# 第13章_事务基础知识

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网： http://www.atguigu.com

## 1. 数据库事务概述

事务是数据库区别于文件系统的重要特性之一，当有了事务就会让数据库始终保持`一致性`，同时还能通过事务的机制`恢复到某个时间点`，这样可以保证已提交到数据库的修改不会因为系统崩溃而丢失

### 1.1 存储引擎支持情况

`SHOW ENGINES`命令来查看当前 MySQL 支持的存储引擎都有哪些，以及这些存储引擎是否支持事务。

```
mysql> show engines;
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
| Engine             | Support | Comment                                                        | Transactions | XA   | Savepoints |
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
| FEDERATED          | NO      | Federated MySQL storage engine                                 | NULL         | NULL | NULL       |
| MEMORY             | YES     | Hash based, stored in memory, useful for temporary tables      | NO           | NO   | NO         |
| InnoDB             | DEFAULT | Supports transactions, row-level locking, and foreign keys     | YES          | YES  | YES        |
| PERFORMANCE_SCHEMA | YES     | Performance Schema                                             | NO           | NO   | NO         |
| MyISAM             | YES     | MyISAM storage engine                                          | NO           | NO   | NO         |
| MRG_MYISAM         | YES     | Collection of identical MyISAM tables                          | NO           | NO   | NO         |
| BLACKHOLE          | YES     | /dev/null storage engine (anything you write to it disappears) | NO           | NO   | NO         |
| CSV                | YES     | CSV storage engine                                             | NO           | NO   | NO         |
| ARCHIVE            | YES     | Archive storage engine                                         | NO           | NO   | NO         |
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
9 rows in set (0.00 sec)
```

能看出在 MySQL 中，只有InnoDB 是支持事务的。

### 1.2 基本概念

**事务**：一组逻辑操作单元，使数据从一种状态变换到另一种状态。

**事务处理的原则**：保证所有事务都作为`一个工作单元`来执行，即使出现了故障，都不能改变这种执行方式。当在一个事务中执行多个操作时，要么所有的事务都被提交(`commit`)，那么这些修改就`永久`地保存下来；要么数据库管理系统将`放弃`所作的所有`修改`，整个事务回滚(`rollback`)到最初状态。

```
#案例：AA用户给BB用户转账100
UPDATE accounts SET money = money - 50 WHERE NAME = 'AA';
#服务器宕机
UPDATE accounts SET money = money + 50 WHERE NAME = 'BB';
```

### 1.3 事务的ACID特性

- 原子性（atomicity）：

  原子性是指事务是一个不可分割的工作单位，要么全部提交，要么全部失败回滚。即要么转账成功，要么转账失败，是不存在中间的状态。如果无法保证原子性会怎么样?就会出现数据不一致的情形，A账户减去100元，而B账户增加100元操作失败，系统将无故丢失100元。

- 一致性（consistency）：

  （国内很多网站上对一致性的阐述有误，具体你可以参考 Wikipedia 对 Consistency的阐述）

  根据定义，一致性是指事务执行前后，数据从一个`合法性状态`变换到另外一个`合法性状态`。这种状态是`语义上`的而不是语法上的，跟具体的业务有关。

那什么是合法的数据状态呢？满足`预定的约束`的状态就叫做合法的状态。通俗一点，这状态是由你自己来定义的（比如满足现实世界中的约束）。满足这个状态，数据就是一致的，不满足这个状态，数据就是不一致的！如果事务中的某个操作失败了，系统就会自动撤销当前正在执行的事务，返回到事务操作之前的状态。

**举例1:**A账户有200元，转账300元出去，此时A账户余额为-100元。你自然就发现了此时数据是不一致的，为什么呢?因为你定义了一个状态，余额这列必须>=0。

**举例2:**A账户200元。转账50元给B账户，A账户的钱扣了，但是B账户因为各种意外，余额并没有增加。你也知道此时数据是不一致的，为什么呢?因为你定义了一个状态，要求A+B的总余额必须不变。

**举例3:**在数据表中将`姓名`字段设置为`唯一性约束`，这时当事务进行提交或者事务发生回滚的时候，如果数据表中的姓名不唯一，就破坏了事务的一致性要求。

- 隔离型（isolation）：

(可以联系JUC中的临界区的概念，为了避免各个线程都执行临界区的代码，必须加`synchronized`)

事务的隔离性是指一个事务的执行`不能被其他事务干扰`，即一个事务内部的操作及使用的数据对`并发`的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

如果无法保证隔离性会怎么样？假设A账户有200元，B账户0元。A账户往B账户转账两次，每次金额为50元，分别在两个事务中执行。如果无法保证隔离性，会出现下面的情形：

```
1 UPDATE accounts SET money = money - 50 WHERE NAME = 'AA';
2 UPDATE accounts SET money = money + 50 WHERE NAME = 'BB';
```

> 根据图解，发现出现了线程安全的问题，从而导致转账前后总金额不一致的情况


- 持久性（durability）：

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是`永久性的`，接下来的其他操作和数据库故障不应该对其有任何影响。

持久性是通过`事务日志`来保证的。日志包括了`重做日志`和`回滚日志`。当我们通过事务对数据进行修改的时候，首先会将数据库的变化信息记录到重做日志中，然后再对数据库中对应的行进行修改。这样做的好处是，即使数据库系统崩溃，数据库重启后也能找到没有更新到数据库系统中的重做日志，重新执行，从而使事务具有持久性。

> 总结

> ACID是事务的四大特性，在这四个特性中，原子性是基础，隔离性是手段，一致性是约束条件，而持久性是目的。
>
> 数据库事务，其实就是数据库设计者为了方便起见，把需要保证原子性、隔离性、一致性和持久性的一个或多个数据库操作称为一个事务。一句话，事务就是ACID

## 1.4 事务的状态

我们现在知道`事务`是一个抽象的概念，它其实对应着一个或多个数据库操作，MySQL根据这些操作所执行的不同阶段把`事务`大致划分成几个状态：

- 活动的（active）

  事务对应的数据库操作正在执行过程中时，我们就说该事务处在`活动的`状态。比如转账的事务（两条DML）在执行~

- 部分提交的（partially committed）

  当事务中的最后一个操作执行完成，但由于操作都在内存中执行，所造成的影响并`没有刷新到磁盘`时，我们就说该事务处在`部分提交的`状态。比如转账的事务执行完成，但是还没有进行提交

- 失败的（failed）

  当事务处在`活动的`或者`部分提交的`状态时，可能遇到了某些错误（数据库自身的错误、操作系统错误或者直接断电等）而无法继续执行，或者人为的停止当前事务的执行，我们就说该事务处在`失败的`状态。比如正在转账时，银行突然断电了，事务就会被停止。

