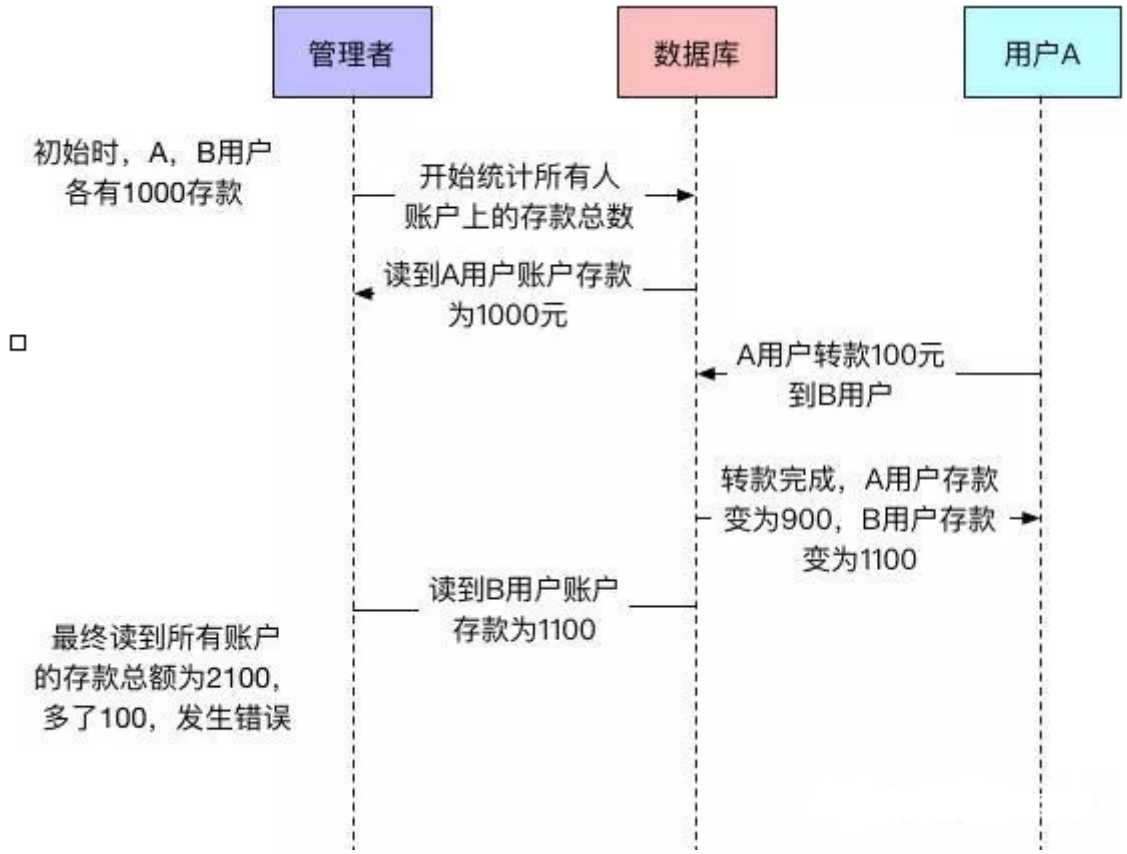
4.1 丢失更新问题

两个事务针对同一数据都发生修改操作时，会存在丢失更新的现象，这个问题称之为丢失更新问题。

举个栗子：

管理者要查询所有用户的存款总额，假设除了用户A和用户B之外，其他用户的存款总额都为0，A、B用户各有存款1000，所以所有用户的存款总额为2000。但是在查询过程中，用户A会向用户B进行转账操作。转账操作和查询总额操作的时序图如下图所示。

