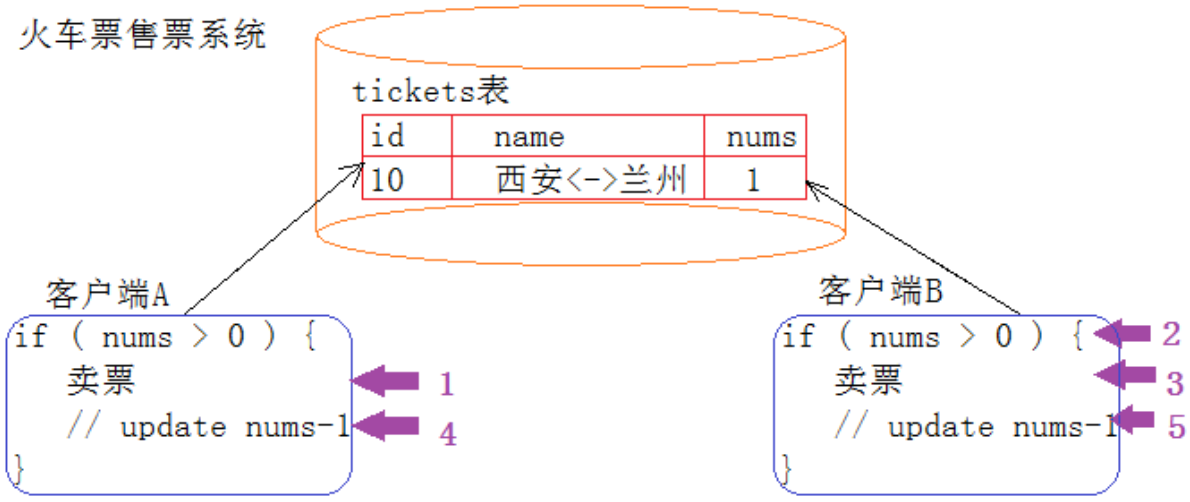
CURD不加控制，会有什么问题?



当客户端A检查还有一张票时，将票卖掉，还没有执行更新数据库时，客户端B检查了票数，发现大于0，于是又卖了一次票。然后A将票数更新回数据库。这是就出现了同一张票被卖了两次。

CURD满足什么属性，能解决上述问题?

1. 买票的过程得是原子的吧
2. 买票互相应该不能影响吧
3. 买完票应该要永久有效吧
4. 买前，和买后都要是确定的状态吧

什么是事务?

事务就是一组DML语句组成，这些语句在逻辑上存在相关性，这一组DML语句要么全部成功，要么全部失败，是一个整体。MySQL提供一种机制，保证我们达到这样的效果。事务还规定不同的客户端看到的数据是不相同的。

事务就是要做的或所做的事情，主要用于处理操作量大，复杂度高的数据。假设一种场景：你毕业了，学校的教务系统后台`MySQL`中，不在需要你的数据，要删除你的所有信息(一般不会:))，那么要删除你的基本信息(姓名，电话，籍贯等)的同时，也删除和你有关的其他信息，比如：你的各科成绩，你在校表现，甚至你在论坛发过的文章等。这样，就需要多条`MySQL`语句构成，那么所有这些操作合起来，就构成了一个事务。

正如我们上面所说，一个`MySQL`数据库，可不止你一个事务在运行，同一时刻，甚至有大量的请求被包装成事务，在向`MySQL`服务器发起事务处理请求。而每条事务至少一条`SQL`，最多很多`SQL`,这样如果大家都访问同样的表数据，在不加保护的情况，就绝对会出现问题。甚至，因为事务由多条`SQL`构成，那么，也会存在执行到一半出错或者不想再执行的情况，那么已经执行的怎么办呢?

所有，一个完整的事务，绝对不是简单的`sql`集合，还需要满足如下四个属性：

- **原子性**：一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
- **一致性**：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。

- **隔离性**：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（`Read uncommitted`）、读提交（`read committed`）、可重复读（`repeatable read`）和串行化（`Serializable`）
- **持久性**：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

上面四个属性，可以简称为`ACID`。

> 原子性（**A**tomicity，或称不可分割性）
>
> 一致性（**C**onsistency）
>
> 隔离性（**I**solation，又称独立性）
>
> 持久性（**D**urability）。

为什么会出现事务

事务被`MySQL`编写者设计出来,本质是为了当应用程序访问数据库的时候,事务能够简化我们的编程模型,不需要我们去考虑各种各样的潜在错误和并发问题.可以想一下当我们使用事务时,要么提交,要么回滚,我们不会去考虑网络异常了,服务器宕机了,同时更改一个数据怎么办对吧?**因此事务本质上是为了应用层服务的**.而不是伴随着数据库系统天生就有的.

备注：我们后面把`MySQL`中的一行信息，称为一行记录

事务的版本支持

在`MySQL`中只有使用了`Innodb`数据库引擎的数据库或表才支持事务，`MyISAM`不支持。

查看数据库引擎

```
mysql> show engines;          -- 表格显示
mysql> show engines \G        -- 行显示
*************************** 1. row ***************************
      Engine: InnoDB    -- 引擎名称
     Support: DEFAULT   -- 默认引擎
     Comment: Supports transactions, row-level locking, and foreign keys -- 描述
Transactions: YES       -- 支持事务
          XA: YES
  Savepoints: YES       -- 支持事务保存点
*************************** 2. row ***************************
      Engine: MRG_MYISAM
     Support: YES
     Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 3. row ***************************
      Engine: MEMORY    --内存引擎
     Support: YES
     Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 4. row ***************************
      Engine: BLACKHOLE
     Support: YES
     Comment: /dev/null storage engine (anything you write to it disappears)
```

```
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 5. row ***************************
      Engine: MyISAM
     Support: YES
     Comment: MyISAM storage engine
Transactions: NO            -- MyISAM不支持事务
          XA: NO
  Savepoints: NO
*************************** 6. row ***************************
      Engine: CSV
     Support: YES
     Comment: CSV storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 7. row ***************************
      Engine: ARCHIVE
     Support: YES
     Comment: Archive storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 8. row ***************************
      Engine: PERFORMANCE_SCHEMA
     Support: YES
     Comment: Performance Schema
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 9. row ***************************
      Engine: FEDERATED
     Support: NO
     Comment: Federated MySQL storage engine
Transactions: NULL
          XA: NULL
  Savepoints: NULL
9 rows in set (0.00 sec)
```

事务提交方式

事务的提交方式常见的有两种：

- 自动提交
- 手动提交

查看事务提交方式

```
mysql> show variables like 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.41 sec)
```

用`SET`来改变`MySQL`的自动提交模式:

```
mysql> SET AUTOCOMMIT=0;            #SET AUTOCOMMIT=0 禁止自动提交
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | OFF   |
+---------------+-------+
1 row in set (0.00 sec)
```

```
mysql> SET AUTOCOMMIT=1;           #SET AUTOCOMMIT=1 开启自动提交
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.01 sec)
```