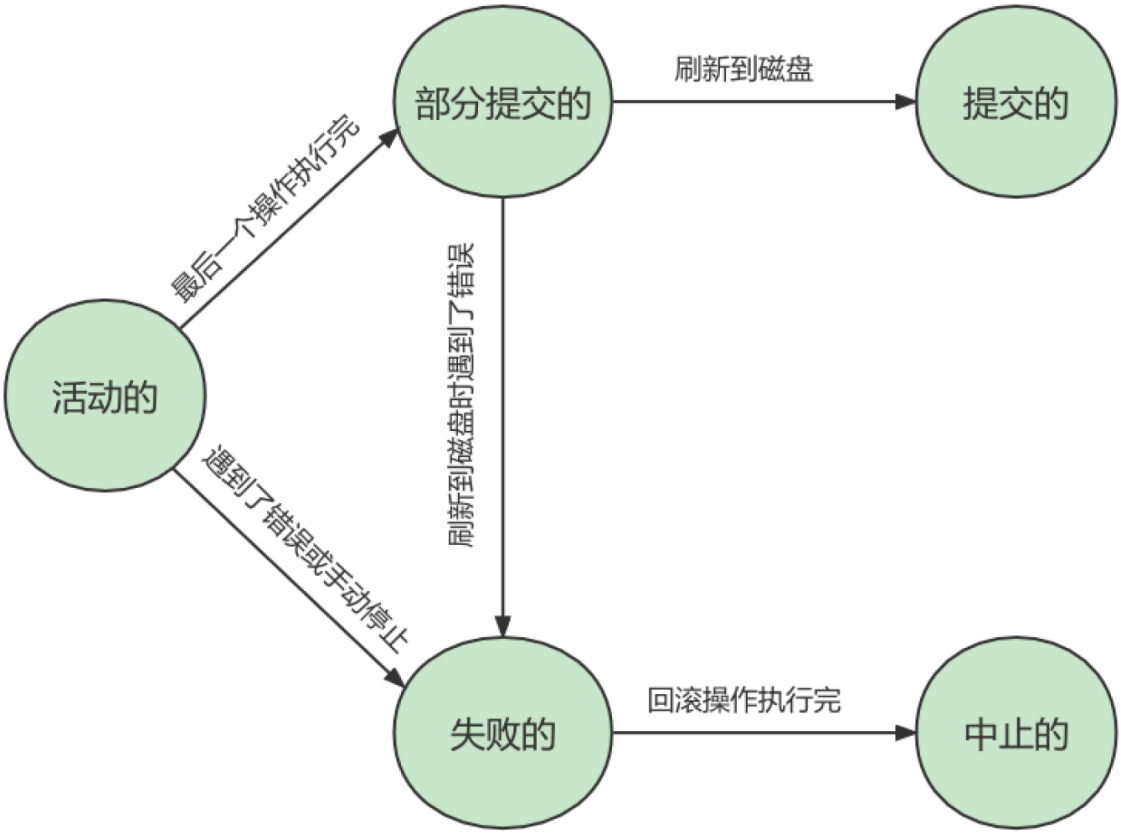
- 中止的（aborted）

  如果事务执行了一部分而变为`失败的`状态，那么就需要把已经修改的事务中的操作还原到事务执行前的状态。换句话说，就是要撤销失败事务对当前数据库造成的影响。我们把这个撤销的过程称之为`回滚`。当`回滚`操作执行完毕时，也就是数据库恢复到了执行事务之前的状态，我们就说该事务处在了`中止的`状态。比如当事务执行失败后，需要进行回滚，回滚完毕后的状态就是中止态
  举例：

  ```
  UPDATE accounts SET money = money - 50 WHERE NAME = 'AA';
  UPDATE accounts SET money = money + 50 WHERE NAME = 'BB';
  ```

- 提交的（committed）
  当一个处在`部分提交的`状态的事务将修改过的数据都`同步到磁盘`上之后，我们就可以说该事务处在了`提交的`状态。

一个基本的状态转换图如下所示：




图中可见，只有当事务处于`提交的`或者`中止的`状态时，一个事务的生命周期才算是结束了。对于已经提交的事务来说，该事务对数据库所做的修改将永久生效，对于处于中止状态的事务，该事务对数据库所做的所有修改被回滚到没执行该事务之前的状态。

# 2.如何使用事务

使用事务有两种方式，分别为`显式事务`和`隐式事务`。

## 2.1显式事务

**事务的完成过程**

- 步骤1：开启事务：
- 步骤2：一系列的DML操作
- ...
- 步骤3：事务结束的状态：提交的状态（COMMIT）、中止的状态（ROLLBACK）

**步骤1：** `START TRANSACTION`或者`BEGIN`，作用是显式开启一个事务。

```
mysql> BEGIN;
#或者
mysql> START TRANSACTION;
```

`START TRANSACTION`语句相较于`BEGIN`特别之处在于，后边能跟随几个`修饰符`：

1. `READ ONLY`：标识当前事务是一个`只读事务`，也就是属于该事务的数据库操作只能读取数据，而不能修改数据。

   > 补充:只读事务中只是不允许修改那些其他事务也能访问到的表中的数据，对于临时表来说(使用CREATE TMEPORARY TABLE创建的表)，由于它们只能在当前会话中可见，所以只读事务其实也是可以对临时表进行增、删、改操作的

2. `READ WRITE`：默认，标识当前事务是一个`读写事务`，也就是属于该事务的数据库操作既可以读取数据，也可以修改数据。
3. `WITH CONSISTENT SNAPSHOT`：启动一致性读。

   举例：

```
START TRANSACTION READ ONLY;#开启一个只读事务
START TRANSACTION READ ONLY,WITH CONSISTENT SNAPSHOT;#开启只读事多和一致性读
START TRANSACTION READ WRITE,WITH CONSISTENT SNAPSHOT;#开启读写事务和一致性读
```

> 注意：
>
> `READ ONLY`和`READ WRITE`是用来设置所谓的`事务访问模式`的，就是以只读还是读写的方式来访问数据库中的数据，一个事务的访问模式不能同时既设置为只读的又设置为读写的，所以不能同时把`READ ONLY`和`READ WRITE`放到`START TRANSACTION`语句后边
>
> 如果不显式指定事务的访问模式，那么该事务的访问模式就是读写模式。

**步骤2：**一系列事务中的操作（主要是DML，不含DDL）

**步骤3：**提交事务或中止事务（即回滚事务）

```
#提交事务。当提交事务后，对数据库的修改是永久性的。
mysql> COMMIT;
#回滚事务。即撤销正在进行的所有没有提交的修改
mysql> ROLLBACK;
#将事务回滚到某个保存点。
mysql> ROLLBACK TO [SAVEPOINT]
```

其中关于SAVEPOINT相关操作有：

```
 #在事务中创建保存点，方便后续针对保存点进行回滚。一个事务中可么存在多个保存点
SAVEPOINT 保存点名称;
#删除某个保存点
RELEASE SAVEPOINT保存点名称;
```

## 2.2隐式事务

MySQL中有一个系统变量`autocommit`：

```
mysql> SHOW VARIABLES LIKE 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.01 sec)
```

默认情况下，如果不显式的使用`START TRANSACTION`或者`BEGIN`语句开启一个事务，那么每一条语句都算是一个独立的事务，这种特性称之为事务的`自动提交`。也就是说，不以`START TRANSACTION`或者`BEGIN`语句显式的开启一个事务，那么下边这两条语句就相当于放到两个独立的事务中去执行

```
# 假设此时autocommit是默认值
UPDATE account SET balance = balance - 10 WHERE id = 1; #此时这条DML操作是一个独立的事务
UPDATE account SET balance = balance + 10 WHERE id = 2; #此时这条DML操作是一个独立的事务
```