转账和查询的时序图：

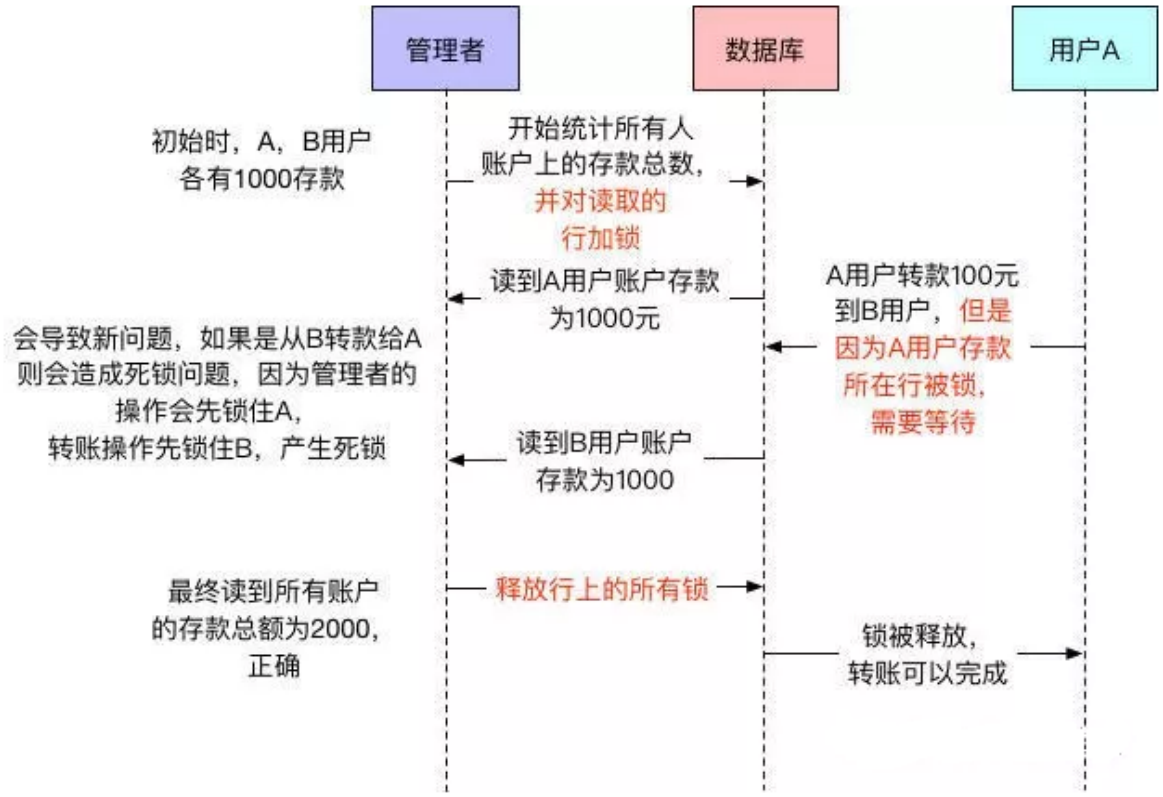


4.2 解决方案

4.2.1 基于锁并发控制LBCC

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

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。
使用锁机制：



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。

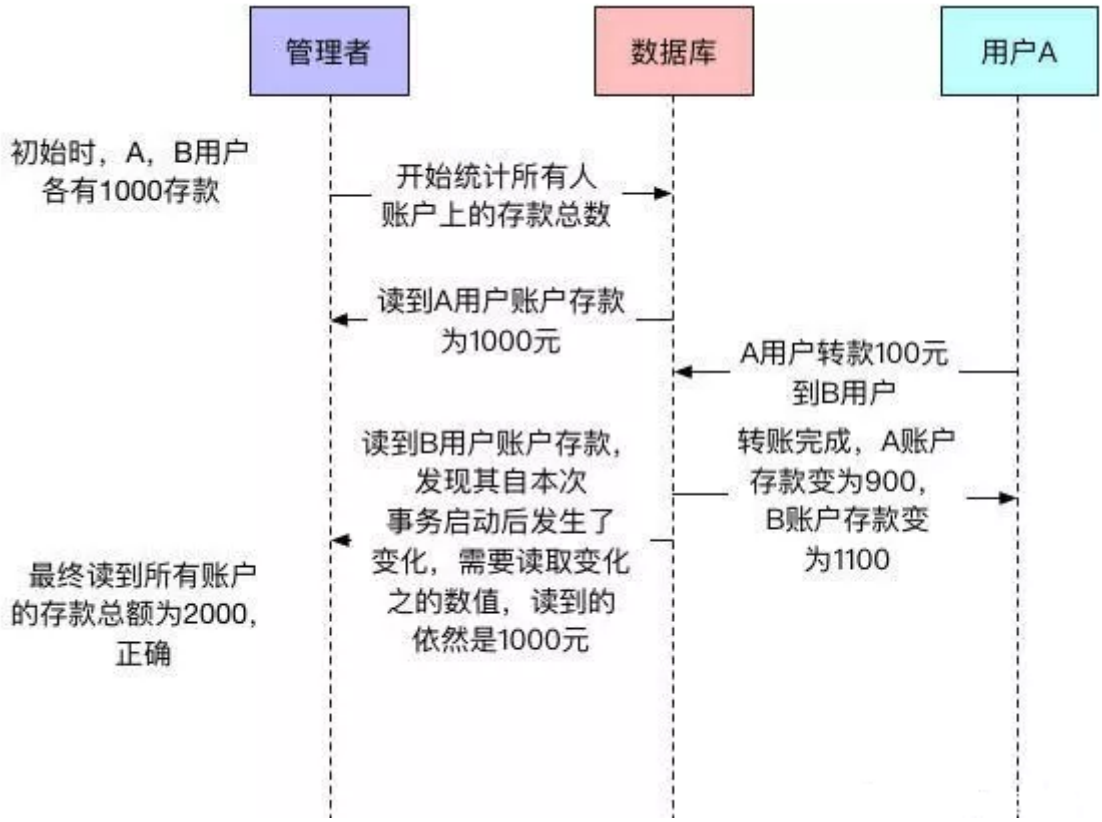
> 关于锁，会在锁篇详细讲解

4.2.2 基于版本并发控制MVCC

当然使用MVCC（MVCC,多版本的并发控制，英文全称：Multi Version Concurrency Control）机制可以解决这个问题。

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】。

使用MVCC机制（RR隔离级别下的演示情况）：



MVCC使得普通的SELECT请求不会加锁，读写不冲突，提高了数据库的并发处理能力。MVCC，保证了ACID中的隔离性。MVCC是怎么实现的，接下来我们详细说明。

