Wallet Demo

目录

1. 设计背景

1.1 目标

此设计的目标是设计一个钱包系统来维护币种，和币种对应的余额信息，一个钱包可以支持多种币，大致支持以下几个功能：

- **钱包系统需要能够支持2种操作，充值入账（增加余额）和提币（减少余额）**
- **钱包系统需要提供记账的能力，能够回溯余额的所有变化进行对账**
- **钱包系统需要提供交易记录查询的功能**

1.2 实现方式

在钱包设计的过程当中，我们需要考虑不同场景下的需求，比如，传统钱包和加密货币钱包的场景，两者因为底层货币形式的不同，往往会有不一样的设计思路

特性	传统钱包	加密货币钱包
账本管理	中心化	去中心化
资产控制权	由机构托管	由用户掌握

特性	传统钱包	加密货币钱包
资金出入问题解决	实时验证+事务锁定	区块链共识算法
安全性	信任中心机构	私钥决定，安全性由用户负责
匿名性与监管	实名，受监管	匿名，弱监管

本文会根据这些异同，尝试给出不一样的设计方案。

2 传统场景下的钱包设计

传统场景下的钱包，用户会在自己的账户下创建钱包账户，存入不同币种的货币，如，美金、人民币等，本节会基于这种场景展开设计，根据设计目标，会提供入账、提币、记账对账和交易记录查询的功能。

2.1 项目设计结构

```
wallet-demo/
├── models/               # 数据库模型
│   ├── base.go           # 公共模型基础结构
│   ├── wallet.go         # 钱包模型
│   ├── transaction.go    # 交易模型
│   └── reconciliation.go # 对账模型
├── services/             # 服务层逻辑
│   ├── wallet_service.go # 钱包服务
│   └── reconciliation_service.go # 对账服务
├── controllers/          # 控制器
│   └── wallet_controller.go # 钱包相关接口
├── main.go               # 项目入口
└── README.md             # 项目介绍文档
```

models 层定义了一些实体类，services 层封装了对应功能的核心代码，controller 则将这些功能通过接口方式暴露出来，实际项目中需要更细粒度拆分，例如 service 层可拆分成 repository 层和 service 层等，但本设计主要目的是 demo 主要功能，对项目结构上做了一定简化，下面会主要介绍表结构设计以及 service 层中核心功能设计。

2.2 表结构设计

除去公共字段的设计，表结构主要有3部分:

2.2.1 wallet

```
type Wallet struct {
        Base
        UserID   uint            `gorm:"not null;index"`
        Currency string          `gorm:"not null;size:10"`
        Balance  decimal.Decimal `gorm:"not null;default:0"`
}
```

基于实现功能的目的，这是 wallet 表的最小实现：

- UserID: 标志该钱包归属于哪一个用户
- Currency: 标志是该用户下的哪一个货币钱包，不同货币的钱包需要相互独立
- Balance: 当前货币钱包余额

考虑到查询效率，可以考虑给 UserID 和 Currency 建立联合索引，这样可以满足查询某个用户下的所有货币钱包和查询某个用户下的某个货币钱包这2种高频查询场景。

2.2.2 transaction

```
type TransactionType string

const (
        TransactionDeposit  TransactionType = "deposit"
```

```
        TransactionWithdraw TransactionType = "withdraw"
)

type Transaction struct {
        Base
        WalletID      uint            `gorm:"not null;index"`
        Type          TransactionType `gorm:"not null;size:20"`
        Amount        decimal.Decimal `gorm:"not null"`
        BalanceBefore decimal.Decimal `gorm:"not null"`
        BalanceAfter  decimal.Decimal `gorm:"not null"`
        Status        string          `gorm:"not null;default:'pending'"`
        TxHash        string          `gorm:"size:100;index"`
        Description   string          `gorm:"size:255"`
}
```

transaction 表会记录钱包中的交易记录，字段介绍如下：

- WalletID: 标记该条交易记录归属于哪一个钱包
- Type: 标记该条交易记录的类型，deposit 或者 withdraw
- Amount: 交易金额
- BalanceBefore: 交易发生前的余额
- BalanceAfter: 交易发生后的余额
- Status: 交易状态，默认 pending，即执行中
- TxHash: 交易的唯一 hash
- Description: 交易描述