当然，如果我们想关闭这种`自动提交`的功能，可以使用下边两种方法之一：

- 在autocommit为true的情况下，显式的的使用`START TRANSACTION`或者`BEGIN`语句开启一个事务。这样在本次事务提交或者回滚前会暂时关闭掉自动提交的功能。

```
START TRANSACTION;

UPDATE account SET balance = balance - 10 WHERE id = 1;

UPDATE account SET balance = balance + 10 WHERE id = 2;

COMMIT; #或rollback;
```

- 把系统变量 autocommit的值设置为 OFF，就像这样：

```
SET autocommit = OFF;
#或
SET autocommit = 0;
#或
SET autocommit = False;
```

```
SET autocommit = FALSE; #针对于DML操作是有效的，对DDL操作是无效的。

UPDATE account SET balance = balance - 10 WHERE id = 1;

UPDATE account SET balance = balance + 10 WHERE id = 2;

COMMIT; #或rollback;
```

这样的话，写入的多条语句就算是属于同一个事务了，直到我们显式的写出`COMMIT`语句来把这个事务提交掉，或者显式的写出`ROLLBACK`语句来把这个事务回滚掉。

> 补充: Oracle 默认不自动提交，需要手写COMMIT命令，而MySQL 默认自动提交。

## 2.3隐式提交数据的情况

- **数据定义语言（Data definition language，缩写为： DDL）**

  数据库对象，指的就是`数据库`、`表`、`视图`、`存储过程`等结构。当使用`CREATE`、`ALTER`、`DROP`等语句去修改数据库对象时，就会隐式的提交前边语句所属于的事务。即:

```
BEGIN;

SELECT ... #事务中的一条语句
UPDATE ...#事务中的一条语句
...  #事务中的其它语句

CREATE TABLE ...# 此语句会隐式的提交前边语句所属于的事务
```

- **隐式使用或修改mysql数据库中的表**

  当使用`ALTER USER`、`CREATE USER`、`DROP USER`、`GRANT`、`RENAME USER`、`REVOKE`、`SET PASSWORD`等语句时也会隐式的提交前边语句所属于的事务。

- **事务控制或关于锁定的语句**

  1. 当我们在一个事务还没提交或者回滚时就又使用`START TRANSACTION`或者`BEGIN`语句开启了另一个事务时，会隐式的提交上一个事务。即：

     ```
     BEGIN;

     SELECT ... #事务中的一条语句
     UPDATE ... #事务中的一条语句
     ...        #事务中的其它语句

     BEGIN;    #此语句会隐式的提交前面语句所属于的事务
     ```

  2. 当前的`autocommit`系统变量的值为`OFF`，我们手动把它调为`ON`时，也会隐式的提交前边语句所属的事务。

  3. 使用`LOCK TABLES`、`UNLOCK TABLES`等关于锁定的语句也会`隐式的提交`前边语句所属的事务。

- 加载数据的语句

  使用`LOAD DATA`语句来批量往数据库中导入数据时，也会`隐式的提交`前边语句所属的事务。

- 关于MySQL复制的一些语句

  使用`START SLAVE`、`STOP SLAVE`、`RESET SLAVE`、`CHANGE MASTER TO`等语句时会`隐式的提交`前边语句所属的事务。

- 其它的一些语句

  使用`ANALYZE TABLE`、`CACHE INDEX`、`CHECK TABLE`、`FLUSH`、`LOAD INDEX INTO CACHE`、`OPTIMIZE TABLE`、`REPAIR TABLE`、`RESET`等语句也会隐式的提交前边语句所属的事务。

## 2.4使用举例 1：提交与回滚

我们看下在 MySQL的默认状态下，下面这个事务最后的处理结果是什么。

先创建user3表

```
USE atguigudb2;
CREATE TABLE user3(name VARCHAR(15) PRIMARY KEY);
```

**情况1：**

```
CREATE TABLE user(name varchar(20), PRIMARY KEY (name)) ENGINE=InnoDB;

BEGIN;
INSERT INTO user SELECT '张三 ';  #此时不会自动提交数据
COMMIT;

BEGIN; #开启一个新的事务
INSERT INTO user SELECT '李四 '; #此时不会自动提交数据
INSERT INTO user SELECT '李四 '; #受主键的影响，不能添加成功
ROLLBACK;

SELECT * FROM user;
```

运行结果（1行数据）：

```
mysql> commit;
Query OK, 0 rows affected (0.00秒)
mysql> BEGIN;
Query OK, 0 rows affected (0.00秒)
mysql> INSERT INTO user SELECT '李四 ';
Query OK, 1 rows affected (0.00秒)
mysql> INSERT INTO user SELECT '李四 ';
Duplicate entry '李四 ' for key 'user.PRIMARY'
mysql> ROLLBACK;
Query OK, 0 rows affected (0.01秒)

mysql> select * from user;
+--------+
| name   |
+--------+
| 张三    |
+--------+
1行于数据集 (0.01秒)
```

**情况2:**

```
CREATE TABLE user (name varchar(20), PRIMARY KEY (name)) ENGINE=InnoDB;

BEGIN;
INSERT INTO user SELECT '张三 '; #此时不会自动提交数据
COMMIT;

INSERT INTO user SELECT '李四 '; # 默认情况下(即autocommit为true)，DML操作也会自动提交数据。
INSERT INTO user SELECT '李四 '; # 事务的失败的状态
ROLLBACK;
```

运行结果（2行数据）：

```
mysql> SELECT * FROM user;
+--------+
| name   |
+--------+
| 张三    |
| 李四    |
+--------+
2行于数据集 (0.01秒)
```

```
BEGIN;
INSERT INTO user3 VALUES('张三'); #此时不会自动提交数据
COMMIT;

BEGIN; #开启一个新的事务
INSERT INTO user3 VALUES('李四'); #此时不会自动提交数据
INSERT INTO user3 VALUES('李四'); #受主键的影响，不能添加成功
ROLLBACK;

SELECT * FROM user3;
/*
+--------+
| NAME   |
+--------+
| 张三   |
+--------+
*/
```

**情况3:**

```
CREATE TABLE user(name varchar(255), PRIMARY KEY (name)) ENGINE=InnoDB;
SET @@completion_type = 1;
BEGIN;
INSERT INTO user SELECT '张三 ';
COMMIT;

INSERT INTO user SELECT '李四 ';
INSERT INTO user SELECT '李四 ';
ROLLBACK;

SELECT * FROM user;
```

运行结果（1行数据）：

```
mysql> SELECT * FROM user;
+--------+
| name   |
+--------+
|张三     |
+--------+
1行于数据集 (0.01秒)
```

你能能看到相同的SQL代码，只是在事务开始之前设置了SET @@completion_type = 1;结果就和第一次处理的一样，只有一个“张三”。这是为什么呢?