4.3 MVCC实现原理【InnoDB】

首先来看一下MVCC的定义：

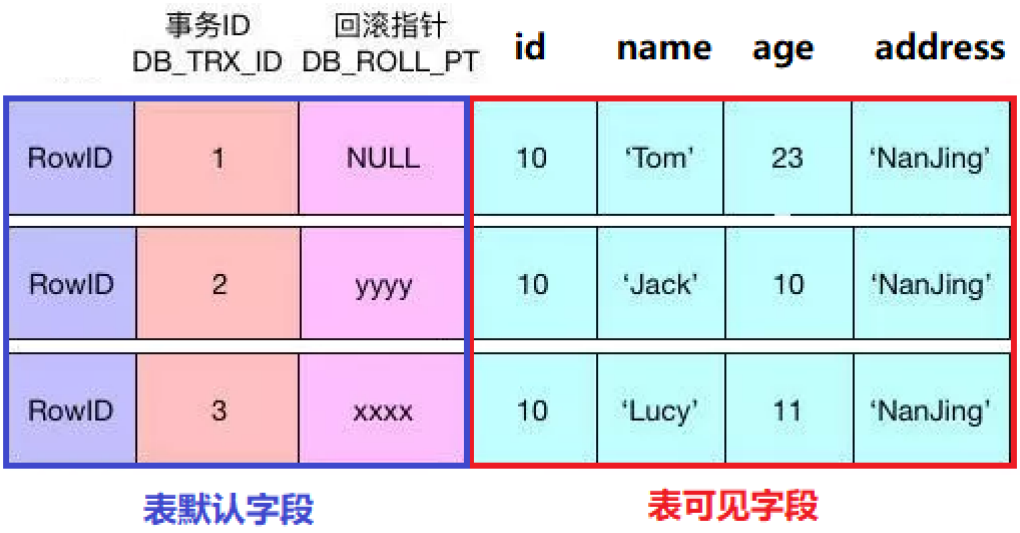
> **Multiversion concurrency control (MVCC)** is a concurrency control method commonly used by database management systems to provide concurrent access to the database and in programming languages to implement transactional memory.

翻译之后可知，MVCC是用于数据库提供并发访问控制的并发控制技术。是相对于LBCC更好的一种并发控制技术。MVCC核心思想是**读不加锁，读写不冲突。**在读多写少的OLTP应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，这也是为什么现阶段，几乎所有的RDBMS，都支持了MVCC原因。

MVCC 核心理念是数据快照，不同的事务访问不同版本的数据快照，从而实现事务的隔离级别。虽然字面上是说具有多个版本的数据快照，但这并不意味着数据库必须拷贝数据，保存多份数据文件，这样会浪费大量的存储空间。InnoDB通过事务的undo log巧妙地实现了多版本的数据快照。

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

> 注意MVCC只在RR和RC两个隔离级别下工作。RU和串行化隔离级别都和 MVCC不兼容 。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。



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。这两个列，**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，会把事务id放到当前事务影响的行事务id中。当查询时，需要用当前查询的事务id和每行记录的事务id进行比较。

4.3.1 undo log

为了更好的支持并发，InnoDB的多版本一致性读是采用了基于回滚段的的方式。另外，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 Log膨胀的问题，需要引入回收机制进行回收。Undo log保存了记录修改前的镜像。

