WeBankBlockchain Gov Documentation

发布 *v2.6.0*

blockchain-dev

2021 年 **02** 月 **03** 日

组件介绍

什么是 WeBankBlockchain-Governance

随着区块链技术的不断发展和应用的加速落地，区块链参与者不再仅仅只关注于共识算法选择、性能扩展等技术问题，也开始关注对等协作的参与者如何解决分歧、减少摩擦、达成共识、多方共治，面向区块链业务和技术体系的治理逐渐成为焦点。

WeBankBlockchain-Governance 是微众银行立足于分布式协作的联盟链场景，提炼和解决了常见的痛点和问题，所打造的一系列简单好用的组件，助力区块链社区、生态和产品协调发展。

WeBankBlockchain-Governance 是一套稳定、高效、安全的区块治理组件解决方案，可无缝适配FISCO BCOS区块链底层平台。 首批开源的有账户治理组件(WeBankBlockchain-Governance-Account)、权限治理组件(WeBankBlockchain-Governance-Auth)、 私钥管理组件(WeBankBlockchain-Governance-Key)和证书管理组件（WeBankBlockchain-Governance-Cert）。

这四个组件分别从私钥丢失重置、合约权限细粒度管控、私钥和证书的全生命周期管控等方面着手，提供了可部署的智能合约代码、易于使用的SDK和可参考的落地实践Demo等交付物。 我们将竭尽所能在实践中持续探索和开发新的治理组件，也希望社区的朋友们一起加入进来，不断为WeBankBlockchain-Gov补充新鲜血液。

设计目标

WeBankBlockchain-Governance 提出了轻量解耦、通用场景、一站式、简洁易用的四大目标。

- **轻量解耦**。所有的治理组件与具体的业务解耦。可轻量化集成，在不侵入底层的前提下可插拔。通过类库、智能合约、SDK等多种方式提供。使用者甚至只需要使用链控制台，就可以部署和管控治理过程。
- **通用场景**。所有治理组件所瞄准的都是所有联盟链治理中的"刚需"场景，例如首批开源的账户重置、合约权限、私钥和证书的生命周期管理，账户、合约、私钥和证书堪称联盟链技术及上层治理的基石。
- **一站式**。链治理通用组件致力于提供一站式的使用体验。以私钥管理组件为例，支持多种私钥生成方式和格式、覆盖几乎所有主流场景，提供基于文件、多数据库等托管方式，并支持私钥派生、分片等加密方式。
- **简洁易用**。致力于提供简洁的使用体验，让用户轻松上手。

WeBankBlockchain-Governance 定位为区块链链治理组件，不仅希望在开发层面提供趁手的工具，更希望在实践层面为区块链参与者提供可参考落地的实践案例，从整体上助力区块链行业治理水平的提升。

组件简介

- **WeBankBlockchain-Governance-Account 账户治理组件**

基于智能合约开发，提供区块链用户账户注册、私钥重置、冻结、解冻等账户全生命周期管理，支持管理员、阈值投票、多签制等多种治理策略。 请参考 文档

- **WeBankBlockchain-Governance-Authority 权限治理组件**

基于智能合约，提供区块链账户、合约、函数等粒度的权限控制的功能的通用组件。 请参考 文档

- **WeBankBlockchain-Governance-Key 私钥管理组件**

提供私钥生成、存储、加解密、加签、验签等私钥全生命周期管理的通用解决方案。 请参考 文档

- **WeBankBlockchain-Governance-Cert 证书管理组件**

提供证书生成、验证、子证书请求等证书全生命周期管理的通用解决方案。 请参考 文档

CHAPTER 1

账户治理组件

简介

WeBankBlockchain-Governance-Account定位为区块链账户治理组件，旨在充分合理地在区块链节点网络中，利用智能合约所提供的图灵完备的计算能力，提供自洽的账户治理能力

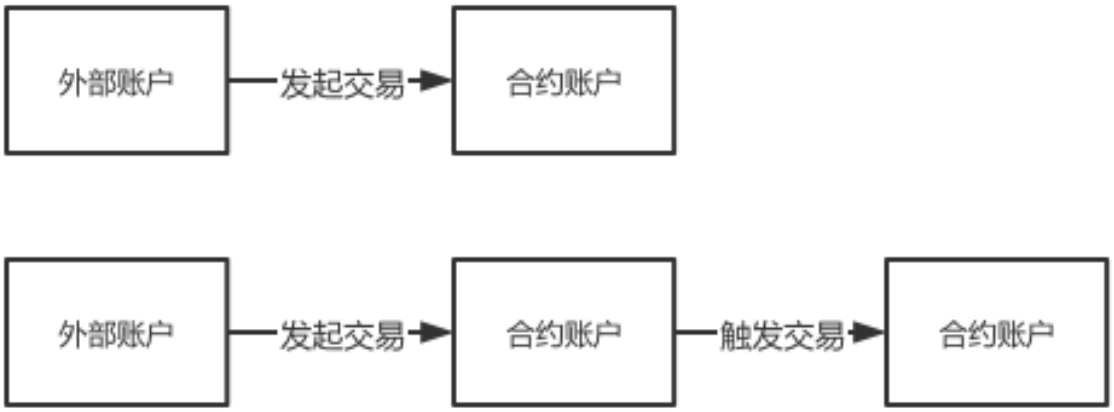
主要特性

- 支持多种治理方式
- 去中心化的分布式协作治理思想
- 账户全生命周期的治理
- 全面、灵活的集成方式
- 支持可插拔的设计，对业务侵入小
- 支持国密
- 支持社交好友重置私钥

1.1 基本概念

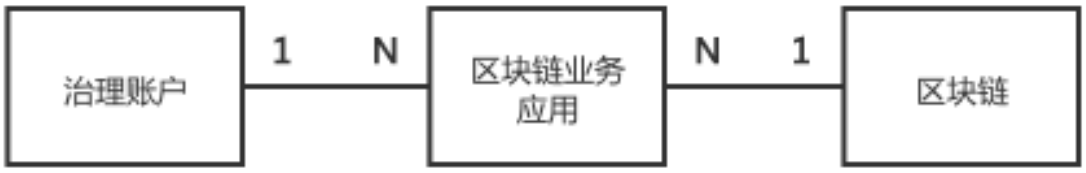
1.1.1 区块链常用术语

- **外部账户(Externally account)**，私钥控制，没有代码关联，可发起交易。外部账户的地址是由私钥所生成的。
- **合约账户(Contract account)**，又称为用户账户(User Account)。合约部署生成，与代码关联，不可发起交易只能被外部账户调用。合约账户的地址是在创建合约是确定的（一般通过计算合约创建者的地址和该地址发出过的交易数量得到）。



1.1.2 账户治理组件所引入的术语

- **区块链业务应用（Blockchain Business Application）**，基于区块链技术，实现和满足特定业务领域的需求。在一个区块链业务应用中，一般会依赖智能合约技术，其执行结果记录在分布式账本中。在一个区块链网络中，可以发布多个区块链业务应用。而在一个区块链业务应用中，最多只能定义一个治理账户。



- **治理账户(Governance Account)**，区块链业务应用治理主体的管理账户。该账户本身为合约账户，其账户关联了合约治理相关的代码，绑定了一个或一组外部账户。只有被绑定的外部账户才能操作此治理账户。此外，只有状态正常的普通账户才能被设置为治理账户。一个治理账户可以管理一个区块链业务应用，也可以管理多个区块链业务应用。
- **普通账户（Normal Account）**，区块链业务应用治理的对象。该账户本身为合约账户，该账户与一个外部账户一一映射并被绑定，关联了状态、配置和操作的代码。只有治理账户和被绑定的外部账户才能操作此治理账户。
- **账户治理者（Account Governance Administer）**，在一个区块链业务应用中，区块链账户治理者负责直接管理和操作治理账户，即所谓的『区块链业务应用管理员』。根据是否是由单一机构管理，又可以划分为带有传统中心化管理特征的超级管理员和去中心化多机构协作的账户治理委员会两类。治理者行使账户的管理职能，支持对普通账户重置私钥、冻结、解冻、销户等操作。
- **治理模式（Governance Mode）**，账户治理者所采用的账户治理的方式，主要有两种，分别为超级管理员模式和治理委员会模式（又被称为投票模式）。其中，投票模式又可以根据投票者权重是否相等，进一步细分为多签投票制（Multi-Vote Mode）和权重投票制（Weighted-Vote Mode）。
- **超级管理员（Super Administer）**，在一个区块链业务应用中，可以指定超级管理员治理模式；在该模式下，治理账户由一个唯一的治理者来控制。
- **账户治理委员会（Account Governance Committee）**，在一个区块链业务应用中，通常是参与区块链业务应用的业务多方共同选出的一个委员会，共同参与账户治理。委员会成员拥有对账户治理事务的投票权，通过投票的方式，来决策和管理相关操作。

- **社交好友重置机制（Social Friends Reset Mechanism）**，是指用户为普通账户设置一组关联的其他普通账户的地址，这些普通账户一般是用户的社交好友们的普通账户。一旦用户需要重置私钥，可通过联系好友发起重置申请，经过多个社交好友同意后，可重置用户普通账户所绑定的私钥。

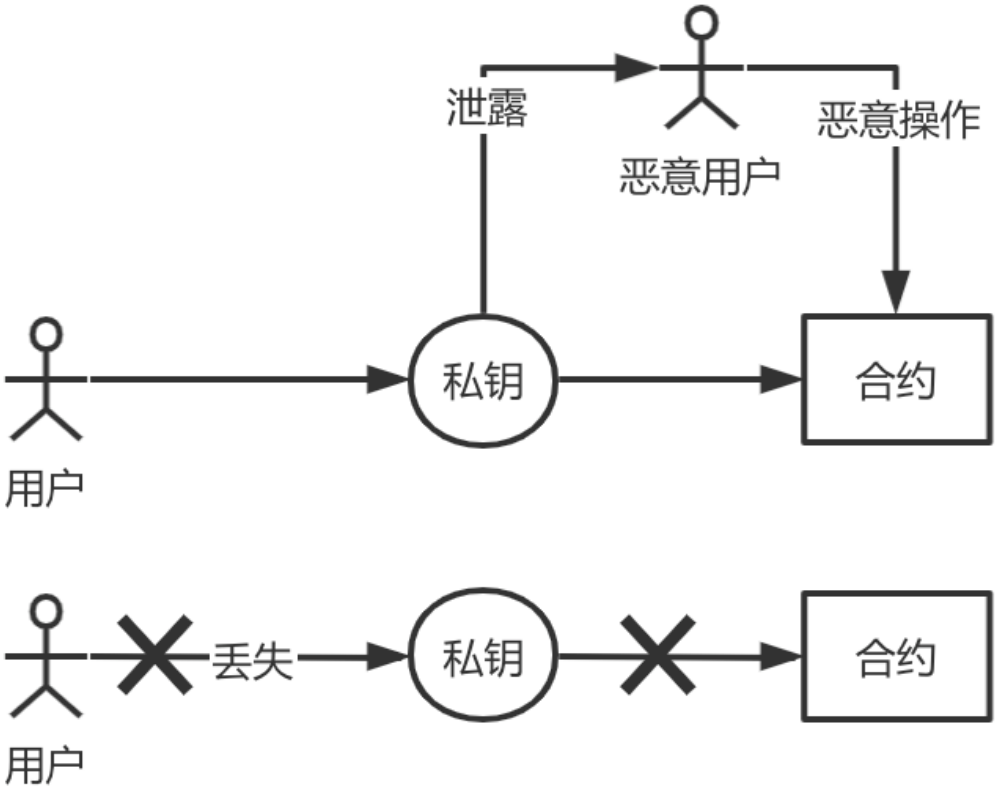
1.2 组件介绍

1.2.1 背景

为什么需要引入账户治理?

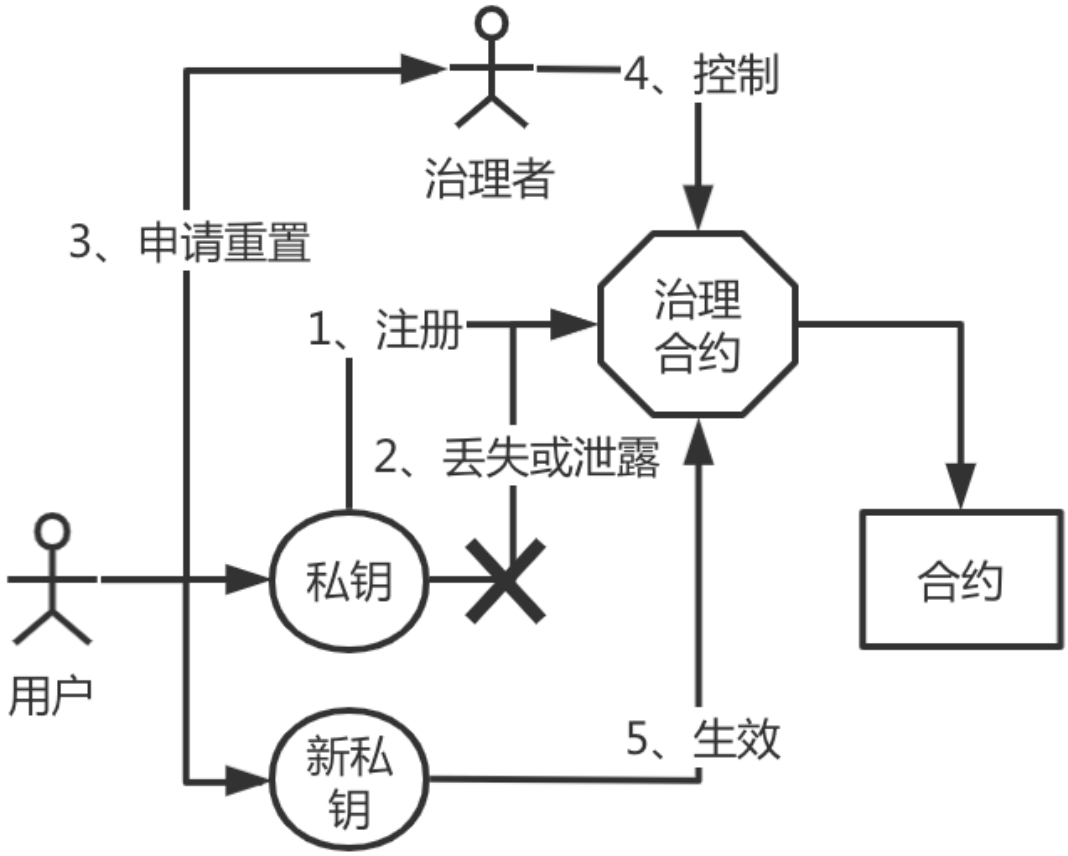
在现有的区块链网络中，是以私钥为控制的中心进行设计的：每个区块链的参与者首先创建一个私钥，通过这个私钥生成公钥并进一步生成外部账户；随后，参与者使用这个私钥对发起的交易报文进行签名并上链。参与者之间也可以通过外部账户的地址来识别和标识身份，完成相关的业务交互。可以说，这个私钥至关重要，一旦泄露或丢失，后果极为严重：

- 泄露以后，盗窃者可以任意控制该外部账户下所有的合约，发送任意的交易指令。
- 丢失以后，原有的用户会失去对该外部账户下所有的合约的控制，无法发送任何的交易指令。



账户治理机制是如何解决上述痛点的?

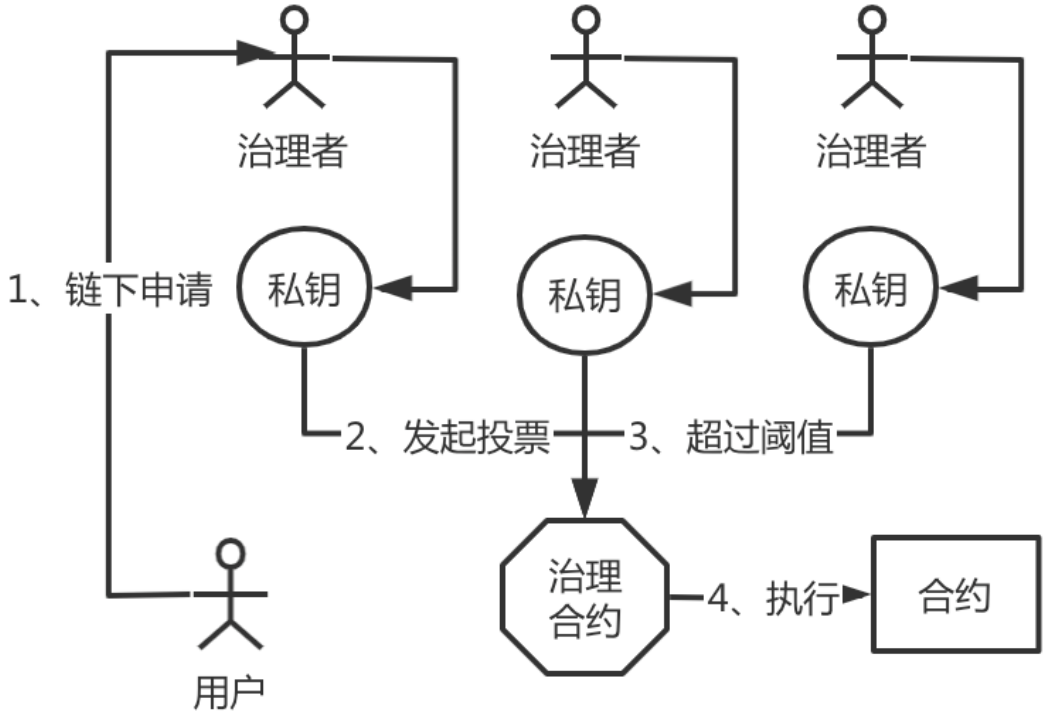
为了解决此问题，需要引入一种账户治理的机制，将原有的账户控制以私钥为中心，改为以普通账户为中心。当用户发现泄露私钥后，可以自助发起重置普通账户所关联映射的私钥。当用户发现丢失私钥后，可以通过向账户治理委员会发起重置申请，更换私钥。而且，当用户更换私钥后，普通账户的地址始终不变，保证了业务平滑运行。



除了解决这个核心痛点外，账户治理体系还额外引入了注册、冻结、解冻、销户等概念，使得在一个分布式协作的业务模式中，账户相关的操作形成了闭环和自洽的治理逻辑。这样，当一个基于区块链的业务系统涉及到链上的账户体系时，可以快速借助账户治理组件的功能，快速、灵活得实现账户体系的治理。

治理者应该如何进行账户治理?

在传统的中心化的解决方案中，存在一个超级管理员的角色，典型的如传统关系型数据库中的root超级管理员，拥有所有的权限。为了兼容这种需求，我们提供了超级管理员的账户治理模式。 但是，传统的中心化解决方案中，存在着权限独大、易被操纵、不利制衡的缺陷，无法满足日益蓬勃的多机构、分布式协作商业模式下的发展。如果链的运营是由一组对等的机构采用分布式协作的方式来管理，则推荐采用治理委员会模式。 治理委员会模式下的治理者通常是参与链的多方共同选出的一个委员会，由多个机构共同进行管理和决策。治理账户维护了一组投票帐号和每个账户对应的投票权重，投票通过的阈值。当某个操作的投票数量超过阈值时，才允许执行该操作。无论是平权还是不同权重，借助这种灵活的投票机制，可以满足大部分复杂场景的治理需求。



1.2.2 概览

为了解决上述痛点，同时践行分布式商业的理念，WeBankBlockchain-Governance-Account应运而生。WeBankBlockchain-Governance-Account是一套开源的区块链账户治理的中间件解决方案，提供了多种区块链账户治理模式、账户生命周期管理、用户自主管理区块链账户治理相关的整体解决方案，提供了包括治理账户创建、多种治理模式选择、治理权限授权，账户创建、冻结、解冻、更换私钥、销户等账户生命周期的各类账户管理功能。

关键特性

WeBankBlockchain-Governance-Account定位为区块链账户治理中间件，旨在充分合理地在区块链节点网络中，利用智能合约所提供的图灵完备的计算能力，提供自洽的账户治理能力。

- **支持多种治理方式**
- **去中心化的分布式协作治理思想**
- **账户全生命周期的治理**
- **全面、灵活的集成方式**
- **支持可插拔的设计，对业务侵入小**
- **支持国密**
- **支持社交好友重置私钥**

组成部分

WeBankBlockchain-Governance-Account包含了以下组成部分：

- **合约代码**，最核心的账户治理实现部分。当前版本提供了基于Solidity语言实现，完全适配FISCO BCOS。 理论上可在任何支持了EVM虚拟机的区块链系统上运行。

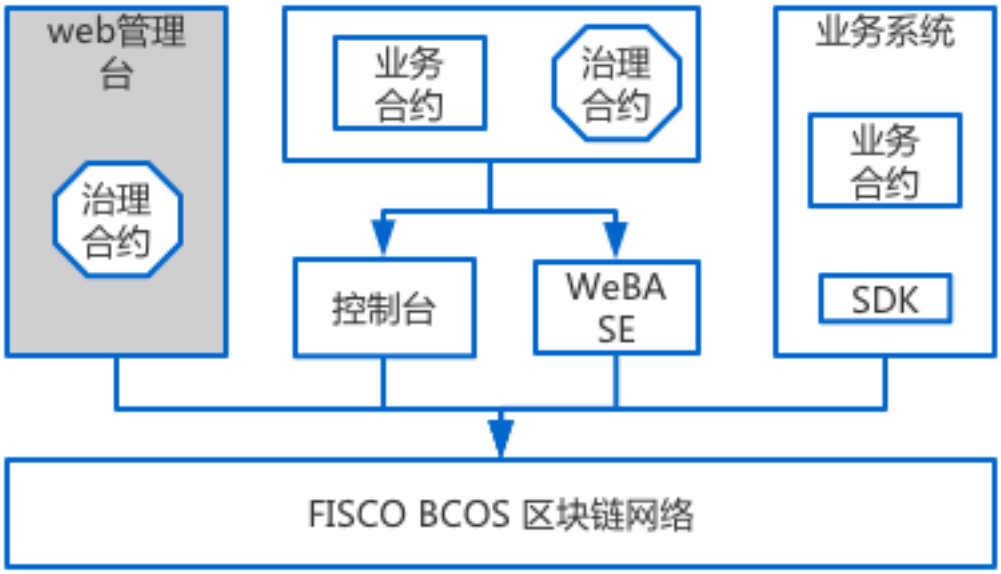
- **Java SDK**。适配了合约代码。通过集成Java SDK以后，可以适配和调用账户治理的所有合约接口。此外，进一步对使用接口进行简化和封装，可符合Java程序员的使用习惯。
- **合约集成Demo**。提供了基于存证和积分转账场景的两个demo。（详情可参考合约中samples目录）
- **SDK集成Demo**。提供了SDK集成和使用的demo，展示了如何使用Java SDK。（详情可参考Governance-Account-Demo）
- **TDD测试代码**。包含了全套的合约测试代码，轻松支持CI/CD。(详情可参考Java SDK中src/test/java目录下的代码)
- **web管理台** 直接通过可视化页面来进行操作，正在开发中.....