这里讲解下 MySQL中completion_type参数的作用，实际上这个参数有3种可能:

- `completion=0`，这是`默认情况`。当执行COMNIT的时候会提交事务，在执行下一个事务时，还需要使`START TRANSACTION`或者`BEGIN`来开启。
- `completion=1`，这种情况下，当提交事务后，相当于执行了`COMMIT AND CHAIN`，也就是开启一个链式事务，即提交事务之后会开启一个相同隔离级别的事务。
- `completion=2`，这种情况下`CONMMIT=COMMIT AND RELEASE`，也就是提交后，会自动与服务器断开连接

> 当我们设置 autocommit=0 时，不论是否采用 START TRANSACTION 或者 BEGIN 的方式来开启事务，都需要用 COMMIT 进行提交，让事务生效，使用 ROLLBACK 对事务进行回滚。
>
> 当我们设置 autocommit=1 时，每条 SQL 语句都会自动进行提交。 不过这时，如果你采用 START TRANSACTION 或者 BEGIN 的方式来显式地开启事务，那么这个事务只有在 COMMIT 时才会生效， 在 ROLLBACK 时才会回滚。

## 2.5使用举例 2：测试不支持事务的 engine

1. 创建测试的表

```
USE atguigudb3;
#举例2：体会INNODB 和 MyISAM

CREATE TABLE test1(i INT) ENGINE = INNODB;

CREATE TABLE test2(i INT) ENGINE = MYISAM;
```

2. 针对于innodb表,ROLLBACK 会生效

```
BEGIN
INSERT INTO test1 VALUES (1);
ROLLBACK;

# 执行完，发现表为空，说明回滚成功~
SELECT * FROM test1;
```

3. 针对于myISAM表:不支持事务，`BEGIN`、`ROLLBACK`这些都会失效

```
BEGIN
INSERT INTO test2 VALUES (1);
ROLLBACK;

# 执行完，发现表中有上面插入的记录，说明MyISAM不支持事务~
SELECT * FROM test2;
```

## 2.6使用举例 3：SAVEPOINT

1. 创建测试表，并简单测试

```sql
CREATE TABLE user3(name VARCHAR(15),balance DECIMAL(10,2));

BEGIN
INSERT INTO user3(name,balance) VALUES('张三',1000);
COMMIT;

# 执行完，发现表中有上面插入的记录，说明默认创建的表是InnoDB的~
SELECT * FROM user3;
```

2. 测试`SAVEPOINT`

```sql
# 开启事务
BEGIN;
UPDATE user3 SET balance = balance - 100 WHERE NAME = '张三';

UPDATE user3 SET balance = balance - 100 WHERE NAME = '张三';

#设置保存点（类似于虚拟机的快照）
SAVEPOINT s1;

UPDATE user3 SET balance = balance + 1 WHERE NAME = '张三';

#回滚到保存点
ROLLBACK TO s1;

# 执行完，发现balance=800，说明回滚到保存点s1成功~
SELECT * FROM user3;

# 由于我们还么有commit，所以本次可以 对此次事务彻底回滚~
ROLLBACK; #回滚操作

# 执行完，发现balance=1000，说明回滚成功~
SELECT * FROM user3;
```

# 3.事务隔离级别

MySQL是一个`客户端 / 服务器`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，就可以称为一个会话（`Session`）。每个客户端都可以在自己的会话中向服务器发出请求语句，一个请求语句可能是某个事务的一部分，也就是对于服务器来说可能同时处理多个事务。事务有`隔离性`的特性，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。但是这样对`性能影响太大`，我们既想保持事务的隔离性，又想让服务器在处理访问同一数据的多个事务时`性能尽量高些`，那就看二者如何权衡取舍了。

## 3.1数据准备

我们需要创建一个表：

```
CREATE TABLE student (
    studentno INT,
    name VARCHAR(20),
    class varchar(20),
    PRIMARY KEY (studentno)
) Engine=InnoDB CHARSET=utf8;
```

然后向这个表里插入一条数据：

```
INSERT INTO student VALUES(1, '小谷', '1班');
```

现在表里的数据就是这样的：

```
mysql> select * from student;
+-----------+--------+-------+
| studentno | name   | class |
+-----------+--------+-------+
|         1 | 小谷    | 1班   |
+-----------+--------+-------+
1 row in set (0.00 sec)
```

## 3.2数据并发问题

针对事务的隔离性和并发性，我们怎么做取舍呢？先看一下访问相同数据的事务在`不保证串行执行`（也就是执行完一个再执行另一个）的情况下可能会出现哪些问题：

### 1. 脏写（Dirty Write）

对于两个事务 SessionA、SessionB，如果事务SessionA`修改了`另一个`未提交`事务SessionB`修改过`的数据，那就意味着发生了`脏写`

脏写示意图

| 发生时间编号 | Session A | Session B |
| --- | --- | --- |
| ① | BEGIN; | |
| ② | | BEGIN; |
| ③ | | UPDATE student SET name='李四' WHERE studentno= 1; |
| ④ | UPDATE student SET name='张三' WHERE studentno =1 ; | |
| ⑤ | COMMIT; | |
| ⑥ | | ROLLBACK; |

Session A和Session B各开启了一个事务，Session B中的事务先将studentno列为1的记录的name列更新为李四，然后Session A中的事务接着又把这条studentno列为1的记录的name列更新为张三'。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。

这时Session A中的事务就没有效果了，明明把数据更新了，最后也提交事务了，最后看到的数据什么变化也没有。这里大家对事务的隔离级比较了解的话，会发现默认隔离级别下，上面SessionA中的更新语句会处于等待状态，这里只是跟大家说明一下会出现这样现象。

## 2. 脏读（Dirty Read）

对于两个事务 Session A、Session B，Session A `读取` 了已经被 Session B `更新` 但还 `没有被提交` 的字段。之后若 Session B `回滚`，Session A `读取` 的内容就是 `临时且无效` 的。

脏读示意图

| 发生时间编号 | Session A | Session B |
| --- | --- | --- |
| ① | BEGIN; | |
| ② | | BEGIN; |
| ③ | | UPDATE student SET name='张三' WHERE studentno= 1; |
| ④ | SELECT * FROM student WHERE studentno = 1;<br>(如果读到列name的值为'张三'，则意味着发生了脏读) | |
| ⑤ | COMMIT; | |
| ⑥ | | ROLLBACK; |

Session A和Session B各开启了一个事务，Session B中的事务先将studentno列为1的记录的name列更新为'张三'，然后SessionA中的事务再去查询这条studentno为1的记录，如果读到列name的值为'张三'，而 SessionB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`。

## 3.不可重复读（Non-Repeatable Read）

对于两个事务Session A、SessionB，Session A `读取` 了一个字段，然后 SessionB `更新` 了该字段。之后 SessionA `再次读取` 同一个字段，`值就不同` 了。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 student WHERE studentno=1;<br>(此时读到的列name的值为'王五') | |
| ③ | | UPDATE student SET name='张三' WHERE studentno = 1; |
| ④ | SELECT * FROM student WHERE studentno=1;<br>(如果读到列name的值为'张三'，则意味着发生了不可重复读) | |
| ⑤ | | UPDATE student SET name='李四' WHERE studentno= 1; |
| ⑥ | SELECT * FROM student WHERE studentno=1;<br>(如果读到列name的值为'李四'，则意味着发生了不可重复读) | |

我们在SessionB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studentno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`。

## 4.幻读（Phantom）

对于两个事务Session A、Session B，SessionA从一个表中`读取`了一个字段,然后 SessionB在该表中`插入`了一些新的行。之后 ,如果 SessionA`再次读取`同一个表，就会多出几行。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 student WHERE studentno>0;<br>(此时读到的列name的值为'张三') | |
| ③ | | INSERT INTO student VALUES(2,'赵六','2班') ; |
| ④ | SELECT * FROM student WHERE studentno>0;<br>(如果读到列name的值为'张三'、'赵六'的记录，则意味着发生了幻读) | |