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

1）insert undo log：

是在 insert 操作中产生的 undo log。

因为 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 user(id,name,age,address) values (10,'tom',23,'nanjing')
```

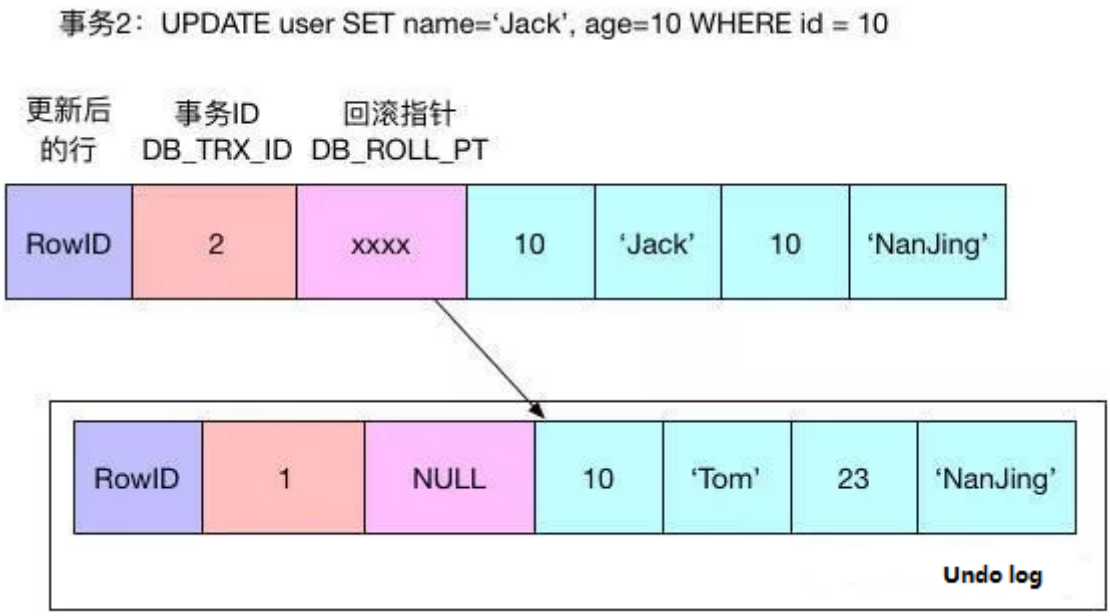
2）update undo log：

是 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。

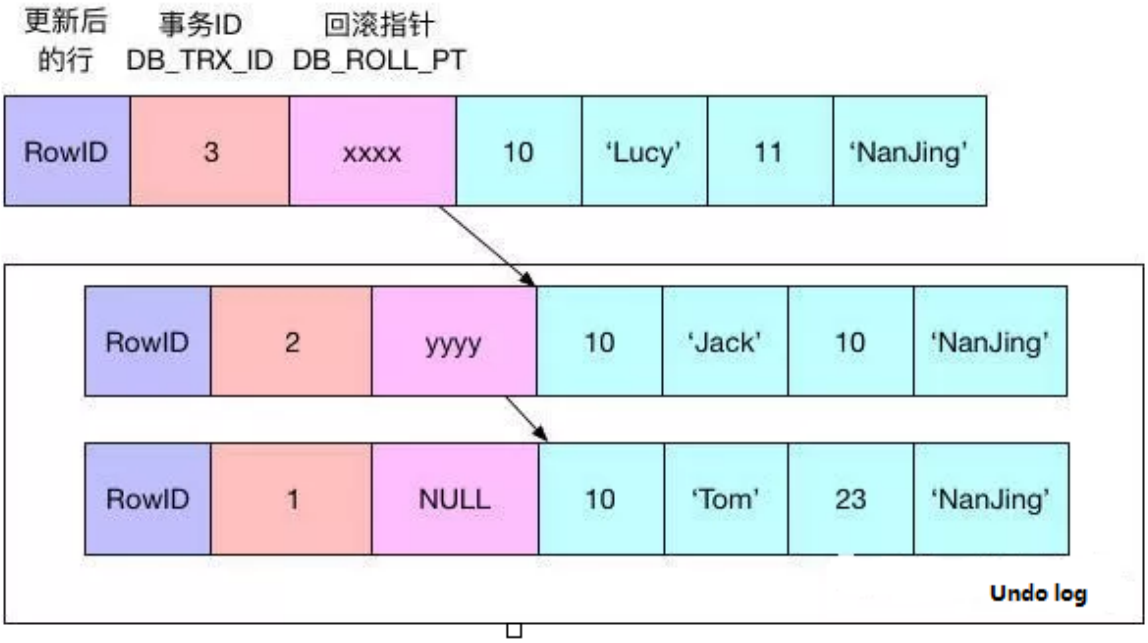
如下图所示（第一次修改）：

```
# 事务2：
update user set name='jack',age=10 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改行，将该行当前的值复制到undo log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 log中修改前的行。
```



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 user set name='Lucy',age=11 where id=10;
```



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，是多个版本数据的链表，也称之为**版本链**了。

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?

所以设计`InnoDB`的设计者提出了一个**ReadView**的概念。结合Undo log的默认字段【事务db_trx_id】来控制那个版本的undo log可被用户看见。这个`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**，并确定三个变量的值：

- **m_up_limit_id**：事务id下限，m_ids事务列表中的最小事务id。
- **m_low_limit_id**：事务id上限，系统中将要产生的下一个事务id的值。
- **m_creator_trx_id**：当前事务id，m_ids中不包含当前事务id。

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`隔离级别的的一个非常大的区别就是它们生成`ReadView`的时机不同
- `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，详情在案例中分析。

3）如何判断可见性?