使用者基于自身业务的实际场景来自由、灵活地使用和集成。

集成方式

在FISCO BCOS生态中，WeBankBlockchain-Governance-Account提供了以下使用和集成的方式：

- 通过部署治理合约来发布合约到链上，获得账户治理的能力；
- 通过在自身业务的合约中引入或集成账户治理合约；
- 通过SDK引入Jar包，集成到自己的Java项目中来调用提供的接口；

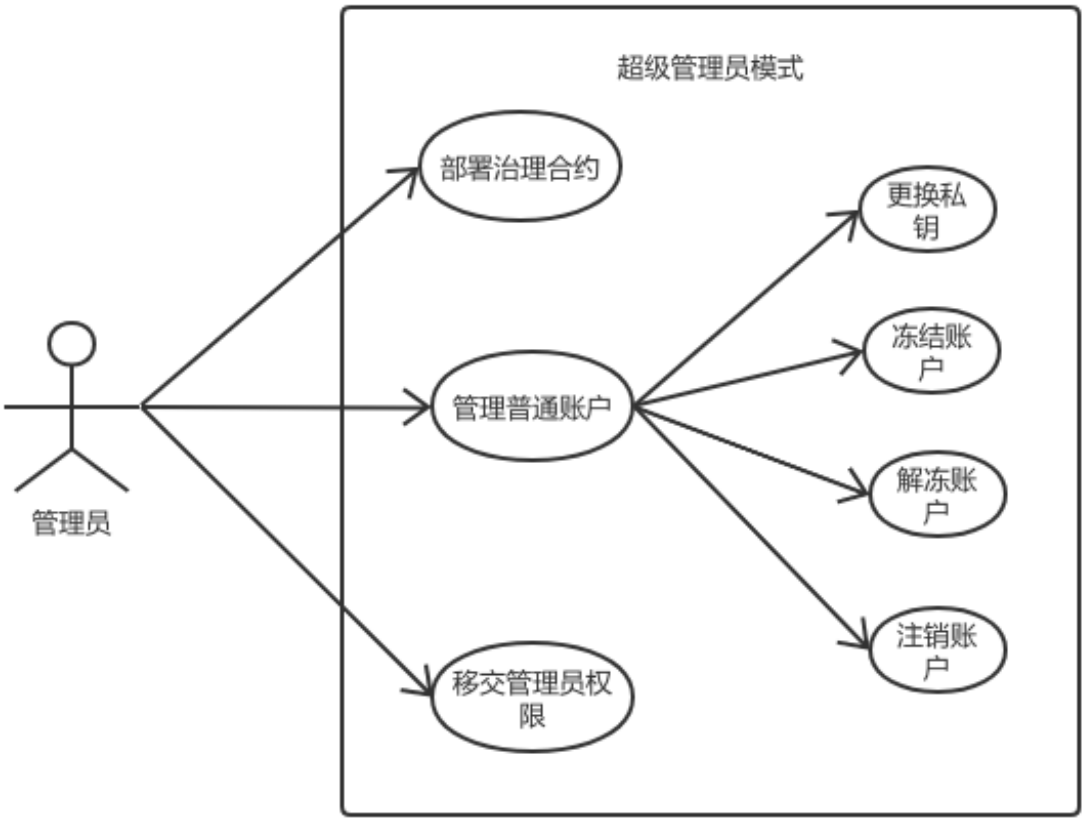


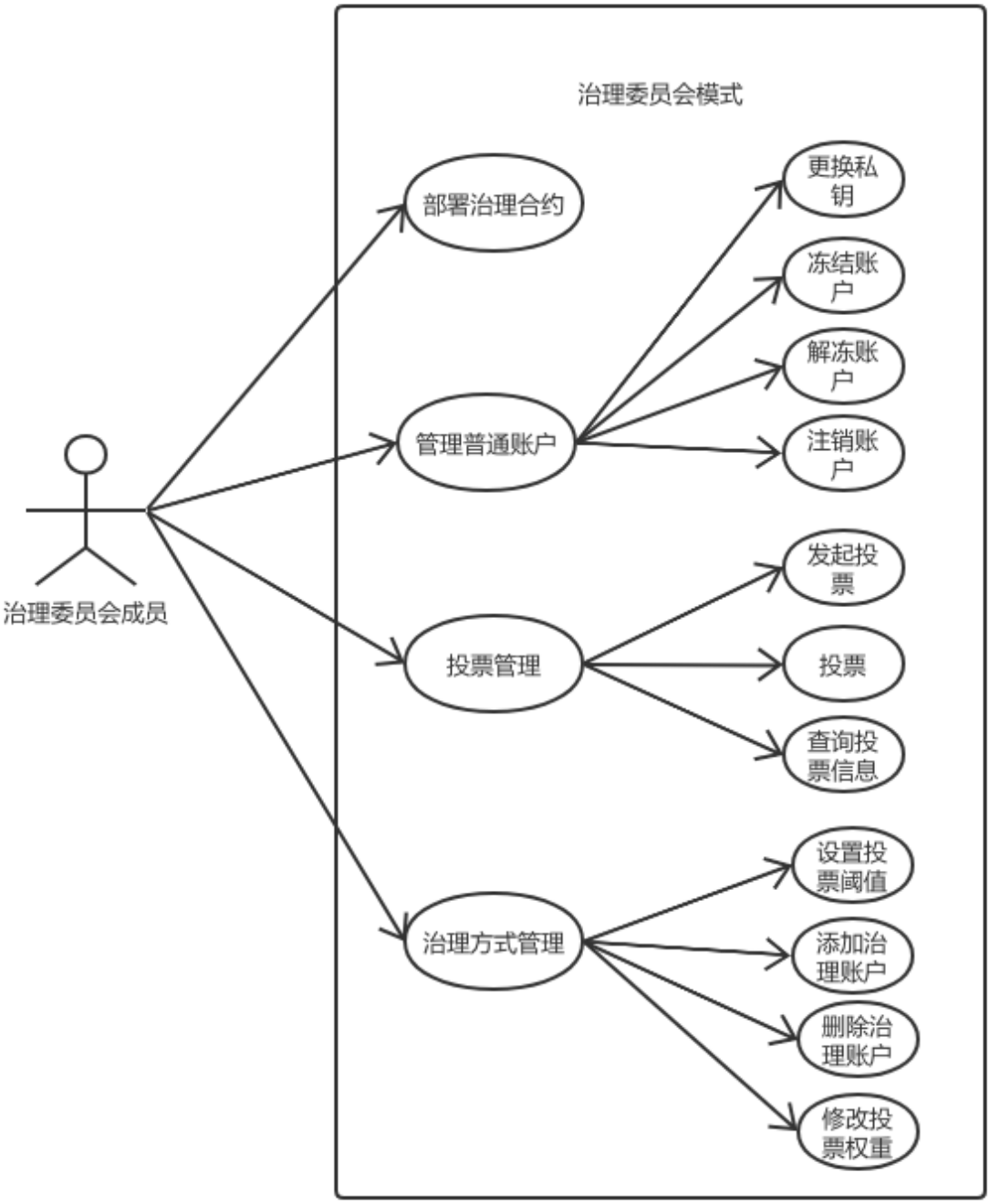
核心功能列表

治理者账户相关核心功能

- 重置用户私钥
- 冻结普通账户
- 解冻普通账户
- 注销普通账户
- 移交管理员权限（超级管理员模式下）
- 添加或修改一个治理委员会的投票账户（治理委员会模式下）

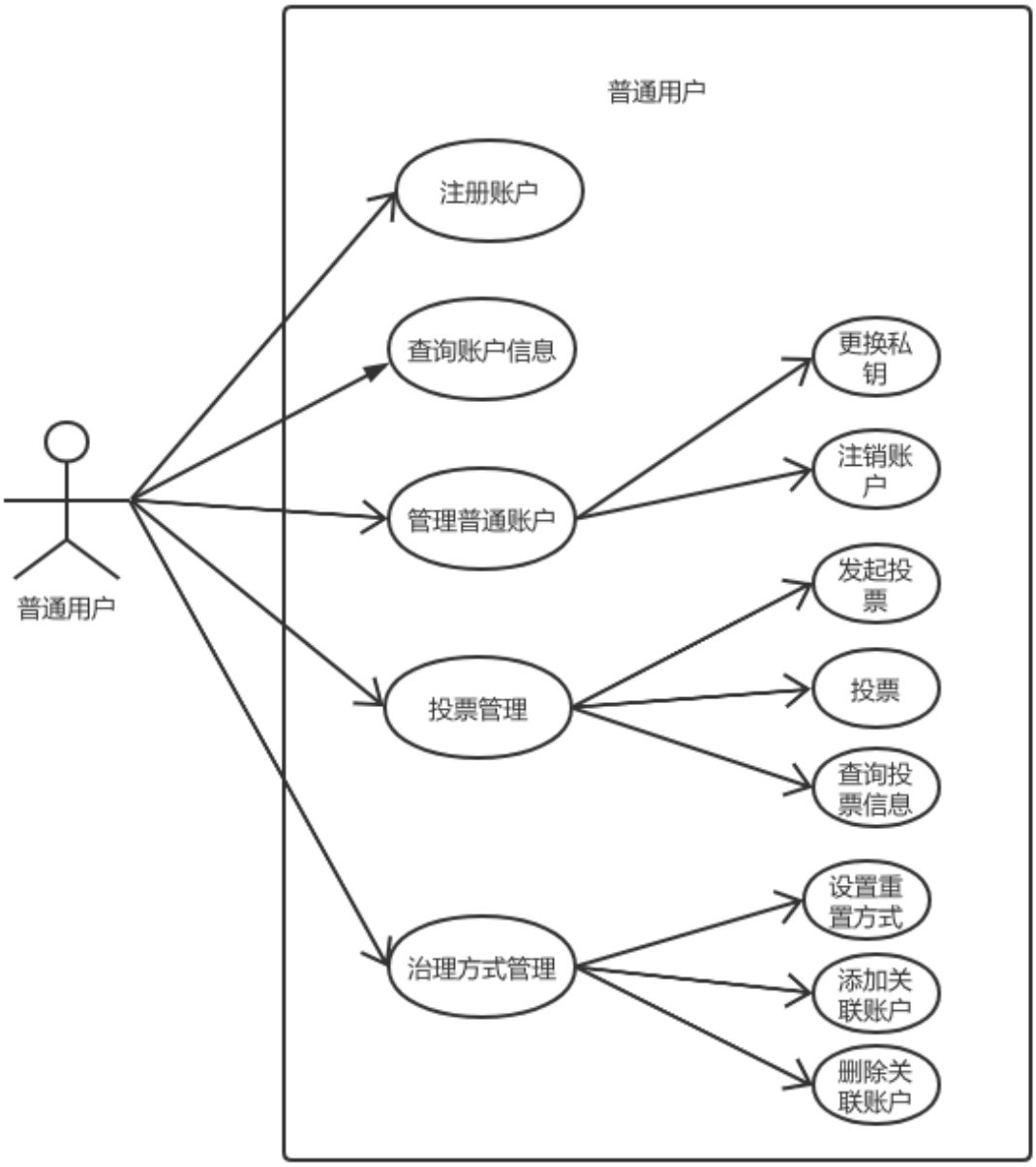
- 删除一个治理委员会的投票账户（治理委员会模式下）
- 设置投票阈值 （治理委员会模式下）





普通账户相关核心功能

- 创建一个新账户
- 重置用户私钥
- 注销账户
- 添加社交好友来支持重置账户私钥
- 添加或删除关联的社交好友
- 查询普通账户状态
- 查询外部账户是否已注册



业务流程介绍

1. 确定治理方式，确定是采用超级管理员模式还是治理委员会模式。所有的治理者需准备和生成机构的私钥。
2. 部署治理合约。根据步骤1确定的治理方式，发布对应的治理合约。
3. 用户普通账户开户。用户预先准备好自己的私钥，通过调用步骤2所部署的治理合约所生成的控制合约来开户。
4. 普通账户地址、状态查询。通过调用步骤2所部署的治理合约所生成的控制合约来查询用户地址和状态。
5. 账户治理常规操作。治理者可发起和操作普通账户。包括了冻结、解冻、销户、重置私钥等普通账户类操作，以及修改投票阈值，修改成员投票权重、修改治理成员等治理账户类的操作。
6. 普通账户自助操作。普通账户可发起重置私钥，设置账户管理模式，添加关联社交好友普通账户等操作。

7. 普通账户销户。普通账户销户完成后，该账户生命周期终结。

1.3 快速开始

本章节以尽量短的时间，为使用者提供最简单的WeBankBlockchain-Governance-Account的快速入门。

1.3.1 前置依赖

注解:

- JDK1.8 或者以上版本，推荐使用OracleJDK。CentOS的yum仓库的OpenJDK缺少JCE(Java Cryptography Extension)，会导致JavaSDK无法正常连接区块链节点。
- 参考 Java环境配置
- FISCO BCOS区块链环境搭建参考 FISCO BCOS安装教程
- 网络连通性。检查所连接的FISCO BCOS节点'channel_listen_port'是否能telnet通，若telnet不通，需要检查网络连通性和安全策略。

1.3.2 获取源码

使用git下载源码:

```
git clone https://github.com/WeBankBlockchain/Governance-Account.git
```

注解:

- 如果因为网络问题导致长时间无法下载，请尝试：git clone https://gitee.com/WeBankBlockchain/Governance-Account.git

- 所有智能合约文件位于src/main/contracts路径下
- 合约demo位于src/main/contracts/samples路径下
- 合约测试代码位于src/test/java路径下
- 其余部分代码为Java SDK代码。

1.3.3 使用组件合约

本章节介绍的是只使用合约本身的方式进行账户治理。如果需要使用SDK来治理的，可跳过本章节。 在完成FISCO BCOS链环境初始化以后，通过将WeBankBlockchain-Governance-Account作为独立的插件，可使用控制台来部署账户治理的合约。可参考控制台的配置与运行。

使用控制台部署和调用合约

本方式可直接通过控制台或WeBASE-Front来操作。 在此我们以控制台为例，进行演示，关于控制台的使用说明请参考控制台命令列表。

准备和登入控制台环境

在进入以下的操作前，首先我们进入控制台，准备至少四组操作私钥。

```
# ./get_account.sh
[INFO] Account Address   : 0x1c7560296c101171eb8015cdc6cfbda26c866189
[INFO] Private Key (pem) : accounts/0x1c7560296c101171eb8015cdc6cfbda26c866189.pem
[INFO] Public  Key (pem) : accounts/0x1c7560296c101171eb8015cdc6cfbda26c866189.
↪public.pem
```

执行以上操作，会在控制台的accounts目录下生成对应的pem私钥文件。连续执行四次后，查看accounts目录，会发现已得到了四组密钥文件。

```
ll accounts
total 32
-rw-r--r-- 1 root root 237 Jul  5 11:46 0x14b0b2e52a156fcd310acee0692501ca23bb8a3e.
↪pem
-rw-r--r-- 1 root root 174 Jul  5 11:46 0x14b0b2e52a156fcd310acee0692501ca23bb8a3e.
↪public.pem
-rw-r--r-- 1 root root 237 Jul  5 11:47 0x1c7560296c101171eb8015cdc6cfbda26c866189.
↪pem
-rw-r--r-- 1 root root 174 Jul  5 11:47 0x1c7560296c101171eb8015cdc6cfbda26c866189.
↪public.pem
-rw-r--r-- 1 root root 237 Jul  5 11:56 0x5b15b41277f4cacfdad39ba06a5dcc1295af0fd8.
↪pem
-rw-r--r-- 1 root root 174 Jul  5 11:56 0x5b15b41277f4cacfdad39ba06a5dcc1295af0fd8.
↪public.pem
-rw-r--r-- 1 root root 237 Jul  5 11:46 0x7abb8086fcffd0417d9a4c9b615426aca5e4182d.
↪pem
-rw-r--r-- 1 root root 174 Jul  5 11:46 0x7abb8086fcffd0417d9a4c9b615426aca5e4182d.
↪public.pem
```

由于每次生成的密钥都是随机的，故而密钥的地址都是不同的。接下来，我们使用最上面的私钥作为操作地址，登入控制台：

```
./start.sh 1 -pem accounts/0x14b0b2e52a156fcd310acee0692501ca23bb8a3e.pem  #1为群
组ID
================================================================================================
Welcome to FISCO BCOS console(1.0.10)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ _______  _______   _______  _______       ________   _______   _______   _______
|        |       \/       \ /       \ /       \     |        \ /       \ /       \ /       ␣
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
| $$__      | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \     | $$  \$$    \| $$      | $$  | $$    | $$    $| $$      | $$  | $$\$$   ␣
↪\
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$\
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__|
↪$$
| $$      |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$
↪$


================================================================================================
[group:1]>
```

部署治理合约

为了便于部署WeBankBlockchain-Governance-Account治理合约，我们共提供了三个Builder合约，来便于快速部署，分别是：

1. AdminGovernBuilder
2. VoteGovernBuilder
3. WeightVoteGovernBuilder

超级管理员模式

```
deploy AdminGovernBuilder
```

在控制台中，执行上述的命令，然后可以调用_governance函数来获得治理账户合约的地址，然后调用WEGovernnance合约的getAccountManager函数来获得账户管理合约的地址和账户管理控制器的地址。

```
# 部署超级管理员模式的治理账户
[group:1]> deploy AdminGovernBuilder
contract address: 0x6b10051756bf259efe7b4c22fad3925700ab2e1e
# 获取治理合约的地址
[group:1]> call AdminGovernBuilder 0x6b10051756bf259efe7b4c22fad3925700ab2e1e␣
↪getGovernance
0x9d8ff088555122e7bcb0827130a9ac64780dff25
# 获取账户管理控制器的地址
[group:1]> call AdminGovernBuilder 0x6b10051756bf259efe7b4c22fad3925700ab2e1e␣
↪getAccountManager
0x26ceac6bb9e727ed69fa391ae2eba9a3f5c0c8d0
```

治理委员会模式

需要输入部署的治理账户的外部账户地址列表和阈值。

```
deploy VoteGovernBuilder(externalAccounts, threshold)
```

我们可以配置每个投票账户拥有相同的投票权重，这种投票方式也被称为多签制。例如，我们部署三个地址，分别为三个不同的地址，三个账户的投票权限相同，阈值为2：

```
# 部署多签制模式的治理账户
[group:1]> deploy VoteGovernBuilder ["0x14b0b2e52a156fcd310acee0692501ca23bb8a3e",
↪"0x1c7560296c101171eb8015cdc6cfbda26c866189",
↪"0x5b15b41277f4cacfdad39ba06a5dcc1295af0fd8"] 2
contract address: 0xe96b9fcb5c9d1e0d9a4baa2ad304578363c05bd6
# 获取治理合约的地址
[group:1]> call VoteGovernBuilder 0xe96b9fcb5c9d1e0d9a4baa2ad304578363c05bd6␣
↪getGovernance
0x003a512af46eb456b1c57a70b95caf104db1df1b
# 获取账户管理控制器的地址
[group:1]> call VoteGovernBuilder 0xe96b9fcb5c9d1e0d9a4baa2ad304578363c05bd6␣
↪getAccountManager
0x50c903e4580682060984b971270f7c7864a6cc81
```

当然我们也可以配置每个投票的账户拥有不同的权重，这种投票方式也被称为权重制。例如，我们部署三个地址，分别为三个不同的地址，对应的权重为1，2，3， 阈值为2：

```
# 部署多签制模式的治理账户
[group:1]> deploy WeightVoteGovernBuilder [
↪"0x14b0b2e52a156fcd310acee0692501ca23bb8a3e",
↪"0x1c7560296c101171eb8015cdc6cfbda26c866189",
↪"0x5b15b41277f4cacfdad39ba06a5dcc1295af0fd8"] [1,2,3] 2
```

(continues on next page)

(续上页)

```
contract address: 0xbf8b5357a01232ab2e3ac8922fbe9a425afba026
# 获取治理合约的地址
[group:1]> call WeightVoteGovernBuilder 0xbf8b5357a01232ab2e3ac8922fbe9a425afba026␣
↪getGovernance
0x9ba26b43e00fb112da14b509715b868bb8fb1a2d
# 获取账户管理控制器的地址
[group:1]> call WeightVoteGovernBuilder 0xbf8b5357a01232ab2e3ac8922fbe9a425afba026␣
↪getAccountManager
0x2c02a11a0fc9eaf1ded01e844b35456aeb352261
```

以上，我们分别展示了如何在链上部署超级管理员模式和治理委员会模式（包括多签制和权重制）的治理合约的部署。 在一个链上，可以基于不同的业务应用或场景来采用不同的管理模式。一般在一种独立的业务或场景中，部署一个治理合约。例如，在链的监管场景中，推荐使用超级管理员模式。在区块链分布式商业中，推荐使用多签制。在类似董事会的治理场景中，推荐采用可设置投票人不同权重的权重投票制。

调用治理合约

在部署了上述合约的前提下，可根据上一章节获得WEGovernance和AccountManager合约的地址。

账户相关的常见操作

上述所有的Manager都继承了BasicManager基础Manager管理器，支持以下操作:

- newAccount 创建普通账户
- getExternalAccount 查询普通账户地址
- hasAccount 查询普通账户是否已开户
- isExternalAccountNormal 查询外部账户状态是否正常

相关的使用实例如下。 创建一个普通账户

```
[group:1]>  call AccountManager 0xf0da09d98fe320371fe30d1bd035f0cabc305eaf␣
↪newAccount "0x1c7560296c101171eb8015cdc6cfbda26c866189"
transaction hash:␣
↪0xeee7badb2ecdddd080aca04cda58498f27709efd0d1773fa770ed1c91f587119
-------------------------------------------------------------------------------------␣
↪----------
Output
function: newAccount(address)
return type: (bool, address)
return value: (true, 0x39be54ce16ccb720aa0ffedea27a7e3621565598)
-------------------------------------------------------------------------------------␣
↪----------
Event logs
event signature: LogSetOwner(address,address) index: 0
event value: (0x1c7560296c101171eb8015cdc6cfbda26c866189,␣
↪0x39be54ce16ccb720aa0ffedea27a7e3621565598)
event signature: LogManageNewAccount(address,address,address) index: 0
event value: (0x1c7560296c101171eb8015cdc6cfbda26c866189,␣
↪0x39be54ce16ccb720aa0ffedea27a7e3621565598,␣
↪0xf0da09d98fe320371fe30d1bd035f0cabc305eaf)
-------------------------------------------------------------------------------------␣
↪----------
```

查询普通账户地址

```
[group:1]> call AccountManager 0xf0da09d98fe320371fe30d1bd035f0cabc305eaf␣
↪getAccount "0x1c7560296c101171eb8015cdc6cfbda26c866189"
0x39be54ce16ccb720aa0ffedea27a7e3621565598
```

查询普通账户是否已开户

```
[group:1]> call AccountManager 0xf0da09d98fe320371fe30d1bd035f0cabc305eaf␣
↪hasAccount "0x1c7560296c101171eb8015cdc6cfbda26c866189"
true
```

查询外部账户状态是否正常

```
[group:1]> call AccountManager 0xf0da09d98fe320371fe30d1bd035f0cabc305eaf␣
↪isExternalAccountNormal "0x1c7560296c101171eb8015cdc6cfbda26c866189"
true
```

治理者的相关操作

治理者可分为超级管理员和治理委员会两种模式，其对应的治理操作也有所不同。 **在超级管理员模式下，包含了以下操作**：获得Governance合约以后，可进行以下操作

- transferOwner 移交超级管理员账户
- setExternalAccount 重置用户私钥
- doOper 账户相关操作， 3-冻结账户 4-解冻账户 5-cancelAccount 注销账户

移交超级管理员账户 在控制台中进行操作。其中，被移交的账户已在上一章节中开户。

```
[group:1]> call WEGovernance 0xbba738f0549cde5e5b0bab2dffc125501556018c␣
↪transferOwner "0x1c7560296c101171eb8015cdc6cfbda26c866189"
transaction hash:␣
↪0x849d1fadfd95be911b6c4cd3be52a3799b51aa9f8d71086039e7ce1b48513fb2
Event logs
event signature: LogSetOwner(address,address) index: 0
event value: (0x1c7560296c101171eb8015cdc6cfbda26c866189,␣
↪0xbba738f0549cde5e5b0bab2dffc125501556018c)
----------------------------------------------------------------------------------
↪----------
```

切换登录管理台的外部账户 可以看到该治理合约的权限已被移交到新的『0x1c7560296c101171eb8015cdc6cfbda26c866189』地址中。故我们需要退出并切换新的私钥来登录控制台。

```
[group:1]> exit
[root@instance-zw7wgjv0 console]# ./start.sh 1 -pem accounts/
↪0x1c7560296c101171eb8015cdc6cfbda26c866189.pem
================================================================================================
Welcome to FISCO BCOS console(1.0.10)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ _______  _______  _______  _______       ________  _______  _______  _______
|        |       \/       \ /       \ /       \     |        \ /       \ /       \ /       ␣
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
| $$__      | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \     | $$  \$$    \| $$      | $$  | $$    | $$    $| $$      | $$  | $$\$$    ␣
↪\
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$\
```

(continues on next page)

(续上页)

```
| $$      _| $$_|   \__| $| $$__/   | $$__/ $$     | $$__/ $| $$__/   | $$__/ $|   \__|
↪$$
| $$     |   $$ \\$$     $$\$$     $$\$$     $$     | $$     $$\$$     $$\$$     $$\$$
↪$$
 \$$      \$$$$$$ \$$$$$$   \$$$$$$   \$$$$$$       \$$$$$$$   \$$$$$$   \$$$$$$   \$$$$$
↪$

================================================================================================
[group:1]>
```

重置用户私钥

```
[group:1]> call WEGovernance 0xbba738f0549cde5e5b0bab2dffc125501556018c␣
↪setExternalAccount 2 "0x7abb8086fcffd0417d9a4c9b615426aca5e4182d"
↪"0x14b0b2e52a156fcd310acee0692501ca23bb8a3e"
transaction hash:␣
↪0x8201112d9bce00d751e9f6fb13ef3182f4c0afafa750334a34820a5f5101fcc7
------------------------------------------------------------------------------------------
↪----------
Output
function: setExternalAccount(uint256,address,address)
return type: (bool)
return value: (true)
------------------------------------------------------------------------------------------
↪----------
Event logs
------------------------------------------------------------------------------------------
↪----------
```