事务常见操作方式

简单银行用户表

- 提前准备

```
## Centos 7 云服务器，默认开启3306 mysqld服务
[whb@VM-0-3-centos ~]$ sudo netstat -nltp
[sudo] password for whb:
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
    PID/Program name
tcp6       0      0 :::3306                 :::*                    LISTEN
   30415/mysqld

## 使用win cmd远程访问Centos 7云服务器，mysqld服务(需要win上也安装了MySQL，这里看到结果即可)
## 注意，使用本地mysql客户端，可能看不到链接效果，本地可能使用域间套接字，查不到链接
C:\Users\whb>mysql -uroot -p -h42.192.83.143
Enter password: ************
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3484
Server version: 5.7.33 MySQL Community Server (GPL)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
```

```
## 使用netstat查看链接情况，可知：mysql本质是一个客户端进程
[whb@VM-0-3-centos ~]$ sudo netstat -ntp
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
    PID/Program name
tcp6       0      0 172.17.0.3:3306         113.132.141.236:19354
ESTABLISHED 30415/mysqld

## 为了便于演示，我们将mysql的默认隔离级别设置成读未提交。
## 具体操作我们后面专门会讲，现在已使用为主。
mysql> set global transaction isolation level READ UNCOMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> quit
Bye

##需要重启终端，进行查看
mysql> select @@tx_isolation;
+------------------+
| @@tx_isolation   |
+------------------+
| READ-UNCOMMITTED |
+------------------+
1 row in set, 1 warning (0.00 sec)
```

- 创建测试表

```
create table if not exists account(
    id int primary key,
    name varchar(50) not null default '',
    blance decimal(10,2) not null default 0.0
)ENGINE=InnoDB DEFAULT CHARSET=UTF8;
```

- 正常演示 - 证明事务的开始与回滚

```
mysql> show variables like 'autocommit';  -- 查看事务是否自动提交。我们故意设置成自动提交，看看该选项是否影响begin
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.00 sec)

mysql> start transaction;                -- 开始一个事务begin也可以，推荐begin
Query OK, 0 rows affected (0.00 sec)

mysql> savepoint save1;                  -- 创建一个保存点save1
Query OK, 0 rows affected (0.00 sec)

mysql> insert into account values (1, '张三', 100);   -- 插入一条记录
Query OK, 1 row affected (0.05 sec)

mysql> savepoint save2;                  -- 创建一个保存点save2
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> insert into account values (2, '李四', 10000);  -- 在插入一条记录
Query OK, 1 row affected (0.00 sec)

mysql> select * from account;              -- 两条记录都在了
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   100.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> rollback to save2;                  -- 回滚到保存点save2
Query OK, 0 rows affected (0.03 sec)

mysql> select * from account;              -- 一条记录没有了
+----+--------+--------+
| id | name   | blance |
+----+--------+--------+
|  1 | 张三   | 100.00 |
+----+--------+--------+
1 row in set (0.00 sec)

mysql> rollback;                           -- 直接rollback，回滚在最开始
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account;              -- 所有刚刚的记录没有了
Empty set (0.00 sec)
```

- 非正常演示1 - 证明未commit，客户端崩溃，MySQL自动会回滚（隔离级别设置为读未提交）

```
-- 终端A
mysql> select * from account;          -- 当前表内无数据
Empty set (0.00 sec)

mysql> show variables like 'autocommit'; -- 依旧自动提交
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.00 sec)

mysql> begin;                              --开启事务
Query OK, 0 rows affected (0.00 sec)

mysql> insert into account values (1, '张三', 100);   -- 插入记录
Query OK, 1 row affected (0.00 sec)

mysql> select * from account;              --数据已经存在，但没有commit，此时同时查看终端B
+----+--------+--------+
| id | name   | blance |
+----+--------+--------+
|  1 | 张三   | 100.00 |
+----+--------+--------+
```

```
1 row in set (0.00 sec)

mysql> Aborted                              -- ctrl + \ 异常终止MySQL

--终端B
mysql> select * from account;               --终端A崩溃前
+----+--------+--------+
| id | name   | blance |
+----+--------+--------+
|  1 | 张三   | 100.00 |
+----+--------+--------+
1 row in set (0.00 sec)

mysql> select * from account;              --数据自动回滚
Empty set (0.00 sec)
```

- 非正常演示2 - 证明commit了，客户端崩溃，MySQL数据不会在受影响，已经持久化

```
--终端 A
mysql> show variables like 'autocommit'; -- 依旧自动提交
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.00 sec)

mysql> select * from account;   -- 当前表内无数据
Empty set (0.00 sec)

mysql> begin;                   -- 开启事务
Query OK, 0 rows affected (0.00 sec)

mysql> insert into account values (1, '张三', 100); -- 插入记录
Query OK, 1 row affected (0.00 sec)

mysql> commit;                              --提交事务
Query OK, 0 rows affected (0.04 sec)

mysql> Aborted                              -- ctrl + \ 异常终止MySQL

--终端 B
mysql> select * from account;              --数据存在了，所以commit的作用是将数据持久化到MySQL中
+----+--------+--------+
| id | name   | blance |
+----+--------+--------+
|  1 | 张三   | 100.00 |
+----+--------+--------+
1 row in set (0.00 sec)
```

- 非正常演示3 - 对比试验。证明begin操作会自动更改提交方式，不会受MySQL是否自动提交影响

```
-- 终端 A
mysql> select *from account;                --查看历史数据
```

```
+----+--------+--------+
| id | name   | blance |
+----+--------+--------+
|  1 | 张三   | 100.00 |
+----+--------+--------+
1 row in set (0.00 sec)

mysql> show variables like 'autocommit';    --查看事务提交方式
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.00 sec)

mysql> set autocommit=0;                     --关闭自动提交
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like 'autocommit';    --查看关闭之后结果
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | OFF   |
+---------------+-------+
1 row in set (0.00 sec)

mysql> begin;                                --开启事务
Query OK, 0 rows affected (0.00 sec)

mysql> insert into account values (2, '李四', 10000);    --插入记录
Query OK, 1 row affected (0.00 sec)

mysql> select *from account;                 --查看插入记录，同时查看终端B
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   100.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> Aborted                               --再次异常终止

-- 终端B
mysql> select * from account;        --终端A崩溃前
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   100.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> select * from account;       --终端A崩溃后，自动回滚
+----+--------+--------+
| id | name   | blance |
+----+--------+--------+
|  1 | 张三   | 100.00 |
```

```
+----+--------+--------+
1 row in set (0.00 sec)
```