可以预想到的高频查询有：

- 查询某个钱包下的所有交易记录
- 查询某个钱包下的某种类型交易记录
- 查询某个钱包下某个状态的交易记录

所以可以考虑给 WalletID + Type 或 Status 添加索引，同时给 TxHash 添加索引可以加速对某条交易的检索速度。

2.2.3 reconciliation

在介绍 reconciliation 表之前需要先说明一下什么是 reconciliation：

> In bookkeeping, a bank reconciliation or Bank Reconciliation Statement (BRS) is the process by which the bank account balance in an entity's books of account is reconciled to the balance reported by the financial institution in the most recent bank statement. Any difference between the two figures needs to be examined and, if appropriate, rectified. Ref

> 总的来说，对账（Reconciliation）指在金融、会计或业务过程中，通过核对和匹配不同来源的记录和数据，确保它们的一致性和准确性

```
type ReconciliationStatus string

const (
        ReconciliationStatusPending  ReconciliationStatus = "pending"
        ReconciliationStatusMatched  ReconciliationStatus = "matched"
        ReconciliationStatusMismatch ReconciliationStatus = "mismatch"
)

type Reconciliation struct {
        Base
        WalletID        uint                `gorm:"not null;index"`
        StartTime       time.Time           `gorm:"not null"`
        EndTime         time.Time           `gorm:"not null"`
        SystemBalance   decimal.Decimal     `gorm:"not null"` // 系统计算的余额
        ExternalBalance decimal.Decimal     `gorm:"not null"` // 外部系统的余额
        Status          ReconciliationStatus `gorm:"not null"`
        Difference      decimal.Decimal     `gorm:"not null"` // 差额
        Notes           string              `gorm:"type:text"`
}
```

字段介绍如下：

- WalletID: 标志哪一个钱包下的对账记录
- StartTime&EndTime: 标志对账的时间区间
- SystemBalance: 标志系统计算的余额
- ExternalBalance: 外部系统，审计等对账余额值
- Status: 对账状态，pending - 对账中；matched - 匹配； mismatch - 账没匹配上
- Difference: 差额，因为精读等原因会有个 tolerance 范围，例如 0.001，所以账匹配上这个字段也可能有值
- Notes: 一些备注等

因为可能会查询某个钱包下的某个时间段的对账记录，所以考虑给 WalletID 和 StartTime&EndTime 添加索引

2.3 核心功能设计

2.3.1 充值入账（增加余额）

代码详见 此处，方法签名如下

```
func (s *WalletService) Deposit(walletID uint,
        amount decimal.Decimal, txHash string) error
```

函数参数：

- `walletID` (uint): 钱包的唯一标识符
- `amount` (decimal.Decimal): 存款金额
- `txHash` (string): 交易哈希，确保每笔交易只有一次处理

函数步骤：

1. **检查存款金额是否有效：**

- 使用 `amount.LessThan(decimal.Zero)` 检查存款金额是否小于零
- 如果是无效金额，返回错误 `"invalid deposit amount"`

2. **开启数据库事务：**

- 通过 `s.db.Transaction` 开启一个数据库事务，确保所有操作要么成功，要么回滚

3. **锁定钱包记录：**

- 使用 `tx.Clauses(clause.Locking{Strength: "UPDATE"})` 对钱包记录加锁，避免并发修改相同钱包的余额。这里用到的是数据库悲观锁，在强一致性要求下，用悲观锁保证余额操作被正确处理
- 通过 `tx.First(&wallet, walletID)` 查找对应的 `walletID` 钱包记录

4. **检查交易哈希是否已存在：**

- 使用 `tx.Where("tx_hash = ?", txHash).First(&existingTx)` 查找数据库中是否已经存在相同的交易哈希
- 如果找到了已有记录，返回错误 `"transaction already processed"` ，避免重复处理交易（即重复入账问题）
- 如果未找到记录，继续执行后续操作

5. **更新钱包余额：**

- 保存操作前的余额 `balanceBefore := wallet.Balance`
- 将钱包余额更新为 `wallet.Balance.Add(amount)` ，即增加存款金额