Session A中的事务先根据条件 studentno >0这个条件查询表 student，得到了name列值为'张三'的记录；之后Session B中提交了一个`隐式事务`，该事务向表student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 studentno>0查询表student，得到的结果集中包含SessionB中的事务新插入的那条记录，这种现象也被称之为`幻读`。我们把新插入的那些记录称之为`幻影记录`。

**注意1:**

有的同学会有疑问，那如果Session B中`删除了`一些符合studentno > 的记录而不是插入新记录，那SessionA之后再根据`studentno > 0`的条件读取的记录变少了，这种现象算不算幻读呢?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`。

**注意2:**

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? 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。

## 3.3SQL中的四种隔离级别

上面介绍了几种并发事务执行过程中可能遇到的一些问题，这些问题有轻重缓急之分，我们给这些问题按照严重性来排一下序：

```
1 脏写 > 脏读 > 不可重复读 >幻读
```

我们愿意舍弃一部分隔离性来换取一部分性能在这里就体现在：设立一些隔离级别，隔离级别越低，并发问题发生的就越多。`SQL标准`中设立了4个`隔离级别`：

- READ UNCOMMITTED：读未提交，在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。不能避免脏读、不可重复读、幻读。
- READ COMMITTED：读已提交，它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。可以避免脏读，但不可重复读、幻读问题仍然存在。
- REPEATABLE READ：可重复读，事务A在读到一条数据之后，此时事务B对该数据进行了修改并提交，那么事务A再读该数据，读到的还是原来的内容。可以避免脏读、不可重复读，但幻读问题仍然存在。这是MySQL的默认隔离级别。
- SERIALIZABLE：可串行化，确保事务可以从一个表中读取相同的行。在这个事务持续期间，禁止其他事务对该表执行插入、更新和删除操作。所有的并发问题都可以避免，但性能十分低下。能避免脏读、不可重复读和幻读。

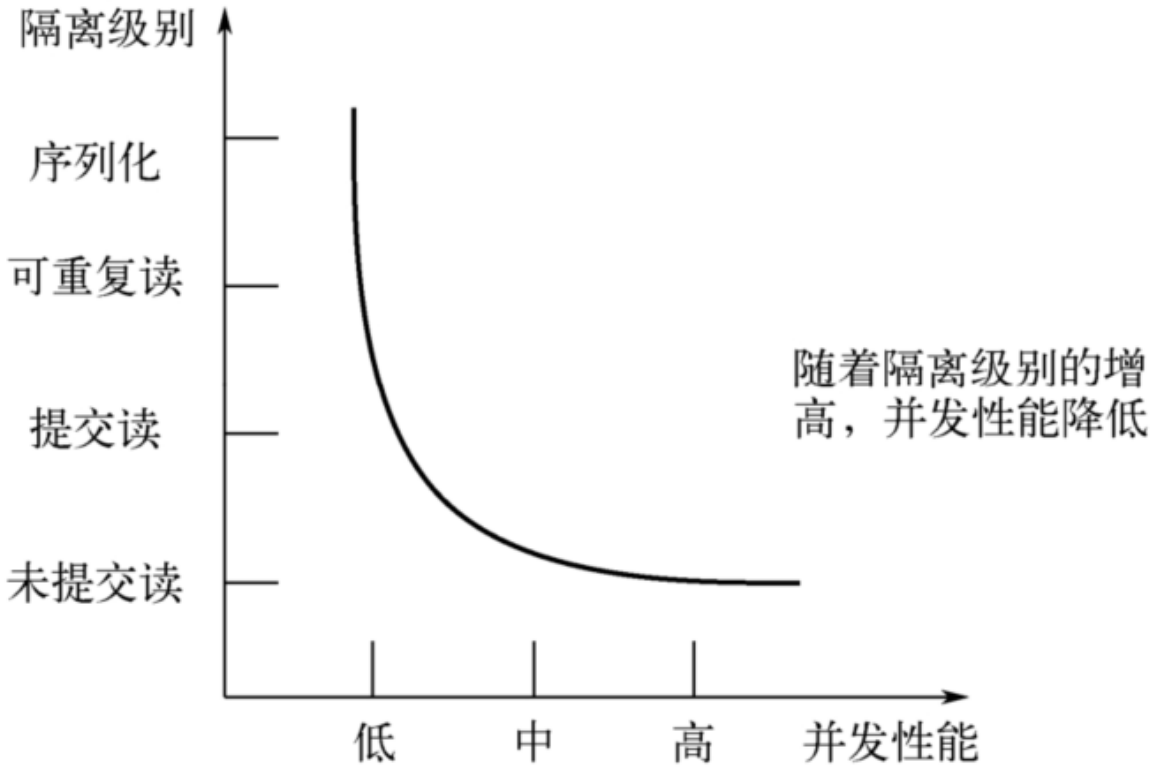
`SQL标准`中规定，针对不同的隔离级别，并发事务可以发生不同严重程度的问题，具体情况如下：

| 隔离级别 | 脏读可能性 | 不可重复读可能性 | 幻读可能性 | 加锁读 |
| --- | --- | --- | --- | --- |
| READ UNCONMITED | Yes | Yes | Yes | No |
| READ COMMITED | No | Yes | Yes | No |
| REPEATABLE READ | No | No | Yes | No |
| SERIALIZABLE | No | No | No | Yes |

> YES表示没有解决

`脏写`怎么没涉及到？因为脏写这个问题太严重了，不论是哪种隔离级别，都不允许脏写的情况发生。

不同的隔离级别有不同的现象，并有不同的锁和并发机制，隔离级别越高，数据库的并发性能就越差，4种事务隔离级别与并发性能的关系如下：




## 3.4MySQL支持的四种隔离级别

不同的数据库厂商对SQL标准中规定的四种隔离级别支持不一样。比如,`Oracle`就只支持`READ COMNITTED`（默认隔离级别）和`SERIALIZABLE`隔离级别。MySQL虽然支持4种隔离级别，但与SQL标准中所规定的各级隔离级别允许发生的问题却有些出入，MySQL在REPEATABLE READ隔离级别下，是可以禁止幻读问题的发生的，禁止幻读的原因在第16章讲解。

MySQL的默认隔离级别为REPEATABLE READ，我们可以手动修改一下事务的隔离级别。

```
# 查看隔离级别， MySQL 5.7.20的版本之前：
mysql> SHOW VARIABLES LIKE 'tx_isolation';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| tx_isolation  | REPEATABLE-READ |
+---------------+-----------------+
1 row in set (0.00 sec)

# MySQL 5.7.20版本之后，引入 transaction_isolation来替换 tx_isolation
#查看隔离级别， MySQL 5.7.20的版本及之后：
mysql> SHOW VARIABLES LIKE 'transaction_isolation';
+-----------------------+-----------------+
| Variable_name         | Value           |
+-----------------------+-----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+-----------------+
1 row in set (0.02 sec)

#或者不同 MySQL版本中都可以使用的：
SELECT @@transaction_isolation;
```

## 3.5如何设置事务的隔离级别

通过下面的语句修改事务的隔离级别：

```
SET [GLOBAL|SESSION] TRANSACTION ISOLATION LEVEL隔离级别 ;
#其中，隔离级别格式：
> READ UNCOMMITTED
> READ COMMITTED
> REPEATABLE READ
> SERIALIZABLE
```

或者：

```
SET [GLOBAL|SESSION] TRANSACTION_ISOLATION = '隔离级别 '
#其中，隔离级别格式：
> READ-UNCOMMITTED
> READ-COMMITTED
> REPEATABLE-READ
> SERIALIZABLE
```

