### **一、数据库表设计**

#### **1. 账户表（account）**

CREATE TABLE `account` (

`account\_no` varchar(32) NOT NULL,

`balance` decimal(10,2) NOT NULL DEFAULT '0.00',

`version` int NOT NULL DEFAULT '0',

`created\_time` datetime NOT NULL DEFAULT CURRENT\_TIMESTAMP,

`updated\_time` datetime NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (`account\_no`),

KEY `idx\_updated\_time` (`updated\_time`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

用下面脚本setup一些账户测试数据：

INSERT INTO `account` (`account\_no`, `balance`)  
SELECT   
 CONCAT('AC', LPAD(FLOOR(RAND() \* 10000000000), 10, '0')) AS account\_no,  
 ROUND(RAND() \* 100000, 2) AS balance  
FROM information\_schema.columns a, information\_schema.columns b   
LIMIT 2000;  
commit;

#### **2. 交易表（transaction）**

CREATE TABLE `transaction` (

`tx\_id` varchar(64) NOT NULL COMMENT '交易ID（雪花算法)',

`source\_account\_no` varchar(64) NOT NULL COMMENT '源账号',

`target\_account\_no` varchar(64) NOT NULL COMMENT '目标账号',

`amount` decimal(10,2) NOT NULL COMMENT '交易金额',

`created\_time` datetime NOT NULL DEFAULT CURRENT\_TIMESTAMP,

`updated\_time` datetime NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (`tx\_id`),

KEY `idx\_source\_account\_no` (`source\_account\_no`),

KEY `idx\_target\_account\_no` (`target\_account\_no`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

### **二、REST API接口设计**

#### **1. 创建交易接口**

POST rbc/api/transaction

Content-Type: application/json

{

"sourceAccountNo": "",

"targetAccountNo": "",

"amount": 100.00

}

**响应**  ：

//成功响应

{

"code": 0,

"msg": "success",

"data": {

"txId": "1911669217665617920"

}

}

//失败响应

{

"code": -101,

"msg": "db exception",

"data": null

}

#### **2. 查询账户余额接口**

GET rbc/api/balance/{accountId}

**响应**  ：

//成功响应

{

"code": 0,

"msg": "success",

"data": {

"accountNo": "AC0000398596",

"balance": 26144.02

}

}

//失败响应

{

"code": -101,

"msg": "账户不存在",

"data": null

}

### **系统架构图**

%% 系统架构图

graph TD

A[客户端] --> B(API网关)

B --> C[余额实时计算服务]

C --> D[[Redis集群实现读写分离]]

C --> E[[MySQL集群实现读写分离]]

D --> G[Sentinel实现集群高可用与故障自动切换]

E --> F[[MHA实现集群高可用与故障自动主从切换]]