6. **保存钱包记录：**

- 使用 `tx.Save(&wallet)` 保存更新后的钱包余额

7. **记录交易：**

- 创建一个 `models.Transaction` 实例，记录交易信息，包括：
  - `WalletID` : 钱包ID
  - `Type` : 交易类型，固定为存款 `models.TransactionDeposit`

- `Amount`：存款金额
- `BalanceBefore`：存款前余额
- `BalanceAfter`：存款后余额
- `Status`：交易状态，设为 `"completed"`
- `TxHash`：交易哈希

8. **保存交易记录：**

- 使用 `tx.Create(&transaction)` 将交易记录保存到数据库

2.3.2 提款（减少余额）

代码详见 此处，方法签名如下

```
func (s *WalletService) Withdraw(walletID uint,
        amount decimal.Decimal, txHash string) error
```

函数参数：

- `walletID` (uint): 钱包的唯一标识符
- `amount` (decimal.Decimal): 提款金额
- `txHash` (string): 交易哈希，确保每笔交易只有一次处理

函数步骤：

1. **检查取款金额是否有效：**

- 使用 `wallet.Balance.LessThan(amount)` 检查取款金额是否合理
- 如果是无效金额，返回错误 `"insufficient balance"`

2. **开启数据库事务：**

- 通过 `s.db.Transaction` 开启一个数据库事务，确保所有操作要么成功，要么回滚

3. **锁定钱包记录：**

- 使用 `tx.Clauses(clause.Locking{Strength: "UPDATE"})` 对钱包记录加锁，避免并发修改相同钱包的余额。这里用到的是数据库悲观锁，在强一致性要求下，用悲观锁保证余额操作被正确处理
- 通过 `tx.First(&wallet, walletID)` 查找对应的 `walletID` 钱包记录

4. **检查交易哈希是否已存在：**

- 使用 `tx.Where("tx_hash = ?", txHash).First(&existingTx)` 查找数据库中是否已经存在相同的交易哈希
- 如果找到了已有记录，返回错误 `"transaction already processed"`，避免重复处理交易（即双花问题）
- 如果未找到记录，继续执行后续操作

5. **更新钱包余额：**

- 保存操作前的余额 `balanceBefore := wallet.Balance`
- 将钱包余额更新为 `wallet.Balance.Sub(amount)`，即减少取款金额

6. **保存钱包记录：**

- 使用 `tx.Save(&wallet)` 保存更新后的钱包余额

7. **记录交易：**

- 创建一个 `models.Transaction` 实例，记录交易信息，包括：
  - `WalletID`：钱包ID
  - `Type`：交易类型，固定为存款 `models.TransactionWithdraw`
  - `Amount`：取款金额
  - `BalanceBefore`：取款前余额
  - `BalanceAfter`：取款后余额

- `Status`：交易状态，设为 `"completed"`
- `TxHash`：交易哈希

8. **保存交易记录：**

- 使用 `tx.Create(&transaction)` 将交易记录保存到数据库

2.3.3 记账对账

代码详见 此处，方法签名如下

```
// PerformReconciliation 执行对账操作
func (s *ReconciliationService) PerformReconciliation(walletID uint,
        startTime, endTime time.Time,
        externalBalance decimal.Decimal) (*models.Reconciliation, error)
```

函数参数：

- `walletID` (uint): 钱包的唯一标识符
- `startTime` & `endTime` (time.Time): 对账的流水时间范围
- `externalBalance` (decimal.Decimal): 外部传入的余额，用于对账核准

函数步骤：

1. **获取开始时间之前的最后一个余额：**

- 获取开始时间之前最近的一笔交易的 BalanceAfter，作为初始余额 initialBalance

2. **计算时间段内的所有变动：**

- 获取在 [startTime, endTime] 时间段内的所有交易记录

3. **计算最终系统余额：**

- 从初始余额开始，根据时间段内的交易类型和金额，计算最终系统余额 systemBalance

4. **判断对账结果：**

- 比较系统余额与外部余额，判断对账状态

5. **创建对账记录：**

- 将对账结果保存到数据库中

2.3.4 交易记录查询

代码详见 此处，方法签名如下

```
func (s *WalletService) GetTransactions(walletID uint,
        startTime, endTime time.Time,
        page, pageSize int) ([]models.Transaction, error)
```