冻结、解冻与销户

```
## 冻结
[group:1]> call WEGovernance 0xbba738f0549cde5e5b0bab2dffc125501556018c doOper 3
↪"0x7abb8086fcffd0417d9a4c9b615426aca5e4182d" 3
transaction hash:␣
↪0x645ccdb7569d53387ab28ede5b3b066971a87ff8ab45ab9ef663b5f4bfde3204
------------------------------------------------------------------------------------------
↪----------
Output
function: doOper(uint256,address,uint8)
return type: (bool)
return value: (true)
------------------------------------------------------------------------------------------
↪----------
Event logs
------------------------------------------------------------------------------------------
↪----------

## 解冻
[group:1]>  call WEGovernance 0xbba738f0549cde5e5b0bab2dffc125501556018c doOper 4
↪"0x7abb8086fcffd0417d9a4c9b615426aca5e4182d" 4
transaction hash:␣
↪0xe5ae51e446836abda81294175d508e0f938f12c53517673aabfa496d4f352b3c
------------------------------------------------------------------------------------------
↪----------
Output
function: doOper(uint256,address,uint8)
return type: (bool)
return value: (true)
------------------------------------------------------------------------------------------
↪----------
```

(continues on next page)

(续上页)

```
Event logs
--------------------------------------------------------------------------------
↪----------

## 注销
[group:1]> call WEGovernance 0xbba738f0549cde5e5b0bab2dffc125501556018c doOper 5
↪"0x7abb8086fcffd0417d9a4c9b615426aca5e4182d" 5
transaction hash:␣
↪0xd13bfc5cd76fb38bea1d05de7b4bec8463ce86a69f4c6b2a3c66ed8bcce582a4
--------------------------------------------------------------------------------
↪----------
Output
function: doOper(uint256,address,uint8)
return type: (bool)
return value: (true)
--------------------------------------------------------------------------------
↪----------
Event logs
--------------------------------------------------------------------------------
↪----------
```

在治理委员会模式下，包含了以下操作：

- register 申请发起治理操作，包括了： 10-申请重设治理委员会投票阈值， 11-申请重设治理委员会投票权重，2-申请重置账户，3-申请冻结账户，4-申请解冻账户， 5-申请注销账户
- vote 对申请项投票
- setExternalAccount 重置用户私钥
- doOper 投票通过后，发起治理操作，包括了冻结账户、解冻账户、注销账户等。
- setThreshold 设置投票的阈值。
- setWeight 设置投票者的权重。
- getRequestInfo 查看投票详情信息。

详细的操作方式可参考治理合约的相关代码接口。 由于使用控制台操作治理委员会相关操作较为繁琐，我们推荐采用Java SDK的方式来进行。后期，我们会提供通过Web管理台的方式，敬请期待。

普通用户的相关操作

普通用户可通过accountManager来重置和注销账户。

- setExternalAccountByUser 重置用户私钥
- cancelByUser 注销账户

如下所示 使用外部账户地址为『0x5b15b41277f4cacfdad39ba06a5dcc1295af0fd8』来注册账户。

```
# 注册账户
[group:1]> call AccountManager 0xf0da09d98fe320371fe30d1bd035f0cabc305eaf␣
↪newAccount "0x5b15b41277f4cacfdad39ba06a5dcc1295af0fd8"
transaction hash:␣
↪0x0d3ff289ca601906f1fa286561755efb405a832c6295061d3d7cdce447884abb
--------------------------------------------------------------------------------
↪----------
Output
function: newAccount(address)
return type: (bool, address)
return value: (true, 0x5196261524954a59c5f23f05883eef59d6e5fd38)
--------------------------------------------------------------------------------
↪----------
```

(continues on next page)

(续上页)

```
Event logs
event signature: LogSetOwner(address,address) index: 0
event value: (0x5b15b41277f4cacfdad39ba06a5dcc1295af0fd8,␣
↪0x5196261524954a59c5f23f05883eef59d6e5fd38)
event signature: LogManageNewAccount(address,address,address) index: 0
event value: (0x5b15b41277f4cacfdad39ba06a5dcc1295af0fd8,␣
↪0x5196261524954a59c5f23f05883eef59d6e5fd38,␣
↪0xf0da09d98fe320371fe30d1bd035f0cabc305eaf)
----------------------------------------------------------------------------------------
↪----------
```

退出并使用该外部账户地址对应的私钥来载入控制台。

```
[group:1]> exit
[root@instance-zw7wgjv0 console]# ./start.sh 1 -pem accounts/
↪0x5b15b41277f4cacfdad39ba06a5dcc1295af0fd8.pem
============================================================================================
Welcome to FISCO BCOS console(1.0.10)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ _______  _______   _______  _______       ________   _______   _______   _______
|        |       \/       \ /       \ /       \     |        \ /       \ /       \ /       ␣
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
| $$__      | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \     | $$  \$$    \| $$      | $$  | $$    | $$    $| $$      | $$  | $$\$$    ␣
↪\
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$\
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__|
↪$$
| $$      |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$
↪$

============================================================================================
```

销户

```
[group:1]> call AccountManager 0xf0da09d98fe320371fe30d1bd035f0cabc305eaf␣
↪cancelByUser
transaction hash:␣
↪0xbced5c2aaf2cbcb0c98ec9877246253f3f594816d456f29a581062d97901e891
----------------------------------------------------------------------------------------
↪----------
Output
function: cancelByUser()
return type: (bool)
return value: (true)
----------------------------------------------------------------------------------------
↪----------
Event logs
event signature: LogManageCancel(address,address) index: 0
event value: (0x5b15b41277f4cacfdad39ba06a5dcc1295af0fd8,␣
↪0xf0da09d98fe320371fe30d1bd035f0cabc305eaf)
----------------------------------------------------------------------------------------
↪----------
```

重新开户并设置外部账户地址为『0x1』

```
[group:1]> call AccountManager 0xf0da09d98fe320371fe30d1bd035f0cabc305eaf␣
↪setExternalAccountByUser "0x1"
transaction hash:␣
↪0x0ff2e49f11767826a311562fec36f81decfc9d67242e8d399473c5fbde53db9d
-------------------------------------------------------------------------------------------
↪----------
Output
function: setExternalAccountByUser(address)
return type: (bool)
return value: (true)
-------------------------------------------------------------------------------------------
↪----------
Event logs
event signature: LogManageExternalAccount(address,address,address) index: 0
event value: (0x0000000000000000000000000000000000000001,␣
↪0x5b15b41277f4cacfdad39ba06a5dcc1295af0fd8,␣
↪0xf0da09d98fe320371fe30d1bd035f0cabc305eaf)
-------------------------------------------------------------------------------------------
↪----------
```

此外，支持用户自主设置账户治理模式，支持以下操作。

- setStatics 修改重置类型，0-不支持社交好友重置私钥，1-支持社交好友重置私钥
- setWeight 设置关联社交好友普通账户的权重
- register 发起投票
- vote 投票
- setExternalAccountBySocial 重置私钥

更多操作，请查看智能合约代码。由于直接操作合约的方式较为繁琐，此处不再进行详细地演示。**我们推荐使用Java SDK集成的方式来进行相关的操作。** 后续，我们将提供web管理台来便于用户使用，敬请期待。

将合约引入到用户自身的业务合约中

组件合约引入

用户在具体的业务合约中，可采用引入的方式来使用WeBankBlockchain-Governance-Account智能合约。

在通用的场景中引入治理合约

```
// 在业务合约中声明import账户管理合约。
import "./AccountManager.sol"

XXContract {
  // 创建AccountManager合约
  AccountManager _accountManager;

  construct(address accountManager) {
            // 传入已部署的AccountManager地址。此地址由上面的部署治理合约步骤获得。
        _accountManager = AccountManager(accountManager);
  }

    // ......
  function doSomeBiz() public {
        // 获得普通用户的地址
        address userAccountAddress = _accountManager.getAccount(msg.sender);
```

(continues on next page)

(续上页)

```
                // **特别注意**: 在具体的业务中，需使用此普通用户的地址来代替以往的外部地址。
        doBiz(userAccountAddress);
    }

}
```

在涉及转账的场景中引入治理合约

在涉及到转账的场景中，引入合约的方式更为典型，本例来源于我们提供的使用demo，限于篇幅，此处略去了和示例非密切相关的部分，获取完整的demo代码可查找src/main/contracts/samples/TransferDemo/路径。

```
import "./AccountManager.sol";
contract TransferDemo {
        // ......
    mapping(address => uint256) private _balances;
    // import AccountManager
    AccountManager _accountManager;

    constructor(address accountManager, uint256 initBalance) public {
        // 初始化accountManager，此地址由上面的部署治理合约步骤获得。
        _accountManager = AccountManager(accountManager);
                // 合约的owner设置为用户普通账户的地址
        address owner = _accountManager.getAccount(msg.sender);
        _balances[owner] = initBalance;
    }

    modifier validateAccount(address addr) {
        require(
            // 调用accountManager来判断该外部账户的地址是否已被注册及账户状态是否正常等。
            _accountManager.isExternalAccountNormal(addr),
            "Account is abnormal!"
        );
        _;
    }
   // ......
    function balance(address owner) public view returns (uint256) {
        // 1.先根据外部账户的地址查询普通账户的地址，然后仔查询余额
        return _balances[_accountManager.getAccount(owner)];
    }
    function transfer(address toAddress, uint256 value)
        public
        // 检查转入方和转出方的外部账户地址是否合法。
        validateAccount(msg.sender)
        validateAccount(toAddress)
        checkTargetAccount(toAddress)
        returns (bool)
    {
        // 1. 查询转出方的普通账户地址
        address fromAccount = _accountManager.getAccount(msg.sender);
        // 2. 查询转出方普通账户余额
        uint256 balanceOfFrom = _balances[fromAccount].sub(value);
        // 3. 转出方金额扣除
        _balances[fromAccount] = balanceOfFrom;
        // 4. 查询转入方的普通账户地址
        address toAccount = _accountManager.getAccount(toAddress);
        // 5. 转入方余额增加
        uint256 balanceOfTo = _balances[toAccount].add(value);
        // 设置转入方普通账户的余额
```

(continues on next page)

(续上页)

```
        _balances[toAccount] = balanceOfTo;
        return true;
    }
}
```

接下来，用户可到FISCO BCOS控制台中，将XXContract的业务合约编译为具体的Java代码。控制台提供一个专门的编译合约工具，方便开发者将Solidity合约文件编译为Java合约文件，具体使用方式参考合约编译工具 也可以通过控制台或WeBASE-Front等工具部署到链上，在此不再赘述。 更加完整地引入治理合约的例子和使用方式可以参考工程中附带的samples demo。我们提供了基于存证和积分转账场景的两个demo。

1.4 Java语言版本的SDK使用说明

为了便于使用，我们提供了Java版本的SDK。相关的使用说明可参考下文。

同时，提供了一个基于本SDK的Governance-Account-Demo。Demo中包含了相关的合约代码和SDK的代码，供参考。

1.4.1 引入工程

将Jar包引入到用户自己的Java业务项目中。

在自己的Java项目中的build.gradle文件中，添加maven仓库

```
    allprojects {
                repositories {
                        ...
                        maven { url 'https://jitpack.io' }
                }
        }
```

引入依赖：

```
    dependencies {
            implementation 'com.github.WeBankBlockchain:Governance-Account:Tag'
        }
```

1.4.2 治理账户功能使用说明

创建治理合约

管理员模式

假如平台方采用管理员的治理模式，那么需要首先生成一个管理员的治理账户。 **具体调用示例：**

```
    // 自动注入AdminModeGovernAccountInitializer对象
    @Autowired
    private GovernAccountInitializer governAccountInitializer;
    // 调用 createGovernAccount 方法生成管理员账户
    WEGovernance govern = adminModeManager.createGovernAccount(u);
```

函数签名：

```
WEGovernance createGovernAccount(Credentials credential)
```

输入参数：

- credential 管理员的私钥

返回参数：

- WEGovernance 返回的治理账户对象

调用成功后，函数会返回对应的WEGovernance治理账户对象，通过getContractAddress()方法可以获得对应的治理合约的地址。

多签制治理模式

例如，以下平台方选择了治理委员会的治理模式，一共有三个参与者参与治理，治理的规则为任意的交易请求获得其中两方的同意，即可获得通过。那么我们接下来将创建一个治理账户。 **具体调用示例：**

```
    // 自动注入GovernAccountInitializer对象
    @Autowired
    private GovernAccountInitializer governAccountInitializer;
    // 准备3个治理账户管理成员的公钥地址，生成治理账户
    List<String> list = new ArrayList<>();
    list.add(address1);
    list.add(address2);
    list.add(address3);
    // 创建账户
    WEGovernance govern = adminModeManager.createGovernAccount(list, 2);
```

函数签名：

```
WEGovernance createGovernAccount(List<Credentials> credentials, int threshold)
```

输入参数：

- externalAccountList 治理委员会成员的外部账户信息。
- threshold 投票阈值，如超过该投票阈值，表示投票通过。

返回参数：

- WEGovernance 返回的治理账户对象

调用成功后，函数会返回对应的WEGovernance治理账户对象，通过getContractAddress()方法可以获得对应的治理合约的地址。

权重投票治理模式

本模式类似于上一种多签制，区别在于每个投票者的投票权重可以是不相同的。 例如，以下平台方选择了治理委员会的权重投票的治理模式，一共有三个参与者参与治理，投票的权重分别为1、2、3，阈值为4，也就是说任意的赞同选票权重相加超过阈值即可获得通过。那么我们接下来将创建一个治理账户。 **具体调用示例：**

```
    // 自动注入GovernAccountInitializer对象
    @Autowired
    private GovernAccountInitializer governAccountInitializer;
    // 准备3个治理账户管理成员的公钥地址，生成治理账户
    List<String> list = new ArrayList<>();
    list.add(address1);
    list.add(address2);
    list.add(address3);
    // 设置3个投票账户的权重分别为1、2、3
    List<BigInteger> weights = new ArrayList<>();
    weights.add(BigInteger.valueOf(1));
```

(continues on next page)

(续上页)

```
    weights.add(BigInteger.valueOf(2));
    weights.add(BigInteger.valueOf(3));
    // 创建账户
    WEGovernance govern = adminModeManager.createGovernAccount(list, weights, 4);
```

函数签名：

```
WEGovernance createGovernAccount(List<String> credentialsList, List<BigInteger>␣
↪weights, int threshold)
```

输入参数：

- externalAccountList 治理委员会成员的外部账户信息。
- weights 治理委员会成员对应的投票权重。
- threshold 投票阈值，如超过该投票阈值，表示投票通过。

返回参数：

- WEGovernance 返回的治理账户对象

调用成功后，函数会返回对应的WEGovernance治理账户对象，通过getContractAddress()方法可以获得对应的治理合约的地址。

调用控制接口

管理员模式

管理员模式下的管理功能均位于GovernAccountInitializer类中。 首先，注入该类：

```
@Autowired
private AdminModeGovernManager adminModeManager;
```

重置用户私钥

具体调用示例：

```
    TransactionReceipt tr = adminModeManager.resetAccount(u1Address, u2Address);
```

函数签名：

```
    TransactionReceipt resetAccount(String oldAccount, String newAccount)
```

输入参数：

- oldAccount 用户的外部账户的原私钥地址。
- newAccount 该账户被重置后的私钥地址。

返回参数：

- TransactionReceipt 交易回执

冻结普通账户

具体调用示例：

```
    TransactionReceipt tr = adminModeManager.freezeAccount(u1Address);
```

函数签名：

```
TransactionReceipt freezeAccount(String account)
```

输入参数：

- account 用户的外部账户的私钥地址。

返回参数：

- TransactionReceipt 交易回执

解冻普通账户

具体调用示例：

```
TransactionReceipt tr = adminModeManager.unfreezeAccount(u1Address);
```

函数签名：

```
TransactionReceipt unfreezeAccount(String account)
```

输入参数：

- account 用户的外部账户的私钥地址。

返回参数：

- TransactionReceipt 交易回执

账户强制注销

具体调用示例：

```
TransactionReceipt tr = governAccountInitializer.cancelAccount(u1Address);
```

函数签名：

```
TransactionReceipt cancelAccount(String account)
```

输入参数：

- account 用户的外部账户的私钥地址。

返回参数：

- TransactionReceipt 交易回执

移交管理员的权限

移交管理员账户时，需要确保被移交的账户已注册，且账户状态正常。 **具体调用示例：**

```
TransactionReceipt tr = adminModeManager.transferAdminAuth(u1Address);
```

函数签名：

```
TransactionReceipt transferAdminAuth(String account)
```

输入参数：

- account 用户的外部账户的私钥地址。

返回参数：

- TransactionReceipt 交易回执

多签制治理模式

治理委员会模式下的管理功能均位于 VoteModeGovernManager 类中。 首先，注入该类：

```
@Autowired
private VoteModeGovernManager voteModeGovernManager;
```

在本模式下，执行任何账户相关的业务操作需要遵循以下步骤：

1. 发起一个投票请求；
2. 治理账户成员赞同该投票；
3. 投票发起者确认投票已经通过后，发起操作。

我们首先来介绍下通用的投票接口：

治理委员会成员投票

具体调用示例：

```
    TransactionReceipt tr = voteModeGovernManager.vote(requestId, true);
```

函数签名：

```
    TransactionReceipt vote(BigInteger requestId, boolean agreed)
```

输入参数：

- requestId 发起投票的requestId。
- agreed 是否同意，true/false

返回参数：

- TransactionReceipt 交易回执

重置用户私钥

参考上文提及的三个步骤：发起投票请求、投票、执行操作。此处，使用了单SDK来处理多用户的操作，使用了changeCredentials函数来切换不同的用户。 **具体调用示例：**

```
    // 发起投票请求
    BigInteger requestId = voteModeGovernManager.requestResetAccount(p2.
↪getAddress(), p1.getAddress());
    // 执行投票
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u1);
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u);
    // 发起重置私钥操作
    TransactionReceipt tr = voteModeGovernManager.resetAccount(requestId, p2.
↪getAddress(), p1.getAddress());
```

发起重置用户私钥投票申请

函数签名：

```
    BigInteger requestResetAccount(String newCredential, String oldCredential)
```

输入参数：

- oldCredential 用户的外部账户的原私钥地址。
- newCredential 该账户被重置后的私钥地址

返回参数：

- BigInteger 投票ID，用户需保存该ID便于后续的交互和其他操作。

重置用户私钥

函数签名：

```
    TransactionReceipt resetAccount(BigInteger requestId, String newCredential,␣
↪String oldCredential)
```

输入参数：

- requestId 之前的投票请求返回的ID
- newCredential 该账户被重置后的私钥地址
- oldCredential 用户的外部账户的原私钥地址。

返回参数：

- TransactionReceipt 交易回执。

冻结普通账户

参考上文提及的三个步骤：发起投票请求、投票、执行操作。此处，使用了单SDK来处理多用户的操作，使用了changeCredentials函数来切换不同的用户。 **具体调用示例：**

```
    // 发起投票请求
    BigInteger requestId = voteModeGovernManager.requestFreezeAccount(p2.
↪getAddress());
    // 执行投票
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u1);
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u);
    // 发起重置私钥操作
    TransactionReceipt tr = voteModeGovernManager.freezeAccount(requestId, p2.
↪getAddress());
```

发起冻结用户账户投票申请

函数签名：

```
   BigInteger requestFreezeAccount(String credential)
```

输入参数：

- externalAccount 用户的外部账户地址。

返回参数：

- BigInteger 投票ID，用户需保存该ID便于后续的交互和其他操作。

冻结用户账户

函数签名：

```
TransactionReceipt freezeAccount(BigInteger requestId, String credential)
```

输入参数：

- requestId 之前的投票请求返回的ID
- externalAccount 用户的外部账户地址。

返回参数：

- TransactionReceipt 交易回执。

解冻普通账户

参考上文提及的三个步骤：发起投票请求、投票、执行操作。此处，使用了单SDK来处理多用户的操作，使用了changeCredentials函数来切换不同的用户。 **具体调用示例：**

```
    // 发起投票请求
    BigInteger requestId = voteModeGovernManager.requestUnreezeAccount(p2.
↪getAddress());
    // 执行投票
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u1);
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u);
    // 发起重置私钥操作
    TransactionReceipt tr = voteModeGovernManager.unfreezeAccount(requestId, p2.
↪getAddress());
```

发起解冻用户账户投票申请

函数签名：

```
BigInteger requestunfreezeAccount(String credential)
```

输入参数：

- externalAccount 用户的外部账户地址。

返回参数：

- BigInteger 投票ID，用户需保存该ID便于后续的交互和其他操作。

解冻用户账户

函数签名：

```
TransactionReceipt unfreezeAccount(BigInteger requestId, String credential)
```

输入参数：

- requestId 之前的投票请求返回的ID
- externalAccount 用户的外部账户地址。

返回参数：

- TransactionReceipt 交易回执。

账户强制注销

参考上文提及的三个步骤：发起投票请求、投票、执行操作。此处，使用了单SDK来处理多用户的操作，使用了changeCredentials函数来切换不同的用户。 **具体调用示例：**

```
    // 发起投票请求
    BigInteger requestId = voteModeGovernManager.requestCancelAccount(p2.
↪getAddress());
    // 执行投票
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u1);
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u);
    // 发起重置私钥操作
    TransactionReceipt tr = voteModeGovernManager.cancelAccount(requestId, p2.
↪getAddress());
```

发起注销用户账户投票申请

函数签名：

```
BigInteger requestCancelAccount(String credential)
```

输入参数：

- externalAccount 用户的外部账户地址。

返回参数：

- BigInteger 投票ID，用户需保存该ID便于后续的交互和其他操作。

注销用户账户

函数签名：

```
TransactionReceipt cancelAccount(BigInteger requestId, String credential)
```

输入参数：

- requestId 之前的投票请求返回的ID
- externalAccount 用户的外部账户地址。

返回参数：

- TransactionReceipt 交易回执。

设置治理账户投票的阈值

参考上文提及的三个步骤：发起投票请求、投票、执行操作。此处，使用了单SDK来处理多用户的操作，使用了changeCredentials函数来切换不同的用户。 **具体调用示例：**

```
    // 发起投票请求
    BigInteger requestId = voteModeGovernManager.
↪requestResetThreshold(newThreshold);
    // 执行投票
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u1);
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u);
    // 发起重置私钥操作
    TransactionReceipt tr = voteModeGovernManager.resetThreshold(requestId,␣
↪newThreshold);
```

发起设置治理账户投票申请

函数签名：

```
    BigInteger requestRemoveGovernAccount(int newThreshold)
```

输入参数：

- newThreshold 新阈值。

返回参数：

- BigInteger 投票ID，用户需保存该ID便于后续的交互和其他操作。

设置新阈值

函数签名：

```
    TransactionReceipt resetThreshold(BigInteger requestId, int threshold)
```

输入参数：

- requestId 之前的投票请求返回的ID
- newThreshold 新阈值。

返回参数：

- TransactionReceipt 交易回执。

治理账户删除一个投票账户

参考上文提及的三个步骤：发起投票请求、投票、执行操作。此处，使用了单SDK来处理多用户的操作，使用了changeCredentials函数来切换不同的用户。 **具体调用示例：**

```
    // 发起投票请求
    BigInteger requestId = voteModeGovernManager.requestRemoveGovernAccount(p2.
↪getAddress());
    // 执行投票
    voteModeGovernManager.vote(requestId, true);
```

(continues on next page)

(续上页)

```
    // 切换投票者
    voteModeGovernManager.changeCredentials(u1);
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u);
    // 发起重置私钥操作
    TransactionReceipt tr = voteModeGovernManager.removeGovernAccount(requestId,␣
↪p2.getAddress());
```

发起删除一个治理账户投票申请

函数签名：

```
   BigInteger requestRemoveGovernAccount(String credential)
```

输入参数：

- externalAccount 用户的外部账户地址。

返回参数：

- BigInteger 投票ID，用户需保存该ID便于后续的交互和其他操作。

删除一个投票账户

函数签名：

```
   TransactionReceipt removeGovernAccount(BigInteger requestId, String credential)
```

输入参数：

- requestId 之前的投票请求返回的ID
- externalAccount 用户的外部账户地址。

返回参数：

- TransactionReceipt 交易回执。

治理账户添加一个投票新账户

参考上文提及的三个步骤：发起投票请求、投票、执行操作。此处，使用了单SDK来处理多用户的操作，使用了changeCredentials函数来切换不同的用户。 **具体调用示例：**

```
    // 发起投票请求
    BigInteger requestId = voteModeGovernManager.requestAddGovernAccount(p2.
↪getAddress());
    // 执行投票
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u1);
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u);
    // 发起重置私钥操作
    TransactionReceipt tr = voteModeGovernManager.addGovernAccount(requestId, p2.
↪getAddress());
```

发起添加一个治理账户投票申请

函数签名：

```
BigInteger requestAddGovernAccount(String credential)
```

输入参数：

- externalAccount 用户的外部账户地址。

返回参数：

- BigInteger 投票ID，用户需保存该ID便于后续的交互和其他操作。

添加一个投票账户

函数签名：

```
TransactionReceipt addGovernAccount(BigInteger requestId, String credential)
```