- 非正常演示4 - 证明单条 SQL 与事务的关系

```
--实验一
-- 终端A
mysql> select * from account;
+----+--------+--------+
| id | name   | blance |
+----+--------+--------+
|  1 | 张三   | 100.00 |
+----+--------+--------+
1 row in set (0.00 sec)

mysql> show variables like 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.00 sec)

mysql> set autocommit=0;                    --关闭自动提交
Query OK, 0 rows affected (0.00 sec)

mysql> insert into account values (2, '李四', 10000);   --插入记录
Query OK, 1 row affected (0.00 sec)

mysql> select *from account;                --查看结果，已经插入。此时可以在查看终端B
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   100.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> ^DBye                                --ctrl + \ or ctrl + d,终止终端

--终端B
mysql> select * from account;               --终端A崩溃前
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   100.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> select * from account;               --终端A崩溃后
+----+--------+--------+
| id | name   | blance |
+----+--------+--------+
```

```
|  1 | 张三   | 100.00 |
+----+--------+--------+
1 row in set (0.00 sec)


-- 实验二
--终端A
mysql> show variables like 'autocommit'; --开启默认提交
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.00 sec)

mysql> select * from account;
+----+--------+--------+
| id | name   | blance |
+----+--------+--------+
|  1 | 张三   | 100.00 |
+----+--------+--------+
1 row in set (0.00 sec)

mysql> insert into account values (2, '李四', 10000);
Query OK, 1 row affected (0.01 sec)

mysql> select *from account;   --数据已经插入
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   100.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> Aborted                 --异常终止

--终端B
mysql> select * from account;         --终端A崩溃前
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   100.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> select * from account;       --终端A崩溃后，并不影响，已经持久化。autocommit
起作用
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   100.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)
```

结论：

> - 只要输入begin或者start transaction，事务便必须要通过commit提交，才会持久化，与是否设置set autocommit无关。
> - 事务可以手动回滚，同时，当操作异常，MySQL会自动回滚
> - 对于`InnoDB`每一条`SQL`语言都默认封装成事务，自动提交。（select有特殊情况，因为`MySQL`有`MVCC`）
> - 从上面的例子，我们能看到事务本身的原子性(回滚)，持久性(commit)
> - 那么隔离性？一致性？

事务操作注意事项

> - 如果没有设置保存点，也可以回滚，只能回滚到事务的开始。直接使用 rollback(前提是事务还没有提交)
> - 如果一个事务被提交了（commit），则不可以回退（rollback）
> - 可以选择回退到哪个保存点
> - `InnoDB`支持事务，`MyISAM`不支持事务
> - 开始事务可以使`start transaction`或者`begin`

事务隔离级别

如何理解隔离性1

- MySQL服务可能会同时被多个客户端进程(线程)访问，访问的方式以事务方式进行
- 一个事务可能由多条SQL构成，也就意味着，任何一个事务，都有执行前，执行中，执行后的阶段。而所谓的原子性，其实就是让用户层，要么看到执行前，要么看到执行后。执行中出现问题，可以随时回滚。所以单个事务，对用户表现出来的特性，就是原子性。
- 但，毕竟所有事务都要有个执行过程，那么在多个事务各自执行多个SQL的时候，就还是有可能会出现互相影响的情况。比如：多个事务同时访问同一张表，甚至同一行数据。
- 就如同你妈妈给你说：你要么别学，要学就学到最好。至于你怎么学，中间有什么困难，你妈妈不关心。那么你的学习，对你妈妈来讲，就是原子的。那么你学习过程中，很容易受别人干扰，此时，就需要将你的学习隔离开，保证你的学习环境是健康的。
- 数据库中，为了保证事务执行过程中尽量不受干扰，就有了一个重要特征：隔离性
- 数据库中，允许事务受不同程度的干扰，就有了一种重要特征：隔离级别

隔离级别

- **读未提交【Read Uncommitted】**： 在该隔离级别，所有的事务都可以看到其他事务没有提交的执行结果。（实际生产中不可能使用这种隔离级别的），但是相当于没有任何隔离性，也会有很多并发问题，如脏读，幻读，不可重复读等，我们上面为了做实验方便，用的就是这个隔离性。
- **读提交【Read Committed】**：该隔离级别是大多数数据库的默认的隔离级别（不是 MySQL 默认的）。它满足了隔离的简单定义:一个事务只能看到其他的已经提交的事务所做的改变。这种隔离级别会引起不可重复读，即一个事务执行时，如果多次 select， 可能得到不同的结果。
- **可重复读【Repeatable Read】**： 这是 MySQL 默认的隔离级别，它确保同一个事务，在执行中，多次读取操作数据时，会看到同样的数据行。但是会有幻读问题。
- **串行化【Serializable】**: 这是事务的最高隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决了幻读的问题。它在每个读的数据行上面加上共享锁，。但是可能会导致超时和锁竞争（这种隔离级别太极端，实际生产基本不使用）

隔离级别如何实现：隔离，基本都是通过锁实现的，不同的隔离级别，锁的使用是不同的。常见有，表锁，行锁，读锁，写锁，间隙锁(GAP),Next-Key锁(GAP+行锁)等。不过，我们目前现有这个认识就行，先关注上层使用。

查看与设置隔离性

```
-- 查看
mysql> SELECT @@global.tx_isolation;    --查看全局隔级别
+-----------------------+
| @@global.tx_isolation |
+-----------------------+
| REPEATABLE-READ       |
+-----------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT @@session.tx_isolation;    --查看会话(当前)全局隔级别
+------------------------+
| @@session.tx_isolation |
+------------------------+
| REPEATABLE-READ        |
+------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT @@tx_isolation;            --默认同上
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
1 row in set, 1 warning (0.00 sec)

--设置
-- 设置当前会话 or 全局隔离级别语法
SET [SESSION | GLOBAL] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ
COMMITTED | REPEATABLE READ | SERIALIZABLE}

--设置当前会话隔离性，另起一个会话，看不多，只影响当前会话
mysql> set session transaction isolation level serializable; -- 串行化
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@global.tx_isolation;     --全局隔离性还是RR
+-----------------------+
| @@global.tx_isolation |
+-----------------------+
| REPEATABLE-READ       |
+-----------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT @@session.tx_isolation;    --会话隔离性成为串行化
+------------------------+
| @@session.tx_isolation |
+------------------------+
| SERIALIZABLE           |
+------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT @@tx_isolation;          --同上
+----------------+
| @@tx_isolation |
+----------------+
| SERIALIZABLE   |
```

```
+----------------+
1 row in set, 1 warning (0.00 sec)


--设置全局隔离性，另起一个会话，会被影响
mysql> set global transaction isolation level READ UNCOMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@global.tx_isolation;
+-----------------------+
| @@global.tx_isolation |
+-----------------------+
| READ-UNCOMMITTED      |
+-----------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT @@session.tx_isolation;
+------------------------+
| @@session.tx_isolation |
+------------------------+
| READ-UNCOMMITTED       |
+------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT @@tx_isolation;
+------------------+
| @@tx_isolation   |
+------------------+
| READ-UNCOMMITTED |
+------------------+
1 row in set, 1 warning (0.00 sec)

-- 注意，如果没有现象，关闭mysql客户端，重新连接。
```