关于设置时使用GLOBAL或SESSION的影响：

- 使用 `GLOBAL` 关键字（在全局范围影响）：

  ```
  SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  #或
  SET GLOBAL TRANSACTION_ISOLATION = 'SERIALIZABLE';
  ```

  则：

  - 当前已经存在的会话无效
  - 只对执行完该语句之后产生的会话起作用

- 使用 `SESSION` 关键字（在会话范围影响）：

  ```
  SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  #或
  SET SESSION TRANSACTION_ISOLATION = 'SERIALIZABLE';
  ```

  则：

  - 对当前会话的所有后续的事务有效
  - 如果在事务之间执行，则对后续的事务有效
  - 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。比如，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`。

> 小结：
> 数据库规定了多种事务隔离级别，不同隔离级别对应不同的干扰程度，隔离级别越高，数据一致性就越好，但并发性越弱。

**演示global**

- 开启会话A

```
mysql> show variables lik 'transaction_isolation';
ERROR 1064 (42000): You have an error in your SQL syntax; check t
k 'transaction_isolation'' at line 1
mysql> show variables like 'transaction_isolation';
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+----------------+
1 row in set (0.01 sec)

mysql> SET GLOBAL TRANSACTION_ISOLATION = 'read-committed';
Query OK, 0 rows affected (0.00 sec)

mysql> select @@transaction_isolation;
+-------------------------+
| @@transaction_isolation |
+-------------------------+
| REPEATABLE-READ         |
+-------------------------+
1 row in set (0.00 sec)
```

当前会话的隔离级别仍然是之前的~

- 再开启另一个会话B

```
mysql> select @@transaction_isolation;
+-------------------------+
| @@transaction_isolation |
+-------------------------+
| READ-COMMITTED          |
+-------------------------+
1 row in set (0.00 sec)
```

新开启的会话生效

- 会话A中退出mysql，再登录，会发现设置的隔离级别已经生效了~

> msql服务器重启 `systemctl restart mysqld` 后，隔离级别又重新回到默认~ 毕竟咱们设置的都是在内存级别的~

**演示session**

- 会话A中

```
mysql> SET SESSION TRANSACTION_ISOLATION = 'read-committed';
Query OK, 0 rows affected (0.00 sec)

mysql> select @@transaction_isolation;
+-------------------------+
| @@transaction_isolation |
+-------------------------+
| READ-COMMITTED          |
+-------------------------+
1 row in set (0.00 sec)
```

当前会话后序的事务都有效~

- 会话B中，事务隔离级别也变更成了我们设置的那个~

## 3.6不同隔离级别举例

1. 创建数据表，并初始化数据

```
use atguigudb3;
create table account(
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(15),
  balance DECIMAL(15)
);

INSERT INTO account VALUES(1 ,'张三', '100'),(2,'李四', '0');
```

2. 在Xshell中开两个Session，模拟两个事务~

3. 将两个session中的隔离级别都设置成`read-uncommitted`

```
mysql> select @@transaction_isolation;
+-------------------------+
| @@transaction_isolation |
+-------------------------+
| REPEATABLE-READ         |
+-------------------------+
1 row in set (0.00 sec)

mysql> set session transaction_isolation = 'read-uncommitted';
Query OK, 0 rows affected (0.00 sec)

mysql> select @@transaction_isolation;
+-------------------------+
| @@transaction_isolation |
+-------------------------+
| READ-UNCOMMITTED        |
+-------------------------+
1 row in set (0.00 sec)
```

**演示1.读未提交之脏读**

设置隔离级别为未提交读：

| 时间 | 事务 1 | 事务 2 |
| --- | --- | --- |
| T1 | set session transaction isolation level read uncommitted;<br>start transaction;(开启事务)<br>update account set balance = balance+100 where id=1;<br>select * from account where id=1;#结果为 200 | |
| T2 | | set session transaction isolation level read uncommitted;<br>start transaction;<br>select * from account where id=1;#查询余额结果为 200，脏读 |
| T3 | rollback; | |
| T4 | commit; | |
| T5 | | select * from account where id=1;查询余额结果为 100 |

事务1和事务2的执行流程如下：

| 时间 | 事务 1 | 事务 2 |
| --- | --- | --- |
| T1 | set session transaction isolation level read uncommitted;<br>start transaction;(开启事务)<br>update account set balance = balance-100 where id=1;<br>update account set balance = balance+100 where id=2;<br>select * from account where id=1;结果为 0 | |
| T2 | | set session transaction isolation level read uncommitted;<br>start transaction;<br>select * from account where id=2; #结果为 100<br>update account set balance = balance-100 where id=2;#更新语句被阻塞 |
| T3 | rollback; | |
| T4 | | commit |

- 案例一

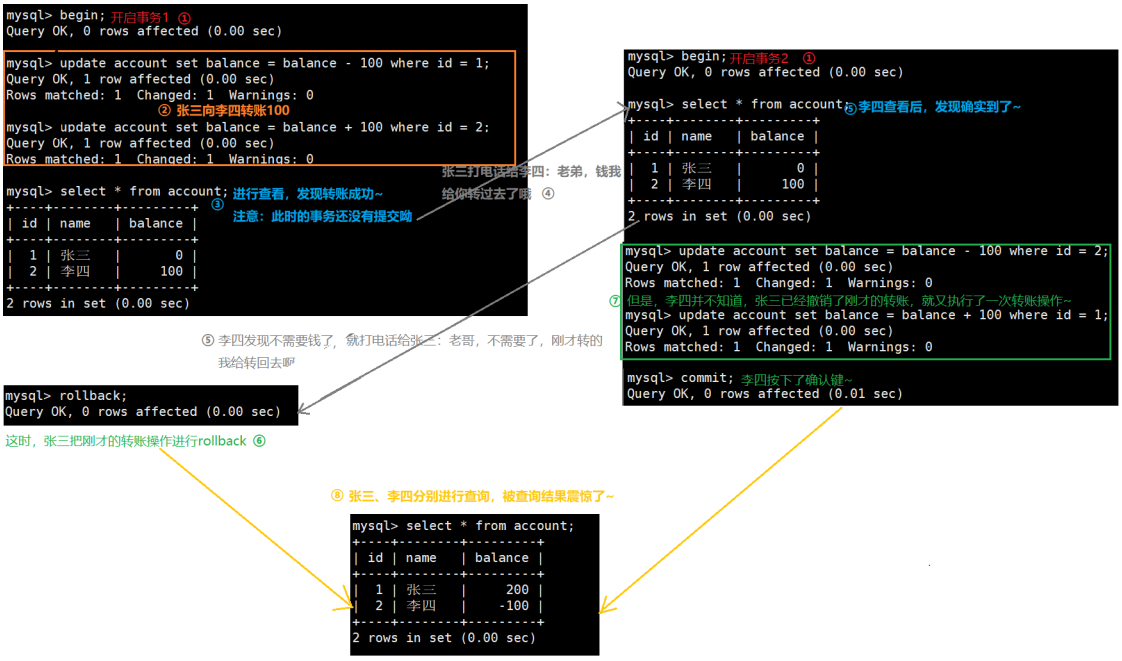
```
mysql> begin;        开启事务的自动提交
Query OK, 0 rows affected (0.00 sec)
                     事务1中更新数据，但是还没提交~
mysql> update account set balance = balance + 100 where id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> select * from account;
+----+--------+---------+
| id | name   | balance |
+----+--------+---------+
|  1 | 张三   |     200 |
|  2 | 李四   |       0 |
+----+--------+---------+
2 rows in set (0.00 sec)
```

