

04-事务：账户余额总是对不上账，怎么办？

你好，我是李玥。今天这节课我们来说一下电商的账户系统。

账户系统负责记录和管理用户账户的余额，这个余额就是每个用户临时存在电商的钱，来源可能是用户充值或者退货退款等多种途径。

账户系统的用途也非常广泛，不仅仅是电商，各种互联网内容提供商、网络游戏服务商，电信运营商等等，都需要账户系统来管理用户账户的余额，或者是虚拟货币。包括银行的核心系统，也同样包含一个账户系统。

从业务需求角度来分析，一个最小化的账户系统，它的数据模型可以用下面这张表来表示：

列名	数据类型	主键	非空	说明
user_id	BIGINT	是	是	用户ID
balance	BIGINT		是	账户余额
timestamp	DATE		是	更新时间

这个表包括用户ID、账户余额和更新时间三个字段。每次交易的时候，根据用户ID去更新这个账户的余额就可以了。

为什么总是对不上账？

每个账户系统都不是孤立存在的，至少要和财务、订单、交易这些系统有着密切的关联。理想情况下，账户系统内的数据应该是自治的。所有用户的账户余额加起来，应该等于这个电商公司在银行专用账户的总余额。账户系统的数据也应该和其他系统的数据能对的上。比如说，每个用户的余额应该能和交易系统中充值记录，以及订单系统中的订单对上。

不过，由于业务和系统的复杂性，现实情况却是，很少有账户系统能够做到一点不差的对上每一笔账。所以，稍微大型一点儿的系统，都会有一个专门的对账系统，来核对、矫正账户系统和其他系统之间的数据差异。

对不上账的原因非常多，比如业务变化、人为修改了数据、系统之间数据交换失败等等。那作为系统的设计者，我们只关注“如何避免由于技术原因导致的对不上账”就可以了，有哪些是因为技术原因导致的呢？比如说：网络请求错误，服务器宕机、系统Bug等。

“对不上账”是通俗的说法，它的本质问题是，**冗余数据的一致性问题**。

这里的冗余数据并不是多余或者重复的数据，而是多份含有相同信息的数据。比如，我们完全可以通过用户的每一笔充值交易数据、消费的订单数据，来计算出这个用户当前的账户余额是多少。也就是说，账户余

额数据和这些账户相关的交易记录，都含有“账户余额”这个信息，那它们之间就互为冗余数据。

在设计系统的存储时，原则上不应该存储冗余数据，一是浪费存储空间，二是让这些冗余数据保持一致是一件非常麻烦的事儿。但有些场景下存储冗余数据是必要的，比如用户账户的余额这个数据。

这个数据在交易过程中会被非常频繁地用到，总不能每次交易之前，先通过所有历史交易记录计算一下当前账户的余额，这样做速度太慢了，性能满足不了交易的需求。所以账户系统保存了每个用户的账户余额，这实际上是一种用**存储空间换计算时间**的设计。

如果说只是满足功能需求，账户系统只记录余额，每次交易的时候更新账户余额就够了。但是这样做有一个问题，如果账户余额被篡改，是没有办法追查的，所以在记录余额的同时，还需要记录每一笔交易记录，也就是账户的流水。流水的数据模型至少需要包含：流水ID、交易金额、交易时间戳以及交易双方的系统、账户、交易单号等信息。

虽然说，流水和余额也是互为冗余数据，但是记录流水，可以有效地修正由于系统Bug或者人为篡改导致的账户余额错误的问题，也便于账户系统与其他外部系统进行对账，所以账户系统记录流水是非常必要的。

在设计账户流水时，有几个重要的原则必须遵守，最好是用技术手段加以限制。

1. 流水记录只能新增，一旦记录成功不允许修改和删除。即使是由于正当原因需要取消一笔已经完成的交易，也不应该去删除交易流水。正确的做法是再记录一笔“取消交易”的流水。
2. 流水号必须是递增的，我们需要用流水号来确定交易的先后顺序。

在对账的时候，一旦出现了流水和余额不一致，并且无法通过业务手段来确定到底是哪儿记错了的情况，一般的处理原则是以交易流水为准来修正余额数据，这样才能保证后续的交易能“对上账”。