在访问某条记录时，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：被访问undo log版本的事务id与ReadView的关系

- 小于ReadView中的`m_up_limit_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。
- 等于ReadView中的`m_creator_trx_id`，**可以**被访问。
- 大于等于ReadView中的`m_low_limit_id`，在生成`ReadView`后才生成，所以该版本**不可以**被当前事务访问。
- 在`m_up_limit_id`和`m_low_limit_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）ReadView案例分析

```
 -- 开启事务：还有一种方式begin
start transaction
-- 提交事务：
commit
-- 回滚事务：
rollback
-- 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
-- 级别字符串：`read uncommitted`、`read committed`、`repeatable read【默认】`、`serializable`
-- 查看当前运行的事务
SELECT
   a.trx_id,a.trx_state,a.trx_started,a.trx_query,
   b.ID,b.USER,b.DB,b.COMMAND,b.TIME,b.STATE,b.INFO,
   c.PROCESSLIST_USER,c.PROCESSLIST_HOST,c.PROCESSLIST_DB,   d.SQL_TEXT
FROM
   information_schema.INNODB_TRX a
LEFT JOIN information_schema.PROCESSLIST b ON a.trx_mysql_thread_id = b.id AND b.COMMAND = 'Sleep'
LEFT JOIN PERFORMANCE_SCHEMA.threads c ON b.id = c.PROCESSLIST_ID
LEFT JOIN PERFORMANCE_SCHEMA.events_statements_current d ON d.THREAD_ID = c.THREAD_ID;
```

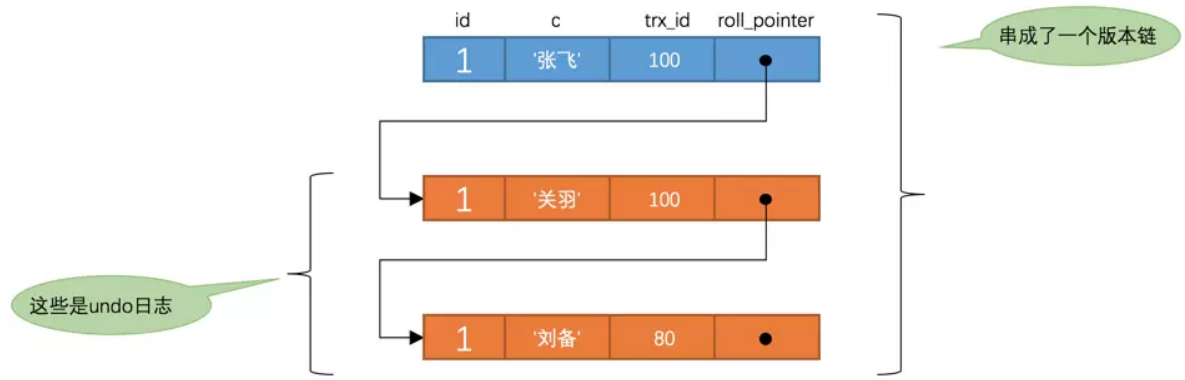
1. 读已提交RC案例

每次读取数据前都生成一个ReadView，默认t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			查询id=1，c为刘备
T7	提交事务01		
T8			查询id=1，c为张飞
T9		提交事务02	
T10			查询id=1，c为诸葛亮

> Tips： 事务id是递增的

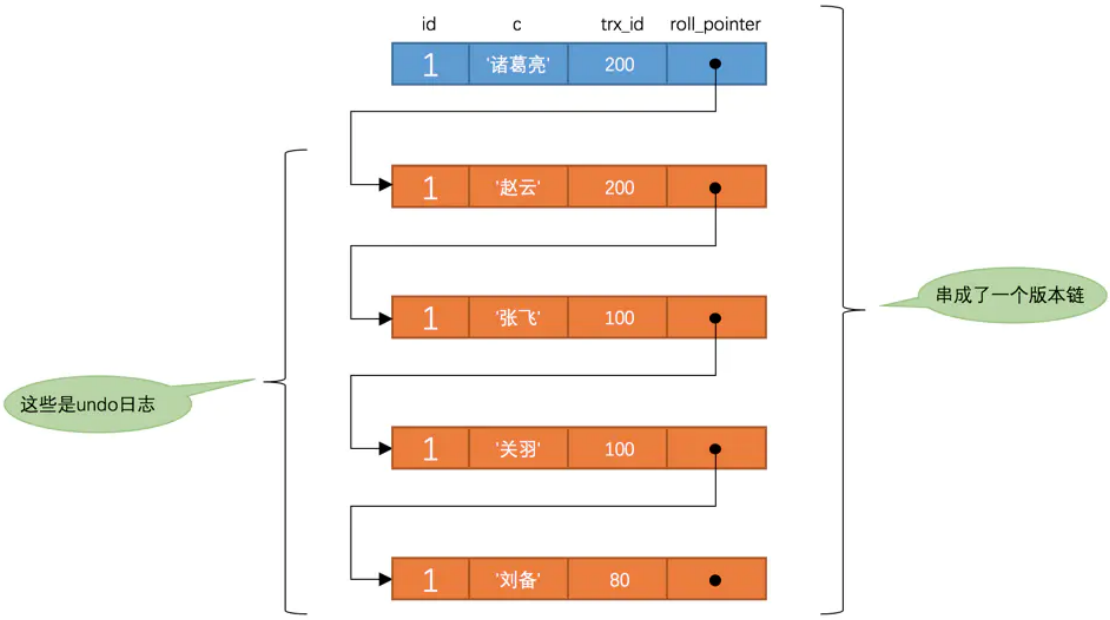
T3时刻，表`t`中`id`为`1`的记录得到的版本链表如下所示：



这个`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`t`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。