该方法提供查询某个钱包某个时间段内的所有交易记录

2.3.5 设计总结与可能的问题

Deposit 和 Withdraw 有类似的设计场景：

- 该函数通过事务保证了数据的一致性和原子性
- 通过悲观锁确保了在并发环境中钱包余额的正确更新
- 检查交易哈希确保每笔交易只处理一次
- 如果所有操作成功，事务提交并保存相关数据；如果过程中出现错误，事务会回滚，确保系统状态的一致性，同时悲观锁和交易唯一 hash 也可以避免双花的问题

- 数据库锁可能导致高并发请求时的性能瓶颈，如果数据库负载较重，考虑使用分布式锁机制（例如：使用 Redis 实现分布式锁），在应用层进行控制，减少数据库锁的负担
- 可能会存在 hash 碰撞的问题，可能性较小，确保哈希生成使用强加密算法（例如 SHA-256），以最大程度减少碰撞的可能性，也可以通过再校验一次业务 ID 来确保交易的唯一性
- 还可能存在超时的问题，无论是网络还是其他应用层面的问题，可能会造成锁长时间未释放或者其他问题，可以在应用层面做超时处理，合理处理相关事务

Deposit/Withdraw 分析：

1. 主要操作：
   - 锁定钱包记录（SELECT ... FOR UPDATE
   - 检查交易哈希是否存在
   - 更新钱包余额
   - 创建交易记录
2. 吞吐量限制：至少需要 2 次读操作（查询钱包和检查交易哈希）和 2 次写操作（更新钱包余额和创建交易记录）
3. 优化方案：
   - 为 tx_hash 添加唯一索引，提高查询效率，也可以引入缓存或布隆过滤器，加速确认是否存在该笔交易
   - 考虑使用分布式锁机制（例如：使用 Redis 实现分布式锁），在应用层进行控制，减少数据库锁的负担
   - 如果系统允许，考虑将多笔小额存款合并为批量操作，减少事务数量
   - 缓存钱包余额，减少数据库查询频率
   - 将存款操作拆分为队列任务，通过消息队列（如 Kafka、RabbitMQ）实现异步更新，减轻数据库即时写入压力

Reconciliation 分析：

1. 主要操作
   - 读取最后余额，查询指定钱包在起始时间前的最后一笔交易记录
   - 读取时间段内交易记录，查询指定时间范围内所有交易记录
   - 计算余额变化，遍历交易记录计算余额变化
   - 写入对账记录，将对账结果插入 Reconciliation 表
2. 吞吐量限制
   - 查询操作如获取最后余额，查询时间段内交易记录依赖 wallet_id 和 created_at 的索引性能，数据量增大时查询效率可能下降，大量对账操作并发时，表扫描和索引查找可能成为瓶颈
   - 遍历交易记录的操作复杂度为 O(n)，当交易记录数量多时对吞吐量产生影响
3. 优化方案：
   - 为 transactions 表创建联合索引 (wallet_id, created_at)
   - 合并对账请求，批量处理多个钱包的对账操作，减少事务提交次数
   - 将最近的余额缓存到 Redis，减少查询 transactions 表的频率
   - 将查询操作分流到从库，降低主库压力
   - 使用消息队列（如 Kafka 或 RabbitMQ）异步处理对账任务
   - 定期计算和存储钱包余额快照，直接使用快照减少实时计算开销

Transactions 查询分析

1. 主要操作
   - 根据 wallet_id 和时间范围查询交易记录
   - 支持分页，通过 Offset 和 Limit 限制每次返回的数据量
   - 按时间逆序排序（created_at DESC）
2. 吞吐量限制
   - 查询范围越大（时间跨度较长），需要扫描的记录越多，可能影响吞吐量
   - 随着 page 增加，Offset 的数值增大，数据库可能扫描更多记录来计算偏移
   - 多用户同时分页查询时，数据库需要处理大量随机查询请求，容易引发资源争用
3. 优化方案
   - 创建基于 wallet_id 和 created_at 的联合索引，加速查询和排序
   - 如果时间跨度较长，可以限制查询范围，如将时间划分为固定周期段（天、月等）
   - 索引覆盖查询，仅返回需要的字段，减少查询返回的数据量，降低 I/O 压力

