FISCO BCOS Documentation

发布 ***v2.5.0***

fisco-dev

2020 年 **06** 月 **23** 日

Contents

1 平台介绍 3

2 版本及兼容 7

3 安装 19

4 教程 25

5 使用手册 41

6 运维部署工具 219

7 SDK 279

8 区块链浏览器 349

9 系统设计 357

10 JSON-RPC API 445

11 常见问题解答 477

12 社区 481

FISCO BCOS 是一个稳定、高效、安全的区块链底层平台，经过多家机构、多个应用，长时间在生产环境运行的实际检验。

- Github主页
- 深度解析系列文章
- 贡献代码
- 反馈问题
- 应用案例集
- 微信群
- 公众号

概览

- 基于FISCO BCOS 2.0+快速构建区块链系统，请参考 安装
- 基于FISCO BCOS 2.0+部署多群组区块链、构建第一个区块链应用，请参考 教程
- 深入了解FISCO BCOS 2.0+功能请看 配置文件和配置项、节点准入、并行交易、分布式存储、国密 等请参考 使用手册
- 控制台：**交互式命令行工具**，可访问区块链节点，查询区块链状态，部署并调用合约等。
- 运维部署工具(Generator)：支持建链、扩容等操作，**推荐构建企业级区块链时使用**，快速使用方法可参考 教程
- SDK：提供访问节点状态、修改区块链系统配置以及节点发送交易等接口。
- 浏览器详细介绍请参考 浏览器
- JSON-RPC接口可参考 JSON-RPC API
- 系统设计文档请参考 系统设计

关键特性

- 多群组: 教程 使用手册 设计文档
- 并行计算: 使用手册 设计文档
- 分布式存储: 使用手册 设计文档

重要:

- 本技术文档只适用FISCO BCOS 2.0+，FISCO BCOS 1.3.x版本的技术文档请查看 1.3系列技术文档
- FISCO BCOS 2.0+版本及兼容性说明 这里

CHAPTER 1

平台介绍

FISCO BCOS是由国内企业主导研发、对外开源、安全可控的企业级金融联盟链底层平台，由金链盟开源工作组协作打造，并于2017年正式对外开源。

社区以开源链接多方，截止2020年5月，汇聚了超1000家企业及机构、逾万名社区成员参与共建共治，发展成为最大最活跃的国产开源联盟链生态圈。底层平台可用性经广泛应用实践检验，数百个应用项目基于FISCO BCOS底层平台研发，超80个已在生产环境中稳定运行，覆盖文化版权、司法服务、政务服务、物联网、金融、智慧社区等领域。

注解： FISCO BCOS以联盟链的实际需求为出发点，兼顾性能、安全、可运维性、易用性、可扩展性，支持多种SDK，并提供了可视化的中间件工具，大幅缩短建链、开发、部署应用的时间。此外，FISCO BCOS通过信通院可信区块链评测功能、性能两项评测，单链TPS可达两万。

1.1 关键特性

1.2 架构

FISCO BCOS 在2.0中，创新性提出“一体两翼多引擎”架构，实现系统吞吐能力的横向扩展，大幅提升性能，在安全性、可运维性、易用性、可扩展性上，均具备行业领先优势。

一体指代群组架构，支持快速组建联盟和建链，让企业建链像建聊天群一样便利。根据业务场景和业务关系，企业可选择不同群组，形成多个不同账本的数据共享和共识，从而快速丰富业务场景、扩大业务规模，且大幅简化链的部署和运维成本。

两翼指的是支持并行计算模型和分布式存储，二者为群组架构带来更好的扩展性。前者改变了区块中按交易顺序串行执行的做法，基于DAG（有向无环图）并行执行交易，大幅提升性能；后者支持企业（节点）将数据存储在远端分布式系统中，克服了本地化数据存储的诸多限制。

多引擎是一系列功能特性的总括，比如预编译合约能够突破EVM的性能瓶颈，实现高性能合约；控制台可以让用户快速掌握区块链使用技巧等。

上述功能特性均聚焦解决技术和体验的痛点，为开发、运维、治理和监管提供更多的工具支持，让系统处理更快、容量更高，使应用运行环境更安全、更稳定。

1.3 核心模块

FISCO BCOS采用高通量可扩展的多群组架构，可以动态管理多链、多群组，满足多业务场景的扩展需求和隔离需求，核心模块包括：

- **共识机制**：可插拔的共识机制，支持PBFT、Raft和rPBFT共识算法，交易确认时延低、吞吐量高，并具有最终一致性。其中PBFT和rPBFT可解决拜占庭问题，安全性更高。
- **存储**：世界状态的存储从原来的MPT存储结构转为分布式存储，避免了世界状态急剧膨胀导致性能下降的问题；引入可插拔的存储引擎，支持LevelDB、RocksDB、MySQL等多种后端存储，支持数据简便快速扩容的同时，将计算与数据隔离，降低了节点故障对节点数据的影响。
- **网络**：支持网络压缩功能，并基于负载均衡的思想实现了良好的分布式网络分发机制，最大化降低带宽开销。

1.4 性能

为提升系统性能，FISCO BCOS从提升交易执行效率和并发两个方面优化了交易执行，使得交易处理性能达到万级以上。

- **基于C++的Precompiled合约**：区块链底层内置C++语言编写的Precompiled合约，执行效率更高。
- **交易并行执行**：基于DAG算法根据交易间互斥关系构建区块内交易执行流，最大化并行执行区块内的交易。
- 交易生命周期的异步并行处理：共识、同步、落盘等各个环节的异步化以及并行处理。

1.5 安全性

考虑到联盟链的高安全性需求，除了节点之间、节点与客户端之间通信采用TLS安全协议外，FISCO BCOS还实现了一整套安全解决方案：

- **网络准入机制**：限制节点加入、退出联盟链，可将指定群组的作恶节点从群组中删除，保障了系统安全性。
- **黑白名单机制**：每个群组仅可接收相应群组的消息，保证群组间网络通信的隔离性；CA黑名单机制可及时与作恶节点断开网络连接，保障了系统安全。
- **权限管理机制**：基于分布式存储权限控制机制，灵活、细粒度地控制外部账户部署合约和创建、插入、删除和更新用户表的权限。
- **支持国密算法**：支持国密加密、签名算法和国密通信协议。
- **落盘加密方案**：支持加密节点落盘数据，保障链上数据的机密性。
- **密钥管理方案**：在落盘加密方案的基础上，采用KeyManager服务管理节点密钥，安全性更强。
- **同态加密**、**群环签名**：链上提供了同态加密、群环签名接口，用于满足更多的业务需求。

1.6 可运维性

联盟链系统中，区块链的运维至关重要，FISCO BCOS提供了一整套运维部署工具，并引入了**合约命名服务**、**数据归档和迁移**、**合约生命周期管理**等工具来提升运维效率。

- **运维部署工具**: 部署、管理和监控多机构多群组联盟链的便捷工具，支持扩容节点、扩容新群组等多种操作。
- **合约命名服务**: 建立合约地址到合约名和合约版本的映射关系，方便调用者通过记忆简单的合约名来实现对链上合约的调用。

- 数据归档、迁移和导出功能: 提供数据导出组件，支持链上数据归档、迁移和导出，增加了链上数据的可维护性，降低了运维复杂度。
- 合约生命周期管理: 链上提供合约生命周期管理功能，便于链管理员对链上合约进行管理。

1.7 易用性

FISCO BCOS引入开发部署工具、交互式控制台、区块链浏览器等工具来提升系统的易用性，大幅缩短建链、部署应用的时间。

- 开发部署工具
- 交互式的命令行工具console
- 区块链浏览器

为了便于不同语言开发者快速开发应用，FISCO BCOS同时支持Java SDK、Node.js SDK、Python SDK和Go SDK

1.8 社区开发工具

依托庞大的开源生态，社区内众伙伴秉承“来自开发者，用于开发者”的共建理念，在FISCO BCOS底层平台之上，自主研发多个趁手开发工具并回馈给社区，从不同业务层面需求上降低区块链应用开发难度和成本。以下作部分列举，欢迎更多机构或开发者向社区反馈更多好用的工具。

- 区块链中间件平台WeBASE：面向多种对象，如开发者、运营者，并根据不同的场景，包括开发、调试、部署、审计等，打造丰富的功能组件和实用工具，提供友好的、可视化的操作环境。
- 分布式身份解决方案WeIdentity：基于区块链的分布式多中心的技术解决方案，提供分布式实体身份标识及管理、可信数据交换协议等一系列的基础层与应用接口，可实现实体对象（人或物）数据的安全授权与交换。
- 分布式事件驱动架构WeEvent：实现了可信、可靠、高效的跨机构、跨平台事件通知机制。在不改变已有商业系统的开发语言、接入协议的情况下，实现跨机构、跨平台的事件通知与处理。
- 跨链协作方案WeCross：支持跨链事务交易，满足跨链交易的原子性，对跨链进行治理，可支持多方协作管理，避免单点风险。
- 场景式隐私保护解决方案WeDPR：针对隐匿支付、匿名投票、匿名竞拍和选择性披露等应用方案，提供即时可用场景式隐私保护高效解决方案，助力各行业合法合规地探索数据相关业务。
- ChainIDE：提供智能合约云端开发工具，帮助开发者节约边际成本，加速推送区块链应用落地。

