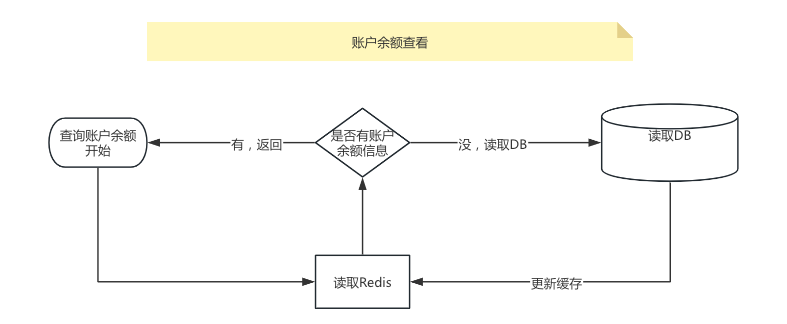
**系统设计**

# 接口设计

## 余额查询接口



## 单个账户余额操作

### 交易的非幂等性

防止客户端重复提交交易，导致多次交易。在每次交易请求时，需要客户端生成唯一的交易ID，同一个交易ID重复提交会提示客户端。

首先在Redis缓存中查询本次交易是否正在执行或者执行完成，如果有，直接返回。

如果没有，可能是缓存已经失效，如果每次去查询数据库来判断交易是否存在，比较耗时，因此在数据库中的交易表transaction\_id的唯一性来决定交易是否存在。

如果可以生成记录，所以没有交易过。如果不能，说明已经交易过。

### 交易的原子性

交易的过程中涉及3张表的写操作，因此需要将三张表的操作放在同一个事务中。

事务的传播：REQUIRED

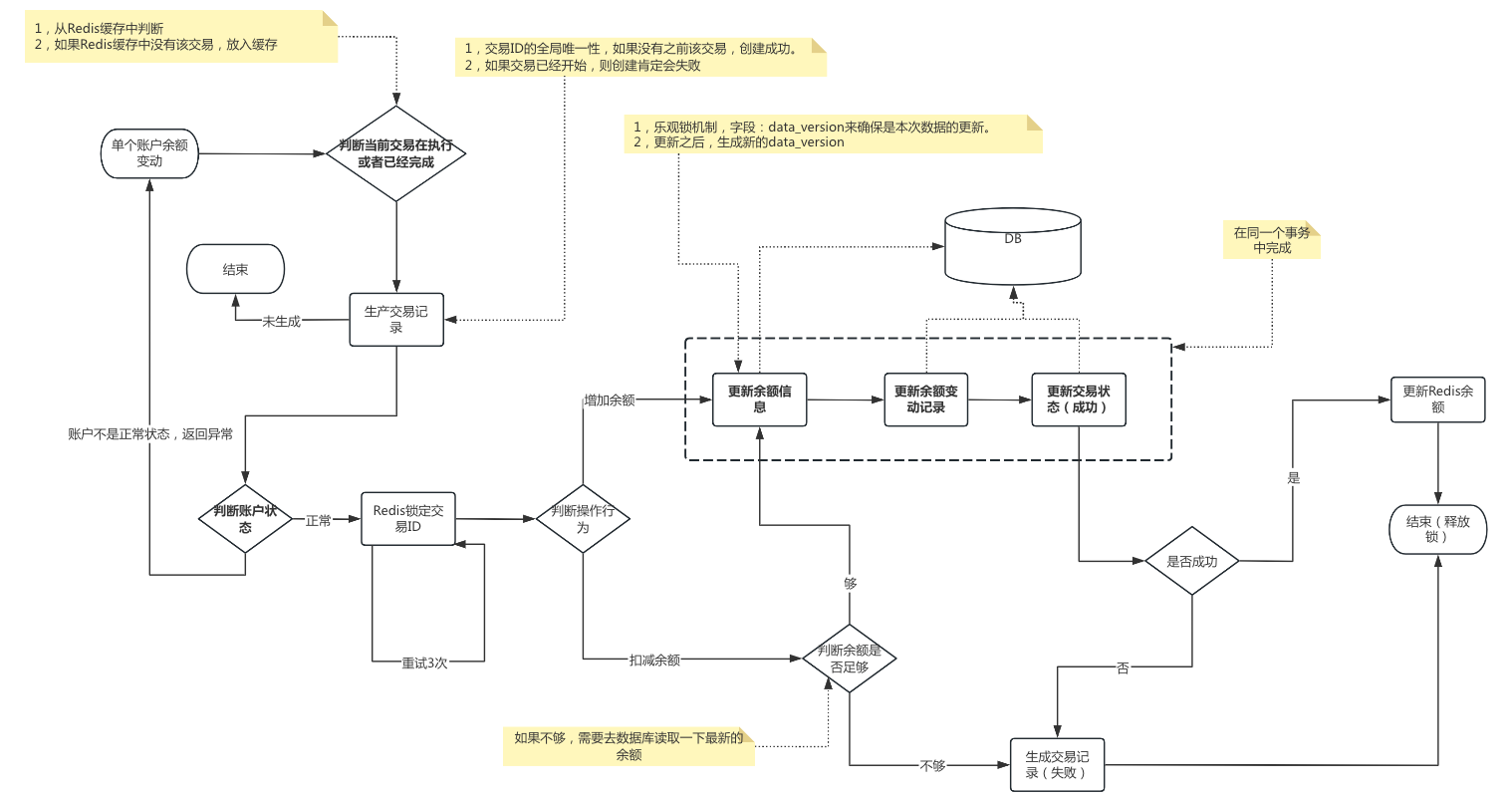
事务的隔离级别： READ\_COMMITTED

有任何异常抛出进行数据回滚。

### 交易的并发性

在同一时刻可能对同一个账户进行并发交易。两个交易获取了用户同样的余额，其中一个交易扣款先完成，另一个交易存款后完成，会导致后面的交易把前面交易给覆盖掉，造成用户余额不对。