输入参数：

- requestId 之前的投票请求返回的ID
- externalAccount 用户的外部账户地址。

返回参数：

- TransactionReceipt 交易回执。

不同权重的投票制治理模式

不同权重的投票制模式总体和多签制非常类似，此处不再做过多的赘述，请参考上节。

治理账户添加一个投票新账户

参考上文提及的三个步骤：发起投票请求、投票、执行操作。此处，使用了单SDK来处理多用户的操作，使用了changeCredentials函数来切换不同的用户。 **具体调用示例：**

```
    // 发起投票请求
    BigInteger requestId = voteModeGovernManager.requestAddGovernAccount(p2.
↪getAddress() , weight);
    // 执行投票
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u1);
    voteModeGovernManager.vote(requestId, true);
    // 切换投票者
    voteModeGovernManager.changeCredentials(u);
    // 发起重置私钥操作
    TransactionReceipt tr = voteModeGovernManager.addGovernAccount(requestId, p2.
↪getAddress(), weight);
```

发起添加一个治理账户投票申请

函数签名：

```
BigInteger requestAddGovernAccount(String credential, int weight)
```

输入参数：

- externalAccount 用户的外部账户地址。
- weight 新加入账户的权重

返回参数：

- BigInteger 投票ID，用户需保存该ID便于后续的交互和其他操作。

添加一个投票账户

函数签名：

```
TransactionReceipt addGovernAccount(BigInteger requestId, String credential,␣
↪int weight)
```

输入参数：

- requestId 之前的投票请求返回的ID
- externalAccount 用户的外部账户地址。
- weight 新加入账户的权重

返回参数：

- TransactionReceipt 交易回执。

其他交易

其余部分颇为相似，可参考基于相同权重的投票模式

1.4.3 普通用户接口

普通账户主要功能

普通账户的功能均位于EndUserOperManager类中。 首先，注入该类：

```
@Autowired
private EndUserOperManager endUserOperManager;
```

创建新账户

具体调用示例：

```
String account = endUserAdminManager.createAccount(p1Address);
```

函数签名：

```
String createAccount(String who)
```

输入参数：

- externalAccount 待创建的账户的外部账户的私钥地址。

返回参数：

- String 新建的账户地址

重置用户私钥

具体调用示例：

```
TransactionReceipt tr = endUserAdminManager.resetAccount(p2Address);
```

函数签名：

```
TransactionReceipt resetAccount(String newCredential)
```

输入参数：

- newCredential 待更换的私钥地址。

返回参数：

- TransactionReceipt 交易回执

账户强制注销

具体调用示例：

```
TransactionReceipt tr = endUserAdminManager.cancelAccount();
```

函数签名：

```
TransactionReceipt cancelAccount()
```

返回参数：

- TransactionReceipt 交易回执

修改普通账户的管理类型

具体调用示例：

```
List<String> voters = Lists.newArrayList();
voters.add(u.getAddress());
voters.add(u1.getAddress());
voters.add(u2.getAddress());
TransactionReceipt tr = endUserAdminManager.modifyManagerType(voters);
```

函数签名：

```
TransactionReceipt modifyManagerType(List<String> voters)
//重载函数，设置为仅自己管理
TransactionReceipt modifyManagerType()
```

输入参数：

- voters （可选） 当变更为支持社交好友投票时需要传入，且voters的大小必须为3。投票本身为2-3的规则。如果voters不传入，则默认不开启投票模式。

返回参数：

- TransactionReceipt 交易回执

添加一个社交好友

具体调用示例：

```
    TransactionReceipt tr = endUserAdminManager.addRelatedAccount(userAddress);
```

函数签名：

```
    TransactionReceipt addRelatedAccount(String account)
```

输入参数：

- account 被添加好友的外部账户地址。

返回参数：

- TransactionReceipt 交易回执

删除一个社交好友

具体调用示例：

```
    TransactionReceipt tr = endUserAdminManager.removeRelatedAccount(userAddress);
```

函数签名：

```
    TransactionReceipt removeRelatedAccount(String account)
```

输入参数：

- account 被删除好友的外部账户地址。

返回参数：

- TransactionReceipt 交易回执

使用社交好友投票重置私钥

重置用户私钥

参考上文提及的三个步骤：发起投票请求、投票、执行操作。此处，使用了单SDK来处理多用户的操作，使用了changeCredentials函数来切换不同的用户。 社交好友投票相关的操作在 SocialVoteManager 类中。 **具体调用示例：**

```
    // 发起投票请求
    TransactionReceipt t = socialVoteManager.requestResetAccount(u1.getAddress(),␣
↪p1.getAddress());
    // 执行投票
    socialVoteManager.vote(requestId, true);
    // 切换投票者
    socialVoteManager.changeCredentials(u1);
    socialVoteManager.vote(requestId, true);
    // 切换投票者
    socialVoteManager.changeCredentials(u);
    // 发起重置私钥操作
    TransactionReceipt tr = socialVoteManager.resetAccount(u1.getAddress(), p1.
↪getAddress());
```

发起重置用户私钥投票申请

函数签名：

```
    TransactionReceipt requestResetAccount(String newCredential, String␣
↪oldCredential)
```

输入参数：

- oldCredential 用户的外部账户的原私钥地址。
- newCredential 该账户被重置后的私钥地址

返回参数：

- TransactionReceipt 交易回执。

投票

函数签名：

```
    TransactionReceipt vote(String oldCredential, boolean agreed)
```

输入参数：

- oldCredential 申请变更账户的外部账户地址。
- agreed 是否同意

返回参数：

- TransactionReceipt 交易回执。

重置用户私钥

函数签名：

```
    TransactionReceipt resetAccount(String newCredential, String oldCredential)
```

输入参数：

- newCredential 该账户被重置后的私钥地址
- oldCredential 用户的外部账户的原私钥地址。

返回参数：

- TransactionReceipt 交易回执。

1.5 组件使用demo

1.5.1 合约集成demo

存证demo

存证合约的demo

```
pragma solidity ^0.4.25;

import "./AccountManager.sol";

contract EvidenceDemo {
    struct EvidenceData {
        string hash;
        address owner;
        uint256 timestamp;
    }
    address public _owner;
    mapping(string => EvidenceData) private _evidences;
    // import AccountManager
    AccountManager _accountManager;

    constructor(address accountManager) public {
        // import accountManager
        _accountManager = AccountManager(accountManager);
        // set user account instead of external account
        _owner = _accountManager.getAccount(msg.sender);
        require(_owner != 0x0, "Invalid account!");
    }

    modifier onlyOwner() {
        // get user account by external account
        address userAccountAddress = _accountManager.getAccount(msg.sender);
        require(userAccountAddress == _owner, "Not admin");
        _;
    }

    function setData(
        string hash,
        address owner,
        uint256 timestamp
    ) public onlyOwner {
        _evidences[hash].hash = hash;
        _evidences[hash].owner = owner;
        _evidences[hash].timestamp = timestamp;
    }

    function getData(string hash)
        public
        view
        returns (
            string,
            address,
            uint256
        )
    {
        EvidenceData storage evidence = _evidences[hash];
        return (evidence.hash, evidence.owner, evidence.timestamp);
    }
}
```

存证合约的demo在控制台的部署指令

```
================================================================================================
Welcome to FISCO BCOS console(1.0.9)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ _______  _______   _______  _______       ________  _______  _______  _______
|        |       \/       \ /       \ /       \     |        \ /       \ /       \ /       ␣
↪\
```

(continues on next page)

(续上页)

```
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
| $$__      | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \     | $$  \$$    \| $$      | $$  | $$    | $$    $| $$      | $$  | $$\$$   ␣
↪\
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$\
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__|
↪$$
| $$      |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$
↪$
```

```
================================================================================
[group:1]> deploy EvidenceDemo "0x08ba44f04df9671a2fc298756bc06f1dbca56e11"
revert instruction

[group:1]> deploy AdminGovernBuilder
contract address: 0xc00504c8dad8f75d08ebceda4f7c7b1d13b327d5

[group:1]> call AdminGovernBuilder 0xc00504c8dad8f75d08ebceda4f7c7b1d13b327d5 _
↪governance
0x24c327f0bf851a936c5049b019142709067b711d

[group:1]> call WEGovernance 0x24c327f0bf851a936c5049b019142709067b711d␣
↪getAccountManager
0x6e869333a1e3dc83501723b7dcb624b09c1757e3

[group:1]>  deploy EvidenceDemo "0x6e869333a1e3dc83501723b7dcb624b09c1757e3"
contract address: 0x99276281a782a199d1998ce7a56927d99dcd6c6a

[group:1]> call EvidenceDemo 0x99276281a782a199d1998ce7a56927d99dcd6c6a setData
↪"hash" "0x6e869333a1e3dc83501723b7dcb624b09c1757e3" 1
transaction hash:␣
↪0xc7f2531ca9e968a97a6035a4fbfa79f2a0cda7adfb4b9f878e36e4a879fa5249

[group:1]> call EvidenceDemo 0x99276281a782a199d1998ce7a56927d99dcd6c6a getData
↪"hash"
[hash, 0x6e869333a1e3dc83501723b7dcb624b09c1757e3, 1]
```

转账demo

转账合约的demo

```
pragma solidity ^0.4.25;


library LibSafeMath {
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a, "SafeMath: subtraction overflow");
        uint256 c = a - b;

        return c;
    }
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");
```

(continues on next page)

(续上页)

```
        return c;
    }
}


pragma solidity ^0.4.25;

import "./LibSafeMath.sol";
import "./AccountManager.sol";


contract TransferDemo {
    using LibSafeMath for uint256;
    mapping(address => uint256) private _balances;
    // import AccountManager
    AccountManager _accountManager;

    constructor(address accountManager, uint256 initBalance) public {
        // import accountManager
        _accountManager = AccountManager(accountManager);
        address owner = _accountManager.getAccount(msg.sender);
        _balances[owner] = initBalance;
    }

    modifier validateAccount(address addr) {
        require(
            // predicate account status
            _accountManager.isExternalAccountNormal(addr),
            "Account is abnormal!"
        );
        _;
    }

    modifier checkTargetAccount(address sender) {
        require(
            msg.sender != sender && sender != address(0),
            "Can't transfer to illegal address!"
        );
        _;
    }

    function balance(address owner) public view returns (uint256) {
        // 1.get account by external account, 2.get balace by account.
        return _balances[_accountManager.getAccount(owner)];
    }

    function transfer(address toAddress, uint256 value)
        public
        // validate source & target account
        validateAccount(msg.sender)
        validateAccount(toAddress)
        checkTargetAccount(toAddress)
        returns (bool)
    {
        // 1. get source account
        address fromAccount = _accountManager.getAccount(msg.sender);
        // 2. sub the balance of source account
        uint256 balanceOfFrom = _balances[fromAccount].sub(value);
        // 3. modify the balance of source account
        _balances[fromAccount] = balanceOfFrom;
```

(continues on next page)

(续上页)

```
        // 4. get target account
        address toAccount = _accountManager.getAccount(toAddress);
        // 5. add balance of target account
        uint256 balanceOfTo = _balances[toAccount].add(value);
        // set the new balance of target account
        _balances[toAccount] = balanceOfTo;
        return true;
    }
}
```

转账合约的demo在控制台的部署指令

```
=====================================================================================================
Welcome to FISCO BCOS console(1.0.9)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ _______  _______   _______  _______       ________   _______   _______   _______
|        |       \/       \ /       \ /       \     |        \ /       \ /       \ /       \
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
| $$__      | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \     | $$  \$$    \| $$      | $$  | $$    | $$    $| $$      | $$  | $$\$$
↪\
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$\
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__|
↪$$
| $$      |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$
↪$

=====================================================================================================
[group:1]> deploy AdminGovernBuilder
contract address: 0xd17a5cdcb5e4b1cc3ce67c71c8e0e1dd07f33914

[group:1]> call AdminGovernBuilder 0xd17a5cdcb5e4b1cc3ce67c71c8e0e1dd07f33914 _
↪governance
0xda80ff2d1cd498c86439cf52c1b1a8bb01da6fbc

[group:1]> call WEGovernance 0xda80ff2d1cd498c86439cf52c1b1a8bb01da6fbc␣
↪getAccountManager
0x199a2b9f43415f1f5ca9da6dc7c3dc124c531fd5

[group:1]> deploy TransferDemo "0x199a2b9f43415f1f5ca9da6dc7c3dc124c531fd5" 1000
contract address: 0x7873756b7a93afed89482040257d332e3fc72336

[group:1]> call TransferDemo 0x7873756b7a93afed89482040257d332e3fc72336 transfer
↪"0x1" 1
The execution of the contract rolled back.

[group:1]> call AccountManager 0x199a2b9f43415f1f5ca9da6dc7c3dc124c531fd5␣
↪newAccount "0x1"
transaction hash:␣
↪0xf5a806396256e01295d014edf5688afdc0e7c1f4363c47f5bd083a14f1398192
-----------------------------------------------------------------------------------------
↪----------
Output
function: newAccount(address)
return type: (bool, address)
```

(continues on next page)

(续上页)

```
return value: (true, 0x941d587493454784874e7d463dc76368f20bd3ff)
-------------------------------------------------------------------------------------------
↪----------
Event logs
event signature: LogSetOwner(address,address) index: 0
event value: (0x0000000000000000000000000000000000000001,␣
↪0x941d587493454784874e7d463dc76368f20bd3ff)
event signature: LogManageNewAccount(address,address,address) index: 0
event value: (0x0000000000000000000000000000000000000001,␣
↪0x941d587493454784874e7d463dc76368f20bd3ff,␣
↪0x199a2b9f43415f1f5ca9da6dc7c3dc124c531fd5)
-------------------------------------------------------------------------------------------
↪----------

[group:1]> call TransferDemo 0x7873756b7a93afed89482040257d332e3fc72336 transfer
↪"0x1" 1
transaction hash:␣
↪0xdbe329df31c18fd54b87139045db1fe2d0358c54139f1b2b649fb730c9a33420
-------------------------------------------------------------------------------------------
↪----------
Output
function: transfer(address,uint256)
return type: (bool)
return value: (true)
-------------------------------------------------------------------------------------------
↪----------

[group:1]> call TransferDemo 0x7873756b7a93afed89482040257d332e3fc72336 balance
↪"0x1"
1

[group:1]>
```

1.5.2 SDK集成Demo

提供了SDK集成和使用的demo，展示了如何使用Java SDK。 详情可参考 Governance-Account-Demo

1.5.3 测试代码说明

链配置

打开src/main/application.properties，修改链配置信息。

```
## 机构ID
system.orgId=org1
## 链的ip端口，多个节点使用;分隔
system.nodeStr=[ip]:[channel_port]
## 群组ID
system.groupId=1
```

自动运行

```
./gradlew test
```

可查看自动化测试的运行结果报告。

CHAPTER 2

权限治理组件

简介

WeBankBlockchain-Governance-Authority定位为区块链权限治理组件，旨在为智能合约开发者提供强大、轻便的权限管理工具，使得开发者可以快速地为自己的业务系统搭建智能合约权限控制体系。

2.1 组件介绍

2.1.1 背景

随着智能合约业务日益丰富，越来越多的合约需要引入权限控制。如果不对智能合约做权限控制，那么无法满足业务的安全性要求。例如，存证场景中，除了上传存证的函数外，还有许多专供审核人员调用的函数，这些函数应仅由审核人员来使用，如果这些函数没有正确设置权限拦截逻辑，整套逻辑就会被攻击者轻易操控。

WeBankBlockchain-Governance-Authority的目的在于为智能合约开发者提供权限控制功能。开发者只要添加少量代码，即可拦截非法调用。同时，有一个专门的权限治理合约用于治理各个业务合约的拦截规则，对规则的修改只需操作权限治理合约，不需要调整业务合约，且修改会实时生效。

2.1.2 特性

函数级的权限粒度

在权限体系中，可以为每个函数单独设置权限，同一个合约中的不同函数可有不同的权限设置。

批量设置用户权限

合约函数的权限配置基本单位为组，使得可以设置整批账户的权限访问规则。

侵入性低

业务合约只需要在代码中引入权限合约地址，并通过在需要权限控制的函数中访问权限合约的权限判断接口，就可以实现权限控制。

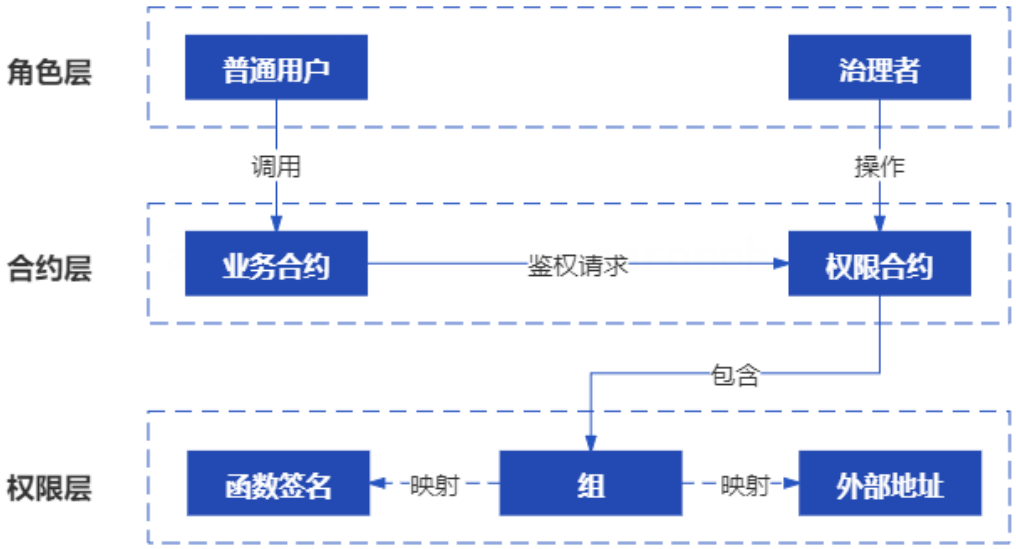
易于集成

对于权限管理人员，既支持通过手动方式来调用权限合约，也支持通过sdk方式进行调用。

支持多种治理方式

允许中心化和去中心化两种方式进行权限治理。在中心化方式中，有一位管理员，可以直接设置各个账户的权限信息；在去中心化方式下，有一个治理委员会，治理委员会成员通过投票来决定如何更改权限信息。

2.1.3 整体原理



整体原理如下：
合约角度来看，合约分为业务合约和权限合约，业务合约包含了业务逻辑，权限合约包含了权限信息。从角色划分来看，参与角色分为普通调用者和管理员，普通调用者用于调用业务合约，而管理员负责修改权限合约中的合约信息。

当普通调用者在调用业务合约的时候，植入在业务合约内部的拦截器会向权限合约查询该用户是否有权限调用该函数。对于权限合约而言，权限判断依赖于存储其内的“组权限信息列表”，每个组都记录了包含哪些地址、哪些函数、运行于什么模式。例如，组A包含了alice和bob两个地址，并关联了某合约的helloworld函数，并且该组运行于白名单模式下，那么就意味着仅有alice和bob可以访问该合约的helloworld函数，而其他账户是不可以访问该函数；组B包含了carol账户，并关联了某个函数functionB，而且设置为黑名单模式，那么意味着carol已经被functionB账户”拉黑”，而其他账户可以正常访问该函数。

2.1.4 场景示例

假设现在要开发一个存证智能合约，要求其中某些函数仅能由审查员来调用，例如验证存证。这时，权限合约的管理员克到权限治理合约中配置“审查员”组，将审查员的地址添加到该组中、添加相应函数，

并将组设为白名单模式。如此配置完成后，权限规则即生效，当审查员之外的账户在调用这些函数的时候，均会被拦截掉，有且仅有审查员可以调用这些函数。

2.2 快速开始

2.2.1 前置依赖

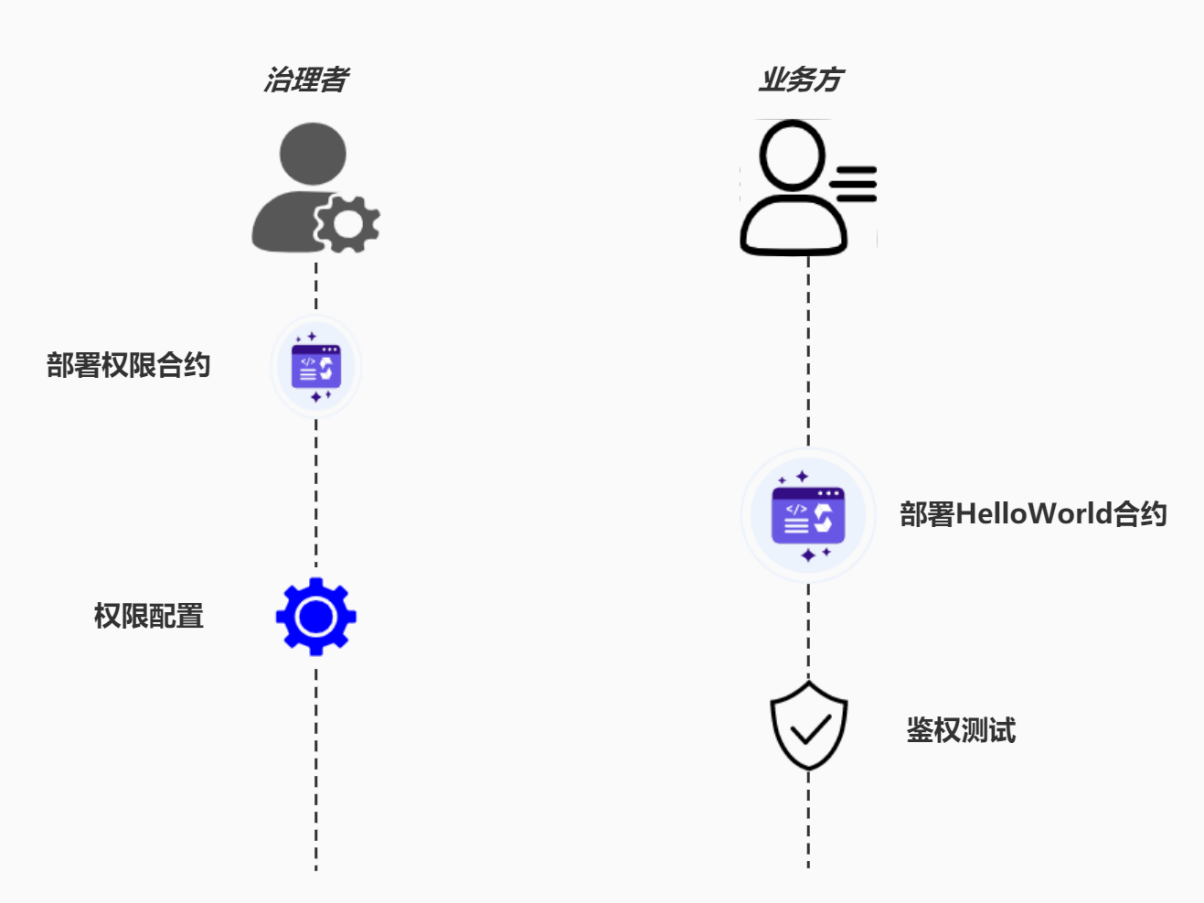
注解:

- JDK1.8 或者以上版本，推荐使用OracleJDK。CentOS的yum仓库的OpenJDK缺少JCE(Java Cryptography Extension)，会导致JavaSDK无法正常连接区块链节点。
- 参考 Java环境配置
- FISCO BCOS区块链环境搭建参考 FISCO BCOS安装教程
- 网络连通性。检查所连接的FISCO BCOS节点'channel_listen_port'是否能telnet通，若telnet不通，需要检查网络连通性和安全策略。

2.2.2 快速开始

权限组件的使用者包括两个角色：治理方和业务方。治理方负责权限合约的部署、配置；业务方负责接入权限合约、拦截非法调用。这一节提供一个简单但完整的示例，通过部署并为业务合约HelloWorld配置权限，以使您了解整个组件的使用流程。这一节的内容包括：

- 【治理方】部署权限合约
- 【业务方】部署HelloWorld合约
- 【治理方】配置HelloWorld的权限
- 【业务方】权限验证



此外，操作权限合约可以通过控制台，或者后文的sdk调用的方式。本节操作示例使用了FISCO BCOS控制台作为客户端来部署合约。如果您对控制台的操作不熟悉，建议参考FISCO BCOS控制台教程。

合约下载

通过git下载源码，获取智能合约

```
git clone https://github.com/WeBankBlockchain/Governance-Authority.git
cd auth-manager/src/main/contracts
```