事务2中读取到了事务1还没有提交的数据，出现了脏读~

> 之后事务1进行回滚，事务2读取的数据就回到了 100~

- 案例二




**演示2：读已提交**

| 时间 | 事务 1 | 事务 2 |
| --- | --- | --- |
| T1 | set session transaction isolation level read committed;<br>start transaction;(开启事务)<br>select * from account where id=2;#结果为 0 | |
| T2 | | set session transaction isolation level read committed;<br>start transaction;<br>update account set balance = balance+100 where id=2;<br>select * from account where id=2;#结果为 100 |
| T3 | select * from account where id=2;#结果仍然为 0，未发生脏读 | |
| T4 | | commit; |
| T5 | select * from account where id=2;#结果为 100<br>commit; | |

设置隔离级别为可重复读，事务的执行流程如下：

| 时间 | 事务 1 | 事务 2 |
| --- | --- | --- |
| T1 | set session transaction isolation level repeatable read;<br>start transaction;(开启事务)<br>select * from account where id=2;#结果为 0 | |
| T2 | | set session transaction isolation level repeatable read;<br>start transaction;<br>update account set balance = balance+100 where id=2;<br>select * from account where id=2;#结果为 100 |
| T3 | | commit; |
| T4 | select * from account where id=2;#结果依然是 0<br>commit;<br>select * from account where id=2; #结果为 100 | |

- 环境准备

```
mysql> truncate table account;
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO account VALUES(1 ,'张三', '100'),(2,'李四', '0');
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> select * from account;
+----+--------+---------+
| id | name   | balance |
+----+--------+---------+
|  1 | 张三   |     100 |
|  2 | 李四   |       0 |
+----+--------+---------+
2 rows in set (0.01 sec)
```

- 将两个session的隔离级别设置为：`read-committed`

```
mysql> set session transaction_isolation = 'read-committed';
Query OK, 0 rows affected (0.00 sec)

mysql> select @@transaction_isolation;
+-------------------------+
| @@transaction_isolation |
+-------------------------+
| READ-COMMITTED          |
+-------------------------+
1 row in set (0.00 sec)
```

- 演示图解


### 演示3：可重复读

- 将两个session的隔离级别设置为：`repeatable-read`]

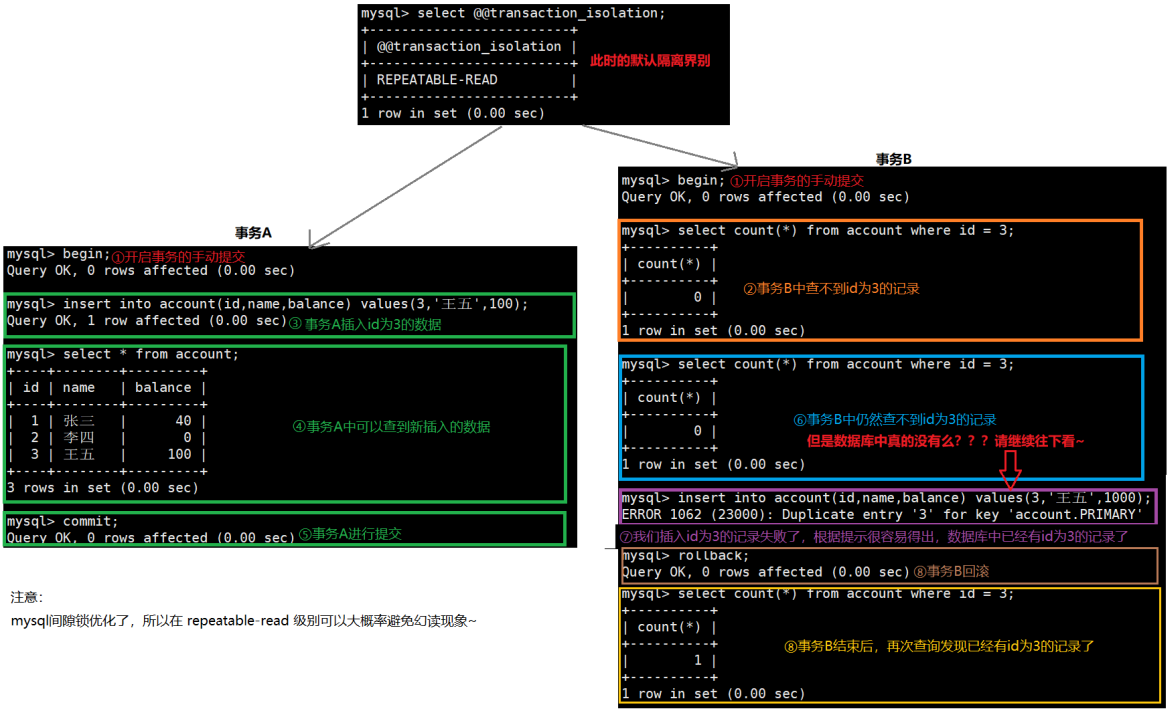
```
mysql> set session transaction_isolation = 'repeatable-read';
Query OK, 0 rows affected (0.00 sec)

mysql> select @@transaction_isolation;
+-------------------------+
| @@transaction_isolation |
+-------------------------+
| REPEATABLE-READ         |
+-------------------------+
1 row in set (0.00 sec)
```

- 演示图解


## 演示4：幻读

| 时间 | 事务 1 | 事务 2 |
| --- | --- | --- |
| T1 | set session transaction isolation level repeatable read;<br>start transaction;(开启事务)<br>select count(*) from account where id=3;#结果为 0 | |
| T2 | | set session transaction isolation level repeatable read;<br>start transaction;<br>insert into account (id,name,balance) values(3,"王五",0);<br>commit; |
| T3 | insert into account (id,name,balance) values(3,"王五",0);<br>#主键重复，插入失败 | |
| T4 | select count(*) from account where id=3;#结果为 0 | |
| T5 | rollback; | |




这里要灵活的理解读取的意思。第一次select是读取，第二次的insert其实也属于隐式的读取，只不过是在mysql的机制中读取的，**插入数据也是要先读取一下有没有主键冲突才能决定是否执行插入**

**幻读，并不是说两次读取获取的结果集不同，幻读侧重的方面是某一次的select 操作得到的结果所表征的数据状态无法支撑后续的业务操作。**更为具体一些：**select某记录是否存在，不存在，准备插入此记录，但执行insert时发现此记录已存在，无法插入，此时就发生了幻读**（如上图所示）。