CHAPTER 2

版本及兼容

FISCO BCOS 2.5.0

变更描述、兼容及升级说明

- FISCO BCOS v2.5.0

FISCO BCOS 2.4.0

变更描述、兼容及升级说明

- FISCO BCOS v2.4.0

FISCO BCOS 2.3.0

变更描述、兼容及升级说明

- FISCO BCOS v2.3.0

FISCO BCOS 2.2.0

变更描述、兼容及升级说明

- FISCO BCOS v2.2.0

FISCO BCOS 2.1.0

变更描述、兼容及升级说明

- FISCO BCOS v2.1.0

FISCO BCOS 2.0.0

变更描述、兼容及升级说明

- FISCO BCOS v2.0.0

FISCO BCOS 2.0.0-rc3

新增特性

- 分布式存储 (操作手册)
- CRUD操作的SDK接口 (操作手册)

变更描述、兼容及升级说明

- FISCO BCOS v2.0.0-rc3

FISCO BCOS 2.0.0-rc2

新增特性

- 并行计算模型 (操作手册) (设计文档)
- 分布式存储 (操作手册)

变更描述、兼容及升级说明

- FISCO BCOS v2.0.0-rc2

FISCO BCOS 2.0.0-rc1

新增特性

- 群组架构 (操作教程) (设计文档)
- 控制台 (安装) (操作手册)
- 虚拟机
- 编译合约 (合约开发)
- 实现CRUD操作的合约 (操作教程)
- 密钥管理服务 (使用手册)
- 准入控制 (设计文档)

变更描述、兼容及升级说明

- FISCO BCOS v2.0.0-rc1

FISCO BCOS 1.x Releases

FISCO BCOS 1.3 正式版:

- FISCO BCOS 1.3.8 Release
- FISCO BCOS 1.3.7 Release
- FISCO BCOS 1.3.6 Release
- FISCO BCOS 1.3.5 Release
- FISCO BCOS 1.3.4 Release
- FISCO BCOS 1.3.3 Release
- FISCO BCOS 1.3.2 Release
- FISCO BCOS 1.3.1 Release

- FISCO BCOS 1.3.0 Release

FISCO BCOS 1.2 正式版:

- FISCO BCOS 1.2.0 Release

FISCO BCOS 1.1 正式版:

- FISCO BCOS 1.1.0 Release

FISCO BCOS 1.0 正式版:

- FISCO BCOS 1.0.0 Release

FISCO BCOS 预览版:

- FISCO-BCOS 1.5.0 pre-release

查看节点和数据版本

- 查看节点二进制版本：`./fisco-bcos --version`
- 数据格式和通信协议的版本：通过配置文件 config.ini的supported_version配置项 获取

2.1 v2.5.0

---

v2.4.x升级到v2.5.0

- **兼容升级**：直接替换v2.4.x节点的二进制为 v2.5.0二进制，升级后的版本修复v2.4.x中的bug，并进一步优化了性能。
- **全面升级**：参考 安装 搭建新链，重新向新节点提交所有历史交易，升级后节点包含v2.5.0所有新特性
- v2.5.0 Release Note

2.1.1 变更描述

新增

- 新增预编译合约，地址0x5，实现模指数运算
- 新增预编译合约，地址0x6，实现椭圆曲线alt_bn128的加法
- 新增预编译合约，地址0x7，实现椭圆曲线alt_bn128的乘法
- 新增预编译合约，地址0x8，实现椭圆曲线配对操作用于zkSNARK验证
- 新增预编译合约，地址0x9，实现blake2哈希函数
- 新增流控功能，实现可配置节点出带宽、最大QPS
- 新增ChainGovernance预编译合约，地址0x1008，实现基于角色的权限管理
- 新增SDK连接节点支持国密SSL，可配置是否启用
- 新增账号管理可冻结解冻私钥对应的账号

更改

- MySQL存储模式下，合约表的字段类型修改为mediumblob
- 国密模式由编译选项修改为配置项

- 更改节点只接受同一机构的SDK连接，可配置是否启用
- 默认开启Paillier同态加密和群签名功能
- build_chain脚本使用私钥为secp256k1的根证书和机构证书
- PBFTBackup的存储由LevelDB修改为RocksDB
- 重构libdevcrypto模块，优化代码结构，使用TASSL实现国密和非国密TLS连接
- 优化存储模块openTable的锁实现
- 优化区块数据编码为并行
- 优化大对象析构耗时为异步
- 优化日志输出机制，降低日志输出对性能的影响
- 优化MHD和交易池模块的线程数，减少内存占用
- 优化MySQL存储适配器实现，优化ZdbStorage的代码实现

修复

- 修复同一个区块内修改Entry，后续交易查询该Entry结果出错的问题
- 修复raft共识下，非leader节点交易池中的交易不能被打包的问题
- 修复CachedStorage的一个死锁问题
- 修复开启Binlog情况下，极端情况下，从Binlog恢复出错的问题
- 修复特定情况下节点重启后Viewchange被拒，无法快速恢复视图的问题

兼容性

向前兼容，旧版本可以直接替换程序升级

兼容模式回滚至v2.4.x方法

当节点采用兼容模式从v2.4.x升级至v2.5.0后，可直接通过将节点二进制替换回v2.4.x完成回滚。

2.2 v2.4.0

---

v2.3.x升级到v2.4.0

- **兼容升级**： 直接替换v2.3.x节点的二进制为 v2.4.0二进制 ，升级后的版本修复v2.3.x中的bug，并新增了2.4.0动态群组生命周期管理功能、网络统计功能，但不会包含2.4.0所有特性，普通场景下可回滚至v2.3.x。回滚方法参考本文最后一节。
- **全面升级**： 参考 安装 搭建新链，重新向新节点提交所有历史交易，升级后节点包含v2.4.0所有新特性
- v2.4.0 Release Note

2.2.1 变更描述

新特性

- 支持动态群组生命周期管理: 相关的RPC接口包括generateGroup、startGroup、stopGroup、removeGroup、recoverGro
- 支持网络流量和Gas统计
- Precompiled合约支持Gas
- 支持EVM Gas衡量标准插件化配置

- 添加新接口sendRawTransactionAndGetProof推送交易回执、交易Merkle证明和交易回执Merkle证明

更新

- CMake最低要求修改为3.7，支持依赖库多源下载

修复

- 修复国密模式下ecRecover接口不可用问题
- 修复国密模式、非国密模式下sha256接口返回值不一致的问题

兼容性

向前兼容，旧版本可以直接替换程序升级，替换后的节点修复v2.3.x中的bug，并新增了2.4.0的动态群组生命周期管理功能、网络统计功能，若要启用v2.4.0版本所有新特性，需重新搭链。

兼容模式回滚至v2.2.x方法

当节点采用兼容模式从v2.3.x升级至v2.4.0后，可直接通过将节点二进制替换回v2.3.x完成回滚。

2.3 v2.3.0

---

v2.2.x升级到v2.3.0

- **兼容升级**：直接替换v2.2.x节点的二进制为 v2.3.0二进制 ，升级后的版本修复v2.2.x中的bug，但不会启用v2.3.0新特性，普通场景下可回滚至v2.2.x。回滚方法参考本文最后一节。
- **全面升级**：参考 安装 搭建新链，重新向新节点提交所有历史交易，升级后节点包含v2.3.0新特性
- v2.3.0 Release Note

2.3.1 变更描述

新特性

- 同态加密：链上支持同态加密功能，启用该功能可参考这里
- 群环签名: 链上支持群签名验证和环签名验证，并提供群环签名服务端和客户端 Demo，实现群环签名机构内生成、上链和链上验证功能
- rPBFT：基于PBFT共识算法，实现一种新型的共识算法rPBFT，尽量减少节点规模对共识算法的影响，配置rPBFT请参考共识配置和rPBFT共识配置
- KVTable：提供基于键值型数据读写方式，相较于Table合约的CRUD接口，更加简单易用、容易维护
- 合约管理功能：提供合约生命周期管理接口，包括合约的冻结、解冻、合约状态查询及其相关的授权、权限查询等操作，方便运维人员对上链合约的管理

更新

- rpc.listen_ip拆分成channel_listen_ip和jsonrpc_listen_ip
- 提供合约写权限控制接口，包括合约写权限授权、撤回和查询
- 简化并行交易配置
- 推荐使用MySQL直连的存储模式替代External存储模式

修复

- 修复特定兼容场景下的内存问题

兼容性

向前兼容，旧版本可以直接替换程序升级，但无法启动此版本的新特性。若需要用此版本的新特性，需重新搭链。

兼容模式回滚至v2.2.x方法

当节点采用兼容模式从v2.2.x升级至v2.3.0后，可直接通过将节点二进制替换回v2.2.x完成回滚。

2.4 v2.2.0

---

v2.1.x升级到v2.2.0

- **兼容升级**： 直接替换v2.1.x节点的二进制为 v2.2.0二进制 ，升级后的版本修复v2.1.x中的bug，但不会启用v2.2.0新特性，普通场景下可回滚至v2.1.x。回滚方法参考本文最后一节。
- **全面升级**： 参考 安装 搭建新链，重新向新节点提交所有历史交易，升级后节点包含v2.2.0新特性
- v2.2.0 Release Note

2.4.1 变更描述

新特性

- 构建交易和回执的默克尔树，提供一种基于SPV的证明方式
- 插件化缓存机制并提供缓存开关

更新

从流程、存储、协议三方面进行优化，提升性能。

1. 流程

- 异步提交RPC交易到交易池
- 并行化对交易池中交易的处理操作
- 优化特定数据的缓存策略
- 优化交易并行执行过程中锁粒度
- 优化部分对象的访问方式，减少拷贝花销