案例代码如下：

```
# 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 t; # 默认是刘备
# Transaction 100
BEGIN;

UPDATE t SET c = '关羽' WHERE id = 1;

UPDATE t SET c = '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 t SET c = '赵云' WHERE id = 1;

UPDATE t SET c = '诸葛亮' WHERE id = 1;

COMMIT;
```

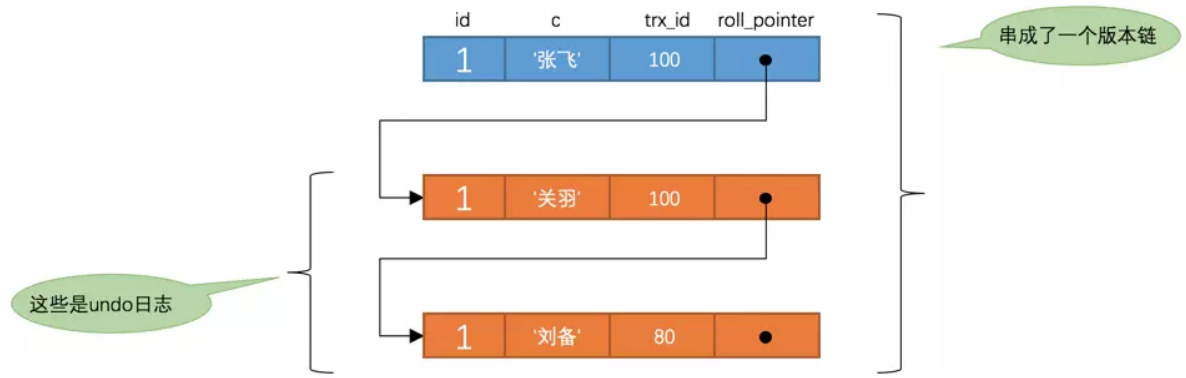
```sql
# 事务03
-- 查询事务隔离级别：
select @@tx_isolation;
-- 设置数据库的隔离级别
set session transaction isolation level read committed;

BEGIN;

# SELECT01：Transaction 100、200未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'

# SELECT02：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'

# SELECT03：Transaction 100、200提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'诸葛亮'
COMMIT;
```