- 合理利用内存进行分页，而不是在数据库层面分页查询

3 加密货币钱包设计

加密货币钱包与传统钱包有相似之处，也有不同之处，用户会在自己的账户下创建钱包账户，存入不同的加密货币 BTC, ETH, etc，本节会基于这种场景展开设计，根据设计目标，会提供入账、提币、记账对账和交易记录查询的功能。

3.1 项目设计结构

```
wallet-demo/
├── models/               # 数据库模型
│   ├── base.go           # 公共模型基础结构
│   ├── crypto_wallet.go         # 钱包模型
│   ├── crypto_transaction.go    # 交易模型
│   └── crypto_reconciliation.go # 对账模型
├── services/             # 服务层逻辑
│   ├── mock_blockchain.go # 模拟链
│   ├── crypto_wallet_service.go # 钱包服务
│   └── crypto_reconciliation_service.go # 对账服务
├── controllers/          # 控制器
│   └── crypto_wallet_controller.go # 钱包相关接口
├── main.go               # 项目入口
└── README.md             # 项目介绍文档
```

与传统钱包不同的是，加密货币钱包会有一个模拟链，模拟加密货币的网络，实现细节，该代码实现了一个 模拟区块链系统 (MockBlockchain)，用于模拟区块链的交易发送、确认、账户余额查询、交易历史查询等功能

模拟区块链核心数据结构

- BlockchainTransaction：代表区块链上的交易，包含以下关键属性：
  - Hash：交易的唯一标识符（哈希值）
  - From/To：交易的发送方和接收方地址
  - Amount：交易金额（*big.Int，适合处理大整数）
  - BlockNumber：交易所在区块的编号
  - Confirmations：交易已确认的区块数
  - Status：交易状态（pending, success, failed）
  - Fee：交易手续费（模拟的 gas 费）
  - Raw：原始交易数据（可以存储序列化后的数据）
- MockBlockchain：模拟区块链网络，主要包含以下成员：
  - mutex：读写锁，用于保证多线程访问安全
  - transactions：交易记录，存储所有交易数据
  - balances：账户余额，存储每个地址的当前余额。
- 关键方法：
  - NewMockBlockchain():
    - 创建新的模拟区块链
    - 初始化交易和余额的空映射
  - SendTransaction()
    - 模拟区块链交易发送
    - 检查发送方余额
    - 生成唯一交易哈希
    - 创建交易记录
    - 更新发送方和接收方余额
    - 启动协程模拟交易确认过程
  - simulateConfirmations()
    - 协程模拟区块链确认过程
    - 6个确认后将交易标记为成功
  - GetTransaction()
    - 通过哈希获取特定交易

        - 使用读锁确保线程安全
    - GetTransactionHistory()
        - 获取特定地址在给定时间范围内的交易
        - 筛选发送方或接收方为指定地址的交易
    - GetAddressBalance()
        - 返回给定地址的当前余额

3.2 表结构设计

3.2.1 wallet

```
type Network string

const (
        NetworkBTC  Network = "BTC"
        NetworkETH  Network = "ETH"
        NetworkBSC  Network = "BSC"
        NetworkTRON Network = "TRON"
)

type CryptoWallet struct {
        Base
        UserID      uint    `gorm:"not null;index"`
        Network     Network `gorm:"size:10;not null"`
        Address     string  `gorm:"size:100;not null;uniqueIndex"`
        Balance     float64 `gorm:"not null;default:0"`
        PrivateKey  string  `gorm:"size:255"`  // Consider encryption
        AddressPath string  `gorm:"size:50"`   // BIP44 derivation path
        ExtraData   string  `gorm:"type:text"` // Network-specific data
}
```

字段介绍：

- UserID: 关联用户的唯一标识
- Network: 区块链网络类型
- Address: 钱包地址
- Balance: 钱包余额
- PrivateKey: 钱包私钥
- AddressPath: BIP44 派生路径
- ExtraData: 网络特定的额外数据