2. 存储

- 限制表名最大长度，从64调整为50
- 以二进制方式对区块数据和nonce数据进行编码存储
- 移除数据落盘阶段对部分表的排序和hash计算

3. 协议

- 优化区块同步策略
- 优化PBFT消息转发策略
- 优化Prepare包结构
- 优化交易广播策略

- 优化交易转发策略

修复

- 修复特定兼容场景下的缓存bug

兼容性

向前兼容，旧版本可以直接替换程序升级，但无法启动此版本的新特性。若需要用此版本的新特性，需重新搭链。

兼容模式回滚至v2.1.x方法

当节点采用兼容模式从v2.1.x升级至v2.2.0后，可直接通过将节点二进制替换回v2.1.x完成回滚。

2.5 v2.1.0

---

v2.0.x升级到v2.1.0

- **兼容升级**：直接替换v2.0.x节点的二进制为 v2.1.0二进制 ，升级后的版本修复v2.0.x中的bug，但不会启用v2.1.0新特性，普通场景下可回滚至v2.0.0。回滚方法参考本文最后一节。
- **全面升级**：参考 安装 搭建新链，重新向新节点提交所有历史交易，升级后节点包含v2.1.0新特性
- v2.1.0 Release Note

2.5.1 变更描述

新特性

- CA白名单功能
- AMOP认证功能
- 合约事件推送
- 运行时启动新群组

更新

- 支持Channel Message v2协议
- 节点连接支持域名配置
- 部署合约的二进制长度放宽至256K
- 交易出错打印更全面的日志
- build_chain.sh生成的SDK证书名更名为sdk.crt和sdk.key
- 为提升性能进行了代码实现细节的调整
- 降低了节点内存的占用

修复

- 修复了在某种场景下channel连接抛异常的错误

兼容性

向前兼容，旧版本可以直接替换程序升级，但无法启动此版本的新特性。若需要用此版本的新特性，需重新搭链。

兼容模式回滚至v2.0.0方法

当节点采用兼容模式从v2.0.x升级至v2.1.0后，可直接通过将节点二进制替换回v2.0.x完成回滚。若在升级到v2.1.0之后部署过较大二进制的合约（在24K-256K之间），回滚至v2.0.x版本则不能重新同步数

据，该条部署合约的交易会执行失败，导致同步失败。此时只能先用v2.1.0同步至最新区块，再回滚至v2.0.x。

2.6 v2.0.0

v2.0.0-rc3升级到v2.0.0

- **兼容升级**：直接替换v2.0.0-rc3节点的二进制为 v2.0.0二进制 ，升级后的版本修复v2.0.0-rc3中的bug，但不会启用v2.0.0新特性， **升级到v2.0.0后，无法回滚到v2.0.0-rc3**
- **全面升级**：参考 安装 搭建新链，重新向新节点提交所有历史交易，升级后节点包含v2.0.0新特性
- v2.0.0 Release Note

2.6.1 变更描述

新特性

- AMOP协议支持多播
- AMOP协议支持二进制传输
- JSON-RPC `getTotalTransactionCount`接口新增历史失败交易数统计

更新

- RocksDB模式支持落盘加密
- 使用TCMalloc优化内存使用

修复

- 修复P2P模块偶现不处理消息的问题
- 修复MySQL或External模式下未赋值字段，查询失败
- 修复某些极端场景下同步错误的问题

兼容性

向前兼容，旧版本可以直接替换程序升级，但无法启动此版本的新特性。若需要用此版本的新特性，需重新搭链。

2.7 v2.0.0-rc3

v2.0.0-rc2升级到v2.0.0-rc3

- **兼容升级**：直接替换v2.0.0-rc2节点的二进制为 rc3二进制 ，升级后的版本修复v2.0.0-rc2中的bug，但不会启用v2.0.0-rc3新特性， **升级到v2.0.0-rc3后，无法回滚到v2.0.0-rc2**
- **全面升级**：参考 安装 搭建新链，重新向新节点提交所有历史交易，升级后节点包含v2.0.0-rc3新特性
- v2.0.0-rc3 Release Note

2.7.1 变更描述

新特性

- 分布式存储：新增支持底层通过数据库连接池直连MySQL
- 分布式存储：新增支持RocksDB引擎，搭建新链时存储默认采用RocksDB
- 分布式存储：新增CRUD接口支持，控制台1.0.3以上版本提供类SQL语句读写区块链数据

更新

- 完善ABI解码模块
- 修改预编译合约和RPC接口错误码，统一为负数
- 优化存储模块，增加缓存层，支持配置缓存大小
- 优化存储模块，允许流水线提交区块。可配置`[storage].max_capacity`控制允许使用的内存空间大小
- 移动分布式存储配置项`[storage]`，从群组genesis文件移动到到群组ini配置文件中
- 默认存储升级到RocksDB，仍支持旧版本LevelDB
- 调整交易互斥变量的拼接逻辑，提高不同合约间交易的并行度

修复

- 修复CRUD接口合约开启并行时可能出现的异常终止

2.7.2 兼容性说明

RC3**向前兼容**，旧版本可以直接替换程序升级，但无法启动此版本的新特性。若需要用此版本的新特性，需重新搭链。

2.8 v2.0.0-rc2

---

v2.0.0-rc1升级到v2.0.0-rc2

- **兼容升级**：直接替换v2.0.0-rc1节点的二进制为 v2.0.0-rc2二进制 ，升级后的版本修复v2.0.0-rc1中的bug，但不会启用v2.0.0-rc2并行计算、分布式存储等新特性， **升级到v2.0.0-rc2后，无法回滚到v2.0.0-rc1**
- **全面升级**：参考 安装 搭建新链，重新向新节点提交所有历史交易，升级后节点包含v2.0.0-rc2新特性
- v2.0.0-rc2 Release Note

2.8.1 变更描述

主要特性

- 并行计算模型：可并行合约开发框架、交易并行执行引擎（PTE）
- 分布式存储：amdb-proxy、SQLStorage

版本优化

- 优化了区块打包交易数的逻辑，根据执行时间动态的调整区块打包交易数
- 优化了区块同步的流程，让区块同步更快

- 并行优化了将交易的编解码、交易的验签和落盘的编码
- 优化了交易执行返回码的逻辑，让返回码更准确
- 升级了存储模块，支持并发读写

其他特性

- 加入网络数据包压缩
- 加入兼容性配置
- 交易编码中加入chainID和groupID
- 交易中加入二进制缓存
- 创世块中加入timestamp信息
- 增加了一些precompile的demo
- 支持用Docker搭链
- 删除不必要的日志
- 删除不必要的重复操作

Bug修复

- RPC中处理参数时asInt异常造成程序退出的Bug
- 交易执行Out of gas时交易一直在交易池中不被处理的Bug
- 不同组间可以用相同的交易二进制重放的Bug
- insert操作造成的性能衰减问题
- 一些稳定性修复

2.8.2 兼容性说明

2.9 v2.0.0-rc1

---

v1.x升级到v2.0.0-rc1

- **v2.0.0-rc2不兼容v1.x，v2.0.0-rc1无法直接解析v1.x产生的历史区块数据**，但可通过在 v2.0.0-rc1 的新链上执行历史交易的方式恢复旧数据
- **搭建2.0的新链**：请参考 安装
- v2.0.0-rc1 Release Note

2.9.1 变更描述

架构

1. 新增群组架构：各群组独立共识和存储，在较低运维成本基础上实现系统吞吐能力横向扩展。
2. 新增分布式数据存储：支持节点将数据存储在远端分布式系统中，实现计算与数据隔离、高速扩容、数据安全等级提升等目标。
3. 新增对预编译合约的支持：底层基于C++实现预编译合约框架，兼容solidity调用方式，提升智能合约执行性能。
4. 引入evmc扩展框架：支持扩展不同虚拟机引擎。
5. 升级重塑P2P、共识、同步、交易执行、交易池、区块管理模块。

协议

1. 实现一套CRUD基本数据访问接口规范合约，基于CRUD接口编写业务合约，实现传统面向SQL方式的业务开发流程。
2. 支持交易上链异步通知、区块上链异步通知以及自定义的AMOP消息通知等机制。
3. 升级以太坊虚拟机版本，支持Solidity 0.5.2版本。
4. 升级RPC模块。

安全

1. 升级落盘加密，提供密钥管理服务。开启落盘加密功能时，依赖KeyManager服务进行密钥管理。
2. 升级准入机制，通过引入网络准入机制和群组准入机制，在不同维度对链和数据访问进行安全控制。
3. 升级权限控制体系，基于表进行访问权限的设计。

其他

1. 提供入门级的搭链工具。
2. 提供模块化的单元测试和端对端集成测试，支持自动化持续集成和持续部署。

2.9.2 兼容性说明

CHAPTER 3

安装

本章介绍FISCO BCOS所需的必要安装和配置。本章通过在单机上部署一条4节点的FISCO BCOS联盟链，帮助用户掌握FISCO BCOS部署流程。请根据这里使用支持的**硬件和平台**操作。

3.1 单群组FISCO BCOS联盟链的搭建

本节以搭建单群组FISCO BCOS链为例操作。使用开发部署工具 `build_chain.sh`脚本在本地搭建一条**4 节点**的FISCO BCOS链，以`Ubuntu 16.04 64bit`系统为例操作。