那从技术上，如何保证账户系统中流水和余额数据一致呢？

使用数据库事务来保证数据一致性

在设计对外提供的服务接口时，不能提供单独更新余额或者流水的功能，只提供交易功能。我们需要在实现交易功能的时候，同时记录流水并修改余额，并且要尽可能保证，在任何情况下，记录流水和修改余额这两个操作，要么都成功，要么都失败。不能有任何一笔交易出现，记录了流水但余额没更新，或者更新了余额但是没记录流水。

这个事儿说起来挺简单，但实际上是非常难实现的。毕竟应用程序只能先后来执行两个操作，执行过程中，可能会发生网络错误、系统宕机等各种异常的情况，所以对于应用程序来说，很难保证这两个操作都成功或者都失败。

数据库提供了事务机制来解决这个问题，实际上事务这个特性最初就是被设计用来解决交易问题的，在英文中，事务和交易就是同一个单词：Transaction。

我们先看一下如何来使用MySQL的事务，实现一笔交易。比如说，在事务中执行一个充值100元的交易，先记录一条交易流水，流水号是888，然后把账户余额从100元更新到200元。对应的SQL是这样的：

```
mysql> begin;  -- 开始事务
```

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

mysql> insert into account_log ...; -- 写入交易流水
Query OK, 1 rows affected (0.01 sec)

mysql> update account_balance ...; -- 更新账户余额
Query OK, 1 rows affected (0.00 sec)

mysql> commit; # 提交事务
Query OK, 0 rows affected (0.01 sec)
```

使用事务的时候，只需要在之前执行begin，标记开始一个事务，然后正常执行多条SQL语句，在事务里面的不仅可以执行更新数据的SQL，查询语句也是可以的，最后执行commit，提交事务就可以了。

我们来看一下，事务可以给我们提供什么样的保证？

首先，它可以保证，记录流水和更新余额这两个操作，要么都成功，要么都失败，即使是在数据库宕机、应用程序退出等等这些异常情况下，也不会出现，只更新了一个表而另一个表没更新的情况。这是事务的**原子性 (Atomic)**。

事务还可以保证，数据库中的数据总是从一个一致性状态（888流水不存在，余额是100元）转换到另外一个一致性状态（888流水存在，余额是200元）。对于其他事务来说，不存在任何中间状态（888流水存在，但余额是100元）。

其他事务，在任何一个时刻，如果它读到的流水中没有888这条流水记录，它读出来的余额一定是100元，这是交易前的状态。如果它能读到888这条流水记录，它读出来的余额一定是200元，这是交易之后的状态。也就是说，事务保证我们读到的数据（交易和流水）总是一致的，这是事务的**一致性(Consistency)**。

实际上，这个事务的执行过程无论多快，它都是需要时间的，那修改流水表和余额表对应的数据，也会有先后。那一定存在一个时刻，流水更新了，但是余额还没更新，也就是说每个事务的中间状态是事实存在的。

数据库为了实现一致性，必须保证每个事务的执行过程中，中间状态对其他事务是不可见的。比如说我们在事务A中，写入了888这条流水，但是还没有提交事务，那在其他事务中，都不应该读到888这条流水记录。这是事务的**隔离性(Isolation)**。

最后，只要事务提交成功，数据一定会被持久化到磁盘中，后续即使发生数据库宕机，也不会改变事务的结果。这是事务的**持久性(Durability)**。

你会发现，我上面讲的就是事务的**ACID**四个基本特性。你需要注意的是，这四个特性之间是紧密关联在一起的，不用去纠结每一个特性的严格定义，更重要的是理解事务的行为，也就是我们的系统在使用事务的时候，各种情况下，事务对你的数据会产生什么影响，这是使用事务的关键。

理解事务的隔离级别

有了数据库的事务机制，只要确保每一笔交易都在事务中执行，我们的账户系统就很容易保证流水和余额数据的一致性。但是，ACID是一个非常严格的定义，或者说是理想的情况。如果要完全满足ACID，一个数据库的所有事务和SQL都只能串行执行，这个性能肯定是不能满足一般系统的要求的。

对账户系统和其他大多数交易系统来说，事务的原子性和持久性是必须要保证的，否则就失去了使用事务的意义，而一致性和隔离性其实可以做适当牺牲，来换取性能。所以，MySQL提供了四种隔离级别，具体来看一下这个表：

隔离级别	脏读 (DR, Dirty Read)	不可重复读 (NR, NonRepeatable Read)	幻读 (PR, Phantom Read)
能读到未提交的数据, RU, READ-UNCOMMITTED	y	y	y
能读到已提交的数据, RC, READ-COMMITTED	N	y	y
可重复读 RR, REPEATABLE-READ	N	N	y
串行执行 SERIALIZABLE	N	N	N

几乎所有讲MySQL的事务隔离级别的文章里面，都有这个表，我们也不能免俗，因为这个表太经典了。很多同学看这个表的时候，面对这么多概念都有点儿晕，确实不太好理解。我来跟你说一下怎么来把这四种隔离级别搞清楚，重点在哪里。

这个表里面自上到下，一共有四种隔离级别：RU、RC、RR和SERIALIZABLE，这四种级别的隔离性越来越严格，性能也越来越差，在MySQL中默认的隔离级别是RR，可重复读。

先说两种不常用的，第一种RU级别，实际上就是完全不隔离。每个进行中事务的中间状态，对其他事务都是可见的，所以有可能出现“脏读”。我们上一个小节充值的例子中，读到了888这条流水，但余额还是转账之前的100元，这种情况就是脏读。这种级别虽然性能好，但是存在脏读的可能，对应用程序来说比较难处理，所以基本不用。

第四种“序列化”级别，具备完美的“隔离性”和“一致性”，性能最差，也很少会用到。

常用的隔离级别其实就是RC和RR两种，其中MySQL默认的隔离级别是RR。这两种隔离级别都可以避免脏读，能够保证在其他事务中是不会读到未提交事务的数据，或者通俗地说，**只要你的事务没有提交，那这个事务对数据做出的更新，对其他会话是不可见的，它们读到的还是你这个事务更新之前的数据。**

RC和RR唯一的区别在于“是否可重复读”，这个概念也有点儿绕口，但其实也很简单。

在一个事务执行过程中，它能不能读到其他已提交事务对数据的更新，如果能读到数据变化，就是“不可重复读”，否则就是“可重复读”。

我们举个例子来说明，比如，我们把事务的隔离级别设为RC。会话A开启了一个事务，读到ID为0的账户，当前账户余额是100元。