启动控制台

现在我们将所有合约代码（即上文auth-manager/src/main/contracts目录中的所有代码）拷贝到控制台的contracts/solidity目录下。

生成两个控制台身份

```
bash get_account.sh
bash get_account.sh
```

该身份会以pem形式保存在控制台的accounts目录下。我们将第一个身份称为“治理者账户”，用于操作权限合约；第二个身份扮演“业务方账户”，用于调用HelloWorld合约。以治理员身份启动控制台：

```
bash start.sh 1 -pem [治理员pem文件，如accounts/xxx.pem]
```

部署权限合约

启动控制台后，下一步就可以部署权限合约了。

```
[group:1]> deploy AuthManager 1 [] [] 0
contact address:[权限合约地址]
```

AuthManager是权限治理合约。部署的时候可传入一组与运行模式有关的参数，按上面填写即可。部署成功后，会返回"contract address"字样和权限合约地址。部署后，仅有治理员身份可以操作该权限合约。

部署HelloWorld合约

在治理方部署了权限治理合约后，业务方现在需要将权限治理合约引入自己的智能合约。

编写HelloWorld合约

现在我们写有一个HelloWorld智能合约，它的代码如下：

```
pragma solidity ^0.4.25;

contract HelloWorld{

    event Hello();
    function hello() public {
        emit Hello();
    }

}
```

接下来引入前面下载的IAuthControl，按下述方式引入该合约：

```
pragma solidity ^0.4.25;

import "./IAuthControl.sol";
contract HelloWorld{
    IAuthControl private _authManager;
    constructor(IAuthControl authManager){
        _authManager = authManager;
    }

    event Hello();
    function hello() public {
        require(_authManager.canCallFunction(address(this), msg.sig, msg.sender));
        emit Hello();
    }
}
```

引入可以分为三步：

- 第一步，导入IAuthControl.sol文件（第3行import）
- 第二步，在构造方法中传入权限治理合约的地址（第5到第8行）
- 第三步，在业务函数中引入权限治理合约的权限判断逻辑（第12行canCallFunction）。

其中，为canCallFunction传入了三个参数，表示向权限合约询问"当前调用者是否有权限调用当前合约的当前函数"。通常，业务方在调用canCallFunction时，只需固定按此写法传参即可，这三个变量已经代表了当前的执行环境。它们含义分别为：

- address(this)：表示当前HelloWorld合约部署后的地址
- msg.sig：当前函数（本例中为hello）的函数签名。
- msg.sender表示当前函数的调用者。

HelloWorld合约部署

将HelloWorld.sol拷贝到控制台contracts/solidity目录下。启动控制台并部署HelloWorld合约：

```
bash start.sh
[group:1]> deploy HelloWorld [权限合约地址]
contact address: [HelloWorld合约地址]
```

其中，构造函数传入的是权限合约地址，这个地址是前面权限合约部署后的地址。

至此，业务方的接入工作已经完成，但现在部署后权限还没有起到效果，因为需要治理方在权限合约中进行权限规则配置。

2.2.3 权限配置

现在以治理方的身份启动控制台，以便操作权限治理合约：

```
bash start.sh 1 -pem [治理者pem文件，如accounts/xxx.pem]
```

由于权限治理合约是基于组的，需要先创立一个组，该组命名为exampleGroup，参数1表示这个组是白名单组，表示它关联的函数都是白名单模式，只有组内成员可以访问它关联的函数。

```
[group:1]> call AuthManager [权限合约地址] createGroup "exampleGroup" 1
```

随后，将测试账户添加到该组：

```
[group:1]> call AuthManager [权限合约地址] addAccountToGroup [业务方的账户地址]
↪"exampleGroup"
```

最后，关联函数和组，这样只有该组允许访问此函数：

```
[group:1]> call AuthManager [权限合约地址] addFunctionToGroup [HelloWorld合约地址]
↪"hello()" "exampleGroup"
```

其中，HelloWorld合约地址是3.2.2节中部署HelloWorld合约后的地址；

经过如此配置后，则仅有业务方被允许访问hello函数。

2.2.4 验证

当权限规则配置完毕，这个时候对HelloWorld的非法访问就会被拦截。例如以随机的身份启动控制台：

```
bash start.sh
[group:1]> call HelloWorld [HelloWorld合约地址] hello
The execution of the contact rolled back.
```

这个时候，由于该身份不在白名单内，访问就会报错。但如果以白名单组员的身份来调用HelloWorld，就可以成功：

```
bash start.sh 1 -pem [业务方账号pem文件，如accounts/xxx.pem]
[group:1]> call HelloWorld [HelloWorld合约地址] hello
0x21dca087cb3e44f44f9b882071ec6ecfcb500361cad36a52d39900ea359d0895
```

2.3 治理手册

这一节主要讲解如何治理权限合约。如果您是业务方，无需阅读接下来的内容；如果您是治理方，想了解如何部署权限合约、配置权限规则，则需要继续阅读本节。

2.3.1 关键概念

业务合约与权限合约

权限治理涉及的合约分为两类：业务合约、权限合约。

- 业务合约是用户自己开发的合约，不属于本组件。业务合约通过访问权限合约的canCallFunctoin函数来拦截非法调用。
- 权限合约是本组件提供的AuthManager合约，用于配置哪些账户可以访问哪些合约的哪些函数。可以治理多个业务合约的权限。

组

组定义了哪些账户可以访问哪些函数。组包含的信息如下：

- 账户列表
- 模式。包含两个可选择的模式：
  - 黑名单模式
  - 白名单模式
- 函数列表

通过上面的配置，即可确定哪些账户可以访问哪些函数:

- 配置了一个黑名单组，意味着仅组内账户不能访问这些函数
- 配置了一个白名单组，意味着仅组内账户可以访问这些函数

治理模式

权限治理合约有两种模式：管理员模式、委员会模式

- 管理员模式下，由单一管理员修改组配置； 还可以转让管理员权限。
- 委员会模式下，所有操作均通过投票进行。委员会成员可以修改组配置。还可以修改委员会列表、投票规则等。

一个权限治理合约的模式在部署时即确定，一旦确定一种模式，就无法更改。

投票模式

投票包含两种规则：多签模式和阈值权重模式。

- 多签模式下，当投票数达到最小签名数时，投票即通过。
- 阈值权重模式下，每个委员均配有对应的权重，当已投票的总权重达到最小阈值时，投票即通过。

多签是阈值权重模式下的一个特例，即所有委员会的权重为1。

2.3.2 AuthManager合约接口列表

- 合约部署

合约部署时需要决定是管理员模式还是治理委员会模式。以下为管理员模式下权限配置接口：

- createGroup 创建组
- addAccountToGroup 将账户添加到组

- addFunctionToGroup 将合约函数关联到组
- removeAccountFromGroup 将账户从组内移除
- removeFunctionFromGroup 将合约函数与组的关联取消

以下为治理委员会模式下的权限配置接口：

- requestCreateGroup 请求创建组
- requestAddAccountToGroup 请求将账户添加到组
- requestAddFunctionToGroup 请求将合约函数关联到组
- requestRemoveAccountFromGroup 请求将账户从组内移除
- requestRemoveFunctionFromGroup 请求将合约函数与组的关联取消
- approveSingle 投票请求
- deleteSingle 删除请求
- getRequestSingle 查看请求
- executeCreateGroup 创建组
- executeAddAccountToGroup 执行将账户添加到组
- executeAddFunctionToGroup 执行将合约函数关联到组
- executeRemoveAccountFromGroup 执行将账户从组内移除
- executeRemoveFunctionFromGroup 执行将合约函数与组的关联取消

以下为通用的查询接口

- containsAccount
- containsFunction
- getGroup
- canCallFunction

以下为管理员模式下的治理接口：

- transferAdminAuth
- isAdmin

以下是治理委员会模式下的治理接口：

- requestSetThreshold
- requestResetGovernors
- executeSetThreshold
- executeResetGovernors
- requestAddGovernor
- deleteAddGovernorReq
- approveAddGovernorReq
- getGovernorsToAdd
- executeAddGovernorReq
- requestRemoveGovernor
- deleteRemoveGovernorReq
- approveRemoveGovernorReq
- getGovernorsToRemove

- executeRemoveGovernorReq

合约部署

说明：部署权限治理合约AuthManager。

参数说明：

- uint mode：1-管理员模式，2-治理委员会模式。如果选择管理员模式，则当前账户为管理员。
- address[] accounts: 初始的治理委员会列表。仅mode为2时有效，如果mode为1，则本参数传空数组即可。
- uint16[] weights: 初始治理委员会权重。仅mode为2时有效，如果mode为1，则本参数传空数组即可。
- uint16 threshold: 投票有效的最小权重。仅mode为2时有效，如果mode为1，则本参数传0即可。

示例1：部署管理员模式下的权限治理合约。后面三个参数如本例一样传空即可。

```
deploy AuthManager 1 [] [] 0
```

示例2：部署委员会模式的权限治理合约，并且采用多签投票方式。比如委员会包含三个地址，分别为”0x1”,”0x2”,”0x3”，要求至少3票投票才能通过，则部署方式为：

```
deploy AuthManager 2 ["0x1", "0x2", "0x3"] [1,1,1] 3
```

此处[1,1,1]表示每个委员权重为1。

示例3：部署委员会方式的权限治理合约，并且采用基于阈值投票方式。比如委员会包含三个地址，分别为”0x1”,”0x2”,”0x3”，权重分别为1，2，3。要求总权重为4，投票才算有效。则部署方式为：

```
deploy AuthManager 2 ["0x1","0x2", "0x3"] [1,2,3] 4
```

createGroup

说明：创建一个组。

调用要求：当前调用者为管理员。该组必须存在。

参数说明：

- string group: 组名
- uint8 mode：组是黑名单还是白名单。1-白名单，2-黑名单

addAccountToGroup

说明：将账户地址拉入到组中。

调用要求：要求当前调用者为管理员。要求组已被创建，且账户尚未在这个组中。

参数说明：

- address account: 账户地址
- string group: 组名

addFunctionToGroup

说明：将某合约的某函数关联到组中。调用后，若当前组为白名单组，则仅有组内账户可以访问该函数；若当前组为黑名单组，则仅有组内账户不允许访问该函数。

调用要求：当前调用者为管理员。要求组已被创建，且函数未被关联到任何组中（无论是本组，还是其他组）

参数说明：

- address contractAddr: 业务合约地址
- string func: 业务合约中待关联函数的签名字符串，如”add(uint256,uint256)”,”hello()”等。
- string group: 组名

removeAccountFromGroup

说明：将账户地址从组中移除。

调用要求：当前调用者为管理员。要求组已被创建，且账户在这个组中。

参数说明：

- address account: 账户地址
- string group: 组名

removeFunctionFromGroup

说明：取消某合约某函数与组的关联。

调用要求：当前调用者为管理员。要求组已被创建，且函数已被关联到组。

参数说明：

- address contractAddr: 业务合约地址
- string func: 业务合约中待关联函数的签名字符串，如”add(uint256,uint256)”,”hello()”等。
- string group: 组名

requestCreateGroup

说明：请求创建一个组。只能同时存在一个同类请求。参数会被缓存，直到请求被执行或删除。

调用要求：当前调用者为治理委员会成员；不存在其他未关闭的同类请求。

参数说明：

- string group: 组名
- uint8 mode：组是黑名单还是白名单。1-白名单，2-黑名单

requestAddAccountToGroup

说明：请求将账户添加到组。只能同时存在一个同类请求。参数会被缓存，直到请求被执行或删除。

调用要求：当前调用者为治理委员会成员；不存在其他未关闭的同类请求。

参数说明：

- address account: 账户地址
- string group: 组名

requestAddFunctionToGroup

说明：请求将函数关联到组。只能同时存在一个同类请求。参数会被缓存，直到请求被执行或删除。

调用要求：当前调用者为治理委员会成员；不存在其他未关闭的同类请求。

参数说明：

- address contractAddr: 业务合约地址
- string func: 业务合约中待关联函数的签名字符串，如”add(uint256,uint256)”,”hello()”等。
- string group: 组名

requestRemoveAccountFromGrop

说明：请求将账户地址从组中移除。只能同时存在一个同类请求。参数会被缓存，直到请求被执行或删除。

调用要求：当前调用者为治理委员会成员；不存在其他未关闭的同类请求。

参数说明：

- address account: 账户地址
- string group: 组名

requestRemoveFunctionFromGroup

说明：请求取消某合约某函数与组的关联。只能同时存在一个同类请求。参数会被缓存，直到请求被执行或删除。

调用要求：当前调用者为治理委员会成员；不存在其他未关闭的同类请求。

参数说明：

- address contractAddr: 业务合约地址
- string func: 业务合约中待关联函数的签名字符串，如”add(uint256,uint256)”,”hello()”等。
- string group: 组名

approveSingle

说明：对某个提案进行投票。

调用要求：当前调用者为治理委员会成员；投票还未关闭。

参数说明：

- uint8 txType：请求类型。见《请求类型列表》一节。

deleteSingle

说明：删除某个提案。

调用要求：当前调用者为治理委员会成员；投票还未关闭。

参数说明：

- uint8 txType：请求类型。见《请求类型列表》一节。

getRequestSingle

说明：获取某个未关闭的投票信息。

调用要求：投票还未关闭。

参数说明：

- uint8 txType：请求类型。见《请求类型列表》一节。

返回值：

- uint256：请求id（可以忽略）
- address：请求的发起地址
- uint256：请求的投票总阈值
- address：（可以忽略）
- uint256：目前投票的总权重
- uint8：请求类型。见《请求类型列表》一节。
- uint8：（可以忽略）

executeCreateGroup

说明：执行创建组。

调用要求：当前调用者为治理委员会成员；请求已被投票通过。

executeAddAccountToGroup

说明：执行将账户添加到组的请求。

调用要求：当前调用者为治理委员会成员；请求已被投票通过。

executeAddFunctionToGroup

说明：执行将业务函数关联到组的请求。

调用要求：当前调用者为治理委员会成员；请求已被投票通过。

executeRemoveAccountFromGroup

说明：执行将账户从组中移除的请求。

调用要求：当前调用者为治理委员会成员；请求已被投票通过。

executeRemoveFunctionFromGroup

说明：执行取消业务函数和组关联的请求。

调用要求：当前调用者为治理委员会成员；请求已被投票通过。

containsAccount

说明：查询某一账户是否位于某一组中。

调用要求：无

参数说明：

- string group:组名
- address account：账户

containsFunction

说明：查询某一合约函数是否已关联到组。

调用要求：无

参数说明：

- string group：组名
- address contractAddr：业务合约地址
- string func:业务合约中待关联函数的签名字符串，如”add(uint256,uint256)”,”hello()”等。

getGroup

说明：查询某一个组的信息

调用要求：无

参数说明：

- string group:组名

返回值说明：

- uint8：组的模式。1-白名单，2-黑名单
- uint256：组包含的账户数目
- uint256：组关联的函数数目

canCallFunction

说明：查询某一账户是否有权限调用某一合约函数

调用要求：无

参数说明：

- address contractAddr：业务合约地址
- bytes4 sig：业务函数的签名字节，由sha3(函数签名字符串)的前4字节得来。和msg.sig一致。
- address caller：调用者

transferAdminAuth

说明：转移管理员权限给另一个账户。

调用要求：当前调用者为管理员。

参数说明：

- address newAdminAddr：新管理员的账户地址。

isAdmin

说明：判断当前调用者是否为合约管理员。

调用要求：无

requestSetThreshold

说明：请求重设投票权重阈值。只能同时存在一个同类请求。参数会被缓存，直到请求被执行或删除。

调用要求：当前调用者为治理委员会成员；不存在其他未关闭的同类请求。

参数说明：

- uint16 newThreshold：新的权重阈值

requestResetGovernors

说明：请求重设治理委员会列表与权重。只能同时存在一个同类请求。参数会被缓存，直到请求被执行或删除。

调用要求：当前调用者为治理委员会成员；不存在其他未关闭的同类请求。

参数说明：

- address[] governors: 新的治理委员会名单
- uint16[] weights: 新的治理委员会对应权重

executeSetThreshold

说明：执行重设阈值请求

调用要求：当前调用者为治理委员会成员；请求已被投票通过。

executeResetGovernAccounts

说明：执行重设治理委员会列表请求

调用要求：当前调用者为治理委员会成员；请求已被投票通过。

requestAddGovernor

说明：请求向治理委员会中添加一个成员

调用要求：当前调用者为治理委员会成员；待添加账户还未没有过相关添加请求

参数说明：

- address account: 待添加成员

deleteAddGovernorReq

说明：删除添加治理委员会成员请求

调用要求：当前调用者为治理委员会成员；待添加账户已经生成了添加请求

参数说明：

- address account: 待添加成员

approveAddGovernorReq

说明：同意添加治理委员会
调用要求：当前调用者为治理委员会成员；待添加账户已经生成了添加请求
参数说明：

- address account: 待添加成员

getGovernorsToAdd

说明：取得所有待添加成员
返回值：

- address[]: 所有待添加的成员名单

executeAddGovernorReq

说明：执行请求
调用要求：当前调用者为治理委员会成员；待添加账户已经生成了添加请求

requestRemoveGovernor

说明：请求从治理委员会中删除一个成员
调用要求：当前调用者为治理委员会成员；待移除账户还未没有过相关移除请求
参数说明：

- address account: 待移除成员

deleteRemoveGovernorReq

说明：删除删除治理委员会成员请求
调用要求：当前调用者为治理委员会成员；待移除账户已经生成了移除请求
参数说明：

- address account: 待移除成员

approveRemoveGovernorReq

说明：同意删除治理委员会成员
调用要求：当前调用者为治理委员会成员；待移除账户已经生成了移除请求
参数说明：

- address account: 待移除成员

getGovernorsToRemove

说明：取得所有待删除成员
返回值：

- address[]: 所有待移除的成员名单

executeAddGovernorReq

说明：执行请求

调用要求：当前调用者为治理委员会成员；待添加账户已经生成了添加请求

2.3.3 常量表

组相关

- 白名单组：1
- 黑名单组：2

请求类型相关

- 重设投票阈值：1
- 重设治理委员会列表：2
- 创建组：3
- 添加账户到组：4
- 关联函数到组：5
- 组内移除函数：6
- 取消函数和组关联：7

2.3.4 集成Sdk

在前面示例中，我们以控制台来操作部署、操作权限合约。除了控制台外，本组件也支持通过java代码来执行这些操作。推荐使用集成SDK的方式来进行权限治理，这样可以在没有控制台的情况下进行权限合约调用。

源码下载

先前章节中，已经通过git下载了源码：

```
git clone git@github.com:WeBankBlockchain/Governance-Authority.git
cd auth-manager
```

这个源码包含：

- 权限治理合约
- 权限治理的java sdk

编译

方式一：如果服务器已安装Gradle

```
gradle build -x test
```

方式二：如果服务器未安装Gradle，则使用gradlew编译。Linux环境：gradlew。Windows环境：gradlew.bat。以Linux环境为例：

```
chmod +x ./gradlew && ./gradlew build -x test
```

编译过后，得到jar包：dist/auth-manager.jar。

集成

在编译好该jar包后，将它引入到一个示例项目中进行调用，以操作权限合约。

新建项目

新建一个java项目

引入auth-manager sdk

前文中，编译auth-manager项目后得到了dist\auth-manager.jar。可以将auth-manager.jar导入到自己的项目中，例如拷贝到libs目录下，然后进行依赖配置，再对自己的项目进行编译。推荐gradle配置如下，

```
repositories {
    maven { url "http://maven.aliyun.com/nexus/content/groups/public/"}
    maven { url "https://oss.sonatype.org/content/repositories/snapshots" }
    maven { url "https://dl.bintray.com/ethereum/maven/" }
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile ('org.fisco-bcos.java-sdk:java-sdk:2.7.0')
    compile fileTree(dir:'libs',include:['*.jar'])

}
```

链连接配置

配置存放在config.toml，参考java sdk配置

使用方式示例

接下来设置黑名单，将一个账户拉黑，使得它无法调用hello函数。

```
package com.webank.authmanager.demo;

import com.webank.authmanager.constant.AuthConstants;
import com.webank.authmanager.contract.AuthManager;
import com.webank.authmanager.factory.AuthManagerFactory;
import com.webank.authmanager.service.AuthByAdminService;
import com.webank.authmanager.utils.HashUtils;
import org.fisco.bcos.sdk.BcosSDK;
import org.fisco.bcos.sdk.client.Client;

/**
 * @author aaronchu
 * @Description
 * @data 2021/01/15
 */
```

(continues on next page)

(续上页)

```
public class SDKDemo {
    public static void main(String[] args) throws Exception{
        BcosSDK bcosSDK = BcosSDK.build("conf/config.toml");
        Client client = bcosSDK.getClient(1);
        AuthManagerFactory authManagerFactory = new AuthManagerFactory(client);
        //Deploy contract
        AuthManager authManager = authManagerFactory.createAdmin();
        //Build facade
        AuthByAdminService authByAdminService = new␣
↪AuthByAdminService(authManager);
        //Create group
        String group = "badGroup";
        authByAdminService.createGroup(group, AuthConstants.ACL_BLACKLIST_MODE);
        String blackAccount = "0x01";
        //Relate function
        String bizContractAddress = "0x2";
        String function = HashUtils.hash("add(uint256,uint256)");//业务函数签名
        authByAdminService.addFunctionToGroup(bizContractAddress, function, group);
        //Verify
        boolean canCall = authByAdminService.canCallFunction(bizContractAddress,␣
↪function, blackAccount);
        System.out.println("Before blocking this account, can the account access␣
↪the function?:"+canCall);

        //Add black account
        authByAdminService.addAccountToGroup(blackAccount, group);

        //Verify
        canCall = authByAdminService.canCallFunction(bizContractAddress, function,␣
↪blackAccount);
        System.out.println("After blocking this account, can the account access␣
↪the function?:"+canCall);

        //Remove black account
        authByAdminService.removeAccountFromGroup(blackAccount, group);

        //Verify
        canCall = authByAdminService.canCallFunction(bizContractAddress, function,␣
↪blackAccount);
        System.out.println("After unblocking, can the account access the function?:
↪"+canCall);
    }
}
```

2.3.5 常见问题

jvm崩溃

现象:

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  Internal Error (sharedRuntime.cpp:834), pid=17781, tid=140031174805248
#  fatal error: exception happened outside interpreter, nmethods and vtable stubs␣
↪at pc 0x00007f5c1d05406f
#
# JRE version: Java(TM) SE Runtime Environment (8.0_45-b14) (build 1.8.0_45-b14)
```

(continues on next page)

(续上页)

```
# Java VM: Java HotSpot(TM) 64-Bit Server VM (25.45-b02 mixed mode linux-amd64␣
↪compressed oops)
# Failed to write core dump. Core dumps have been disabled. To enable core dumping,
↪ try "ulimit -c unlimited" before starting Java again
#
# If you would like to submit a bug report, please visit:
#   http://bugreport.java.com/bugreport/crash.jsp
#

---------------  T H R E A D  ---------------

Current thread (0x00007f5bac160800):  JavaThread "nioEventLoopGroup-3-12" [_thread_
↪in_Java, id=18017, stack(0x00007f5b8c5e8000,0x00007f5b8c6e9000)]

Stack: [0x00007f5b8c5e8000,0x00007f5b8c6e9000],  sp=0x00007f5b8c6e6150,  free␣
↪space=1016k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [libjvm.so+0xaac99a]  VMError::report_and_die()+0x2ba
V  [libjvm.so+0x4f2de9]  report_fatal(char const*, int, char const*)+0x59
V  [libjvm.so+0x9ab5ba]  SharedRuntime::continuation_for_implicit_
↪exception(JavaThread*, unsigned char*,␣
↪SharedRuntime::ImplicitExceptionKind)+0x33a
V  [libjvm.so+0x914f1a]  JVM_handle_linux_signal+0x48a
V  [libjvm.so+0x90b493]  signalHandler(int, siginfo*, void*)+0x43
C  [libpthread.so.0+0xf100]
J 10415 C2 com.sun.crypto.provider.GCTR.doFinal([BII[BI)I (130 bytes) @␣
↪0x00007f5c1e10c096 [0x00007f5c1e10be40+0x256]
```

这是jdk的bug，需要将jdk版本升级到jdk8u51版本。

函数签名是什么

函数签名是函数的标识符，格式为“函数名(参数类型列表)”。例如，有下面这个solidity函数：

```
function add(uint256 a, uint256 b) public pure returns(uint256){}
```

那么它的函数签名为：