注解:

- 若需在已有区块链上进行升级，请转至 版本及兼容 章节。
- 搭建多群组的链操作类似， 参考这里 。
- 本节使用预编译的静态‘fisco-bcos‘二进制文件，在CentOS 7和Ubuntu 16.04 64bit上经过测试。

3.1.1 准备环境

- 安装依赖

开发部署工具 `build_chain.sh`脚本依赖于`openssl, curl`，使用下面的指令安装。 若为CentOS，将下面命令中的`apt`替换为`yum`执行即可。macOS执行`brew install openssl curl`即可。

```
sudo apt install -y openssl curl
```

- 创建操作目录

```
cd ~ && mkdir -p fisco && cd fisco
```

- 下载`build_chain.sh`脚本

```
curl -LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.5.0/build_
↪chain.sh && chmod u+x build_chain.sh
```

注解:

- 如果因为网络问题导致长时间无法下载build_chain.sh脚本，请尝试 *curl -LO https://gitee.com/FISCO-BCOS/FISCO-BCOS/raw/master/tools/build_chain.sh && chmod u+x build_chain.sh*

3.1.2 搭建单群组4节点联盟链

在fisco目录下执行下面的指令，生成一条单群组4节点的FISCO链。请确保机器的30300~30303，20200~20203，8545~8548端口没有被占用。

```
bash build_chain.sh -l "127.0.0.1:4" -p 30300,20200,8545 -v 2.5.0
```

注解:

- 其中-p选项指定起始端口，分别是p2p_port,channel_port,jsonrpc_port
- 出于安全性和易用性考虑，v2.3.0版本最新配置将listen_ip拆分成jsonrpc_listen_ip和channel_listen_ip，但仍保留对listen_ip的解析功能，详细请参考 这里
- 为便于开发和体验，channel_listen_ip参考配置是 *0.0.0.0* ，出于安全考虑，请根据实际业务网络情况，修改为安全的监听地址，如：内网IP或特定的外网IP

命令执行成功会输出All completed。如果执行出错，请检查nodes/build.log文件中的错误信息。

```
Checking fisco-bcos binary...
Binary check passed.
============================================================
Generating CA key...
============================================================
Generating keys ...
Processing IP:127.0.0.1 Total:4 Agency:agency Groups:1
============================================================
Generating configurations...
Processing IP:127.0.0.1 Total:4 Agency:agency Groups:1
============================================================
[INFO] Execute the download_console.sh script in directory named by IP to get␣
↪FISCO-BCOS console.
e.g.  bash /home/ubuntu/fisco/nodes/127.0.0.1/download_console.sh
============================================================
[INFO] FISCO-BCOS Path   : bin/fisco-bcos
[INFO] Start Port        : 30300 20200 8545
[INFO] Server IP         : 127.0.0.1:4
[INFO] Output Dir        : /home/ubuntu/fisco/nodes
[INFO] CA Key Path       : /home/ubuntu/fisco/nodes/cert/ca.key
============================================================
[INFO] All completed. Files in /home/ubuntu/fisco/nodes
```

3.1.3 启动FISCO BCOS链

- 启动所有节点

```
bash nodes/127.0.0.1/start_all.sh
```

启动成功会输出类似下面内容的响应。否则请使用netstat -an | grep tcp检查机器的30300~30303，20200~20203，8545~8548端口是否被占用。

```
try to start node0
try to start node1
try to start node2
try to start node3
 node1 start successfully
 node2 start successfully
 node0 start successfully
 node3 start successfully
```

3.1.4 检查进程

- 检查进程是否启动

```
ps -ef | grep -v grep | grep fisco-bcos
```

正常情况会有类似下面的输出； 如果进程数不为4，则进程没有启动（一般是端口被占用导致的）

```
fisco       5453     1  1 17:11 pts/0    00:00:02 /home/ubuntu/fisco/nodes/127.0.0.
↪1/node0/../fisco-bcos -c config.ini
fisco       5459     1  1 17:11 pts/0    00:00:02 /home/ubuntu/fisco/nodes/127.0.0.
↪1/node1/../fisco-bcos -c config.ini
fisco       5464     1  1 17:11 pts/0    00:00:02 /home/ubuntu/fisco/nodes/127.0.0.
↪1/node2/../fisco-bcos -c config.ini
fisco       5476     1  1 17:11 pts/0    00:00:02 /home/ubuntu/fisco/nodes/127.0.0.
↪1/node3/../fisco-bcos -c config.ini
```

3.1.5 检查日志输出

- 如下，查看节点node0链接的节点数

```
tail -f nodes/127.0.0.1/node0/log/log*  | grep connected
```

正常情况会不停地输出链接信息，从输出可以看出node0与另外3个节点有链接。

```
info|2019-01-21 17:30:58.316769| [P2P][Service] heartBeat,connected count=3
info|2019-01-21 17:31:08.316922| [P2P][Service] heartBeat,connected count=3
info|2019-01-21 17:31:18.317105| [P2P][Service] heartBeat,connected count=3
```

- 执行下面指令，检查是否在共识

```
tail -f nodes/127.0.0.1/node0/log/log*  | grep +++
```

正常情况会不停输出++++Generating seal，表示共识正常。

```
info|2019-01-21 17:23:32.576197|␣
↪[g:1][p:264][CONSENSUS][SEALER]++++++++++++++++Generating seal on,blkNum=1,tx=0,
↪myIdx=2,hash=13dcd2da...
info|2019-01-21 17:23:36.592280|␣
↪[g:1][p:264][CONSENSUS][SEALER]++++++++++++++++Generating seal on,blkNum=1,tx=0,
↪myIdx=2,hash=31d21ab7...
info|2019-01-21 17:23:40.612241|␣
↪[g:1][p:264][CONSENSUS][SEALER]++++++++++++++++Generating seal on,blkNum=1,tx=0,
↪myIdx=2,hash=49d0e830...
```

3.2 配置及使用控制台

在控制台通过Web3SDK链接FISCO BCOS节点，实现**查询区块链状态**、**部署调用合约**等功能，能够快

速获取到所需要的信息。 控制台指令详细介绍参考这里。

3.2.1 准备依赖

- Java环境配置

参考Java环境要求。

- 获取控制台并回到fisco目录

```
cd ~/fisco && curl -LO https://github.com/FISCO-BCOS/console/releases/download/v1.
↪0.9/download_console.sh && bash download_console.sh
```

注解:

- 如果因为网络问题导致长时间无法下载，请尝试 *cd ~/fisco && curl -LO https://gitee.com/FISCO-BCOS/console/raw/master/tools/download_console.sh*

- 拷贝控制台配置文件

若节点未采用默认端口，请将文件中的20200替换成节点对应的channle端口。

```
cp -n console/conf/applicationContext-sample.xml console/conf/applicationContext.
↪xml
```

- 配置控制台证书

```
cp nodes/127.0.0.1/sdk/* console/conf/
```

3.2.2 启动控制台

- 启动

```
cd ~/fisco/console && bash start.sh
```

输出下述信息表明启动成功 否则请检查conf/applicationContext.xml中节点端口配置是否正确

```
=====================================================================================
Welcome to FISCO BCOS console(1.0.3)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________  _______   _______   _______   _______        ________   _______   _______␣
↪   _______
|        \|       \ /       \ /       \ /       \       |        \ /       \ /       ␣
↪\  /       \
| $$$$$$$$ \$$$$$$|  $$$$$$\|  $$$$$$\|  $$$$$$\      | $$$$$$$$\|  $$$$$$\|  $$$$$
↪$$\|  $$$$$$\
| $$__       | $$  | $$___\$$| $$   \$$| $$  | $$      | $$__/ $$| $$   \$$| $$  | $
↪$| $$___\$$
| $$  \      | $$   \$$     \ | $$       | $$  | $$      | $$    $$| $$       | $$  | $
↪$ \$$     \
| $$$$$      | $$   _\$$$$$$$\| $$   __ | $$  | $$      | $$$$$$$$\| $$   __ | $$  | $
↪$ _\$$$$$$$\
| $$_____   _| $$_ |  \__| $$| $$__/  \| $$__/ $$      | $$__/ $$| $$__/  \| $$__/ $
↪$|  \__| $$
| $$     \ |   $$ \ \$$    $$ \$$    $$ \$$    $$      | $$    $$ \$$    $$ \$$    $
↪$ \$$    $$
 \$$$$$$$$  \$$$$$$$  \$$$$$$$  \$$$$$$$  \$$$$$$$        \$$$$$$$  \$$$$$$$  \$$$$
↪$   \$$$$$$$
```

(continues on next page)

(续上页)