读未提交【Read Uncommitted】

```
--几乎没有加锁，虽然效率高，但是问题太多，严重不建议采用
--终端A
-- 设置隔离级别为 读未提交
mysql> set global transaction isolation level read uncommitted;
Query OK, 0 rows affected (0.00 sec)

--重启客户端

mysql> select @@tx_isolation;
+------------------+
| @@tx_isolation   |
+------------------+
| READ-UNCOMMITTED |
+------------------+
1 row in set, 1 warning (0.00 sec)

mysql> select * from account;
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   100.00 |
```

```
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> begin;                              --开启事务
Query OK, 0 rows affected (0.00 sec)

mysql> update account set blance=123.0 where id=1;  --更新指定行
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

--没有commit哦！！！

--终端B
mysql> begin;
mysql> select * from account;
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   123.00 |             --读到终端A更新但是未commit的数据[insert，
delete同样]
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

--一个事务在执行中，读到另一个执行中事务的更新(或其他操作)但是未commit的数据，这种现象叫做脏读
(dirty read)
```

读提交【Read Committed】

```
-- 终端A
mysql> set global transaction isolation level read committed;
Query OK, 0 rows affected (0.00 sec)
--重启客户端
mysql> select * from account;   --查看当前数据
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   123.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> begin;                                  --手动开启事务，同步的开始终端B事务
Query OK, 0 rows affected (0.00 sec)

mysql> update account set blance=321.0 where id=1;   --更新张三数据
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

--切换终端到终端B，查看数据。

mysql> commit;                             --commit提交!
Query OK, 0 rows affected (0.01 sec)

--切换终端到终端B，再次查看数据。
```

```
--终端B
mysql> begin;                                --手动开启事务，和终端A一前一后
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account;                --终端A commit之前，查看不到
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   123.00 |                  --老的值
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

--终端A commit之后，看到了！
--but，此时还在当前事务中，并未commit，那么就造成了，同一个事务内，同样的读取，在不同的时间段(依旧还在事务操作中！)，读取到了不同的值，这种现象叫做不可重复读(non reapeatable read)！！（这个是问题吗？？）
mysql> select *from account;
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   321.00 |                  --新的值
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)
```

可重复读【Repeatable Read】

```
--终端A
mysql> set global transaction isolation level repeatable read;  --设置全局隔离级别RR
Query OK, 0 rows affected (0.01 sec)

--关闭终端重启

mysql> select @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |                        --隔离级别RR
+-----------------+
1 row in set, 1 warning (0.00 sec)

mysql> select *from account;   --查看当前数据
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   321.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> begin;                      --开启事务，同步的，终端B也开始事务
Query OK, 0 rows affected (0.00 sec)

mysql> update account set blance=4321.0 where id=1; --更新数据
Query OK, 1 row affected (0.00 sec)
```

```
Rows matched: 1  Changed: 1  Warnings: 0

--切换到终端B，查看另一个事务是否能看到

mysql> commit;   --提交事务

--切换终端到终端B，查看数据。

--终端B
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account;  --终端A中事务 commit之前，查看当前表中数据，数据未更新
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   321.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> select * from account; --终端A中事务 commit 之后，查看当前表中数据，数据未更新
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   321.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

--可以看到，在终端B中，事务无论什么时候进行查找，看到的结果都是一致的，这叫做可重复读！

mysql> commit;              --结束事务
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account;   --再次查看，看到最新的更新数据
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |  4321.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

-----------------------------------------------------------------------

--如果将上面的终端A中的update操作，改成insert操作，会有什么问题？？
--终端A
mysql> select *from account;
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |   321.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> begin;           --开启事务，终端B同步开启
```

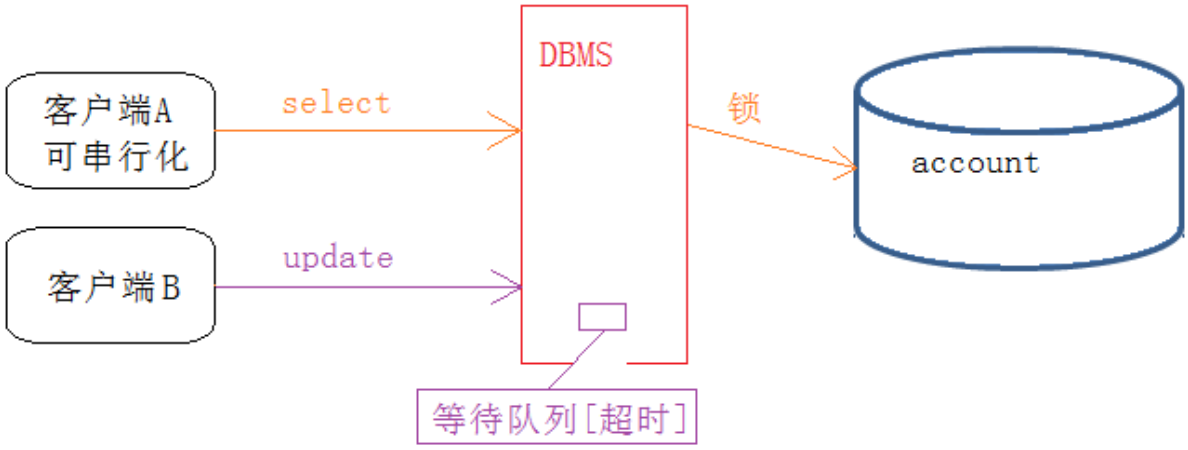
```
Query OK, 0 rows affected (0.00 sec)

mysql> insert into account (id,name,blance) values(3, '王五', 5432.0);
Query OK, 1 row affected (0.00 sec)

--切换到终端B，查看另一个事务是否能看到

mysql> commit; --提交事务
Query OK, 0 rows affected (0.00 sec)

--切换终端到终端B，查看数据。

mysql> select * from account;
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |  4321.00 |
|  2 | 李四   | 10000.00 |
|  3 | 王五   |  5432.00 |
+----+--------+----------+
3 rows in set (0.00 sec)

--终端B
mysql> begin;   --开启事务
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account; --终端A commit前 查看
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |  4321.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> select * from account; --终端A commit后 查看
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |  4321.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)

mysql> select * from account; --多次查看，发现终端A在对应事务中insert的数据，在终端B的事务周期中，也没有什么影响，也符合可重复的特点。但是，一般的数据库在可重复读情况的时候，无法屏蔽其他事务insert的数据(为什么？因为隔离性实现是对数据加锁完成的，而insert待插入的数据因为并不存在，那么一般加锁无法屏蔽这类问题),会造成虽然大部分内容是可重复读的，但是insert的数据在可重复读情况被读取出来，导致多次查找时，会多查找出来新的记录，就如同产生了幻觉。这种现象，叫做幻读(phantom read)。很明显，MySQL在RR级别的时候，是解决了幻读问题的(解决的方式是用Next-Key锁(GAP+行锁)解决的。这块比较难，有兴趣同学了解一下)。
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |  4321.00 |
|  2 | 李四   | 10000.00 |
+----+--------+----------+
2 rows in set (0.00 sec)
```

```
mysql> commit;   --结束事务
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account; --看到更新
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |  4321.00 |
|  2 | 李四   | 10000.00 |
|  3 | 王五   |  5432.00 |
+----+--------+----------+
3 rows in set (0.00 sec)
```