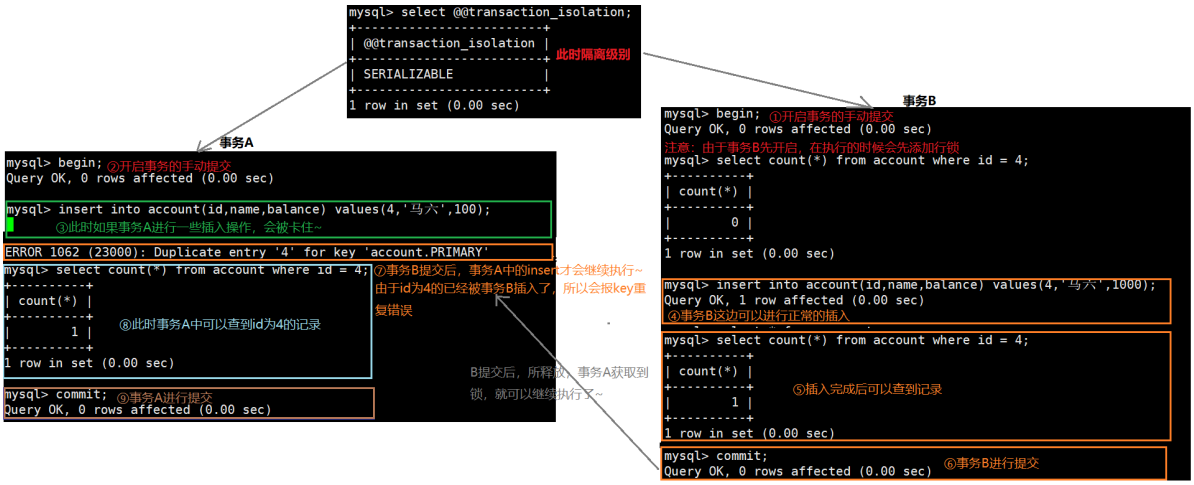
在RR隔离级别下，step1、step2是会正常执行的，step3则会报错主键冲突，对于事务B的业务来说是执行失败的，这里事务B就是发生了幻读，因为事务B在step1中读取的数据状态并不能支撑后续的业务操作，事务B:“见鬼了，我刚才读到的结果应该可以支持我这样操作才对啊，为什么现在不可以"。事务B不敢相信的又执行了step4，发现和step1读取的结果是一样的（RR下的`MVCC`机制)。此时，幻读无疑已经发生，事务B无论读取多少次，都查不到id=3的记录，但它的确无法插入这条他通过读取来认定不存在的记录（此数据已被事务A插入)，对于事务B来说，它幻读了。

**其实RR（Mysql默认隔离级别）也是可以避免幻读的**，通过对select操作手动加`行X锁(独占锁)`(SELECT ... FOR UPDATE这也正是SERIALIZABLE 隔离级别下会隐式为你做的事情)，同时，即便当前记录不存在，比如id = 3是不存在的，当前事务也会获得一把记录锁（因为InnoDB的行锁锁定的是索引，故记录实体存在与否没关系，存在就加`行X锁`，不存在就加`间隙锁`），其他事务则无法插入此索引的记录，故杜绝了幻读。

在`SERIALIZABLE隔离级别`下，step1执行时是会隐式的添加`行(X)锁/gap(X)锁`的，从而step2会被阻塞，step3 会正常执行，待事务1提交后，事务2才能继续执行（主键冲突执行失败)，对于事务1来说业务是正确的，成功的阻塞扼杀了扰乱业务的事务2，对于事务1来说他前期读取的结果是可以支撑其后续业务的。

所以MySQL的幻读并非什么读取两次返回结果集不同，而是事务在插入事先检测不存在的记录时，惊奇的发现这些数据已经存在了，之前的检测读获取到的数据如同鬼影一般。

补充：隔离级别是`SERIALIZABLE`时的效果：




# 4.事务的常见分类

从事务理论的角度来看，可以把事务分为以下几种类型：

- 扁平事务（FlatTransactions）
- 带有保存点的扁平事务（FlatTransactionswithSavepoints）
- 链事务（ChainedTransactions）
- 嵌套事务（NestedTransactions）
- 分布式事务（DistributedTransactions）

下面简单介绍这几种类型：

1. `扁平事务`是事务类型中最简单的一种，但是在实际生产环境中，这可能是使用最频繁的事务，在扁平事务中，所有操作都处于同一层次，其由BEGIN WORK开始，由COMMIT WORK或ROLLBACK WORK结束，其间的操作是原子的，要么都执行，要么都回滚，因此，扁平事务是应用程序成为原子操作的基本组成模块。扁平事务虽然简单，但是在实际环境中使用最为频繁，也正因为其简单，使用频繁，故每个数据库系统都实现了对扁平事务的支持。扁平事务的主要限制是不能提交或者回滚事务的某一部分，或分几个步骤提交。

   扁平事务一般有三种不同的结果：

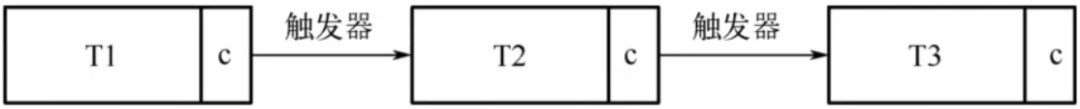
   - 事务成功完成。在平常应用中约占所有事务的96%。
   - 应用程序要求停止事务。比如应用程序在捕获到异常时会回滚事务，约占事务的3%。
   - 外界因素强制终止事务。如连接超时或连接断开，约占所有事务的1%。

2. `带有保存点的扁平事务`除了支持扁平事务支持的操作外，还允许在事务执行过程中回滚到同一事务中较早的一个状态。这是因为某些事务可能在执行过程中出现的错误并不会导致所有的操作都无效，放弃整个事务不合乎要求，开销太大。

   `保存点（Savepoint）`用来通知事务系统应该记住事务当前的状态，以便当之后发生错误时，事务能回到保存点当时的状态。对于扁平的事务来说，隐式的设置了一个保存点，然而在整个事务中，只有这一个保存点，因此，回滚只能会滚到事务开始时的状态。

3. `链事务`是指一个事务由多个子事务链式组成，它可以被视为保存点模式的一个变种。带有保存点的扁平事务，当发生系统崩溃时，所有的保存点都将消失，这意味着当进行恢复时，事务需要从开始处重新执行，而不能从最近的一个保存点继续执行。链事务的思想是：在提交一个事务时，释放不需要的数据对象，将必要的处理上下文隐式地传给下一个要开始的事务，前一个子事务的提交操作和下一个子事务的开始操作合并成一个原子操作，这意味着下一个事务将看到上一个事务的结果，就好像在一个事务中进行一样。这样，**在提交子事务时就可以释放不需要的数据对象，而不必等到**

**整个事务完成后才释放。**其工作方式如下：




链事务与带有保存点的扁平事务的不同之处体现在：

1. 带有保存点的扁平事务能回滚到任意正确的保存点，而链事务中的回滚仅限于当前事务，即只能恢复到最近的一个保存点。
2. 对于锁的处理，两者也不相同，链事务在执行COMMIT后即释放了当前所持有的锁，而带有保存点的扁平事务不影响迄今为止所持有的锁。

4. `嵌套事务`是个层次结构框架，由一个顶层事务（Top-Level Transaction）控制着各个层次的事务，顶层事务之下嵌套的事务被称为子事务（Subtransaction），其控制着每一个局部的变换，子事务本身也可以是嵌套事务。因此，嵌套事务的层次结构可以看成是一棵树。
5. `分布式事务`通常是在一个分布式环境下运行的扁平事务，因此，需要根据数据所在位置访问网络中不同节点的数据库资源。例如，一个银行用户从招商银行的账户向工商银行的账户转账1000元，这里需要用到分布式事务，因为不能仅调用某一家银行的数据库就完成任务。