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

控制台启动失败，参考 附录：JavaSDK启动失败场景

3.2.3 使用控制台获取信息

```
# 获取客户端版本
[group:1]> getNodeVersion
{
    "Build Time":"20200619 06:32:10",
    "Build Type":"Linux/clang/Release",
    "Chain Id":"1",
    "FISCO-BCOS Version":"2.5.0",
    "Git Branch":"HEAD",
    "Git Commit Hash":"72c6d770e5cf0f4197162d0e26005ec03d30fcfe",
    "Supported Version":"2.5.0"
}
# 获取节点链接信息
[group:1]> getPeers
[
    {
        "IPAndPort":"127.0.0.1:49948",
        "NodeID":
↪"b5872eff0569903d71330ab7bc85c5a8be03e80b70746ec33cafe27cc4f6f8a71f8c84fd8af9d7912cb5ba068901fe4
↪",
        "Topic":[]
    },
    {
        "IPAndPort":"127.0.0.1:49940",
        "NodeID":
↪"912126291183b673c537153cf19bf5512d5355d8edea7864496c257630d01103d89ae26d17740daebdd20cbc645c9a5
↪",
        "Topic":[]
    },
    {
        "IPAndPort":"127.0.0.1:49932",
        "NodeID":
↪"db75ab16ed7afa966447c403ca2587853237b0d9f942ba6fa551dc67ed6822d88da01a1e4da9b51aedafb8c64e9d208
↪",
        "Topic":[]
    }
]
```

3.3 部署及调用HelloWorld合约

3.3.1 HelloWorld合约

HelloWorld合约提供两个接口，分别是`get()`和`set()`，用于获取/设置合约变量`name`。合约内容如下:

```
pragma solidity ^0.4.24;

contract HelloWorld {
    string name;

    function HelloWorld() {
```

(continues on next page)

(续上页)

```
        name = "Hello, World!";
    }

    function get()constant returns(string) {
        return name;
    }

    function set(string n) {
        name = n;
    }
}
```

3.3.2 部署HelloWorld合约

为了方便用户快速体验，HelloWorld合约已经内置于控制台中，位于控制台目录下contracts/solidity/HelloWorld.sol，参考下面命令部署即可。

```
# 在控制台输入以下指令 部署成功则返回合约地址
[group:1]> deploy HelloWorld
contract address:0xb3c223fc0bf6646959f254ac4e4a7e355b50a344
```

3.3.3 调用HelloWorld合约

```
# 查看当前块高
[group:1]> getBlockNumber
1

# 调用get接口获取name变量 此处的合约地址是deploy指令返回的地址
[group:1]> call HelloWorld 0xb3c223fc0bf6646959f254ac4e4a7e355b50a344 get
Hello, World!

# 查看当前块高，块高不变，因为get接口不更改账本状态
[group:1]> getBlockNumber
1

# 调用set设置name
[group:1]> call HelloWorld 0xb3c223fc0bf6646959f254ac4e4a7e355b50a344 set "Hello,␣
↪FISCO BCOS"
0x21dca087cb3e44f44f9b882071ec6ecfcb500361cad36a52d39900ea359d0895

# 再次查看当前块高，块高增加表示已出块，账本状态已更改
[group:1]> getBlockNumber
2

# 调用get接口获取name变量，检查设置是否生效
[group:1]> call HelloWorld 0xb3c223fc0bf6646959f254ac4e4a7e355b50a344 get
Hello, FISCO BCOS

# 退出控制台
[group:1]> quit
```

注：

1. 部署合约还可以通过deployByCNS命令，可以指定部署的合约版本号，使用方式参考这里。
2. 调用合约通过callByCNS命令，使用方式参考这里。

CHAPTER 4

教程

本章将介绍使用FISCO BCOS快速上手开发DApp的基本流程和相关的核心概念。 同时，我们还提供了便于企业用户开发部署的工具包的使用指南。

4.1 关键概念

区块链是由多个学科交叉组合形成的一门技术，本章将阐述区块链相关的基本概念，对涉及的基本理论进行科普介绍。如果您已经对这些基本技术很熟悉，可以跳过本章。

4.1.1 区块链是什么

区块链（blockchain）是在比特币之后提出的一个概念，在中本聪关于比特币的论文中没有直接引入blockchain的概念，而是以chain of block来描述一种数据结构。

Chain of block是指由多个区块通过哈希（hash）串联成一条链式结构的数据组织方式。区块链则是采用多项技术交叉组合，维护管理这个chain of block数据结构，形成一个不可篡改的分布式账本的综合技术领域。

区块链技术是一种在对等网络环境下，通过透明和可信规则，构建不可伪造、难以篡改和可追溯的块链式数据结构，实现和管理可信数据的产生、存取和使用的模式。技术架构上，区块链是由分布式架构与分布式存储、块链式数据结构、点对点网络、共识算法、密码学算法、博弈论、智能合约等多种信息技术共同组成的整体解决方案。

区块链技术和生态起源于比特币，随着金融、司法、供应链、文化娱乐、社会管理、物联网等更多行业对此领域技术的关注，希望将其技术价值应用到更广泛的分布式协作中，区块链技术和产品模式也在持续进化，FISCO BCOS区块链底层平台在区块链技术基础上，专注提升安全、性能、可用性、易用性、隐私保护、合规监管等方面的能力，和业界生态共同发展，体现多方参与、智能协同、专业分工、价值分享的效能。

账本

账本顾名思义，用于管理账户、交易流水等数据，支持分类记账、对账、清结算等功能。在多方合作中，多个参与方希望共同维护和共享一份及时、正确、安全的分布式账本，以消除信息不对称，提升运作效率，保证资金和业务安全。而区块链通常被认为是用于构建“分布式共享账本”的一种核心技术，通

过链式的区块数据结构、多方共识机制、智能合约、世界状态存储等一系列技术的共同作用，可实现一致、可信、事务安全、难以篡改可追溯的共享账本。

账本里包含的基本内容有区块，交易，账户，世界状态。

区块

区块是按时间次序构建的数据结构，区块链的第一个区块称为“创世块”（genesis block），后续生成的区块用“高度”标识，每个区块高度逐一递增，新区块都会引入前一个区块的hash信息，再用hash算法和本区块的数据生成唯一的数据指纹，从而形成环环相扣的块链状结构，称为“Blockchain”也即区块链。精巧的数据结构设计，使得链上数据按发生时间保存，可追溯可验证，如果修改任何一个区块里的任意一个数据，都会导致整个块链验证不通过，从而篡改的成本会很高。

一个区块的基本数据结构是区块头和区块体，区块头包含区块高度，hash、出块者签名、状态树根等一些基本信息，区块体里包含一批交易数据列表已经相关的回执信息，根据交易列表的大小，整个区块的大小会有所不同，考虑到网络传播等因素，一般不会太大，在1M~几M字节之间。

交易

交易可认为是一段发往区块链系统的请求数据，用于部署合约，调用合约接口，维护合约的生命周期，以及管理资产，进行价值交换等，交易的基本数据结构包括发送者，接受者，交易数据等。用户可以构建一个交易，用自己的私钥给交易签名，发送到链上（通过sendRawTransaction等接口），由多个节点的共识机制处理，执行相关的智能合约代码，生成交易指定的状态数据，然后将交易打包到区块里，和状态数据一起落盘存储，该交易即为被确认，被确认的交易被认为具备了事务性和一致性。

随着交易确认相应还会有交易回执（receipt）产生，和交易一一对应且保存在区块里，用于保存一些交易执行过程生成的信息如结果码、日志、消耗的gas量等。用户可以使用交易hash检查交易回执，判定交易是否完成。

和“写操作”的交易对应，还有一种”只读”调用方式，用于读取链上数据，节点收到请求后会根据请求的参数访问状态信息并返回，并不会将请求加入共识流程，也不会导致修改链上的数据。

账户

在采用账户模型设计的区块链系统里，账户这个术语代表着用户、智能合约的唯一性存在。

在采用公私钥体系的区块链系统里，用户创建一个公私钥对，经过hash等算法换算即得到一个唯一性的地址串，代表这个用户的账户，用户用该私钥管理这个账户里的资产。用户账户在链上不一定有对应的存储空间，而是由智能合约管理用户在链上的数据，因此这种用户账户也会被称为“外部账户”。

对智能合约来说，一个智能合约被部署后，在链上就有了一个唯一的地址，也称为合约账户，指向这个合约的状态位、二进制代码、相关状态数据的索引等。智能合约运行过程中，会通过这个地址加载二进制代码，根据状态数据索引去访问世界状态存储里对应的数据，根据运行结果将数据写入世界状态存储，更新合约账户里的状态数据索引。智能合约被注销时，主要是更新合约账户里的合约状态位，将其置为无效，一般不会直接清除该合约账户的实际数据。

世界状态

FISCO BCOS采用“账户模型”的设计，即除了区块和交易的存储空间外，还会有一块保存智能合约运行结果的存储空间。智能合约执行过程产生的状态数据，经过共识机制确认，分布式的保存在各节点上，数据全局一致，可验证难篡改，所以称为“世界状态”。

状态存储空间的存在，使得区块链上可以保存各种丰富的数据，包括用户账户信息如余额等，智能合约二进制码，智能合约运行结果等相关的各种数据，智能合约执行过程中会从状态存储中获取一些数据参与运算，为实现复杂的合约逻辑提供了基础。

另一方面，维护状态数据需要付出不少存储成本，随着链的持续运行，状态数据会持续膨胀，如采用复杂的数据结构如帕特里夏树（Patricia Tree），状态数据的的容量会进一步扩大，根据不同的场景需要，可对状态数据进行裁剪优化，或采用分布式数据仓库等方案存储，以支持更海量的状态数据容量。

共识机制

共识机制是区块链领域的核心概念，无共识，不区块链。区块链作为一个分布式系统，可以由不同的节点共同参与计算、共同见证交易的执行过程，并确认最终计算结果。协同这些松散耦合、互不信任的参与者达成信任关系，并保障一致性，持续性协作的过程，可以抽象为“共识”过程，所牵涉的算法和策略统称为共识机制。