2. 可重复读RR案例

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。

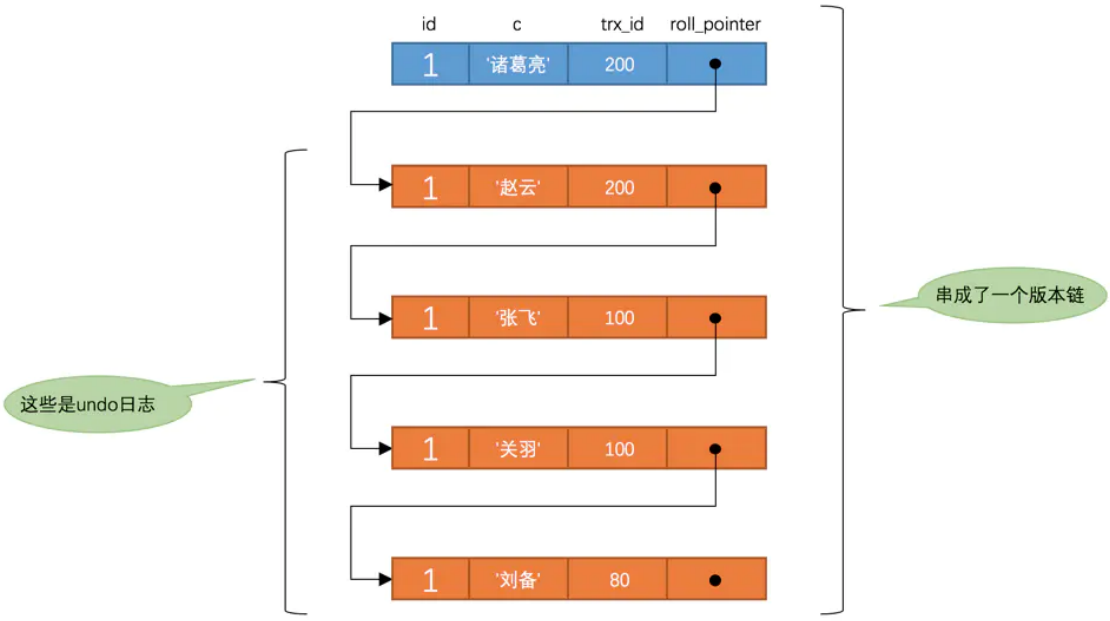
T3时刻，表`t`中`id`为`1`的记录得到的版本链表如下所示：



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。

4.4 MVCC下的读操作

在一个支持MVCC并发控制的系统中，哪些读操作是快照读？哪些操作又是当前读呢？

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

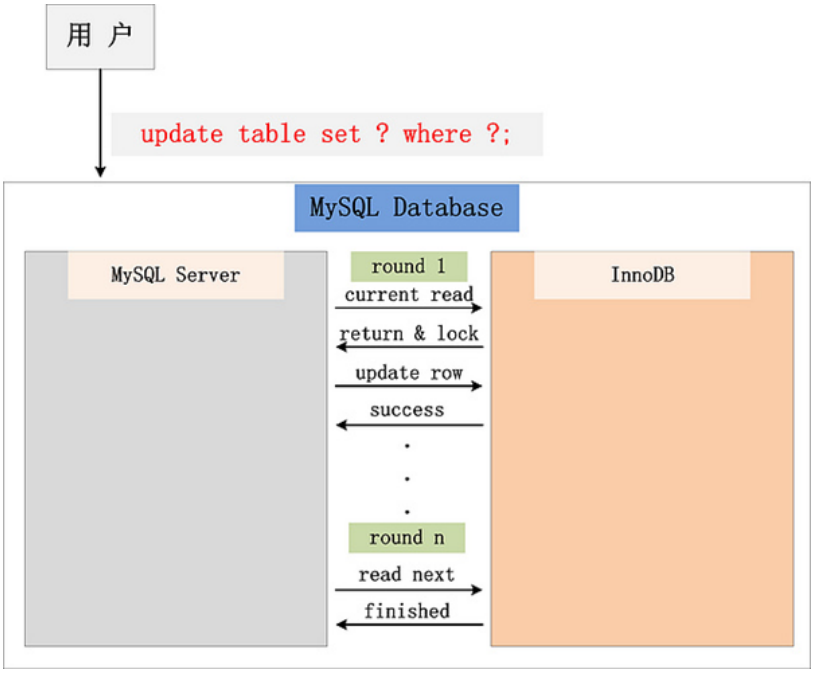
以MySQL InnoDB为例：

- **快照读**：简单的select操作，属于快照读，不加锁。

```
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**。
```



从图中，可以看到，一个Update操作的具体流程。

- **当Update SQL被发给MySQL后，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。

案例代码：

```
BEGIN;

# SELECT1：Transaction 100、200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'

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

COMMIT;
```

4.4.2 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；

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`。

五、事务回滚与数据恢复

> 正常情况下，事务如何回滚数据?
>
> 异常情况下，如何回滚数据?

事务的隔离性由**MVCC**和**锁**实现。

事务的原子性，持久性和一致性主要是通过redo log、undo log和Force Log at Commit机制机制来完成的。

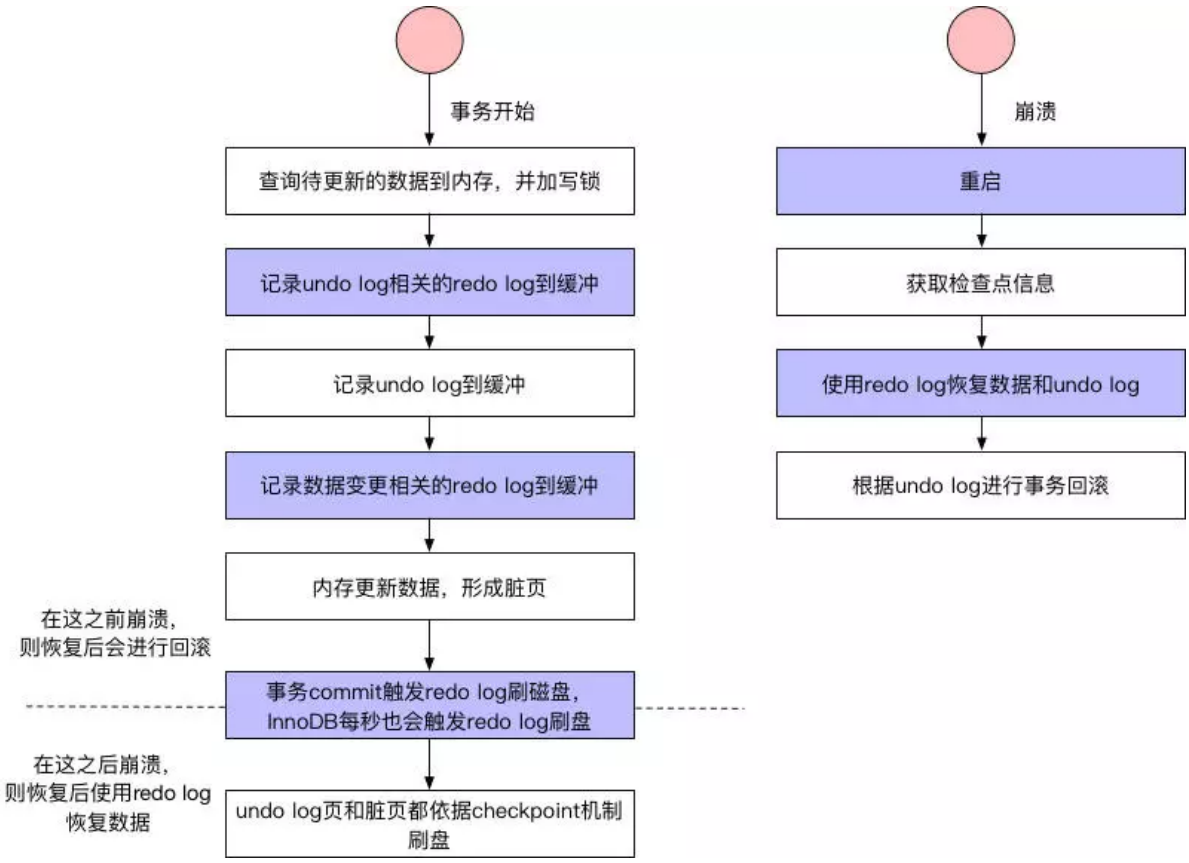
- Force Log at Commit机制保证事务提交后redo log日志都已经持久化。
- undo log用于对事务的影响进行撤销，也可用于多版本控制。
- redo log用于在崩溃时恢复数据，**事务回滚和数据恢复是如何实现的呢?**