串行化【serializable】

```
--对所有操作全部加锁，进行串行化，不会有问题，但是只要串行化，效率很低，几乎完全不会被采用
--终端A
mysql> set global transaction isolation level serializable;
Query OK, 0 rows affected (0.00 sec)
mysql> select @@tx_isolation;
+----------------+
| @@tx_isolation |
+----------------+
| SERIALIZABLE   |
+----------------+
1 row in set, 1 warning (0.00 sec)
mysql> begin;                          --开启事务，终端B同步开启
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account; --两个读取不会串行化，共享锁
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |  4321.00 |
|  2 | 李四   | 10000.00 |
|  3 | 王五   |  5432.00 |
+----+--------+----------+
3 rows in set (0.00 sec)

mysql> update account set blance=1.00 where id=1; --终端A中有更新或者其他操作，会阻塞。直到终端B事务提交。
Query OK, 1 row affected (18.19 sec)
Rows matched: 1  Changed: 1  Warnings: 0

--终端B
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account;    --两个读取不会串行化
+----+--------+----------+
| id | name   | blance   |
+----+--------+----------+
|  1 | 张三   |  4321.00 |
|  2 | 李四   | 10000.00 |
|  3 | 王五   |  5432.00 |
+----+--------+----------+
```

```
3 rows in set (0.00 sec)

mysql> commit; --提交之后，终端A中的update才会提交。
Query OK, 0 rows affected (0.00 sec)
```



总结：

- 其中隔离级别越严格，安全性越高，但数据库的并发性能也就越低，往往需要在两者之间找一个平衡点。
- 不可重复读的重点是修改和删除：同样的条件, 你读取过的数据,再次读取出来发现值不一样了
  幻读的重点在于新增：同样的条件, 第1次和第2次读出来的记录数不一样
- 说明：`mysql`默认的隔离级别是可重复读,一般情况下不要修改
- 上面的例子可以看出，事务也有长短事务这样的概念。事务间互相影响，指的是事务在并行执行的时候，即都没有commit的时候，影响会比较大。

隔离级别	脏读	不可重复读	幻读	加锁读
读未提交 (read uncommited)	✔	✔	✔	不加锁
读已提交 (read commited)	✘	✔	✔	不加锁
可重复读 (repeatable read)	✘	✘	✘	不加锁
可串行化 (serializable)	✘	✘	✘	加锁

✔：会发生该问题
✘：不会发生该问题

一致性(Consistency)

- 事务执行的结果，必须使数据库从一个一致性状态，变到另一个一致性状态。当数据库只包含事务成功提交的结果时，数据库处于一致性状态。如果系统运行发生中断，某个事务尚未完成而被迫中断，而改未完成的事务对数据库所做的修改已被写入数据库，此时数据库就处于一种不正确（不一致）的状态。因此一致性是通过原子性来保证的。
- 其实一致性和用户的业务逻辑强相关，一般MySQL提供技术支持，但是一致性还是要用户业务逻辑做支撑，也就是，一致性，是由用户决定的。
- 而技术上，通过AID保证C

推荐阅读

```
https://www.jianshu.com/p/398d788e1083
https://tech.meituan.com/2014/08/20/innodb-lock.html
https://www.cnblogs.com/aspirant/p/9177978.html
```

备注：
基本上，了解了上面的知识，在MySQL事务使用上，肯定没有问题。不过，这块设计很优秀，也是面试中可能被问到的，一般学生，如果能说出上面的内容，就已经不错了。但是如果我们能更详细，更深入的谈论这个问题，那么对我们的面试与学习肯定是大大的有帮助。

不过接下来的内容，会比较难一些，听的不明白，也没有太大问题。
这块内容，也要结合同学们听课的情况，如果时间紧张，我们就不讲了。

如果有时间，可以现场给学生演示一下，在RR级别的时候，多个事务的update，多个事务的insert,多个事务的delete，是否会有加锁现象。

现象结果是，update，insert，delete之间是会有加锁现象的，但是select和这些操作是不冲突的。这就是通过读写锁(锁有行锁或者表锁)+MVCC完成隔离性。

试学内容-如何理解隔离性2

数据库并发的场景有三种：

- `读-读`：不存在任何问题，也不需要并发控制
- `读-写`：有线程安全问题，可能会造成事务隔离性问题，可能遇到脏读，幻读，不可重复读
- `写-写`：有线程安全问题，可能会存在更新丢失问题，比如第一类更新丢失，第二类更新丢失(后面补充)

读-写

多版本并发控制（`MVCC`）是一种用来解决 `读-写冲突` 的**无锁并发控制**

为事务分配单向增长的事务ID，为每个修改保存一个版本，版本与事务ID关联，读操作只读该事务开始前的数据库的快照。 所以 `MVCC` 可以为数据库解决以下问题

- 在并发读写数据库时，可以做到在读操作时不用阻塞写操作，写操作也不用阻塞读操作，提高了数据库并发读写的性能
- 同时还可以解决脏读，幻读，不可重复读等事务隔离问题，但不能解决更新丢失问题

理解 `MVCC` 需要知道三个前提知识：

- 3个记录隐藏字段
- `undo` 日志
- `Read View`

3个记录隐藏列字段

- `DB_TRX_ID`：6 byte，最近修改(`修改/插入`)事务ID，记录创建这条记录/最后一次修改该记录的事务ID
- `DB_ROLL_PTR`：7 byte，回滚指针，指向这条记录的上一个版本（简单理解成，指向历史版本就行，这些数据一般在 `undo log` 中）
- `DB_ROW_ID`：6 byte，隐含的自增ID（隐藏主键），如果数据表没有主键，`InnoDB` 会自动以 `DB_ROW_ID` 产生一个聚簇索引

- 补充：实际还有一个删除flag隐藏字段, 既记录被更新或删除并不代表真的删除，而是删除flag变了

假设测试表结构是：

```
mysql> create table if not exists student(
            name varchar(11) not null,
            age int not null
        );

mysql> insert into student (name, age) values ('张三', 28);
Query OK, 1 row affected (0.05 sec)

mysql> select * from student;
+--------+-----+
| name   | age |
+--------+-----+
| 张三   |  28 |
+--------+-----+
1 row in set (0.00 sec)
```

上面描述的意思是：

name	age	DB_TRX_ID(创建该记录的事务ID)	DB_ROW_ID(隐式主键)	DB_ROLL_PTR(回滚指针)
张三	28	null	1	null