subgraph 数据层

D

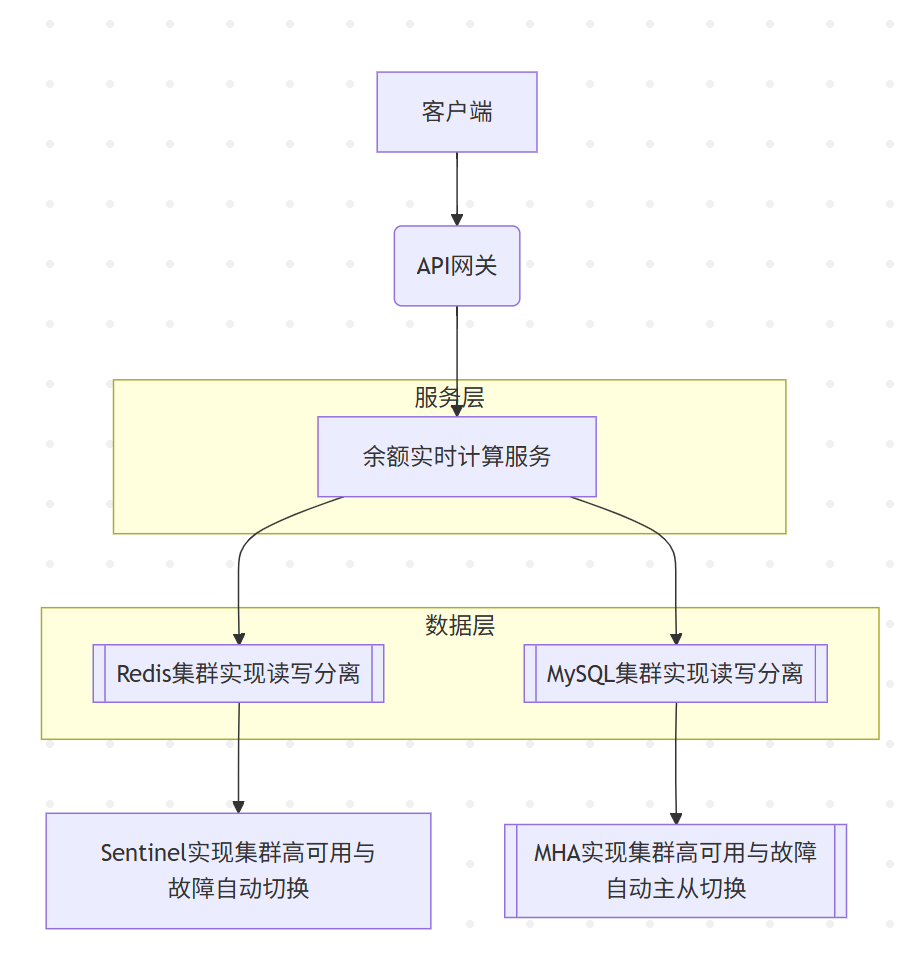
E

end

subgraph 服务层

C

end



因为此项目为demo，受限于机器成本，所以mysql和redis选择使用单节点部署来完成功能测试，如果是生产环境，则必须使用上图的高可用架构部署方案，mysql和redis都可以使用主从架构做读写分离。

且项目暂时只使用一个应用RBC(Realtime Balance Caculation)来进行开发和部署，如果是生产环境，有更多的业务功能，可以拆分成两个服务，一个account-service和一个transaction-service,目前demo项目暂时采用一个应用来完成业务逻辑。后续如果需要迭代，功能更多，可做服务拆分。

生产环境如果数据量大可使用shardingsphere的shardingjdbc对account表进行分表读写处理以进一步提升性能，此demo项目暂时简化这个步骤。

### **系统调用业务时序图**

%% 交易处理时序图

sequenceDiagram

participant 用户

participant 网关

participant 交易服务

participant Redis

participant MySQL

交易服务->>Redis: 启动时将最近高频的更新账户写入redis做缓存预热

用户->>网关: POST /transactions

网关->>交易服务: 交易请求(源账户,目标账户,金额)