```
add(uint256,uint256)
```

CHAPTER 3

私钥管理组件

简介

WeBankBlockchain-Governance-Key组件旨在让用户简便、安全的使用私钥。该组件包含椭圆曲线私钥的生成、使用、加解密保管等功能，覆盖私钥全生命周期，并支持国密标准。WeBankBlockchain-Governance-Key既适合个人用户，也适合企业级用户，对于个人用户，可以方便地生成、使用、加密私钥；对于企业级用户，提供了多层次、多维度、大规模的私钥托管解决方案。

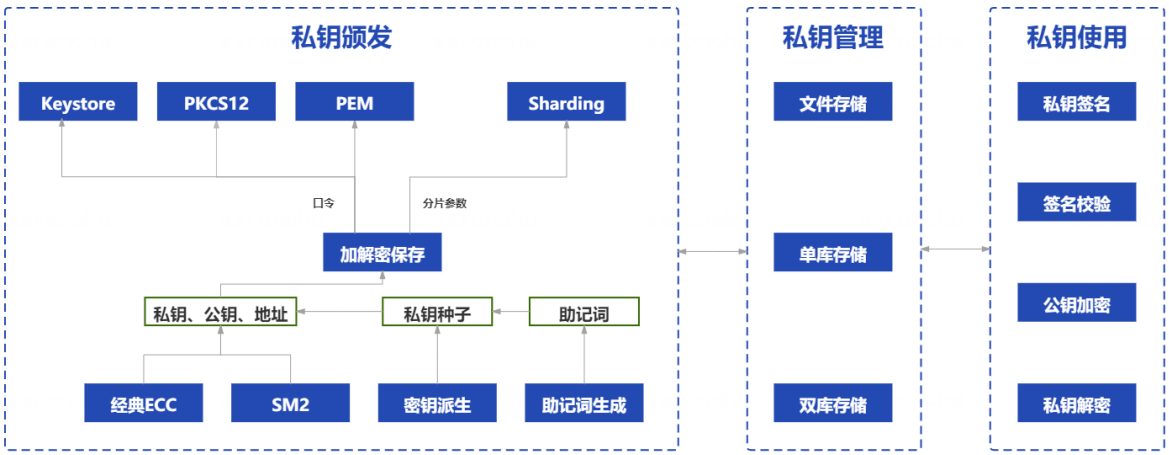
主要特性

- 多种主流密钥生成方式
- 多种主流密钥生成方式
- 支持私钥加密导出
- 支持企业级密钥托管方案
- 支持分片与还原
- 支持可视化操作界面
- 支持通用密码学操作
- 国密支持

3.1 组件介绍

WeBankBlockchain-Governance-Key组件旨在让用户便捷、安全的使用私钥，其功能覆盖私钥颁发、私钥托管、私钥使用等功能，覆盖私钥全生命周期，并支持国密标准。Governance-Key包含key-core和key-mgr两个组件，key-toolkit用于私钥的生成、加密、常规密码学操作，支持多种主流协议。key-mgr用于私钥托管，包含多种托管模式。

3.1.1 关键特性



多种密钥生成方式

提供三种密钥生成方式：随机数、助记词、密钥派生。随机数方式是指椭圆曲线方式，在曲线域内生成一个随机数作为私钥。助记词遵循BIP-32协议，通过助记词和口令可获取私钥，降低了私钥保存的难度。而密钥派生模式下，支持通过父级私钥和一串数据来确定性地生成下级私钥，适用于根据不同场景使用不同私钥的方案。

支持密钥导出

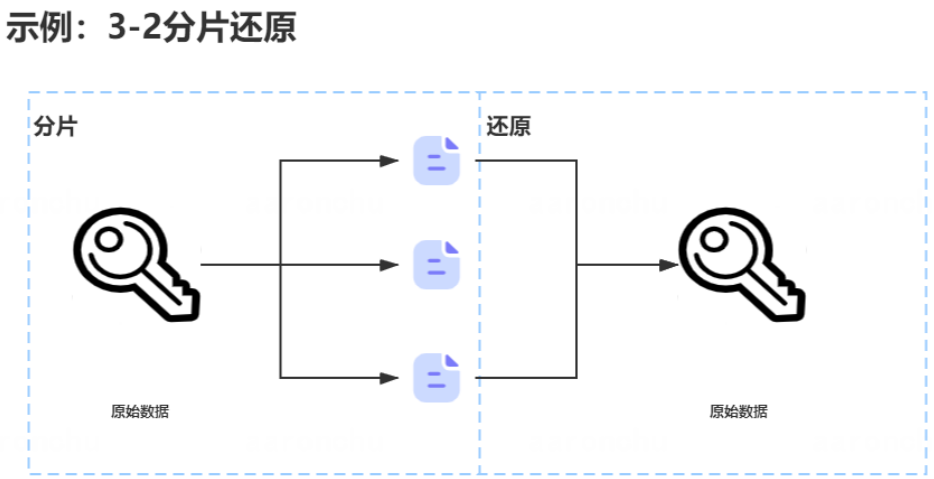
支持用户对私钥进行导出，目前支持PEM、PKCS12、ETH Keystore等标准。PEM格式下，私钥信息被编码到.pem文件中；PKCS12格式下，私钥通过口令加密，按RFC7292规范导出到p12文件中，与openssl等主流工具相兼容；ETH Keystore格式下，私钥通过口令加密，按以太坊标准keystore格式导出到json文件中。

密钥托管方案

支持企业机构来托管用户私钥，适用于B2B2C等业务模式。该模式下，允许将用户的私钥加密并保存，其中加密后的密钥可保存到文件，也可以保存到数据库。此外，加密口令也可以单独保存到另一个数据库，通过双库的方式提高了安全性。

私钥分片、还原

支持将私钥进行门限分片和还原。使用者可以将私钥分解成n个碎片，只有集齐指定数量的碎片，才能恢复出原始私钥。



常规密码学操作

支持常规密码学操作，例如签名、加解密、哈希。

国密支持

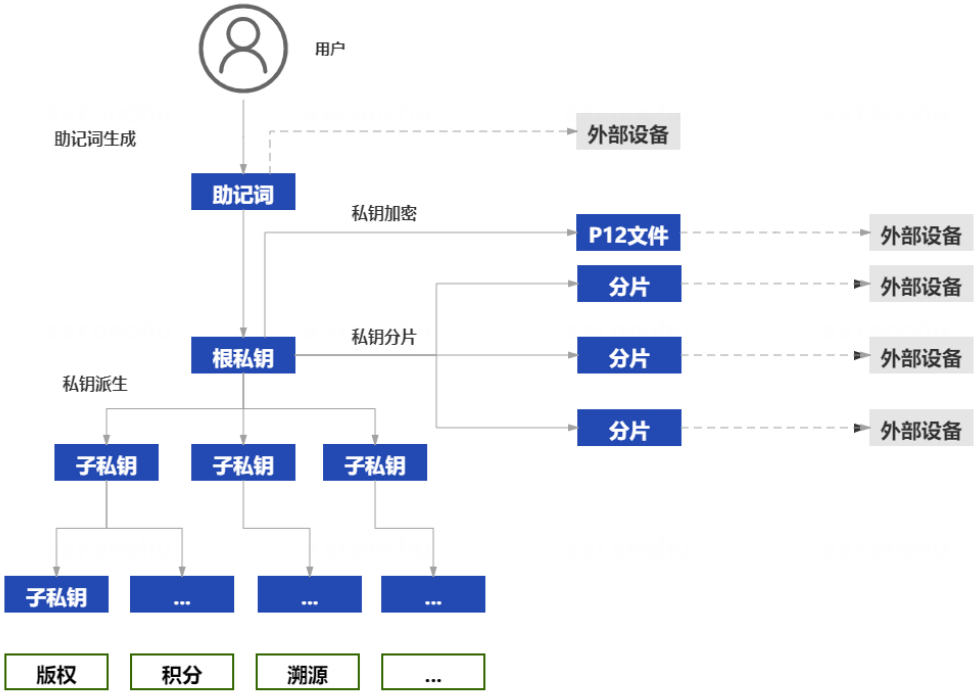
支持国密算法。例如，对于私钥生成、签名、加解密等非对称密码学操作，我们采用标准国密曲线sm2p256v1；对于哈希，则采用SM3标准。

3.1.2 场景示例

个人使用场景示例

某用户想使用私钥来处理不同的业务需求，出于安全性考虑，该用户希望有一个根私钥，然后在不同的场景下使用其派生的子私钥。首先该用户使用助记词方式生成了一个私钥作为根私钥，将该助记词抄写在纸上妥善保管。随后，为了私钥明文在不被黑客攻击，该用户还用自己的口令将其加密为P12格式保存好。最后，该用户还是担心该私钥泄露，便采用门限分片将其分解为多片，置于不同的设备上，如果丢失，可以通过这些设备还原。

经过如此处理后，该用户认为根私钥已得到足够的保护。于是，对于不同的业务场景，该用户派生不同的子私钥来进行签名等操作。



3.2 前置依赖

在使用本组件前，请确认系统环境已安装相关依赖软件，清单如下：

如果您还未安装这些依赖，请参考附录。

3.3 基本术语

使用本组件前，请先确保理解下述基本术语：

3.4 Key-core快速开始

Key-core支持两种使用模式：可视化模式和sdk模式。在可视化模式下，会启动一个本地的网站，上面包含了私钥操作的核心功能。在sdk模式下，用户可将sdk引入到自己的项目中，并根据sdk提供的私钥功能来实现自己的业务代码。

3.4.1 源码下载

通过git 下载源码.

```
cd ~
git clone https://github.com/WeBankBlockchain/Governance-Key.git
cd Governance-Key
```

3.4.2 使用可视化界面

进入目录：

```
cd key-core-web
```

编译代码：

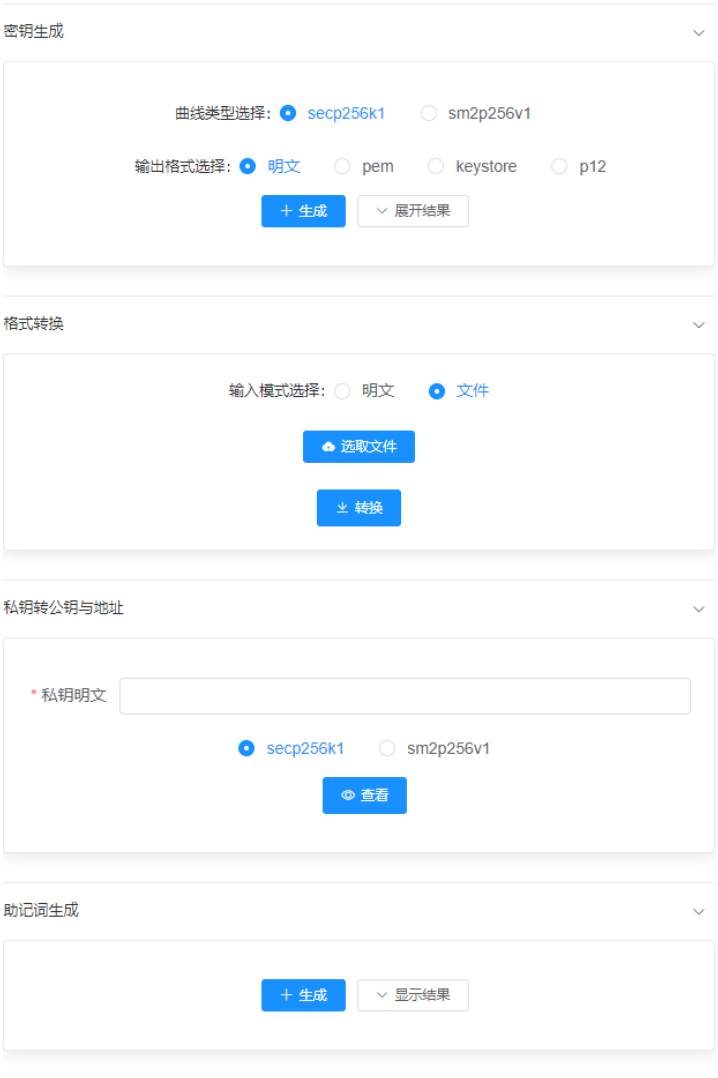
```
gradle bootJar
```

编译后，会生成dist目录，包含key-core-web.jar包。

启动可视化界面：

```
cd dist
java -jar key-core-web.jar
```

启动成功后，会自动弹出浏览器页面。如果未自动弹出，也可以访问localhost:8001端口。网页效果如下：



3.4.3 使用sdk

源码编译

进入目录：

```
cd ~/Governance-Key/key-core
```

编译代码：

```
gradle build -x test
```

完成编译之后，在根目录下会生成dist文件夹，文件夹中包含key-core.jar。

引入jar包

将dist目录中的key-core.jar包导入到自己的项目中，例如放到libs目录下。然后进行依赖配置，以gradle为例，依赖配置如下：

```
repositories {
    maven {
        url "http://maven.aliyun.com/nexus/content/groups/public/"
    }
    maven { url "https://oss.sonatype.org/service/local/staging/deploy/maven2"}
    maven { url "https://oss.sonatype.org/content/repositories/snapshots" }
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile 'com.webank:webankblockchain-crypto-core:1.0.0-SNAPSHOT'

    compile "org.apache.commons:commons-lang3:3.6"
    compile group: 'org.bouncycastle', name: 'bcprov-jdk15on', version: '1.60'
    compile group: 'org.bouncycastle', name: 'bcpkix-jdk15on', version: '1.60'
    compile 'org.web3j:core:3.4.0'
    compile ('org.fisco-bcos.java-sdk:java-sdk:2.7.0')
    compile "commons-io:commons-io:2.6"
    compile 'com.lambdaworks:scrypt:1.4.0'
    compile 'commons-codec:commons-codec:1.9'
    testCompile group: 'junit', name: 'junit', version: '4.12'
    compile fileTree(dir:'libs',include:['*.jar'])
}
```

常用场景示例

随机数方式生成私钥

```
import com.webank.keygen.model.PkeyInfo;
import com.webank.keygen.service.PkeyByRandomService;
import com.webank.keygen.service.PkeySM2ByRandomService;
import com.webank.keygen.utils.KeyPresenter;

public class KeyGenerationDemo {

    public static void main(String[] args) throws Exception{
        //生成非国密私钥
        PkeyByRandomService eccService = new PkeyByRandomService();
        PkeyInfo eccKey = eccService.generatePrivateKey();
        System.out.println("private key:"+ KeyPresenter.asString(eccKey.
↪getPrivateKey()));
        System.out.println("public key:" + KeyPresenter.asString(eccKey.
↪getPublicKey().getPublicKey()));
        System.out.println("address:" +eccKey.getAddress());
```

(continues on next page)

(续上页)

```
        //生成国密私钥
        PkeySM2ByRandomService gmService = new PkeySM2ByRandomService();
        PkeyInfo gmPkey = gmService.generatePrivateKey();
        System.out.println("private key:"+ KeyPresenter.asString(gmPkey.
↪getPrivateKey()));
        System.out.println("public key:" + KeyPresenter.asString(gmPkey.
↪getPublicKey().getPublicKey()));
        System.out.println("address:" +gmPkey.getAddress());
    }
}
```

私钥转换为公钥及地址

当持有一个私钥明文时，可将其转换为公钥和地址。

```
import com.webank.keygen.enums.EccTypeEnums;
import com.webank.keygen.model.PkeyInfo;
import com.webank.keygen.utils.KeyPresenter;

public class KeyToPubAndAddress {

    public static void main(String[] args) throws Exception{
        String pkeyStr =
↪"0xed1d9dc98c8496b9837cb8c46a2302b9d479aab08f536dc0785115c11990d7f3";
        byte[] privateKey = KeyPresenter.asBytes( pkeyStr);
        PkeyInfo pkeyInfo
                = PkeyInfo.builder()
                .privateKey(privateKey)
                .eccName(EccTypeEnums.SECP256K1.getEccName())
                .build();

        System.out.println("public key :" + KeyPresenter.asString(pkeyInfo.
↪getPublicKey().getPublicKey()));
        System.out.println("address :" + pkeyInfo.getAddress());
    }
}
```

助记词使用

用户可以通过助记词和口令来恢复私钥明文。下述示例中包含了助记词生成、助记词恢复私钥的代码:

```
import com.webank.keygen.enums.EccTypeEnums;
import com.webank.keygen.model.PkeyInfo;
import com.webank.keygen.service.PkeyByMnemonicService;
import com.webank.keygen.utils.KeyPresenter;

public class MnemonicUsage {

    public static void main(String[] args) throws Exception{
        //助记词生成
        PkeyByMnemonicService mnemonicService = new PkeyByMnemonicService();
        String mnemonic = mnemonicService.createMnemonic();
        System.out.println("Mnemonic:"+mnemonic);
        //生成私钥
        PkeyInfo pkey = mnemonicService.generatePrivateKeyByMnemonic(mnemonic,
↪"passphrase", EccTypeEnums.SECP256K1);
```

(continues on next page)

(续上页)

```
        System.out.println("ECC Private Key:"+ KeyPresenter.asString(pkey.
↪getPrivateKey()));
        System.out.println("ECC Chaincode:"+ KeyPresenter.asString(pkey.
↪getChainCode()));
        System.out.println("ECC Address:"+ pkey.getAddress());
    }
}
```

密钥派生

当用户要在不同场景，甚至不同区块链间使用私钥时，若每次都生成一个私钥，那么随着场景的增多，要保管的私钥势必也会增多，增加了私钥丢失、泄露的可能。所以，用户可以精心保管一个（或多个）根私钥，对于每个特殊场景，通过派生的方式得到子私钥。

就派生方式而言，支持私钥派生私钥，公钥派生公钥。私钥派生的私钥，与私钥对应公钥所派生的公钥是匹配的。此外，BIP-44规范提供了一组按场景组织的规范化派生方式。

```
import com.webank.keygen.hd.bip32.ExtendedPrivateKey;
import com.webank.keygen.hd.bip32.ExtendedPublicKey;
import com.webank.keygen.hd.bip44.path.Purpose44Path;
import com.webank.keygen.model.PkeyInfo;
import com.webank.keygen.service.PkeyByRandomService;
import com.webank.keygen.service.PkeyHDDeriveService;
import com.webank.keygen.utils.KeyPresenter;

public class KeyDerive {

    public static void main(String[] args) throws Exception{
        //获得一个根私钥
        PkeyByRandomService generateService = new PkeyByRandomService();
        PkeyInfo pkeyInfo = generateService.generatePrivateKey();

        PkeyHDDeriveService deriveService = new PkeyHDDeriveService();
        ExtendedPrivateKey rootKey = deriveService.
↪buildExtendedPrivateKey(pkeyInfo);
        //私钥派生子私钥
        ExtendedPrivateKey subPrivKey = rootKey.deriveChild(2);
        System.out.println("child no.2: "+ KeyPresenter.asString(subPrivKey.
↪getPkeyInfo().getPrivateKey()));
        //子私钥转子公钥
        ExtendedPublicKey subPubKey= subPrivKey.neuter();
        System.out.println("pubkey for child no.2: "+ KeyPresenter.
↪asString(subPubKey.getPubInfo().getPublicKey()));
        //BIP-44派生
        Purpose44Path derivePath = deriveService.getPurpose44PathBuilder().m()
                .purpose44().sceneType(2)
                .account(3).change(4).addressIndex(5).build();
        ExtendedPrivateKey derived = derivePath.deriveKey(rootKey);
        System.out.println("derived for bip 44 "+ KeyPresenter.asString(derived.
↪getPkeyInfo().getPrivateKey()));
    }
}
```

私钥加密导出到目录

```
import com.webank.keygen.enums.EccTypeEnums;
import com.webank.keygen.model.PkeyInfo;
```

(continues on next page)

(续上页)

```
import com.webank.keygen.service.PkeyEncryptService;
import org.web3j.utils.Numeric;

public class KeyExport {
    public static void main(String [] args) throws Exception{
        //生成一个私钥
        PkeyInfo pkeyInfo
                = PkeyInfo.builder()
                .privateKey(Numeric.hexStringToByteArray(
↪"252ffefe4e3856eb84a4fba5f07fc2066d3043a763cb74ed16ff093ac79b52d6"))
                .eccName(EccTypeEnums.SECP256K1.getEccName())
                .build();
        byte[] privateKeyBytes = pkeyInfo.getPrivateKey();
        EccTypeEnums eccTypeEnums = EccTypeEnums.getEccByName(pkeyInfo.
↪getEccName());
        //pem导出到
        PkeyEncryptService encryptService = new PkeyEncryptService();
        encryptService.encryptPEMFormat(privateKeyBytes, eccTypeEnums, System.
↪getProperty("user.dir"));
        //keystores导出
        String password = "123456";
        encryptService.encryptKeyStoreFormat(password, privateKeyBytes,
↪eccTypeEnums, System.getProperty("user.dir"));
        //p12导出
        encryptService.encryptP12Format(password, privateKeyBytes,eccTypeEnums,␣
↪System.getProperty("user.dir"));
    }
}
```

数据分片与还原

支持将数据分片，分片的数据可还原为原始数据。分片模式可指定为(n, t)，其中n表示数据要分为多少片，t表示只需要多少片即可还原出原始数据，例如分片模式若为（3，2），那么原始数据可分解为3片，当且仅当持有其中2片的时候，即可还原出原始数据。

```
import com.webank.keygen.model.PkeyInfo;
import com.webank.keygen.service.PkeyByRandomService;
import com.webank.keygen.service.PkeyShardingService;
import com.webank.keygen.utils.KeyPresenter;
import java.util.ArrayList;
import java.util.List;

public class Shard {
    public static void main(String[] args) throws Exception{
        //生成一个私钥
        PkeyByRandomService generateService = new PkeyByRandomService();
        PkeyInfo pkeyInfo = generateService.generatePrivateKey();
        System.out.println("Before sharding "+KeyPresenter.asString(pkeyInfo.
↪getPrivateKey()));
        //开始分片，分解为5片，凑齐任意3片才能还原
        PkeyShardingService shardingService
                = new PkeyShardingService();
        List<String> shards = shardingService.shardingPKey(pkeyInfo.
↪getPrivateKey(), 5, 3);
        //还原
        List<String> recoveredShards = new ArrayList<>();
        recoveredShards.add(shards.get(0));
        recoveredShards.add(shards.get(2));
        recoveredShards.add(shards.get(3));
```

(continues on next page)

(续上页)

```
        byte[] recovered = shardingService.recoverPKey(recoveredShards);
        System.out.println("After recovered "+ KeyPresenter.asString(recovered));
    }
}
```

密码学操作

下述例子包含了签名、验签、数据加密、数据解密。

```
import com.webank.keysign.service.ECCEncryptService;
import com.webank.keysign.service.ECCSignService;
import com.webank.keysign.service.SM2EncryptService;
import com.webank.keysign.service.SM2SignService;

public class Crypto {
    public static void main(String[] args){
        //Case 1: Ecc(secp256k1) sign and verify
        String eccMsg = "HelloEccSign";
        String eccPrivateKey =
↪"28018238ac7eec853401dfc3f31133330e78ac27a2f53481270083abb1a126f9";
        String eccPublicKey =
↪"0460fc2bce5795ee2ac34d1f584f603b4e2920a95d8d3db5f5c664244a99fd76405831ffaf932f64eae3ec67bc8ff7
↪";

        ECCSignService eccSignService = new ECCSignService();
        String eccSignature = eccSignService.sign(eccMsg, eccPrivateKey);
        System.out.println("ecc signature:"+eccSignature);

        boolean eccVerifyResult = eccSignService.verify(eccMsg, eccSignature,␣
↪eccPublicKey);
        System.out.println("ecc verify result:"+eccVerifyResult);

        //Case 2: Ecc(secp256k1) encryption and decryption
        ECCEncryptService eccEncryptService = new ECCEncryptService();
        String eccCipherText = eccEncryptService.encrypt(eccMsg, eccPublicKey);
        System.out.println("ecc encryption cipher:"+eccCipherText);

        String eccPlainText = eccEncryptService.decrypt(eccCipherText,␣
↪eccPrivateKey);
        System.out.println("ecc decryption result:"+eccPlainText);

        //Case3: Gm(sm2p256v1) sign and verify
        String gmMsg = "HelloGM";
        String gmPrivateKey =
↪"73c8a8054b5e42b0d089e24f16c665bc82a132082d258c5efb54c49a3b7273f9";
        String gmPublicKey =
↪"0451c895673d372267a565c4a7711102108138132b21f22ed556df08fb4c8cfdcaf17dcb605f8a6394f8684aa1916d
↪";

        SM2SignService sm2SignService = new SM2SignService();
        String gmSignature = sm2SignService.sign(gmMsg, gmPrivateKey);
        System.out.println("gm signature:"+gmSignature);

        boolean gmVerifyResult = sm2SignService.verify(gmMsg, gmSignature,␣
↪gmPublicKey);
        System.out.println("gm verify result:"+gmVerifyResult);

        //Case4: Gm(sm2p256v1) encryption and decryption
        SM2EncryptService gmEncryptService = new SM2EncryptService();
```

(continues on next page)

(续上页)

```
        String gmCipherText = gmEncryptService.encrypt(gmMsg, gmPublicKey);
        System.out.println("gm encryption cipher:"+gmCipherText);