我们目前并不知道创建该记录的事务ID，隐式主键，我们就默认设置成null，1。第一条记录也没有其他版本，我们设置回滚指针为null。

undo日志

这里不想细讲，但是有一件事情得说清楚，MySQL将来是以服务进程的方式，在内存中运行。我们之前所讲的所有机制：索引，事务，隔离性，日志等，都是在内存中完成的，即在MySQL内部的相关缓冲区中，保存相关数据，完成各种判断操作。然后在合适的时候，将相关数据刷新到磁盘当中的。

所以，我们这里理解undo log，简单理解成，就是MySQL中的一段内存缓冲区，用来保存日志数据的就行。

模拟MVCC

现在有一个事务10(仅仅为了好区分)，对student表中记录进行修改(update)：将name(张三)改成name(李四)。

- 事务10,因为要修改，所以要先给该记录加行锁。
- 修改前，现将改行记录拷贝到undo log中，所以，undo log中就有了一行副本数据。(原理就是写时拷贝)
- 所以现在MySQL中有两行同样的记录。现在修改原始记录中的name，改成 '李四'。并且修改原始记录的隐藏字段DB_TRX_ID为当前事务10的ID, 我们默认从10开始，之后递增。而原始记录的回滚指针DB_ROLL_PTR列，里面写入undo log中副本数据的地址，从而指向副本记录，既表示我的上一个版本就是它。
- 事务10提交，释放锁。

事务10 执行 完毕 后

name	age	DB_TRX_ID(修改该记录的事务ID)	DB_ROW_ID(隐式主键)	DB_ROLL_PTR(回滚指针)
李四	28	10	1	0x11223344(示意)

undo log

name	age	DB_TRX_ID(创建该记录的事务ID)	DB_ROW_ID(隐式主键)	DB_ROLL_PTR(回滚指针)
张三	28	null	1	null

备注：此时，最新的记录是'李四'那条记录。

现在又有一个事务11，对student表中记录进行修改(update)：将age(28)改成age(38)。

- 事务11,因为也要修改，所以要先给该记录加行锁。（该记录是那条？）
- 修改前，现将改行记录拷贝到undo log中，所以，undo log中就又有了一行副本数据。此时，新的副本，我们采用头插方式，插入undo log。
- 现在修改原始记录中的age，改成 38。并且修改原始记录的隐藏字段 `DB_TRX_ID` 为当前 事务11 的 ID。而原始记录的回滚指针 `DB_ROLL_PTR` 列，里面写入undo log中副本数据的地址，从而指向副本记录，既表示我的上一个版本就是它。
- 事务11提交，释放锁。

事务11 执行 完毕 后 最新数据

name	age	DB_TRX_ID(修改该记录的事务ID)	DB_ROW_ID(隐式主键)	DB_ROLL_PTR(回滚指针)
李四	38	11	1	0x11223366(示意)

undo log

name	age	DB_TRX_ID(修改该记录的事务ID)	DB_ROW_ID(隐式主键)	DB_ROLL_PTR(回滚指针)
李四	28	10	1	0x11223344(示意)

name	age	DB_TRX_ID(创建该记录的事务ID)	DB_ROW_ID(隐式主键)	DB_ROLL_PTR(回滚指针)
张三	28	null	1	null

这样，我们就有了一个基于链表记录的历史版本链。所谓的回滚，无非就是用历史数据，覆盖当前数据。

上面的一个一个版本，我们可以称之为一个一个的快照。

##一些思考

上面是以更新（`upadte`）主讲的,如果是`delete`呢？一样的，别忘了，删数据不是清空，而是设置flag为删除即可。也可以形成版本。

如果是`insert`呢？因为`insert`是插入，也就是之前没有数据，那么`insert`也就没有历史版本。但是一般为了回滚操作，insert的数据也是要被放入undo log中，如果当前事务commit了，那么这个undo log 的历史insert记录就可以被清空了。

总结一下，也就是我们可以理解成，`update`和`delete`可以形成版本链，`insert`暂时不考虑。

那么`select`呢？

首先，`select`不会对数据做任何修改，所以，为`select`维护多版本，没有意义。不过，此时有个问题，就是：

select读取，是读取最新的版本呢？还是读取历史版本？

当前读：读取最新的记录，就是当前读。增删改，都叫做当前读，select也有可能当前读，比如：select lock in share mode(共享锁)，select for update （这个好理解，我们后面不讨论）

快照读：读取历史版本(一般而言)，就叫做快照读。(这个我们后面重点讨论)

我们可以看到，在多个事务同时删改查的时候，都是当前读，是要加锁的。那同时有select过来，如果也要读取最新版(当前读)，那么也就需要加锁，这就是串行化。
但如果是快照读，读取历史版本的话，是不受加锁限制的。也就是可以并行执行！换言之，提高了效率，即MVCC的意义所在。

那么，是什么决定了，select是当前读，还是快照读呢？隔离级别！

那为什么要有隔离级别呢？
事务都是原子的。所以，无论如何，事务总有先有后。

但是经过上面的操作我们发现，事务从begin->CURD->commit，是有一个阶段的。也就是事务有执行前，执行中，执行后的阶段。但，不管怎么启动多个事务，总是有先有后的。

那么多个事务在执行中，CURD操作是会交织在一起的。那么，为了保证事务的“有先有后”，是不是应该让不同的事务看到它该看到的内容，这就是所谓的隔离性与隔离级别要解决的问题。

先来的事务，应不应该看到后来的事务所做的修改呢?

那么，如何保证，不同的事务，看到不同的内容呢？也就是如何如何实现隔离级别？

`Read View`

Read View就是事务进行`快照读`操作的时候生产的`读视图`(Read View)，在该事务执行的快照读的那一刻，会生成数据库系统当前的一个快照，记录并维护系统当前活跃事务的ID(当每个事务开启时，都会被分配一个ID, 这个ID是递增的，所以最新的事务，ID值越大)

`Read View`在`MySQL`源码中,就是一个类，本质是用来进行可见性判断的。 即当我们某个事务执行快照读的时候，对该记录创建一个`Read View`读视图，把它比作条件,用来判断当前事务能够看到哪个版本的数据，既可能是当前最新的数据，也有可能是该行记录的`undo log`里面的某个版本的数据。

下面是`ReadView`结构,但为了减少同学们负担，我们简化一下

```
class ReadView {
  // 省略...
 private:
  /** 高水位，大于等于这个ID的事务均不可见*/
  trx_id_t m_low_limit_id
```