节点

安装了区块链系统所需软硬件，加入到区块链网络里的计算机，可以称为一个“节点”。节点参与到区块链系统的网络通信、逻辑运算、数据验证，验证和保存区块、交易、状态等数据，并对客户端提供交易处理和数据查询的接口。节点的标识采用公私钥机制，生成一串唯一的NodeID，以保证它在网络上的唯一性。

根据对计算的参与程度和数据的存量，节点可分为共识节点和观察节点。共识节点会参与到整个共识过程，做为记账者打包区块、做为验证者验证区块以完成共识过程。观察节点不参与共识，同步数据，进行验证并保存，可以做为数据服务者提供服务。

共识算法

共识算法需要解决的几个核心问题是：

1. 选出在整个系统中具有记账权的角色，做为leader发起一次记账。
2. 参与者采用不可否认和不能篡改的算法，进行多层面验证后，采纳Leader给出的记账。
3. 通过数据同步和分布式一致性协作，保证所有参与者最终收到的结果都是一致的，无错的。

区块链领域常见的共识算法有公链常用的工作量证明（Proof of Work）,权益证明（Proof of Stake），委托权益证明（Delegated Proof of Stake），以及联盟链常用的实用性拜占庭容错共识PBFT（Practical Byzantine Fault Tolerance），Raft等，另外一些前沿性的共识算法通常是将随机数发生器和上述几个共识算法进行有机组合，以改善安全、能耗以及性能和规模问题。

FISCO BCOS共识模块采用插件化的设计，可支持多种共识算法，当前包括PBFT和Raft，后续将会持续实现更大规模，速度更快的共识算法。

智能合约

智能合约概念于1995年由Nick Szabo首次提出，指以数字形式定义的能自动执行条款的合约，数字形式意味着合约必须用计算机代码实现，因为只要参与方达成协定，智能合约建立的权利和义务，就会被自动执行，且结果不能被否认。

FISCO BCOS运用智能合约不仅用于资产管理、规则定义和价值交换，还可以用来进行全局配置、运维治理、权限管理等。

智能合约生命周期

智能合约的生命周期为设计，开发,测试,部署,运行,升级,销毁等几个步骤。

开发人员根据需求，进行智能合约代码的编写，编译，单元测试。合约开发语言可包括solidity,C++,java,go,javascript,rust等，语言的选择根据平台虚拟机选型而定。在合约通过测试后，采用部署指令发布到链上，经过共识算法确认后，合约生效并被后续的交易调用。

当合约需要更新升级时，重复以上开发到部署的步骤，发布新版合约，新版合约会有一个新的地址和独立的存储空间，并不是覆盖掉旧合约。新版合约可通过旧合约数据接口访问旧版本合约里保存的数据，或者通过数据迁移的方式将旧合约的数据迁移到新合约的存储里，最佳实践是设计执行流程的“行为合约”和保存数据的“数据合约”，将数据和合约解耦，当业务流程产生改变，而业务数据本身没有改变时，新行为合约直接访问原有的“数据合约”即可。

销毁一个旧合约并不意味着清除合约的所有数据，只是将其状态置为“无效”，该合约则不可再被调用。

智能合约虚拟机

为了运行数字智能合约，区块链系统必须具备可编译、解析、执行计算机代码的编译器和执行器，统称为虚拟机体系。合约编写完毕后，用编译器编译，发送部署交易将合约部署到区块链系统上，部署交易共识通过后，系统给合约分配一个唯一地址和保存合约的二进制代码，当某个合约被另一个交易调用后，虚拟机执行器从合约存储里加载代码并执行，并输出执行结果。

在强调安全性、事务性和一致性的区块链系统里，虚拟机应具有沙盒特征，屏蔽类似随机数、系统时间、外部文件系统、网络等可能导致不确定性的因素，且可以抵抗恶意代码的侵入，以保证在不同节点上同一个交易和同一个合约的执行生成的结果是一致的，执行过程是安全的。

当前流行的虚拟机机制包括EVM， 受控的Docker，WebAssembly等，FISCO BCOS的虚拟机模块采用模块化设计，已经支持受到社区广泛欢迎的EVM，将会支持更多的虚拟机。

图灵完备

图灵机和图灵完备是计算机领域的经典概念，由数学家艾伦·麦席森·图灵（1912～1954）提出的一种抽象计算模型，引申到区块链领域，主要指合约支持判断、跳转、循环、递归等逻辑运算，支持多种数据类型如整形、字符串、结构体的数据处理能力，甚至有一定的面向对象特性如继承、派生、接口等，这样才能支持复杂的业务逻辑和完备的契约执行，与只支持栈操作的简单脚本进行区分。

2014年后出现的区块链大多支持图灵完备的智能合约，使得区块链系统具备更高的可编程性，在区块链既有的基本特性（如多方共识，难以篡改，可追溯等，安全性等）基础上，还可以实现具有一定业务逻辑的业务契约，如李嘉图合约（The Ricardian Contract），也可以使用智能合约来实现。

合约的执行还需要处理“停机问题”，即判断程序是否会在有限的时间之内解决输入的问题，并结束执行，释放资源。想象一下，一个合约在全网部署，在被调用时在每个节点上都会执行，如果这个合约是个无限循环，就意味着可能会耗尽整个体系的资源。所以停机问题的处理也是区块链领域里图灵完备计算体系的一个重要关注点。

4.1.2 联盟链概念分析

行业里通常将区块链的类型分为公有链，联盟链，私有链。公有链指所有人都可以随时随地参与甚至是匿名参与的链；私有链指一个主体（如一个机构或一个自然人）所有，私有化的管理和使用的链；联盟链通常是指多个主体达成一定的协议，或建立了一个业务联盟后，多方共同组建的链，加入联盟链的成员需要经过验证，一般是身份可知的。正因为有准入机制，所以联盟链也通常被称为“许可链”。

因为联盟链从组建、加入、运营、交易等环节有准入和身份管理，在链上的操作可以用权限进行管控，共识方面一般采用PBFT等基于多方多轮验证投票的共识机制，不采用POW挖矿的高能耗机制，网络规模相对可控，在交易时延性、事务一致性和确定性、并发和容量方面都可以进行大幅的优化。

联盟链在继承区块链技术的优势的同时，更适合性能容量要求高，强调监管、合规的敏感业务场景，如金融、司法、以及大量和实体经济相关的业务。联盟链的路线，兼顾了业务合规稳定和业务创新，也是国家和行业鼓励发展的方向。

性能

性能指标

软件系统的处理性能指标最常见的是TPS（Transaction Per Second），即系统每秒能处理和确认的交易数，TPS越高，性能越高。区块链领域的性能指标除了TPS之外，还有确认时延，网络规模大小等。

确认时延是指交易发送到区块链网络后，经过验证、运算和共识等一系列流程后，到被确认时所用的时间，如比特币网络一个区块是10分钟，交易被大概率确认需要6个区块，即一个小时。采用PBFT算法的话，可以使交易在秒级确认，一旦确认即具有最终确定性，更适合金融等业务需求。

网络规模指在保证一定的TPS和确认时延前提下，能支持多少共识节点的协同工作。业界一般认为采用PBFT共识算法的系统，节点规模在百级左右，再增加就会导致TPS下降，确认时延增加。目前业界有通过随机数算法选择记账组的共识机制，可以改善这个问题。

性能优化

性能的优化有两个方向,向上扩展（Scale up）和平行扩展（Scale out）。向上扩展指在有限的资源基础上优化软硬件配置，极大提升处理能力，如采用更有效率的算法，采用硬件加速等。平行扩展指系统架构具有良好的可扩展性，可以采用分片、分区的方式承载不同的用户、业务流的处理，只要适当增加软硬件资源，就能承载更多的请求。

性能指标和软件架构，硬件配置如CPU、内存、存储规格、网络带宽都密切相关，且随着TPS的增加，对存储容量的压力也会相应增加，需要综合考虑。

安全性

安全性是个很大的话题，尤其是构建在分布式网络上多方参与的区块链系统。在系统层面，需要关注网络攻击、系统渗透、数据破坏和泄漏的问题，在业务层面需要关注越权操作、逻辑错误、系统稳定性造成的资产损失、隐私被侵害等问题。

安全性的保障要关注”木桶的短板“，需要有综合性的防护策略，提供多层面，全面的安全防护，满足高要求的安全标准，并提供安全方面的最佳实践，对齐所有参与者的安全级别，保障全网安全。

准入机制

准入机制指在无论是机构还是个人组建和加入链之前，需要满足身份可知、资质可信，技术可靠的标准，主体信息由多方共同审核后，才会启动联盟链组建工作，然后将经过审核的主体的节点加入到网络，为经过审核的人员分配可发送交易的公私钥。 在准入完成后，机构、节点、人员的信息都会登记到链上或可靠的信息服务里，链上的一切行为都可以追溯到机构和人。

权限控制

联盟链上权限控制即不同人员对各种敏感级别的数据读写的控制，细分可以罗列出如合约部署、合约内数据访问、区块数据同步、系统参数访问和修改、节点启停等不同的权限，根据业务需要，还可以加入更多的权限控制点。

权限是分配给角色的，可沿用典型的基于角色的权限访问控制（Role-Based Access Control）设计，一个参考设计是将角色分为运营管理者，交易操作员，应用开发者，运维管理者，监管方，每个角色还可以根据需要细分层级，完备的模型可能会很庞大复杂，可以根据场景需要进行适当的设计，能达到业务安全可控的程度即可。