```
mysql> -- 会话 A
mysql> -- 确认当前设置的隔离级别是RC
mysql> SELECT @@global.transaction_isolation, @@transaction_isolation;
+-----+-----+
| @@global.transaction_isolation | @@transaction_isolation |
+-----+-----+
| READ-COMMITTED                  | READ-COMMITTED          |
+-----+-----+
1 row in set (0.00 sec)

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

mysql> select log_id, amount, timestamp from account_log order by log_id;
+-----+-----+-----+
| log_id | amount | timestamp          |
+-----+-----+-----+
|      3 |    100 | 2020-02-07 09:40:37 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from account_balance; -- 账户余额是100元;
+-----+-----+-----+-----+
| user_id | balance | timestamp          | log_id |
+-----+-----+-----+-----+
|      0 |    100 | 2020-02-07 09:47:39 |      3 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

这时候另外一个会话B，对这个账户完成了一笔转账交易，并且提交了事务。把账户余额更新成了200元。

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

mysql> select log_id, amount, timestamp from account_log order by log_id;
+-----+-----+-----+
| log_id | amount | timestamp          |
+-----+-----+-----+
|      3 |    100 | 2020-02-07 09:40:37 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> -- 写入流水
mysql> insert into account_log values (NULL, 100, NOW(), 1, 1001, NULL, 0, NULL, 0, 0);
Query OK, 1 row affected (0.00 sec)

mysql> -- 更新余额
mysql> update account_balance
    -> set balance = balance + 100, log_id = LAST_INSERT_ID(), timestamp = NOW()
    -> where user_id = 0 and log_id = 3;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> -- 当前账户有2条流水记录
mysql> select log_id, amount, timestamp from account_log order by log_id;
+-----+-----+-----+
| log_id | amount | timestamp          |
+-----+-----+-----+
```