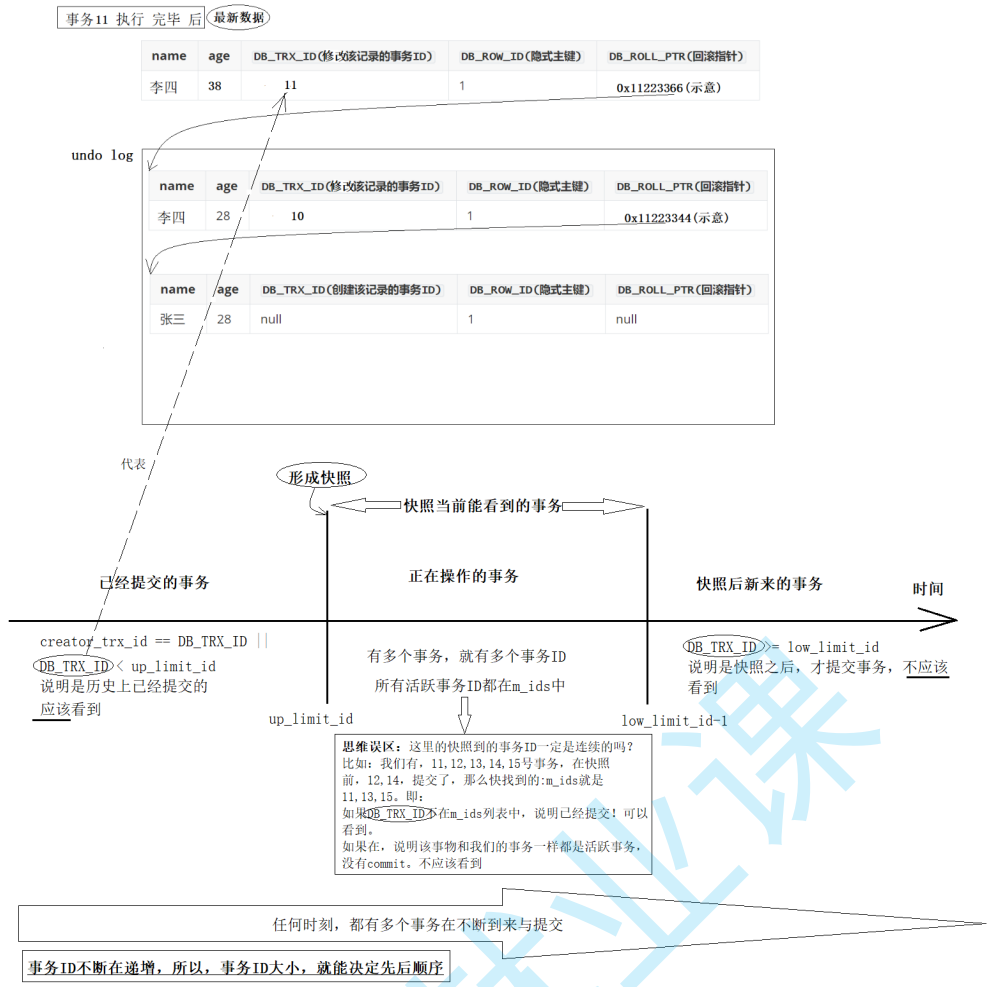
```
  /** 低水位：小于这个ID的事务均可见 */
  trx_id_t m_up_limit_id;

  /** 创建该 Read View 的事务ID*/
  trx_id_t m_creator_trx_id;

  /** 创建视图时的活跃事务id列表*/
  ids_t m_ids;

  /** 配合purge，标识该视图不需要小于m_low_limit_no的UNDO LOG，
   * 如果其他视图也不需要，则可以删除小于m_low_limit_no的UNDO LOG*/
  trx_id_t m_low_limit_no;

  /** 标记视图是否被关闭*/
  bool m_closed;

  // 省略...
};
```

```
m_ids;            //一张列表，用来维护Read View生成时刻，系统正活跃的事务ID
up_limit_id;      //记录m_ids列表中事务ID最小的ID(没有写错)
low_limit_id;     //ReadView生成时刻系统尚未分配的下一个事务ID，也就是目前已出现过的事务ID的最大值+1(也没有写错)
creator_trx_id    //创建该ReadView的事务ID
```

我们在实际读取数据版本链的时候，是能读取到每一个版本对应的事务ID的，即：当前记录的`DB_TRX_ID`。

那么，我们现在手里面有的东西就有，当前快照读的`ReadView`和版本链中的某一个记录的`DB_TRX_ID`。

所以现在的问题就是，当前快照读，应不应该读到当前版本记录。一张图，解决所有问题！



对应源码策略：

```
157         /** Check whether the changes by id are visible.
158         @param[in]      id      transaction id to check against the view
159         @param[in]      name    table name
160         @return whether the view sees the modifications of id. */
161         bool changes_visible(
162                 trx_id_t                id,
163                 const table_name_t&     name) const
164                 MY_ATTRIBUTE((warn_unused_result))
165         {
166                 ut_ad(id > 0);
167
168                 if (id < m_up_limit_id || id == m_creator_trx_id) {
169
170                         return(true);
171                 }
172
173                 check_trx_id_sanity(id, name);
174
175                 if (id >= m_low_limit_id) {
176
177                         return(false);
178
179                 } else if (m_ids.empty()) {
180
181                         return(true);
182                 }
183
184                 const ids_t::value_type*        p = m_ids.data();
185
186                 return(!std::binary_search(p, p + m_ids.size(), id));
187         }
```


如果查到不应该看到当前版本，接下来就是遍历下一个版本，直到符合条件，即可以看到。上面的`readview`是当你进行select的时候，会自动形成。

整体流程

假设当前有条记录：

name	age	DB_TRX_ID(创建该记录的事务ID)	DB_ROW_ID(隐式主键)	DB_ROLL_PTR(回滚指针)
张三	28	null	1	null

事务操作：

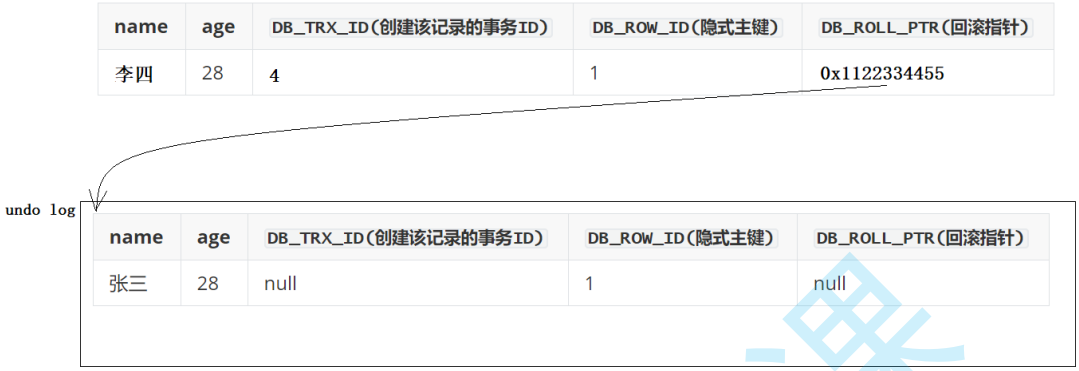
事务1 [id=1]	事务2 [id=2]	事务3 [id=3]	事务4 [id=4]
事务开始	事务开始	事务开始	事务开始
...	...	...	修改且已提交
进行中	快照读	进行中	
...	...	...	