redo log的原理:

- redo log顾名思义，就是重做日志，每次数据库的SQL操作导致的数据变化它都会记录一下，具体来说，**redo log是物理日志，记录的是数据库页的物理修改操作。**如果数据发生了丢失，数据库可以根据redo log进行数据恢复。
- InnoDB通过Force Log at Commit机制实现事务的持久性，即当事务COMMIT时，必须先将该事务的所有日志都写入到redo log文件进行持久化之后，COMMIT操作才算完成。
- 当事务的各种SQL操作执行时，即会在缓冲区中修改数据，也会将对应的redo log写入它所属的缓存。当事务执行COMMIT时，与该事务相关的redo log缓冲必须都全部刷新到磁盘中之后COMMIT才算执行成功。

数据回滚与恢复流程:

- 开启一个事务后，用户可以使用COMMIT来提交，也可以用ROLLBACK来回滚。
- 其中COMMIT或者ROLLBACK执行成功之后，数据一定是会被全部保存或者全部回滚到最初状态的，这也体现了事务的原子性。
- 但是也会有很多的异常情况，比如说**事务执行中途连接断开**，或者是**执行COMMIT或者ROLLBACK时发生错误**，**Server Crash**等，此时数据库会自动进行回滚或者重启之后进行恢复。回滚与恢复流程如下:



数据库崩溃重启后需要从redo log中把未落盘的脏页数据恢复出来，重新写入磁盘，保证用户的数据不丢失。当然，在崩溃恢复中还需要回滚没有提交的事务。由于回滚操作需要undo log的支持，undo log的完整性和可靠性需要redo log来保证，所以崩溃恢复先做redo恢复数据，然后做undo回滚。

在事务执行的过程中，除了记录redo log，还会记录一定量的undo log。undo log记录了数据在每个操作前的状态，如果事务执行过程中需要回滚，就可以根据undo log进行回滚操作。

undo log的存储不同于redo log，它存放在数据库内部的一个**特殊的段(segment)**中，这个段称为**回滚段**。回滚段位于共享表空间中。undo段中以undo page为更小的组织单位。undo page和存储数据库数据索引的页类似。因为redo log是物理日志，记录的是数据库页的物理修改操作。所以undo log（也看成数据库数据）的写入也会产生redo log，也就是undo log的产生会伴随着redo log的产生，这是因为undo log也需要持久性的保护。

事务进行过程中，每次DML sql语句执行，都会记录undo log和redo log，然后更新数据形成脏页，然后redo log按照时间或者空间等条件进行落盘，undo log和脏页按照checkpoint进行落盘，落盘后相应的redo log就可以删除了。此时，事务还未COMMIT，如果发生崩溃，则首先检查checkpoint记录，使用相应的redo log进行数据和undo log的恢复，然后查看undo log的状态发现事务尚未提交，然后就使用undo log进行事务回滚。事务执行COMMIT操作时，会将本事务相关的所有redo log都进行落盘，只有所有redo log落盘成功，才算COMMIT成功。然后内存中的数据脏页继续按照checkpoint进行落盘。如果此时发生了崩溃，则使用redo log恢复数据。