        String gmPlainText = gmEncryptService.decrypt(gmCipherText, gmPrivateKey);
        System.out.println("gm decryption result:"+gmPlainText);
    }
}
```

SDK核心接口

下面包含核心接口：

- PkeyByRandomService:随机数方式私钥生成（非国密）
- PkeySM2ByRandomService：随机数方式私钥生成（国密）
- PkeyByMnemonicService：助记词生成；助记词恢复私钥
- PkeyHDDeriveService：密钥派生
- PkeyEncryptService：私钥按不同格式导出和导入
- PkeyShardingService：分片与还原

3.5 Key-mgr快速开始

Key-mgr用于托管密钥，适合企业级用户使用。在文件存储模式下，加密后的密钥会被存入到本地。在数据库存储模式下，加密后的密钥会被存储到数据库中，这里分为单库模式和双库模式。在单库模式下，仅保存加密后的私钥，加密口令由C端用户自行保存，适合C端用户对托管机构半信任的场景；在双库模式下，会保存加密后的密钥和加密口令，二者分别存储在不同的数据库中，适用于C端用户对托管机构完全信任的场景。

注解： 使用前请先阅读[使用必读](depend.md)，确保已理解相关概念等。

3.5.1 编译源码

进入目录：

```
cd ~/Governance-Key/key-mgr
```

编译：

```
gradle build -x test
```

3.5.2 引入jar包

完成编译之后，在根目录下会生成dist文件夹，文件夹中包含key-mgr.jar。将其导入到自己的项目中，例如放到libs目录下。然后进行依赖配置，以gradle为例，依赖配置如下：

```
repositories {
    maven {
        url "http://maven.aliyun.com/nexus/content/groups/public/"
    }
    maven { url "https://oss.sonatype.org/service/local/staging/deploy/maven2"}
```

(continues on next page)

(续上页)

```
    maven { url "https://oss.sonatype.org/content/repositories/snapshots" }
    mavenLocal()
    mavenCentral()
}

dependencies {

    compile 'org.springframework.boot:spring-boot-starter'
    compile 'org.springframework.boot:spring-boot-starter-data-jpa'

    testCompile('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
        //exclude group: 'junit', module: 'junit'
    }
    compile 'org.springframework.boot:spring-boot-starter-jta-atomikos'
    compile ('org.projectlombok:lombok:1.18.8')
    compile ('org.projectlombok:lombok:1.18.8')
    annotationProcessor 'org.projectlombok:lombok:1.18.8'
    compile "org.apache.commons:commons-lang3:3.6"
    compile "commons-io:commons-io:2.6"

    compile "com.fasterxml.jackson.core:jackson-core:2.9.6"
    compile "com.fasterxml.jackson.core:jackson-databind:2.9.6"
    compile "com.fasterxml.jackson.core:jackson-annotations:2.9.6"

    compile 'com.lhalcyon:bip32:1.0.0'
    //compile 'io.github.novacrypto:BIP44:0.0.3'

    compile group: 'org.bouncycastle', name: 'bcprov-jdk15on', version: '1.60'
    compile group: 'org.bouncycastle', name: 'bcpkix-jdk15on', version: '1.60'
    compile 'org.web3j:core:3.4.0'
    compile 'com.lambdaworks:scrypt:1.4.0'
    compile 'commons-codec:commons-codec:1.9'
    compile ('org.fisco-bcos.java-sdk:java-sdk:2.7.0')
    compile 'mysql:mysql-connector-java'
    compile 'com.webank:webankblockchain-crypto-core:1.0.0-SNAPSHOT'
    compile fileTree(dir:'libs',include:['*.jar'])
}
```

3.5.3 配置

如果仅出于体验的目的，无需做任何配置，托管后的加密密钥会被保存到~/.pkeymgr。如果需要更高级的配置，请参考下面的模板，配置application.properties。

```
# true-存储密码（全信任模式），false-不存储密码（半信任模式）
system.storePwd=true
# 托管方式：db-数据库托管方式，file-文件托管方式
system.mgrStyle=db

### 该配置仅为system.mgrType=file时的配置
#### system.dataFileDir=~/myKeys

## 加密格式，支持p12或keystore
system.keyEncType=p12
## 可以用secp256k1 or sm2p256v1。后者为国密曲线。
system.eccType=sm2p256v1

## 加密后的私钥存储url
spring.datasource.encryptkeydata.url=jdbc:mysql://[ip]:[port]/pkey_mgr?
↪autoReconnect=true&characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2b8
```

(continues on next page)

(续上页)

```
spring.datasource.encryptkeydata.username=
spring.datasource.encryptkeydata.password=

## 若存储密码
spring.datasource.keypwd.url=jdbc:mysql://[ip]:[port]/pkey_mgr_pwd?
↪autoReconnect=true&characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2b8
spring.datasource.keypwd.username=
spring.datasource.keypwd.password=

## spring jpa config
spring.jpa.properties.hibernate.hbm2ddl.auto=update
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect
```

3.5.4 建表

如果在上述配置中指定了**spring.jpa.properties.hibernate.hbm2ddl.auto=update**，则jpa会帮助用户自动建立数据表。

如果不希望自动建立数据表，请先关闭jpa建表开关：

```
spring.jpa.properties.hibernate.hbm2ddl.auto=validate
```

然后按下面方式手动建表。

1） 在encryptkeydata数据源运行下述建表语句：

```
CREATE TABLE `encrypt_keys_info` (
   `pk_id` bigint(20) NOT NULL AUTO_INCREMENT,
   `encrypt_key` longtext,
   `key_address` varchar(255) DEFAULT NULL,
   `key_name` varchar(255) DEFAULT NULL,
   `parent_address` varchar(255) DEFAULT NULL,
   `user_id` varchar(255) DEFAULT NULL,
   `ecc_type` varchar(255) DEFAULT NULL,
   `enc_type` varchar(255) DEFAULT NULL,
   PRIMARY KEY (`pk_id`),
   KEY `user_id` (`user_id`),
   KEY `key_address` (`key_address`)
 ) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8
```

2） 请在keypwd数据源运行下述建表语句：

```
CREATE TABLE `key_pwds_info` (
   `pk_id` bigint(20) NOT NULL AUTO_INCREMENT,
   `key_address` varchar(255) DEFAULT NULL,
   `key_pwd` varchar(255) DEFAULT NULL,
   `user_id` varchar(255) DEFAULT NULL,
   PRIMARY KEY (`pk_id`),
   KEY `user_id` (`user_id`),
   KEY `key_address` (`key_address`)
 ) ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=utf8
```

3.5.5 接口使用

KeysManagerService类是整个pkey-mgr模块的入口，覆盖私钥管理的全生命周期，包含如下功能：

建议您通过Spring自动注入KeysManagerService服务，示例如下：

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class Example {

    @Autowired
    private KeysManagerService keysManagerService;

}
```

createPrivateKey

该方法用于随机生成私钥，并进行托管存储。

```
@Test
public void demo() throws Exception {
    String userId = "1000";
    String password = "123456";
    String keyName = "MyKey";
    keysManagerService.createPrivateKey(userId, password, keyName);
}
```

执行过后，可在encryptkeydata数据源的encrypt_keys_info表查看存储的私钥；如果配置了双库（system.storePwd=true），可通过keypwd数据源的key_pwds_info表查看密码存储结果。

importPrivateKeyFile

该方法用于从p12或keystore文件中提取私钥，并进行托管存储。按哪种格式提取取决于applicaiton.properties中配置的是p12还是keystore，故要注意确保文件格式和配置相符。

```
@Test
public void demo() throws Exception {
    String userId = "1000";
    String password = "123456";
    String path = "~/keys/enryptFile.p12";
    keysManagerService.importPrivateKeyFile(userId, password, path);
}
```

执行过后，可在encryptkeydata数据源的encrypt_keys_info表查看存储的私钥；如果配置了双库（system.storePwd=true），可通过keypwd数据源的key_pwds_info表查看密码存储结果。

importPrivateKey

该方法用于直接导入私钥原文，并对其托管存储。

```
@Test
public void demo() throws Exception {
    String userId = "1000";
    String password = "123456";
    String privateKey =
↪"0xeec37c95fb78c49132513882bb293fd6ecdef194e85efcc654af0d9c291995ec";
    String keyName = "MyKey";
    keysManagerService.importPrivateKey(userId, password, privateKey, keyName);
}
```

执行过后，可在encryptkeydata数据源的encrypt_keys_info表查看存储的私钥；如果配置了双库（system.storePwd=true），可通过keypwd数据源的key_pwds_info表查看密码存储结果。

createPrivateKeyByParent

该方法通过父私钥原文和一个chaincode来确定性地派生子私钥。派生过后的子私钥也会被导入托管。

```
@Test
public void demo() throws Exception {
    String userId = "1200";
    String parentKey =
↪"eec37c95fb78c49132513882bb293fd6ecdef194e85efcc654af0d9c291995ec";
    //首先导入父私钥
    keysManagerService.importPrivateKey(userId, "123456",parentKey , "parentKey");
    //派生子私钥并导入
    String chaincode1 = "123";
    String password1 = "pwd1";
    keysManagerService.createPrivateKeyByParent(userId, parentKey,chaincode1,␣
↪password1);
    String chaincode2 = "xyz";
    String password2 = "pwd2";
    keysManagerService.createPrivateKeyByParent(userId, parentKey,chaincode2,␣
↪password2);
}
```

执行过后，可在encryptkeydata数据源的encrypt_keys_info表查看存储的子私钥；如果配置了双库（system.storePwd=true），可通过keypwd数据源的key_pwds_info表查看密码存储结果。

queryChildKeys

该方法用于查询一个父私钥所关联的下级子私钥。

```
@Test
public void demo() throws Exception {
    String userId = "1200";
    String parentAddress = "9349e27ae2202202bd0487d4aa08705a14c52710";
    //Query child keys
    List<EncryptKeyInfo> childs = keysManagerService.queryChildKeys(userId,␣
↪parentAddress);
    log.info("Child keys {}", JacksonUtils.toJson(childs));
}
```

exportPrivateKeyFile

该方法根据用户id、私钥地址从库中读取私钥，以所配置的格式导出到目标目录。

```
@Test
public void demo() throws Exception {
    String userId = "1200";
    String keyAddress = "9349e27ae2202202bd0487d4aa08705a14c52710";
    String destinationDirectory = "C:\\Users\\unnamed\\Desktop\\keys";
    keysManagerService.exportPrivateKeyFile(userId, keyAddress,␣
↪destinationDirectory);
}
```

执行过后，将在destinationDirectory指定的目录内，得到一个加密的p12或keystore文件（取决于system.keyEncType配置）

decryptPrivateKey

该方法将一个密文解密为原文。密文格式取决于system.keyEncType配置。下面示例中，是解密p12的示例，要求system.keyEncType=p12:

```
@Test
public void testDecryptP12() throws Exception{
    String enc =
↪"0x3082070c020103308206c506092a864886f70d010701a08206b6048206b2308206ae3081e306092a864886f70d01
↪";
    String pk = keysManagerService.decryptPrivateKey("123456", enc);
    log.info("recovered {}", pk);
}
```

下面示例中，是解密keystore的示例，要求system.keyEncType=keystore:

```
 @Test
public void testDecryptKeystore() throws Exception{
    String enc = "{\"address\":\"0x68e1d0ad5ebc77feeaaac712e55cd274a2b1d33a\",\"id\
↪":\"c5f82b87-b25a-42b7-baa1-51871ffae0eb\",\"version\":3,\"crypto\":{\"cipher\":\
↪"aes-128-ctr\",\"ciphertext\":\
↪"0ca68b721be5d1556cbd9be2b3b043e7e1ddf1458d89a43339a24b71d2f2bb3c\",\
↪"cipherparams\":{\"iv\":\"7fa497df863424072f7964d480095bfd\"},\"kdf\":\"scrypt\",
↪\"kdfparams\":{\"dklen\":32,\"n\":262144,\"p\":1,\"r\":8,\"salt\":\
↪"6e039e411821bde52b17481a8759ff66548ba97f4d2e094262d06f93a9f6844c\"},\"mac\":\
↪"04e8d244a4ed9c5206952ba17cb9a49560a2ce141c52a60dac15fbca1f3544db\"}}";
    String pk = keyManagerService.decryptPrivateKey("password", enc);
    log.info("recovered {}", pk);
}
```

getEncryptPrivateKeyList

该方法读取某一用户的所有私钥密文。

```
@Test
public void demo() throws Exception {
    String userId = "1200";
    List<EncryptKeyInfo> keys = keysManagerService.
↪getEncryptPrivateKeyList(userId);
    log.info("keyStrings {}", JacksonUtils.toJson(keys));
}
```

getEncryptPrivateKeyByUserIdAndAddress

该方法读取某一用户下某地址对应的私钥密文。

```
@Test
public void demo() throws Exception {
    String userId = "1000";
    String address = "e4cbf2581d2fc8541613079440cfbdbf2595b127";
    EncryptKeyInfo key = keysManagerService.
↪getEncryptPrivateKeyByUserIdAndAddress(userId, address);
    log.info("keyStrings {}", JacksonUtils.toJson(key));
}
```

updateKeyName

updateKeyName更新库中的私钥名称。

```
@Test
public void demo() throws Exception {
    String userId = "1200";
```

(continues on next page)

(续上页)

```
    String address = "9349e27ae2202202bd0487d4aa08705a14c52710";
    String newKeyName = "newKeyName";
    keysManagerService.updateKeyName(userId, address, newKeyName);
}
```

执行过后，可在encryptkeydata数据源的encrypt_keys_info表的key_name字段查看新的私钥名。

updateKeyPassword

更新私钥密码。

```
@Test
public void demo() throws Exception {
    String userId = "1200";
    String address = "9349e27ae2202202bd0487d4aa08705a14c52710";
    String oldPwd = "123456";
    String newPwd = "654321";
    keysManagerService.updateKeyPassword(userId, address, oldPwd, newPwd);
}
```

执行过后，可在encryptkeydata数据源的encrypt_keys_info表查看修改过的密文；如果配置了双库（system.storePwd=true），可通过keypwd数据源的key_pwds_info表的key_pwd字段看到新的密码。

deleteUserKey示例

根据用户Id和私钥地址来删除私钥。

```
@Test
public void demo() throws Exception {
    String userId = "1200";
    String keyAddress = "9349e27ae2202202bd0487d4aa08705a14c52710";
    keysManagerService.deleteUserKey(userId, keyAddress);
}
```

执行过后，可在encryptkeydata数据源的encrypt_keys_info表查看到私钥已删除；如果配置了双库（system.storePwd=true），可通过keypwd数据源的key_pwds_info表看到密码已删除。

3.6 Java doc

key-core和key-mgr的java doc

3.7 常见问题

CHAPTER 4

证书管理组件

简介

WeBankBlockchain-Governance-Cert提供了证书生命周期管理的解决方案，规范证书签发流程，支持证书托管，支持多种签名算法，方便个人或企业使用。

主要特性

- 支持多种密钥和签名算法
- 支持证书托管
- 支持多级证书签发
- 支持证书重置

4.1 基本概念

4.1.1 私钥和公钥

在非对称加密领域，对数据的加解密、签名都依赖于密钥对。在密钥对中，公开的密钥叫公钥，只有自己知道的叫私钥。 非对称加密有许多体系，最著名的是RSA、DH、ECDSA。其中ECDSA是区块链领域采纳的密钥体系，也称为椭圆曲线体系，该体系中，私钥是一个某范围内的整数，公钥则是曲线上的一个点。

4.1.2 曲线

在椭圆曲线体系中，密钥生成依赖于曲线参数，一条曲线中合法的密钥对，在另一条曲线的环境中，就是非法密钥。最经典的曲线之一就是secp256k1，为多种主流区块链采纳；另一种曲线是国密曲线sm2p256v1，该曲线满足我国的密码学标准。

4.1.3 证书

数字证书也称公开密钥证书，是指用于电子信息活动中电子文件行为主体的验证和证明，并可实现电子文件保密性和完整性的电子数据。数字证书是一个经证书认证中心（Certification Authority,简称CA）发行的文件。

4.1.4 数字签名

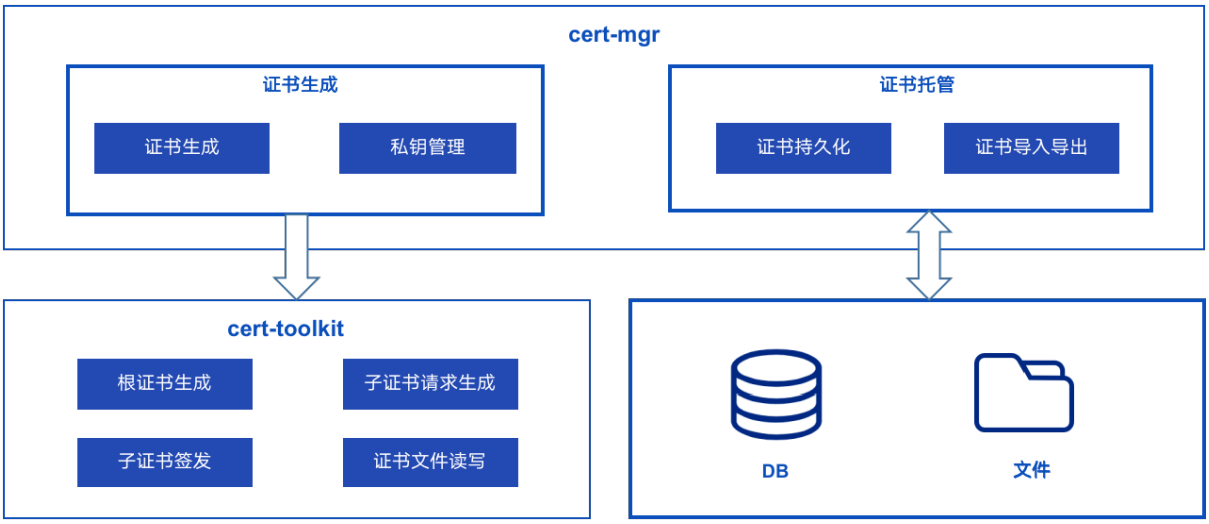
数字签名(Digital Signature)（又称公钥数字签名、电子签章）是一种类似写在纸上的普通的物理签名，但是使用了公钥加密领域的技术实现，用于鉴别数字信息的方法。一套数字签名通常定义两种互补的运算，一个用于签名，另一个用于验证。

4.2 组件介绍

WeBankBlockchain-Governance-Cert提供了证书生命周期管理的解决方案，规范证书签发流程，支持证书托管，支持多种签名算法，方便个人或企业使用。

4.2.1 设计概要

WeBankBlockchain-Governance-Cert包含两个模块，cert-toolkit和cert-mgr，cert-toolkit作为证书生成工具，可作为独立工具包使用，cert-mgr基于cert-toolkit工具包，提供了证书的托管能力，并支持证书的生命周期管理，对签发的流程统一规范。



4.2.2 关键特性

支持多种密钥和签名算法

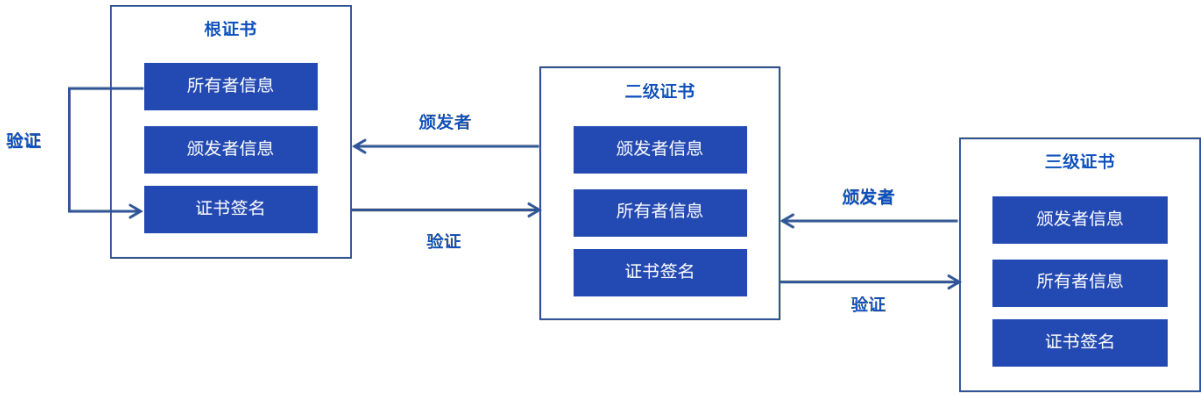
支持RSA，EC等密钥算法，支持SM2国密算法 支持SHA256WITHRSA、SHA256WITHECDSA、SM3WITHSM2等签名算法

支持证书托管

证书和子证书请求会持久化在数据库中，同时证书相关的私钥也会保存，对证书信息统一管理，可对证书进行多维度查询，并支持证书的导出

支持多级证书签发

证书可进行多级签发，可选择上级证书并请求签发，生成证书链，使用方便，操作便捷



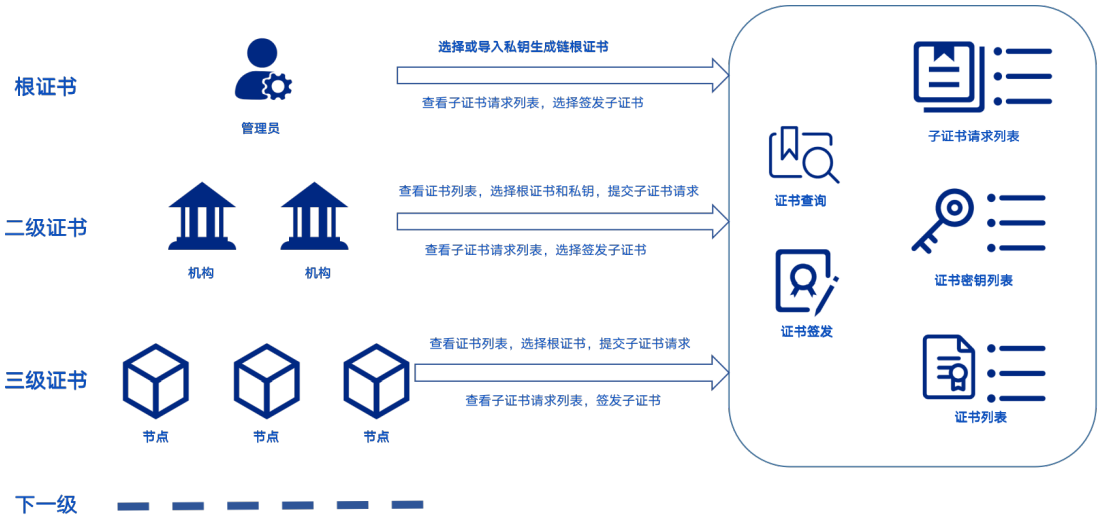
支持证书重置

支持对证书进行重置，重置信息包括有效时间、证书用途等

4.2.3 场景示例

链上节点准入证书管理

链上节点证书的签发统一由WeBankBlockchain-Governance-Cert来完成，Governance-Cert可以集成或者独立部署，并由权威机构来管理服务。在链初始化时可由部署者调用接口完成根证书的生成，新增机构或节点可以通过Governance-Cert提供的查询接口，来查询根证书，并提交子证书请求，根证书管理者可从通过查询请求列表，来获取准入请求，并选择签发子证书，子证书签发完成，下一级证书采取同样逻辑处理 通过WeBankBlockchain-Governance-Cert对于证书的管理，可以规范流程，提升效率，并保证证书安全



证书工具包使用

WeBankBlockchain-Governance-Cert中cert-toolkit可作为独立JAVA工具包在项目中引用，代替命令行完成证书的生成和签发。企业或个人项目可集成WeBankBlockchain-Governance-Cert作为证书签发工具包

4.3 快速开始

注解： 本章介绍证书组件的使用方式。

4.3.1 cert-toolkit使用

功能介绍

cert-toolkit用于证书生成。支持轻量级jar包接入。
支持如下方式：

- 根证书生成
- 证书请求生成
- 子证书生成
- 证书文件的读写

前置依赖

在使用本组件前，请确认系统环境已安装相关依赖软件，清单如下：
如果您还未安装这些依赖，请参考附录。

部署说明

目前支持从源码进行部署。

获取源码

通过git下载源码：

```
git clone https://github.com/WeBankBlockchain/Governance-Cert.git
```

注解：

- 如果因为网络问题导致长时间无法下载，请尝试：git clone https://gitee.com/WeBankBlockchain/Governance-Cert.git

进入目录：

```
cd Governance-Cert
cd cert-toolkit
```

编译源码

方式一：如果服务器已安装Gradle

```
gradle build -x test
```

方式二：如果服务器未安装Gradle，使用gradlew编译

```
chmod +x ./gradlew && ./gradlew build -x test
```