- 事务4：修改name(张三) 变成name(李四)

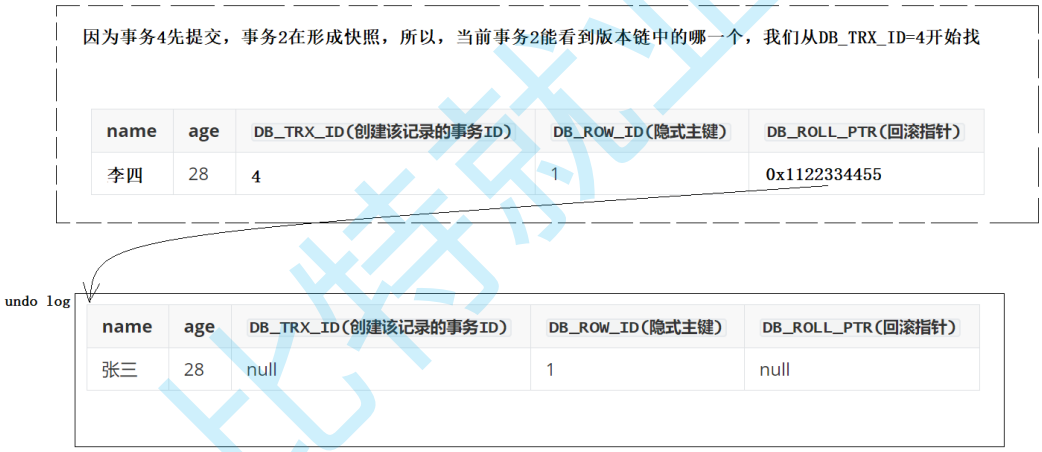
- 当`事务2`对某行数据执行了`快照读`，数据库为该行数据生成一个`Read View`读视图

```
//事务2的 Read View

m_ids;            // 1,3
up_limit_id;      // 1
low_limit_id;     // 4 + 1 = 5，原因：ReadView生成时刻，系统尚未分配的下一个事务ID
creator_trx_id    // 2
```

此时版本链是：



- 只有事务4修改过该行记录，并在事务2执行快照读前，就提交了事务。



- 我们的事务2在快照读该行记录的时候，就会拿该行记录的`DB_TRX_ID`去跟`up_limit_id,low_limit_id和活跃事务ID列表(trx_list)`进行比较，判断当前事务2能看到该记录的版本。

```
//事务2的 Read View
m_ids;            // 1,3
up_limit_id;      // 1
low_limit_id;     // 4 + 1 = 5，原因：ReadView生成时刻，系统尚未分配的下一个事务ID
creator_trx_id    // 2

//事务4提交的记录对应的事务ID
DB_TRX_ID=4

//比较步骤
DB_TRX_ID（4）< up_limit_id（1） ?  不小于，下一步
DB_TRX_ID（4）>= low_limit_id(5) ? 不大于，下一步
m_ids.contains(DB_TRX_ID) ? 不包含，说明，事务4不在当前的活跃事务中。
```

```
//结论
故，事务4的更改，应该看到。
所以事务2能读到的最新数据记录是事务4所提交的版本，而事务4提交的版本也是全局角度上最新的版本
```

RR 与 RC的本质区别

当前读和快照读在RR级别下的区别

下面的代码经过测试，是完全没有问题的。要不要现场测试，主要看上课的时间允不允许

> `select * from user lock in share mode`,以加共享锁方式进行读取，对应的就是当前读。此处只作为测试使用，不重讲。

测试表：

```
 --设置RR模式下测试
mysql> set global transaction isolation level REPEATABLE READ;
Query OK, 0 rows affected (0.00 sec)

--重启终端

mysql> select @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
1 row in set, 1 warning (0.00 sec)

--依旧用之前的表
create table if not exists account(
    id int primary key,
    name varchar(50) not null default '',
    blance decimal(10,2) not null default 0.0
)ENGINE=InnoDB DEFAULT CHARSET=UTF8;

--插入一条记录，用来测试
mysql> insert into user (id, age, name) values (1, 15,'黄蓉');
Query OK, 1 row affected (0.00 sec)
```

测试用例1-表1：

事务A操作	事务A描述	事务B描述	事务B操作
`begin`	开启事务	开启事务	`begin`
`select * from user`	快照读(无影响)查询	快照读查询	`select * from user`
`update user set age=18 where id=1;`	更新age=18	-	-
`commit`	提交事务	-	-
		`select` 快照读,没有读到age=18	`select * from user`
		`select lock in share mode`当前读, 读到age=18	`select * from user lock in share mode`

测试用例2-表2：

事务A操作	事务A描述	事务B描述	事务B操作
`begin`	开启事务	开启事务	`begin`
`select * from user`	快照读，查到age=18	-	-
`update user set age=28 where id=1;`	更新age=28	-	-
`commit`	提交事务	-	-
		`select` 快照读 age=28	`select * from user`
		`select lock in share mode`当前读 age=28	`select * from user lock in share mode`

- 用例1与用例2：唯一区别仅仅是`表1`的事务B在事务A修改age前`快照读`过一次age数据
- 而`表2`的事务B在事务A修改age前没有进行过快照读。

结论：

- 事务中快照读的结果是非常依赖该事务首次出现快照读的地方，即某个事务中首次出现快照读，决定该事务后续快照读结果的能力
- delete同样如此

RR 与 RC的本质区别

- 正是Read View生成时机的不同，从而造成RC,RR级别下快照读的结果的不同
- 在RR级别下的某个事务的对某条记录的第一次快照读会创建一个快照及Read View, 将当前系统活跃的其他事务记录起来

- 此后在调用快照读的时候，还是使用的是同一个Read View，所以只要当前事务在其他事务提交更新之前使用过快照读，那么之后的快照读使用的都是同一个Read View，所以对之后的修改不可见；
- 即RR级别下，快照读生成Read View时，Read View会记录此时所有其他活动事务的快照，这些事务的修改对于当前事务都是不可见的。而早于Read View创建的事务所做的修改均是可见
- 而在RC级别下的，事务中，每次快照读都会新生成一个快照和Read View, 这就是我们在RC级别下的事务中可以看到别的事务提交的更新的原因
- 总之在RC隔离级别下，是每个快照读都会生成并获取最新的Read View；而在RR隔离级别下，则是同一个事务中的第一个快照读才会创建Read View, 之后的快照读获取的都是同一个Read View。
- 正是RC每次快照读，都会形成Read View，所以，RC才会有不可重复读问题。

读-读

- 不讨论

写-写

- 现阶段，直接理解成都是当前读，当前不做深究

推荐阅读

```
关于这块，有很好的文章，推荐大家阅读
https://blog.csdn.net/SnailMann/article/details/94724197
https://www.cnblogs.com/f-ck-need-u/archive/2018/05/08/9010872.html
https://blog.csdn.net/chenghan_yang/article/details/97630626
```