```

|      3 |      100 | 2020-02-07 09:40:37 |
|      4 |      100 | 2020-02-07 10:06:15 |
+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> -- 当前账户余额是200元;
mysql> select * from account_balance;
+-----+-----+-----+-----+
| user_id | balance | timestamp           | log_id |
+-----+-----+-----+-----+
|      0 |      200 | 2020-02-07 10:06:16 |      4 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
mysql> commit;
Query OK, 0 rows affected (0.00 sec)

```

注意，这个时候会话A之前开启的事务是一直未关闭的。我们再来会话A中看一下账户的余额，你觉得应该是多少？

我们来看一下实际的结果。

```

mysql> -- 会话 A
mysql> -- 当前账户有2条流水记录
mysql> select log_id, amount, timestamp from account_log order by log_id;
+-----+-----+-----+
| log_id | amount | timestamp           |
+-----+-----+-----+
|      3 |      100 | 2020-02-07 09:40:37 |
|      4 |      100 | 2020-02-07 10:06:15 |
+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> -- 当前账户余额是200元;
mysql> select * from account_balance;
+-----+-----+-----+-----+
| user_id | balance | timestamp           | log_id |
+-----+-----+-----+-----+
|      0 |      200 | 2020-02-07 10:06:16 |      4 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
mysql> commit;
Query OK, 0 rows affected (0.00 sec)

```

可以看到，当我们把隔离级别设置为RC时，会话A第二次读到的账户余额是200元，也就是会话B更新后的数据。对于会话A来说，**在同一个事务内两次读取同一条数据，读到的结果可能会不一样，这就是“不可重复读”**。

如果把隔离级别设置为RR，会话A第二次读到的账户余额仍然是100元，交易流水也只有一条记录。**在RR隔离级别下，在一个事务进行过程中，对于同一条数据，每次读到的结果总是相同的，无论其他会话是否已经更新了这条数据，这就是“可重复读”**。

理解了RC和RR这两种隔离级别的区别，就足够应对绝大部分业务场景了。

最后，我来简单说一下“幻读”。在实际业务中，很少能遇到幻读，即使遇到，也基本不会影响到数据准确性，所以你简单了解一下即可。在RR隔离级别下，我们开启一个事务，之后直到这个事务结束，在这个事务内其他事务对数据的更新是不可见的，这个我们刚刚讲过。

比如我们在会话A中开启一个事务，准备插入一条ID为1000的流水记录。查询一下当前流水，不存在ID为1000的记录，可以安全地插入数据。

```
mysql> -- 会话 A
mysql> select log_id from account_log where log_id = 1000;
Empty set (0.00 sec)
```

这时候，另外一个会话抢先插入了这条ID为1000的流水记录。

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

mysql> insert into account_log values
    -> (1000, 100, NOW(), 1, 1001, NULL, 0, NULL, 0, 0);
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

然后会话A再执行相同的插入语句时，就会报主键冲突错误，但是由于事务的隔离性，它执行查询的时候，却查不到这条ID为1000的流水，就像出现了“幻觉”一样，这就是幻读。

```
mysql> -- 会话 A
mysql> insert into account_log values
    -> (1000, 100, NOW(), 1, 1001, NULL, 0, NULL, 0, 0);
ERROR 1062 (23000): Duplicate entry '1000' for key 'account_log.PRIMARY'
mysql> select log_id from account_log where log_id = 1000;
Empty set (0.00 sec)
```

理解了这几种隔离级别，最后我们给出一种兼顾并发、性能和数据一致性的交易实现。这个实现在隔离级别为RC和RR时，都是安全的。

1. 我们给账户余额表增加一个log_id属性，记录最后一笔交易的流水号。
2. 首先开启事务，查询并记录当前账户的余额和最后一笔交易的流水号。
3. 然后写入流水记录。
4. 再更新账户余额，需要在更新语句的WHERE条件中限定，只有流水号等于之前查询出的流水号时才更新。

5. 然后检查更新余额的返回值，如果更新成功就提交事务，否则回滚事务。

需要特别注意的一点是，更新账户余额后，不能只检查更新语句是不是执行成功了，还需要检查返回值中变更的行数是不是等于1。因为即使流水号不相等，余额没有更新，这条更新语句的执行结果仍然是成功的，只是更新了0条记录。

下面是整个交易的SQL，供你参考：

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> -- 查询当前账户的余额和最后一笔交易的流水号。
mysql> select balance, log_id from account_balance where user_id = 0;
+-----+-----+
| balance | log_id |
+-----+-----+
|      100 |       3 |
+-----+-----+
1 row in set (0.00 sec)

mysql> -- 插入流水记录。
mysql> insert into account_log values
      -> (NULL, 100, NOW(), 1, 1001, NULL, 0, NULL, 0, 0);
Query OK, 1 row affected (0.01 sec)

mysql> -- 更新余额，注意where条件中，限定了只有流水号等于之前查询出的流水号3时才更新。
mysql> update account_balance
      -> set balance = balance + 100, log_id = LAST_INSERT_ID(), timestamp = NOW()
      -> where user_id = 0 and log_id = 3;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> -- 这里需要检查更新结果，只有更新余额成功（Changed: 1）才提交事务，否则回滚事务。
mysql> commit;
Query OK, 0 rows affected (0.01 sec)
```