1. 悲观锁定账户：每次交易时，锁定该账户【行锁】，等交易完成，释放该锁。该方案可以有效避免交易错乱，但是每次锁定该账户，会造成其他交易（与该账户相关）的等待。
2. 乐观锁定数据版本：给账户表增加一个：data\_version字段，该字段是UUID，每次数据变动之前，读取本次数据版本，在进行数据修改时，需要确保是修改当前data\_version一致的数据，修改赋值新的data\_version。 如果data\_version不一致，修改失败。 该方案有效避免交易的等待。然后通过适当随机的重试机制可以尽量确保并发交易的成功。

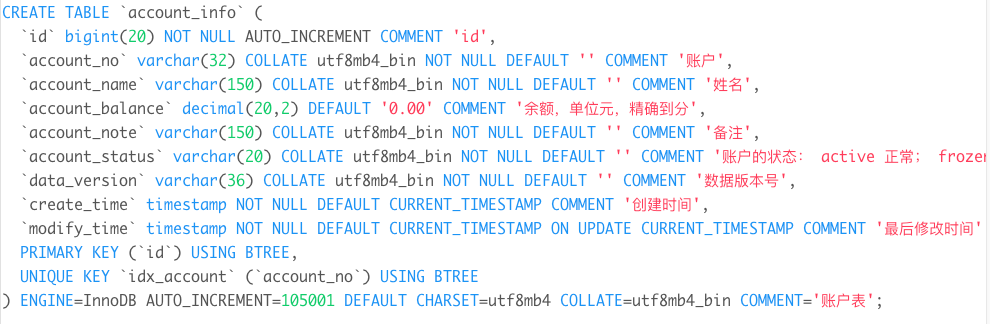


## 两个账户之间转账

两个账户之间转账涉及表的操作更多一些，但核心的设计思想和单个账户余额操作的一致。

# 数据库设计

## 账户表



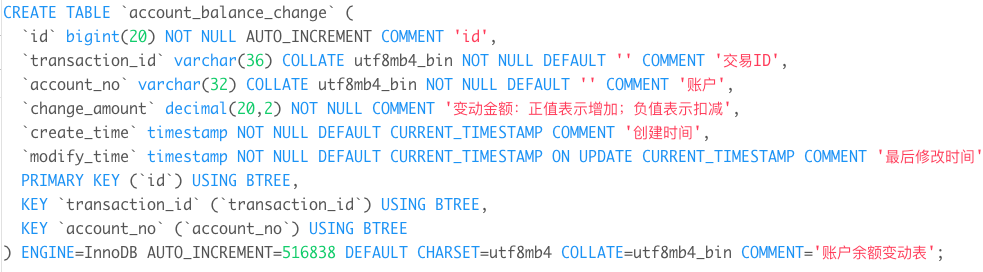
注意：

account\_no 唯一性

account\_status 账户的状态

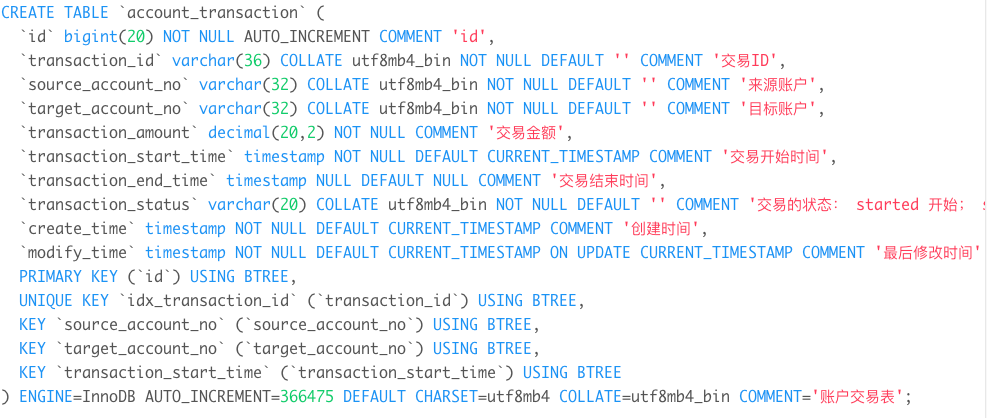
data\_version 数据版本

## 账户余额变动表



每次账户余额的变动需要生成记录，方便后续进行对账，追溯资金的来源。

# 交易表



每次交易生成一条记录，记录交易的信息。

transaction\_id : 唯一性。防止交易重复。