隐私保护

基于区块链架构的业务场景要求各参与方都输出和共享相关数据，以共同计算和验证，在复杂的商业环境中，机构希望自己的商业数据受控，在越来越被重视的个人数据隐私保护的形势下，个人对隐私保护

的诉求也日益增强。如何对共享的数据牵涉隐私的部分进行保护，以及在避免运作过程泄漏隐私，是一个很重要的问题。

隐私保护首先是个管理问题，要求在构建系统开展业务时，把握“最小授权，明示同意的原则”，对数据的收集、存储、应用、披露、删除、恢复全生命周期进行管理，建立日常管理和应急管理制度，在高敏感业务场景设定监管角色，引入第三方检视和审计，从事先事中事后全环节进行管控。

在技术上，可以采用数据脱敏，业务隔离或者系统物理隔离等方式控制数据分发范围，同时也可以引入密码学方法如零知识证明、安全多方计算、环签名、群签名、盲签名等，对数据进行高强度的加密保护。

物理隔离

这个概念主要用于隐私保护领域，“物理隔离”是避免隐私数据泄露的彻底手段，物理隔离指只有共享数据的参与者在网络通信层互通，不参与共享数据的参与者在网络互相都不能通信，不交换哪怕一个字节的数据。

相对而言的是逻辑隔离，参与者可以接收到和自己无关的数据，但数据本身带上权限控制或加密保护，使得没有授权或密钥的参与者不能访问和修改。但随着技术的发展，所受到的权限受控数据或加密数据在若干年后依旧有可能被破解。

对极高敏感性的数据，可以采用“物理隔离”的策略，从根源上杜绝被破解的可能性。相应的成本是需要仔细甄别数据的敏感级别，对隔离策略进行周密的规划，并分配足够的硬件资源承载不同的数据。

治理与监管

联盟链治理

联盟链治理牵涉到多参与方协调工作，激励机制，安全运营，监管审计等一系列的问题，核心是理清各参与方的责权利，工作流程，构建顺畅的开发和运维体系，以及保障业务的合法合规，对包括安全性在内的问题能事先防范事后应急处理。为达成治理，需要制定相关的规则且保证各参与方达成共识并贯彻执行。

一个典型的联盟链治理参考模型是各参与方共同组建联盟链委员会，共同讨论和决议，根据场景需要设定各种角色和分配任务，如某些机构负责开发，某些机构参与运营管理，所有机构参与交易和运维，采用智能合约实现管理规则和维护系统数据，委员会和监管机构可掌握一定的管理权限，对业务、机构、人员进行审核和设置，并在出现紧急情况时，根据事先约定的流程，通过共识过的智能合约规则，进行应急操作，如账户重置，业务调整等，在需要进行系统升级时，委员会负责协调各方进行系统更新。

在具备完善治理机制的联盟链上，各参与方根据规则进行点对点的对等合作，包括资产交易、数据交换，极大程度提升运作效率，促进业务创新，同时合规性和安全性等方面也得到了保障。

快速部署

构建一个区块链系统的大致步骤包括：获取硬件资源包括服务器、网络、内存、硬盘存储等，进行环境配置包括选择指定操作系统、开通网络端口和相关策略、带宽规划、存储空间分配等，获取区块链二进制可运行软件或者从源码进行编译，然后进行区块链系统的配置，包括创世块配置、运行时参数配置，日志配置等，进行多方互联配置，包括节点准入配置、端口发现、共识参与方列表等，客户端和开发者工具配置，包括控制台、SDK等，这个过程会包括许多细节，如各种证书和公私钥的管理等，很容易出现环境、版本、配置的差错，导致整个过程复杂、繁琐和反复，形成了较高的使用门槛。

如何将以上步骤简化和加速，使构建和组链过程变得简便，快速，不容易出错，且低成本，需要从以下几方面进行考虑： 首先，标准化目标部署平台，事先将操作系统、依赖软件列表、网络带宽和存储容量、网络策略等关键的软硬件准备好，对齐版本和参数，使得平台可用，依赖完备。当下流行的云服务，docker等方式都可以帮助构建这样的标准化平台。

然后，从使用者的视角出发，优化区块链软件的构建、配置和组链流程，提供快速构建，自动组链的工具，使得使用者不需要关注诸多细节，简单的几步操作即可运行起供开发调试、上线运行的链。

FISCO BCOS非常重视使用者的部署体验，提供了一键部署的命令行，帮助开发者快速搭建开发调试环境，提供企业级搭链工具，面向多机构联合组链的场景，灵活的进行主机、网络等参数配置，管理相关的证书，便于多个企业之间协同工作。经过快速部署的优化，将使用者搭起区块链的时间缩短到几分钟到半小时以内。

数据治理

区块链强调数据层层验证，历史记录可追溯，常见的方案是从创世块以来，所有的数据都会保存在所有的参与节点上（轻节点之外），导致的结果是数据膨胀，容量紧张，尤其是在承载海量服务的场景里，在一定时间之后，一般的存储方案已经无法容纳数据，而海量存储成本很高，另一个角度是安全性，全量数据永久保存，可能面临历史数据泄露的风险，所以需要在数据治理方面进行设计。

数据治理主要是几个策略：裁剪迁移，平行扩容，分布式存储。如何选择需要结合场景分析。

对具有较强时间特征的数据，如某业务的清结算周期是一个星期，那么一个星期前的数据不需要参与在线计算和验证，旧的数据则可以从节点迁移到大数据存储里，满足数据可查询可验证的需求以及业务保存年限的要求，线上节点的数据压力大幅降低，历史数据离线保存，在安全策略上也可以进行更严密的保护。

对规模持续扩大的业务，如用户数或合同存证量剧增，可以针对不同的用户和合同，分配到不同的逻辑分区，每个逻辑分区有独立的存储空间，只承载一定容量的数据，当接近容量的上限，则再分配更多资源容纳新的数据。分区的的设计使得在资源调配，成本管理上都更容易把控。

结合数据裁剪迁移和平行扩容，数据的容量成本，安全级别都得到很好的控制，便于开展海量规模的业务。

运维监控

区块链系统从构建和运行逻辑上都具有较高一致性，不同节点的软硬件系统基本一致。其标准化特性给运维人员带来了便利，可使用通用的工具、运维策略和运维流程等对区块链系统进行构建、部署、配置、故障处理，从而降低运维成本以及提升效率。

运维人员对联盟链的操作会被权限系统控制，运维人员有修改系统配置、启停进程、查看运行日志、排查故障等权限，但不参与到业务交易中，也不能直接查看具有较高安全隐私等级的用户数据，交易数据。

系统运行过程中，可通过监控系统对各种运行指标进行监控，对系统的健康程度进行评估，当出现故障时发出告警通知，便于运维快速反应，进行处理。

监控的维度包括基础环境监控,如CPU占比、系统内存占比和增长、磁盘IO情况、网络连接数和流量等。

区块链系统监控包括如区块高度、交易量和虚拟机计算量，共识节点出块投票情况等。

接口监控包括如接口调用计数、接口调用耗时情况、接口调用成功率等。

监控数据可以通过日志或网络接口进行输出，便于和机构的现有的监控系统进行对接，复用机构的监控能力和既有的运维流程。运维人员收到告警后，采用联盟链提供的运维工具，查看系统信息、修改配置、启停进程、处理故障等。

监管审计

随着区块链技术和业务形态探索的发展，需要在区块链技术平台上提供支持监管的功能，避免区块链系统游离于法律法规以及行业规则之外，成为洗钱、非法融资或犯罪交易的载体。

审计功能主要用于满足区块链系统的审计内控、责任鉴定和事件追溯等要求，需要以有效的技术手段，配合业务所属的行业标准进行精确的审计管理。

监管者可以做为节点接入到区块链系统里，或者通过接口和区块链系统进行交互，监管者可同步到所有的数据进行审计分析，跟踪全局的业务流程，如发现异常，可以向区块链发出具备监管权限的指令，对业务、参与人、账户等进行管控，实现“穿透式监管”。

FISCO BCOS在角色和权限设计，功能接口，审计工具等方面都对监管审计进行了支持。

4.2 构建第一个区块链应用

本章将会介绍一个基于FISCO BCOS区块链的业务应用场景开发全过程，从业务场景分析，到合约的设计实现，然后介绍合约编译以及如何部署到区块链，最后介绍一个应用模块的实现，通过我们提供的Web3SDK实现对区块链上合约的调用访问。

本教程要求用户熟悉Linux操作环境，具备Java开发的基本技能，能够使用Gradle工具，熟悉Solidity语法。

通过学习教程，你将会了解到以下内容：

1. 如何将一个业务场景的逻辑用合约的形式表达
2. 如何将Solidity合约转化成Java类
3. 如何配置Web3SDK
4. 如何构建一个应用，并集成Web3SDK到应用工程
5. 如何通过Web3SDK调用合约接口，了解Web3SDK调用合约接口的原理