最后，我给出流水和余额两个表的DDL，你自己执行例子的时候可以使用。

```
CREATE TABLE `account_log` (
  `log_id` int NOT NULL AUTO_INCREMENT COMMENT '流水号',
  `amount` int NOT NULL COMMENT '交易金额',
  `timestamp` datetime NOT NULL COMMENT '时间戳',
  `from_system` int NOT NULL COMMENT '转出系统编码',
  `from_system_transaction_number` int DEFAULT NULL COMMENT '转出系统的交易号',
  `from_account` int DEFAULT NULL COMMENT '转出账户',
  `to_system` int NOT NULL COMMENT '转入系统编码',
  `to_system_transaction_number` int DEFAULT NULL COMMENT '转入系统的交易号',
  `to_account` int DEFAULT NULL COMMENT '转入账户',
  `transaction_type` int NOT NULL COMMENT '交易类型编码',
  PRIMARY KEY (`log_id`)
);

CREATE TABLE `account_balance` (
  `user_id` int NOT NULL COMMENT '用户ID',
```



```
`balance` int NOT NULL COMMENT '余额',
`timestamp` datetime NOT NULL COMMENT '时间戳',
`log_id` int NOT NULL COMMENT '最后一笔交易的流水号',
PRIMARY KEY (`user_id`)
);
```

小结

账户系统用于记录每个用户的余额，为了保证数据的可追溯性，还需要记录账户流水。流水记录只能新增，任何情况下都不允许修改和删除，每次交易的时候需要把流水和余额放在同一个事务中一起更新。

事务具备原子性、一致性、隔离性和持久性四种基本特性，也就是ACID，它可以保证在一个事务中执行的数据更新，要么都成功，要么都失败。并且在事务执行过程中，中间状态的数据对其他事务是不可见的。

ACID是一种理想情况，特别是要完美地实现CI，会导致数据库性能严重下降，所以MySQL提供的四种可选的隔离级别，牺牲一定的隔离性和一致性，用于换取高性能。这四种隔离级别中，只有RC和RR这两种隔离级别是常用的，它们的唯一区别是在进行的事务中，其他事务对数据的更新是否可见。

思考题

课后希望你能动手执行一下我们今天这节课中给出的例子，看一下多个事务并发更新同一个账户时，RC和RR两种不同的隔离级别，在行为上有什么不同？

欢迎你在留言区与我讨论，如果你觉得今天的内容对你有帮助，也欢迎把它分享给你的朋友。

精选留言：

- 李玥 2020-03-05 10:35:13
hi，我是李玥。

我们还是回顾一下上节课留的思考题，是这样的。

既然用户的购物车数据存放在MySQL或者是Redis中各有优劣势。那能不能把购物车数据存在MySQL中，并且用Redis来做缓存呢？这样不就可以兼顾两者的优势了么？这样做是不是可行？如果可行，如何来保证Redis中的数据和MySQL中的数据是一样的呢？