交易服务->>Redis: 获取源账户和目标账户信息

alt 源账户和目标账户在redis存在

Redis-->>交易服务: 返回源账户和目标账户的余额，版本号

else 源账户和目标账户在redis不存在

交易服务-->>MySQL: 获取源账户和目标账户信息

MySQL-->>交易服务: 返回源账户和目标账户的余额，版本号

alt 源账户或目标账户在DB不存在

交易服务->>Redis: 对不存在的账户信息写入一个"NULL"缓存防止缓存击穿

交易服务-->>用户: 返回错误代码

else 源账户或目标账户在DB存在

交易服务->>Redis:将源账户和目标账户写入缓存

end

end

alt 余额充足

交易服务->>Redis: 删除原账户和目标账户的redis缓存

交易服务->>MySQL: 基于乐观锁来对源账户余额做扣减和目标账户做增加

MySQL->>交易服务: 返回更新结果的updateCount

opt 如果updateCount任意一个不为1，

交易服务-->>交易服务: 抛出乐观锁加锁异常，并通过spring retry进行重试3次

end

alt 账户更新成功

交易服务->>Redis: 延迟双删方式第二次删除原账户和目标账户的redis缓存

交易服务-->>用户: 返回成功代码

else 账户更新失败

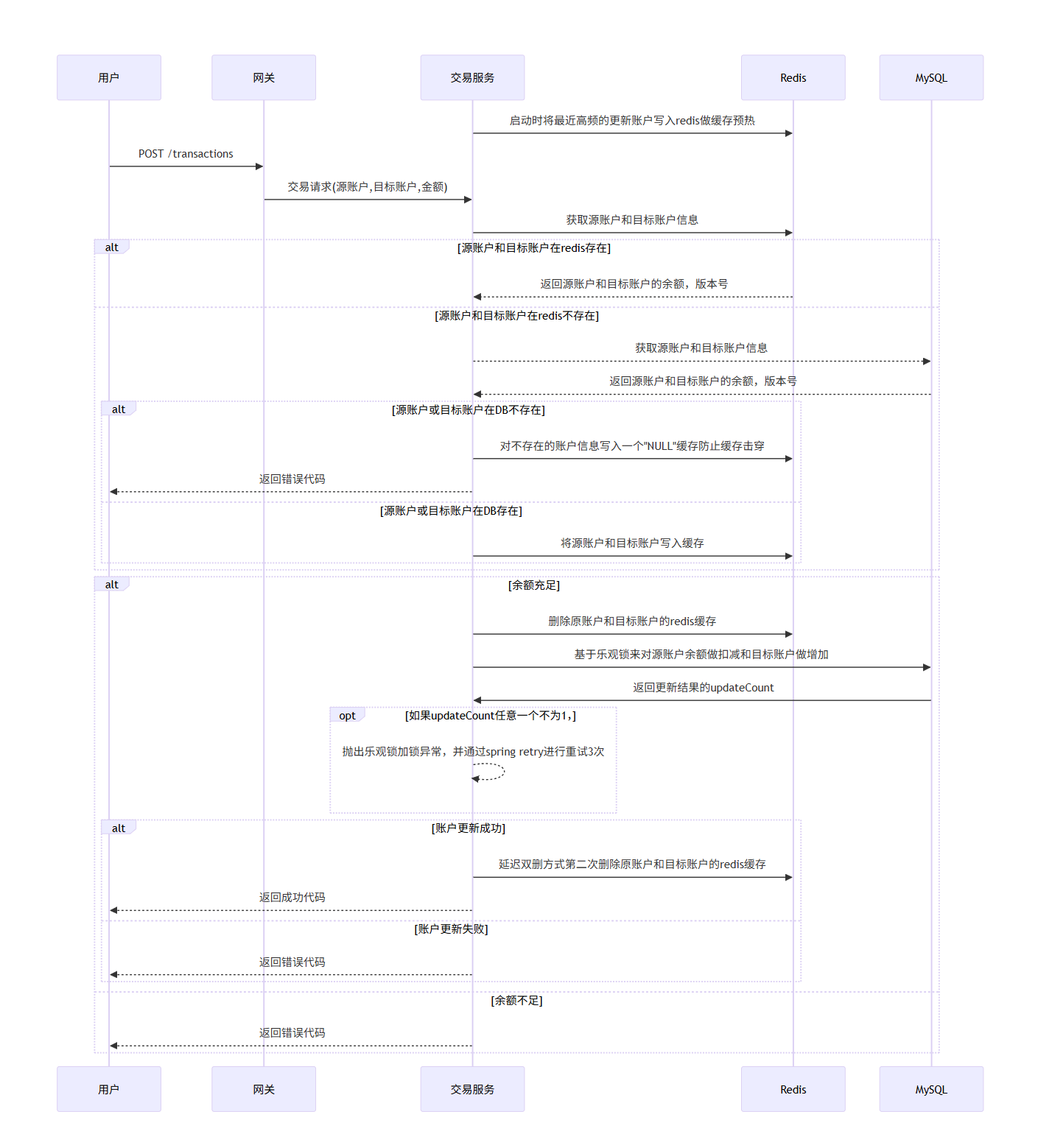
交易服务-->>用户: 返回错误代码

end

else 余额不足

交易服务-->>用户: 返回错误代码

end



缓存使用cach aside+延迟双删的方式来解决数据库和缓存的数据一致性问题，

账户更新使用乐观锁+spring retry重试机制来尽力使账户更新成功。