> PrivateKey（私钥）
>
> - 区块链中最关键的安全凭证
> - 用于签名交易和证明资产所有权
> - 对应公钥可以生成钱包地址

> AddressPath（地址派生路径）
>
> - 地址派生路径（BIP44 Derivation Path）是一种在分层确定性钱包（HD Wallet）中生成和管理加密货币地址的标准方法
> - 使用一个种子（Seed）生成所有地址
> - 从单一种子派生出多个地址
> - 只需备份初始种子即可恢复所有地址

3.2.2 transaction

```
type CryptoTransaction struct {
        Base
        WalletID      uint            `gorm:"not null;index"`
        Type          TransactionType `gorm:"not null;size:20"`
        Network       Network         `gorm:"size:10;not null"`
        FromAddress   string          `gorm:"size:100;not null"`
```

```
        ToAddress     string          `gorm:"size:100;not null"`
        Amount        float64         `gorm:"not null"`
        Status        string          `gorm:"not null;default:'pending'"`
        TxHash        string          `gorm:"size:100;index"`
        Confirmations int             `gorm:"default:0"`
        BlockNumber   uint64          `gorm:"default:0"`
        GasPrice      string          `gorm:"size:50"`
        GasUsed       uint64          `gorm:"default:0"`
        Raw           string          `gorm:"type:text"`
}
```

字段介绍：

- WalletID: 关联交易的钱包ID
- Type: 交易类型 deposit/withdraw
- Network: 交易所属区块链网络
- FromAddress: 发送方地址
- ToAddress: 接收方地址
- Amount: 交易金额
- Status: 交易状态
- TxHash: 交易哈希
- Confirmations: 区块确认数
- BlockNumber: 交易所在区块号
- GasPrice: 交易gas价格
- GasUsed: 实际消耗的gas数量
- Raw: 原始交易数据

> Gas 是以太坊中衡量计算工作量的单位，类似于汽车中的"燃料"。每一个在以太坊虚拟机（EVM）中执行的操作都会消耗一定数量的 Gas

> GasUsed（Gas 消耗量）,交易实际消耗的 Gas 数量,取决于交易的复杂度,不同操作消耗的 Gas 不同

3.2.3 reconciliation

```
type CryptoReconciliation struct {
        Base
        WalletID       uint `gorm:"not null;index"`
        StartTime      time.Time
        EndTime        time.Time
        SystemBalance  float64
        ChainBalance   float64
        Status         ReconciliationStatus
        Difference     float64
        MismatchReason string `gorm:"type:text"`
        UnmatchedTxs   string `gorm:"type:text"` // JSON array of unmatched transaction hashes
}
```

字段介绍：

- WalletID: 关联特定钱包的唯一标识
- tartTime&EndTime: 对账的时间范围
- SystemBalance: 系统内部记录的余额
- ChainBalance: 区块链上实际的余额
- Status: 对账状态
- Difference: 系统余额与链上余额的差值
- MismatchReason: 余额不匹配的原因
- UnmatchedTxs: 未匹配的交易哈希

3.3 核心功能设计

3.3.1 充值入账（增加余额）

代码详见 此处，方法签名如下

```
// ProcessDeposit 充值
func (s *CryptoWalletService) ProcessDeposit(walletID uint, txHash string) error
```

函数参数：

- `walletID` (uint): 钱包的唯一标识符
- `txHash` (string): 交易对应的 hash 值

函数步骤：

1. **重复交易检查：**

- 防止同一笔交易重复处理，使用交易哈希作为唯一标识，如果交易已存在，直接返回错误

2. **区块链交易获取：**

- 从区块链获取交易详细信息，获取失败则返回错误

3. **确认数验证：**

- 要求至少6个区块确认，提高安全性，防止双花攻击

4. **钱包和地址验证：**

- 验证钱包是否存在，校验收款地址与钱包地址一致

5. **数据库事务处理：**

- 使用数据库事务确保原子性，余额更新和交易记录同时成功或失败

> 入金（Deposit）场景是钱包被动接受资产，所以只需要验证这笔交易即可，不需要通过链上发起交易（Send transaction）

3.3.2 提款（减少余额）

代码详见 此处，方法签名如下

```
// Withdraw 提现功能
func (s *CryptoWalletService) Withdraw(walletID uint,
        toAddress string, amount float64) (string, error)
```

函数参数：

- `walletID` (uint): 钱包的唯一标识符
- `toAddress` (string): 接收方地址
- `amount` (float64): 出金金额