导入jar包

cert-toolkit编译之后在cert-toolkit目录下会生成dist文件夹，文件夹中包含cert-toolkit.jar。可以将cert-toolkit.jar导入到自己的项目中，例如libs目录下。然后进行依赖配置。gradle依赖配置如下，然后再对自己的项目进行编译。

```
repositories {
        maven {
                url "http://maven.aliyun.com/nexus/content/groups/public/"
        }
        maven { url "https://oss.sonatype.org/content/repositories/snapshots" }
        maven { url "https://dl.bintray.com/ethereum/maven/" }
        mavenLocal()
        jcenter()
}

dependencies {
        testCompile 'junit:junit:4.12'
        compile 'org.slf4j:slf4j-api:1.7.30'
        compile ('org.projectlombok:lombok:1.18.6')
        annotationProcessor ('org.projectlombok:lombok:1.18.6')
        compile('ch.qos.logback:logback-core:1.2.3')
        compile('ch.qos.logback:logback-classic:1.2.3')
        compile "org.apache.commons:commons-lang3:3.6"
        compile "commons-io:commons-io:2.6"
        compile 'commons-codec:commons-codec:1.4'
        compile 'com.lhalcyon:bip32:1.0.0'
        compile 'org.web3j:core:3.4.0'
        compile 'com.lambdaworks:scrypt:1.4.0'
        compile group: 'org.bouncycastle', name: 'bcprov-jdk15on', version: '1.60'
        compile group: 'org.bouncycastle', name: 'bcpkix-jdk15on', version: '1.60'
        compile fileTree(dir:'libs',include:['*.jar'])
}
```

示例说明

下面介绍下证书的生成流程

根证书生成

根据生成的私钥自签名生成根证书，可以指定私钥也可选择自动生成，组件封装了多种入参方法，可按需使用，参照根证书生成

以generateKPAndRootCert方法为例，示例代码如下：

```
    CertService certService = new CertService();
    X500NameInfo info = X500NameInfo.builder()
            .commonName("chain")
            .organizationName("fisco-bcos")
            .organizationalUnitName("chain")
            .build();
    //自动生成私钥（默认为RSA），并写入指定路径"out/ca",
    certService.generateKPAndRootCert(info,"out/ca");
```

执行上述方法，会在控制台打印出证书、私钥文件保存结果和路径，证书会保存在out/ca/ca.crt文件中

子证书csr生成

csr全称为Certificate Signing Request，即证书请求文件，根（父）证书通过其私钥对请求文件签名来颁发子证书

csr的生成提供了多种入参方法，参照证书申请生成

以generateCertRequestByDefaultConf为例，可快速生成csr，示例代码如下：

```
    CertService certService = new CertService();
    X500NameInfo info = X500NameInfo.builder()
            .commonName("chain")
            .organizationName("fisco-bcos")
            .organizationalUnitName("chain")
            .build();
    //自动生成RSA私钥，KeyUtils为证书组件密钥工具类
    KeyPair keyPair = KeyUtils.generateKeyPair();
    //CertUtils工具提供了证书读写解析的相关能力
    String priStr = CertUtils.readPEMAsString(keyPair.getPrivate());
    String csrStr = certService.generateCertRequestByDefaultConf(info, priStr,
↪"out/agency", "agency");
    System.out.println(csrStr);
```

执行上述方法会在控制台打印出csr文件内容，并写入out/agency/agency.csr文件中

其中涉及的CertUtils工具类，该类提供了证书读写解析的相关能力，参照CertUtils详情

子证书颁发

通过根证书和其私钥对子证书申请进行签发,提供多种入参方法，参照子证书签发

以文件路径为入参示例，示例代码如下：

```
    //参数为生成相关文件路径
    CertService certService = new CertService();
    String childStr2 = certService.generateChildCertByDefaultConf("out/ca/ca.crt",
↪"out/agency/agency.csr","out/ca/ca_pri.key", "out/agency" ,
            "agency");
    System.out.println(childStr2);
```

执行上述方法会在控制台打印出子证书内容,并写入out/agency/agency.crt文件中，可从第二步开始，继续下一级证书的签发。

证书链验证

上述步骤中我们生成了多级证书，对生成的证书链进行验证，查看证书是否有效

示例代码如下：

```
    CertService certService = new CertService();
    try {
        X509Certificate root = null;
        root = CertUtils.readCrt("out/ca/ca.crt");
        X509Certificate child = CertUtils.readCrt("out/agency/agency.crt");
        List<X509Certificate> certChain = new ArrayList<>();
        //可添加多级证书...这里以上述步骤中生成的两个证书为例
        certChain.add(root);
        certChain.add(child);
        System.out.println("证书链验证结果 = " + certService.verify(root, certChain));
    } catch (CertificateException | FileNotFoundException e) {
```

(continues on next page)

(续上页)

```
        e.printStackTrace();
    }
```

执行上述方法，可以在控制台看到证书链的验证结果为true，表明证书链的有效性

证书撤销

我们可以对有效期内的证书进行吊销操作

```
    CertService certService = new CertService();
    try {
        //从文件中读取证书（上述步骤中生成的证书路径）
        X509Certificate root = CertUtils.readCrt("out/ca/ca.crt");
        X509Certificate child = CertUtils.readCrt("out/agency/agency.crt");
        //从文件中读取私钥（上述步骤中生成的私钥路径）
        PrivateKey caPrivateKey = (PrivateKey) CertUtils.readRSAKey("out/ca/ca_pri.
↪key");
        List<X509Certificate> revokeCertificates = new ArrayList<>();
        revokeCertificates.add(child);
        //撤销上述步骤中签发的子证书
        X509CRL X509Crl = certService.createCRL(root,caPrivateKey,
↪revokeCertificates,"SHA256WITHRSA");
        System.out.println("吊销证书路径：out/agency/agency.crt");
        X509Crl.getRevokedCertificates().forEach(x509CRLEntry -> {
            System.out.println("吊销证书序列号：" +  x509CRLEntry.getSerialNumber());
        });

        //验证吊销证书后的证书链
        List<X509Certificate> certChain = new ArrayList<>();
        //可添加多级证书...这里以上述步骤中生成的两个证书为例
        certChain.add(root);
        certChain.add(child);
        System.out.println("证书链验证结果 = " + certService.verify(root,certChain,
↪X509Crl));
    } catch (Exception e) {
        e.printStackTrace();
    }
```

执行上述方法，可以在控制台看到吊销后的证书链验证结果为false，表明该子证书已经失效

PFX证书读写

crt证书只包含公钥信息

pfx由PKCS#12（Public Key Cryptography Standards #12）标准定义，包含了公钥和私钥信息。

CertUtils工具类提供了pfx的生成和读取方法，示例代码如下:

```
    try {
        List<X509Certificate> list = new ArrayList<>();
        list.add(CertUtils.readCrt("out/ca/ca.crt"));
        //生成pfx文件，参数分别为：证书别名，私钥，keyStore密码，证书信息，保存路径，证书名
        CertUtils.savePfx("fisco",(PrivateKey) CertUtils.readRSAKey("out/ca/ca_pri.
↪key"),"123",list,"out/ca","ca");
        //从pfx中导出私钥信息
        PrivateKey key = CertUtils.readPriKeyFromPfx("out/ca/ca.pfx","123");
        //在控制台输出导出私钥的BASE64编码信息
        System.out.println(Base64.toBase64String(key.getEncoded()));
    } catch (Exception e) {
```

(continues on next page)

(续上页)

```
        e.printStackTrace();
    }
```

执行上述方法，可以在out/ca路径下看到ca.pfx证书文件的生成，以及从该pfx文件中导出的私钥Base64编码信息

接口说明

CertService提供了多种功能接口：

- createRootCertificate：生成根证书，即自签名证书
- createCertRequest：生成证书请求
- createChildCertificate：生成子证书
- verify：验证证书
- createCRL：吊销证书

为方便调用，针对上述接口封装了默认配置（签名算法：SHA256WITHRSA,有效期10年）的生成接口：

- generateRootCertByDefaultConf：生成根证书
- generateCertRequestByDefaultConf：生成证书请求
- generateChildCertByDefaultConf：生成子证书
- generateKPAndRootCert：生成密钥对和根证书

KeyUtils和CertUtils两个工具类，提供了对证书和私钥的相关读写操作。

更多参照Java doc

4.3.2 cert-mgr使用

功能介绍

cert-mgr用于证书托管。

证书管理中，证书相关的私钥由单独的私钥表保存，还包含了证书表和请求表，生成的证书会保存在证书表中，子证书的请求会保存在请求表中。

证书签名算法目前支持：

- SHA256WITHRSA
- SHA256WITHECDSA
- SM3WITHSM2

前置依赖

在使用本组件前，请确认系统环境已安装相关依赖软件，清单如下：

如果您还未安装这些依赖，请参考附录。

部署教程

目前支持从源码进行部署。

获取源码

通过git下载源码：

```
https://github.com/WeBankBlockchain/Governance-Cert.git
```

注解：

- 如果因为网络问题导致长时间无法下载，请尝试：git clone https://gitee.com/WeBankBlockchain/Governance-Cert.git

进入目录：

```
cd Governance-Cert/cert-mgr
```

编译源码

方式一：如果服务器已安装Gradle

```
gradle build -x test
```

方式二：如果服务器未安装Gradle，使用gradlew编译

```
chmod +x ./gradlew && ./gradlew build -x test
```

导入jar包

cert-mgr编译之后在根目录下会生成dist文件夹，文件夹中包含cert-mgr.jar。可以将cert-mgr.jar导入到自己的项目中，例如拷贝到libs目录下，然后进行依赖配置。gradle推荐依赖配置如下，然后再对自己的项目进行编译。

```
repositories {
    mavenCentral()
    mavenLocal()
    maven {
        url "http://maven.aliyun.com/nexus/content/groups/public/"
    }
}
dependencies {
        compile 'org.springframework.boot:spring-boot-starter'
            compile 'org.springframework.boot:spring-boot-starter-data-jpa'
            compile 'org.springframework.boot:spring-boot-starter-jta-atomikos'

            testCompile('org.springframework.boot:spring-boot-starter-test') {
                    exclude group: 'org.junit.vintage', module: 'junit-vintage-
↪engine'
                    //exclude group: 'junit', module: 'junit'
            }
            compile ('org.projectlombok:lombok:1.18.6')
            testCompile ('org.projectlombok:lombok:1.18.6')
            annotationProcessor 'org.projectlombok:lombok:1.18.6'
            testAnnotationProcessor 'org.projectlombok:lombok:1.18.6'
            testCompile 'junit:junit:4.12'
            compile "org.apache.commons:commons-lang3:3.6"
            compile "commons-io:commons-io:2.6"
            compile 'com.lhalcyon:bip32:1.0.0'
```

(continues on next page)

(续上页)

```
            compile 'org.web3j:core:3.4.0'
            compile 'com.lambdaworks:scrypt:1.4.0'
            compile 'commons-codec:commons-codec:1.9'
            compile 'mysql:mysql-connector-java'
            compile group: 'org.bouncycastle', name: 'bcprov-jdk15on', version: '1.
↪60'
            compile group: 'org.bouncycastle', name: 'bcpkix-jdk15on', version: '1.
↪60'
            compile fileTree(dir:'libs',include:['*.jar'])
}
```

使用详解

依赖注入使用前需添加组件扫描，如下所示：

```
@SpringBootApplication
@ComponentScan(basePackages = { "com.webank.cert" })
public class CertTestApplication {
        public static void main(String[] args) {
                SpringApplication.run(CertTestApplication.class, args);
        }
}
```

配置

请参考下面的模板，配置application.properties。

```
## 证书存储db
spring.datasource.url=jdbc:mysql://[ip]:[port]/pkey_mgr?autoReconnect=true&
↪characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2b8
spring.datasource.username=
spring.datasource.password=

## spring jpa config
spring.jpa.properties.hibernate.hbm2ddl.auto=update
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect
```

建表

如果在上述配置中指定了**spring.jpa.properties.hibernate.hbm2ddl.auto=update**，则jpa会帮助用户自动建立数据表。

如果不希望自动建立数据表，请先关闭jpa建表开关：

```
spring.jpa.properties.hibernate.hbm2ddl.auto=validate
```

然后按下面方式手动建表，默认开启自动建表

1） 在数据源运行下述建表语句：

```
 -- Create syntax for TABLE 'cert_keys_info'
 drop table if exists cert_keys_info;
 CREATE TABLE `cert_keys_info` (
   `pk_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
   `user_id` varchar(255) NOT NULL,
```

(continues on next page)

(续上页)

```
  `key_alg` varchar(8) NOT NULL,
  `key_pem` longtext NOT NULL,
  `creat_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
↪TIMESTAMP,
  PRIMARY KEY (`pk_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- Create syntax for TABLE 'cert_info'
drop table if exists cert_info;
CREATE TABLE `cert_info` (
  `pk_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `user_id` varchar(255) NOT NULL,
  `subject_pub_key` longtext NOT NULL,
  `cert_content` longtext NOT NULL,
  `issuer_key_id` bigint(20) NOT NULL,
  `subject_key_id` bigint(20) NOT NULL,
  `parent_cert_id` bigint(20),
  `serial_number` varchar(255) NOT NULL,
  `issuer_org` varchar(255) NOT NULL,
  `issuer_cn` varchar(255) NOT NULL,
  `subject_org` varchar(255) NOT NULL,
  `subject_cn` varchar(255) NOT NULL,
  `is_ca_cert` int(4) NOT NULL,
  `creat_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
↪TIMESTAMP,
  PRIMARY KEY (`pk_id`),
  UNIQUE KEY (`parent_cert_id`,`serial_number`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- Create syntax for TABLE 'cert_request_info'
drop table if exists cert_request_info;
CREATE TABLE `cert_request_info` (
  `pk_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `parent_cert_id` bigint(20),
  `subject_key_id` bigint(20) NOT NULL,
  `user_id` varchar(255) NOT NULL,
  `cert_request_content` longtext NOT NULL,
  `subject_org` varchar(255) NOT NULL,
  `subject_cn` varchar(255) NOT NULL,
  `creat_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
↪TIMESTAMP,
  PRIMARY KEY (`pk_id`),
  UNIQUE KEY (`parent_cert_id`,`subject_key_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

接口说明

CertManagerService类是证书管理的统一入口，覆盖证书管理的全生命周期，包含如下功能：

- createRootCert：生成根证书
- createRootCertByHexPriKey：私钥Hex格式作为入参生成证书
- createCertRequest：生成请求
- createCertRequestByHexPriKey：私钥Hex格式作为入参生成请求
- createChildCert：生成子证书

- resetCertificate：证书重置
- queryCertList：证书列表查询
- queryCertRequestList：请求列表查询
- queryCertKeyList：证书私钥列表查询
- queryCertInfoByCertId：根据id查询证书
- queryCertRequestByCsrId：根据id证书请求
- exportCertToFile：证书导出

参考Java doc

示例说明

下面介绍使用的例子

实例注入

在Java项目test中创建一个测试类，在测试类中注入依赖

建议通过Spring自动注入KeysManagerService服务，示例如下：

```
    @SpringBootTest
    @RunWith(SpringRunner.class)
    public class Example {
        @Autowired
        private CertManagerService certManagerService;
    }
```

生成根证书

生成根证书，提供了多种封装接口，可按需使用

示例代码如下

```
    @Test
    public void testCreateRootCert() throws Exception {
        X500NameInfo issuer = X500NameInfo.builder()
                .commonName("chain")
                .organizationName("fisco-bcos")
                .organizationalUnitName("chain")
                .build();
        //用户id，如Bob
        String userId = "Bob";
        //有效期
        Date beginDate = new Date();
        Date endDate = new Date(beginDate.getTime() + CertConstants.DEFAULT_
↪VALIDITY);
        //默认配置生成：采用RSA密钥对，也可调用其他封装接口，自定义密钥类型，支持pem和Hex格式
        CertVO cert = certManagerService.createRootCert(userId,issuer,beginDate,
↪endDate);
        System.out.println("证书id: " + cert.getPkId());
        System.out.println("证书签发者: " + cert.getUserId());
        System.out.println("父证书id: " + cert.getPCertId());
        System.out.println("签发私钥id: " + cert.getIssuerKeyId());
        System.out.println("证书内容: " + cert.getCertContent());
    }
```

执行上述代码，可在控制台看到生成的证书内容，证书和相关联的签名私钥会保存在db中

证书列表查询

可以通过接口按一定条件查询已签发的证书列表

示例代码如下

```
    @Test
    public void testQueryCertList() throws Exception {
        List<CertVO> list = certManagerService.queryCertInfoList();
        list.forEach(certVO -> {
            System.out.println("证书id: " + certVO.getPkId() + "\t");
            System.out.println("证书id: " +certVO.getUserId() + "\t");
            System.out.println("父证书id: " + certVO.getPCertId() + "\t");
            System.out.println("签发私钥id: " + certVO.getIssuerKeyId() + "\t");
            System.out.println("----------");
        });
    }
```

执行上述代码，可在控制台看到所有签发的证书列表及相关信息

子证书csr生成

从上述步骤查询结果中，选择证书作为根证书，请求子证书。

示例代码如下

```
    @Test
    public void testCreateCertRequest() throws Exception {
        X500NameInfo subject = X500NameInfo.builder()
                .commonName("agancy")
                .organizationName("fisco-bcos")
                .organizationalUnitName("agancy")
                .build();
        String userId = "Alice";
        //自动生成密钥对，入参为：当前用户userId，父证书issuerCertId（可根据上述步骤查询得
到），申请机构信息subject
        CertRequestVO csr = certManagerService.createCertRequest(userId, 1,␣
↪subject);
        System.out.println("证书申请id: " + csr.getPkId());
        System.out.println("父证书签发者: " + csr.getPCertUserId());
        System.out.println("证书申请者: " + csr.getUserId());
        System.out.println("父证书id: " + csr.getPCertId());
        System.out.println("证书申请内容" + csr.getCertRequestContent());
    }
```

执行上述代码，证书申请和相关联的签名私钥会保存在db中，同时在db中将证书申请和其对应的父证书关联，便于后续子证书签发。

证书申请列表查询

可以通过接口按一定条件查询证书申请列表

示例代码如下

```
    @Test
    public void testQueryCertRequestList() throws Exception {
        List<CertRequestVO> list = certManagerService.queryCertRequestList();
        list.forEach(csr -> {
            System.out.println("证书申请id: " + csr.getPkId());
            System.out.println("父证书签发者: " + csr.getPCertUserId());
            System.out.println("证书申请者: " + csr.getUserId());
            System.out.println("父证书id: " + csr.getPCertId());
            System.out.println("----------");
        });
    }
```

执行上述代码，可在控制台看到所有证书申请的列表及相关信息

子证书签发

从上述步骤查询结果中，可选择证书申请签发子证书。

示例代码如下

```
    @Test
    public void testCreateChildCert() throws Exception {
        String userId = "Bob";
        //参数：签发者userId，证书申请id
        CertVO child = certManagerService.createChildCert(userId, 1);
        System.out.println("证书id: " + child.getPkId());
        System.out.println("证书签发者: " + child.getUserId());
        System.out.println("父证书id: " + child.getPCertId());
        System.out.println("签发私钥id: " + child.getIssuerKeyId());
        System.out.println("证书内容: " + child.getCertContent());
    }
```

执行上述代码，可在控制台看到签发的子证书详情，可从第二步开始，继续签发下一级证书

证书重置

证书重置支持对证书有效期和用途配置进行重置，方法为exportCertToFile。 可通过第二步的查询证书列表，选择证书重置，如选择证书id为1的证书进行重置，示例代码如下：

```
    @Test
    public void testResetCertificate() throws Exception {
        String userId = "John";
        Date beginDate = new Date();
        Date endDate = new Date(beginDate.getTime() + CertConstants.DEFAULT_
↪VALIDITY);
        CertVO cert = certManagerService.resetCertificate(userId,1,new␣
↪KeyUsage(KeyUsage.dataEncipherment),beginDate,endDate);
        System.out.println("证书id: " + cert.getPkId());
        System.out.println("证书签发者: " + cert.getUserId());
        System.out.println("父证书id: " + cert.getPCertId());
        System.out.println("签发私钥id: " + cert.getIssuerKeyId());
        System.out.println("证书内容: " + cert.getCertContent());
    }
```

执行上述代码可以在控制台看到重置后的证书详情

证书导出

可通过第二步的查询证书列表，选择证书导出，如选择证书id为1的证书进行导出，示例代码如下：

```
    @Test
    public void testExportCertToFile() throws Exception {
        String filePath = "src/ca.crt";
        certManagerService.exportCertToFile(1L,filePath);
        System.out.println("导出完成，保存路径为：" + filePath);
    }
```

更多示例

参考证书管理接口示例

参考Java doc

4.4 Java doc

cert-toolkit和cert-mgr的java doc

4.5 常见问题

CHAPTER 5

附录

5.1 MySql的安装

此处以Centos安装MariaDB为例。MariaDB数据库是 MySQL 的一个分支，主要由开源社区在维护，采用 GPL 授权许可。MariaDB完全兼容 MySQL，包括API和命令行。其他安装方式请参考MySQL官网。

（1）安装MariaDB

- 安装命令

```
sudo yum install -y mariadb*
```

（2）启停

```
启动：sudo systemctl start mariadb.service
停止：sudo systemctl stop  mariadb.service
```

（3）设置开机启动

```
sudo systemctl enable mariadb.service
```

（4）初始化root用户

```
执行以下命令：
sudo mysql_secure_installation
以下根据提示输入：
Enter current password for root (enter for none):<-初次运行直接回车
Set root password? [Y/n] <- 是否设置root用户密码，输入y并回车或直接回车
New password: <- 设置root用户的密码
Re-enter new password: <- 再输入一次你设置的密码
Remove anonymous users? [Y/n] <- 是否删除匿名用户，回车
Disallow root login remotely? [Y/n] <-是否禁止root远程登录，回车
Remove test database and access to it? [Y/n] <- 是否删除test数据库，回车
Reload privilege tables now? [Y/n] <- 是否重新加载权限表，回车
```

- 使用root用户登录，密码为初始化设置的密码

```
mysql -uroot -p -h localhost -P 3306
```

- 授权root用户远程访问

  注意，以下语句仅适用于开发环境，不能直接在实际生产中使用！！！ 以下操作仅供参考，请勿直接拷贝，请自定义设置复杂密码。

```
mysql > GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY '123456' WITH␣
↪GRANT OPTION;
mysql > flush PRIVILEGES;
```

安全温馨提示:

- 例子中给出的数据库密码（123456）仅为样例，强烈建议设置成复杂密码
- 例子中root用户的远程授权设置会使数据库在所有网络上都可以访问，请按具体的网络拓扑和权限控制情况，设置网络和权限帐号

（5）创建test用户并授权本地访问

```
mysql > GRANT ALL PRIVILEGES ON *.* TO 'test'@localhost IDENTIFIED BY '123456'␣
↪WITH GRANT OPTION;
mysql > flush PRIVILEGES;
```

（6）测试是否成功

- 登录数据库

```
mysql -utest -p123456 -h localhost -P 3306
```

- 创建数据库

```
mysql > create database datastash;
mysql > use datastash;
```

以上语句仅适用于开发环境，不能直接在实际生产中使用！！！以上设置会使数据库在所有网络上都可以访问，请按具体的网络拓扑和权限控制情况，设置网络和权限帐号

5.2 Java安装

5.2.1 Ubuntu环境安装Java

```
# 安装默认Java版本(Java 8或以上)
sudo apt install -y default-jdk
# 查询Java版本
java -version
```

5.2.2 CentOS环境安装Java

```
# 查询centos原有的Java版本
$ rpm -qa|grep java
# 删除查询到的Java版本
$ rpm -e --nodeps java版本
# 查询Java版本，没有出现版本号则删除完毕
$ java -version
# 创建新的文件夹，安装Java 8或以上的版本，将下载的jdk放在software目录
# 从openJDK官网(https://jdk.java.net/java-se-ri/8)或Oracle官网(https://www.oracle.
↪com/technetwork/java/javase/downloads/index.html)选择Java 8或以上的版本下载，例如下
载jdk-8u201-linux-x64.tar.gz
$ mkdir /software
# 解压jdk
```

(continues on next page)

(续上页)

```
$ tar -zxvf jdk-8u201-linux-x64.tar.gz
# 配置Java环境，编辑/etc/profile文件
$ vim /etc/profile
# 打开以后将下面三句输入到文件里面并退出
export JAVA_HOME=/software/jdk-8u201-linux-x64.tar.gz
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
# 生效profile
$ source /etc/profile
# 查询Java版本，出现的版本是自己下载的版本，则安装成功。
java -version
```

5.3 Git安装

git：用于拉取最新代码

centos:

```
sudo yum -y install git
```

ubuntu:

```
sudo apt install git
```