关于这个问题，我这样看。

用Redis给购物车库做缓存，技术上肯定是可行的。但是有两个问题需要思考一下。

第一个问题是，值不值得这样做？因为每个人的购物车都是不一样的，所以这个缓存它的读写比差距不会很大，缓存的命中率不会太高，缓存的收益有限，为了维护缓存，还要增加系统的复杂度。所以我们就要自行权衡一下，是不是值得的问题。我的观点是，除了超大规模的系统以外，没有必要设置这个缓存。

第二个问题是，如果我们非要做这样一个缓存，用什么缓存更新策略更好呢？这里我先卖个关子，在《11 | MySQL如何应对高并发（一）：使用缓存保护MySQL》这节课中，我们会专门讲到几种常用的缓存策略，你可以学完这节课之后，再回过头来想一下这个问题。 [3赞]

- 观弈道人 2020-03-05 07:15:58

这篇讲事务的文章，是遇到的讲的最清楚明白、恰到好处的一篇，没有提看似更本质、更唬人的各种锁，比如间隙锁等，可以说大部人也都不能清楚理解各种锁，对于非专职dba理解此篇的知识技巧足够了。[3赞]

- 慌张而黑糖 2020-03-05 08:32:04

其中在更新账户余额时的where log_id=3这部分感觉和乐观锁中的version起到的作用很像 [2赞]

- o9 2020-03-06 00:57:52

深入浅出！

对于 RC 隔离级别而言，并发高的情况下会出现，由于更新余额时有 where log_id = 更新之前查询的log_id 所以更新条数为 0，回滚，充值失败，不影响数据的一致性。

对于 RR 隔离级别而言，由于是可重复读的，所以 更新余额的 sql 可以执行成功。

- 公号-云原生程序员 2020-03-05 23:04:14

老师的文章是采用了乐观锁；在采用乐观锁除了要避免出现ABA问题外，还需要注意可能会出现失败的场景。此时，一般可以采用重试的策略；如果并发不是很高，可以在重试多次不行的情况下，考虑降级为悲观锁策略。

- 镜子 2020-03-05 21:31:05

假如两个事务一起执行，把余额扣成负数了，而两个事务都执行了，该如何处理？

- 约书亚 2020-03-05 18:05:25

幻读的定义一直不太明确，有争论，不同定义对于MySQL到底RR解决没解决幻读，结论也不同。

我更倾向权威的定义，引自<<A Critique of ANSI SQL Isolation Levels>>:

P3 (Phantom): Transaction T1 reads a set of data items satisfying some <search condition>. Transaction T2 then creates data items that satisfy T1's <search condition> and commits. If T1 then repeats its read with the same <search condition>, it gets a set of data items different from the first read.

这与本课给的定义不同。

当然不影响本课的核心内容，大家也没必要钻牛角尖

- 知非 2020-03-05 16:30:57

两个事务A,B同时修改余额，RC的情况下只有先执行update语句的事务A能执行成功，B事务的update语句由于读到了A事务更新后的log_id会失败。RR的情况下两个事务都会成功，但是要注意一定是对余额进行加减操作，不能直接把余额变为某个值。

- 大秦皇朝 2020-03-05 14:45:34

李Sir，能否大概提一下RC和RR在一般情况下在多少数据量级别，性能差距能有多少？作为小白来说，自己做这种测试貌似受到其它外界因素影响较大，实际测试出来好像感觉不是很明显。因为您可能比较有经验或者经历过，所以想请教下您能否大概说说？感谢！

- 肥low 2020-03-05 10:07:16

首先赞一下排版 问题难道是想问MVCC么 两种级别生成快照时机不同？

- suke 2020-03-05 09:29:52

老师，那更新失败的情况一般电商系统该如何处理？是要继续尝试？还是直接返回失败

- 每天晒白牙 2020-03-05 09:09:01

又巩固了一遍事务的问题，不过这块确实容易蒙，尤其是面试中，还是需要自己下去梳理整理一下，加深印象

- 墨雨 2020-03-05 08:51:36

我在考虑的是，更新账户余额的时候是不是应该同时更新一下最后一笔交易流水的id呢？

- 赵冲 2020-03-05 08:38:47

老师，DDL语句，为什么插入流水表时，不记录账户余额表的主键呢？

- Cranliu 2020-03-05 08:21:29

RR级别下加的是next-key锁，可以解决幻读的吧？