函数步骤：

1. **钱包验证：**

- 检查钱包是否存在

2. **余额检查：**

- 验证钱包余额是否充足，防止透支

3. **区块链交易：**

- 调用区块链服务发送交易，失败则回滚事务

4. **余额更新：**

- 扣减钱包余额，使用数据库表达式确保原子性

5. **交易记录：**

- 创建提现交易记录，初始状态为 "processing"

> 因为是出金（Withdraw），需要上链发起交易，这里都是模拟过程

3.3.3 记账对账

代码详见 此处，方法签名如下

```
func (s *CryptoReconciliationService) PerformReconciliation(walletID uint,
        startTime, endTime time.Time) (*models.CryptoReconciliation, error)
```

函数参数：

- `walletID` (uint): 钱包的唯一标识符
- `startTime` & `endTime` (time.Time): 对账的时间范围

函数步骤：

1. **钱包验证：**

- 检查钱包是否存在，获取钱包基本信息

2. **系统交易获取：**

- 获取指定时间范围内的系统交易记录，根据钱包ID和时间范围筛选

3. **链上交易获取：**

- 从区块链获取指定地址和时间范围的交易记录

4. **链上余额获取：**

- 获取区块链上的实际余额

5. **创建对账记录：**

- 初始化对账记录，记录系统余额和链上余额，计算余额差异

6. **差异分析：**

- 比较系统余额和链上余额，果差异超过阈值，标记为不匹配，调用差异分析方法

7. **保存对账记录：**

- 将对账记录保存到数据库

> 与传统钱包不同的时，在对账时，加密货币钱包可以与链上余额进行对比，但也只是数据源的差异，实际上的对账逻辑差不多

3.3.4 交易记录查询

代码详见 此处，方法签名如下

```
// GetTransactions 获取交易信息历史
func (s *CryptoWalletService) GetTransactions(walletID uint,
        page, pageSize int) ([]models.CryptoTransaction, int64, error)
```

该方法提供查询加密货币钱包某个时间段内的所有交易记录

3.3.5 设计总结与可能的问题

Deposit 分析：

1. 主要操作
    - 检查重复交易

- 获取区块链交易
- 验证确认数
- 获取钱包信息
- 验证接收地址
- 更新余额
- 记录交易

2. 性能分析（耗时操作）
   - 重复交易查询
   - 钱包信息获取
   - 余额更新
   - 交易记录创建
   - 区块链交互，GetTransaction 调用耗时
3. 优化方案
   - 将最近处理的交易缓存起来，提高重复交易查询的速度
   - 钱包信息添加缓存，更新信息时同步更新缓存
   - 把 deposit 放入消息队列异步处理，减少接口等待速度
   - 区块链确认及数据库事务本身可以防止双花的情况，我们仍可以换成分布式锁，提高应用的锁处理速度
   - 批量处理请求，减少网络、磁盘的 IO 次数

Withdraw 分析：

加密货币的 Withdraw 多出在链上发起交易这一步，所以会增加不确定性

1. 主要操作
   - 检查钱包
   - 验证余额
   - 发起区块链交易
   - 扣减余额
   - 记录交易
2. 性能分析（耗时操作）
   - 钱包信息查询
   - 余额扣减
   - 交易记录创建
   - 交易记录创建
   - 区块链交互，SendTransaction 调用耗时，交易广播，网络延迟
3. 优化方案
   - 基本可以参考 Deposit

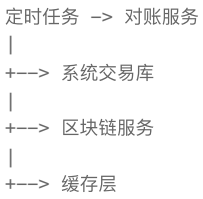
Reconciliation 分析：

1. 主要操作

   - 获取钱包信息
   - 获取系统交易记录
   - 获取链上交易记录
   - 获取链上余额
   - 创建对账记录
   - 分析差异
2. 性能分析

   - 大量数据处理
   - 跨数据源查询
   - 复杂的对账算法
   - 区块链交互延迟
3. 优化方案

- 增量对账，考虑只对增量部分进行对账，例如从上次对账完成之后，进行增量对账
- 并行处理



- 缓存钱包信息及交易数据

Transactions 查询分析：

与传统方式钱包基本一致