教程中会提供示例的完整项目源码，用户可以在此基础上快速开发自己的应用。

重要： 请参考 安装文档 完成FISCO BCOS区块链的搭建和控制台的下载工作，本教程中的操作假设在该文档搭建的环境下进行。

4.2.1 示例应用需求

区块链天然具有防篡改，可追溯等特性，这些特性决定其更容易受金融领域的青睐，本文将会提供一个简易的资产管理的开发示例，并最终实现以下功能：

- 能够在区块链上进行资产注册
- 能够实现不同账户的转账
- 可以查询账户的资产金额

4.2.2 合约设计与实现

在区块链上进行应用开发时，结合业务需求，首先需要设计对应的智能合约，确定合约需要储存的数据，在此基础上确定智能合约对外提供的接口，最后给出各个接口的具体实现。

存储设计

FISCO BCOS提供合约CRUD接口开发模式，可以通过合约创建表，并对创建的表进行增删改查操作。针对本应用需要设计一个存储资产管理的表`t_asset`，该表字段如下：

- account: 主键，资产账户(string类型)
- asset_value: 资产金额(uint256类型)

其中account是主键，即操作`t_asset`表时需要传入的字段，区块链根据该主键字段查询表中匹配的记录。`t_asset`表示例如下：

接口设计

按照业务的设计目标，需要实现资产注册，转账，查询功能，对应功能的接口如下：

```
// 查询资产金额
function select(string account) public constant returns(int256, uint256)
// 资产注册
function register(string account, uint256 amount) public returns(int256)
// 资产转移
function transfer(string from_asset_account, string to_asset_account, uint256␣
↪amount) public returns(int256)
```

完整源码

```
pragma solidity ^0.4.24;

import "./Table.sol";

contract Asset {
    // event
    event RegisterEvent(int256 ret, string account, uint256 asset_value);
    event TransferEvent(int256 ret, string from_account, string to_account,␣
↪uint256 amount);

    constructor() public {
        // 构造函数中创建t_asset表
        createTable();
    }

    function createTable() private {
        TableFactory tf = TableFactory(0x1001);
        // 资产管理表, key : account, field : asset_value
        // |  资产账户(主键)      |     资产金额       |
        // |-------------------- |-------------------|
        // |        account      |    asset_value    |
        // |---------------------|-------------------|
        //
        // 创建表
        tf.createTable("t_asset", "account", "asset_value");
    }

    function openTable() private returns(Table) {
        TableFactory tf = TableFactory(0x1001);
        Table table = tf.openTable("t_asset");
        return table;
    }

    /*
    描述 : 根据资产账户查询资产金额
    参数 :
            account : 资产账户

    返回值:
            参数一: 成功返回0, 账户不存在返回-1
            参数二: 第一个参数为0时有效，资产金额
    */
    function select(string account) public constant returns(int256, uint256) {
        // 打开表
        Table table = openTable();
        // 查询
```

(continues on next page)

(续上页)

```
        Entries entries = table.select(account, table.newCondition());
        uint256 asset_value = 0;
        if (0 == uint256(entries.size())) {
            return (-1, asset_value);
        } else {
            Entry entry = entries.get(0);
            return (0, uint256(entry.getInt("asset_value")));
        }
    }

    /*
    描述 : 资产注册
    参数 :
            account : 资产账户
            amount  : 资产金额
    返回值:
            0  资产注册成功
            -1 资产账户已存在
            -2 其他错误
    */
    function register(string account, uint256 asset_value) public returns(int256){
        int256 ret_code = 0;
        int256 ret= 0;
        uint256 temp_asset_value = 0;
        // 查询账户是否存在
        (ret, temp_asset_value) = select(account);
        if(ret != 0) {
            Table table = openTable();

            Entry entry = table.newEntry();
            entry.set("account", account);
            entry.set("asset_value", int256(asset_value));
            // 插入
            int count = table.insert(account, entry);
            if (count == 1) {
                // 成功
                ret_code = 0;
            } else {
                // 失败? 无权限或者其他错误
                ret_code = -2;
            }
        } else {
            // 账户已存在
            ret_code = -1;
        }

        emit RegisterEvent(ret_code, account, asset_value);

        return ret_code;
    }

    /*
    描述 : 资产转移
    参数 :
            from_account : 转移资产账户
            to_account :  接收资产账户
            amount :  转移金额
    返回值:
            0  资产转移成功
            -1 转移资产账户不存在
            -2 接收资产账户不存在
```

(continues on next page)

(续上页)

```
            -3 金额不足
            -4 金额溢出
            -5 其他错误
    */
    function transfer(string from_account, string to_account, uint256 amount)␣
↪public returns(int256) {
        // 查询转移资产账户信息
        int ret_code = 0;
        int256 ret = 0;
        uint256 from_asset_value = 0;
        uint256 to_asset_value = 0;

        // 转移账户是否存在?
        (ret, from_asset_value) = select(from_account);
        if(ret != 0) {
            ret_code = -1;
            // 转移账户不存在
            emit TransferEvent(ret_code, from_account, to_account, amount);
            return ret_code;

        }

        // 接受账户是否存在?
        (ret, to_asset_value) = select(to_account);
        if(ret != 0) {
            ret_code = -2;
            // 接收资产的账户不存在
            emit TransferEvent(ret_code, from_account, to_account, amount);
            return ret_code;
        }

        if(from_asset_value < amount) {
            ret_code = -3;
            // 转移资产的账户金额不足
            emit TransferEvent(ret_code, from_account, to_account, amount);
            return ret_code;
        }

        if (to_asset_value + amount < to_asset_value) {
            ret_code = -4;
            // 接收账户金额溢出
            emit TransferEvent(ret_code, from_account, to_account, amount);
            return ret_code;
        }

        Table table = openTable();

        Entry entry0 = table.newEntry();
        entry0.set("account", from_account);
        entry0.set("asset_value", int256(from_asset_value - amount));
        // 更新转账账户
        int count = table.update(from_account, entry0, table.newCondition());
        if(count != 1) {
            ret_code = -5;
            // 失败? 无权限或者其他错误?
            emit TransferEvent(ret_code, from_account, to_account, amount);
            return ret_code;
        }

        Entry entry1 = table.newEntry();
        entry1.set("account", to_account);
```

(continues on next page)

(续上页)

```
        entry1.set("asset_value", int256(to_asset_value + amount));
        // 更新接收账户
        table.update(to_account, entry1, table.newCondition());

        emit TransferEvent(ret_code, from_account, to_account, amount);

        return ret_code;
    }
}
```

注： Asset.sol合约的实现需要引入FISCO BCOS提供的一个系统合约接口文件 Table.sol ，该系统合约文件中的接口由FISCO BCOS底层实现。当业务合约需要操作CRUD接口时，均需要引入该接口合约文件。Table.sol 合约详细接口参考这里。

4.2.3 合约编译

上一小节，我们根据业务需求设计了合约Asset.sol的存储与接口，给出了完整实现，但是Java程序无法直接调用Solidity合约，需要先将Solidity合约文件编译为Java文件。

控制台提供了编译工具，可以将Asset.sol合约文件存放在console/contracts/solidity目录。利用console目录下提供的sol2java.sh脚本进行编译，操作如下：

```
# 切换到fisco/console/目录
$ cd ~/fisco/console/
# 编译合约，后面指定一个Java的包名参数，可以根据实际项目路径指定包名
$ ./sol2java.sh org.fisco.bcos.asset.contract
```

运行成功之后，将会在console/contracts/sdk目录生成java、abi和bin目录，如下所示。

```
|-- abi # 生成的abi目录，存放solidity合约编译生成的abi文件
|   |-- Asset.abi
|   |-- Table.abi
|-- bin # 生成的bin目录，存放solidity合约编译生成的bin文件
|   |-- Asset.bin
|   |-- Table.bin
|-- contracts # 存放solidity合约源码文件，将需要编译的合约拷贝到该目录下
|   |-- Asset.sol # 拷贝进来的Asset.sol合约，依赖Table.sol
|   |-- Table.sol # 实现系统CRUD操作的合约接口文件
|-- java  # 存放编译的包路径及Java合约文件
|   |-- org
|        |--fisco
|             |--bcos
|                  |--asset
|                       |--contract
|                            |--Asset.java  # Asset.sol合约生成的Java文件
|                            |--Table.java  # Table.sol合约生成的Java文件
|-- sol2java.sh
```

java目录下生成了org/fisco/bcos/asset/contract/包路径目录，该目录下包含Asset.java和Table.java两个文件，其中Asset.java是Java应用调用Asset.sol合约需要的文件。

Asset.java的主要接口：

```
package org.fisco.bcos.asset.contract;

public class Asset extends Contract {
    // Asset.sol合约 transfer接口生成
    public RemoteCall<TransactionReceipt> transfer(String from_account, String to_
↪account, BigInteger amount);
    // Asset.sol合约 register接口生成
```

(continues on next page)

(续上页)

```
    public RemoteCall<TransactionReceipt> register(String account, BigInteger␣
↪asset_value);
    // Asset.sol合约 select接口生成
    public RemoteCall<Tuple2<BigInteger, BigInteger>> select(String account);

    // 加载Asset合约地址，生成Asset对象
    public static Asset load(String contractAddress, Web3j web3j, Credentials␣
↪credentials, ContractGasProvider contractGasProvider);

    // 部署Assert.sol合约，生成Asset对象
    public static RemoteCall<Asset> deploy(Web3j web3j, Credentials credentials,␣
↪ContractGasProvider contractGasProvider);
}
```

其中load与deploy函数用于构造Asset对象，其他接口分别用来调用对应的solidity合约的接口，详细使用在下文会有介绍。

4.2.4 SDK配置

我们提供了一个Java工程项目供开发使用，首先获取Java工程项目：

```
    # 获取Java工程项目压缩包
    $ cd ~
    $ curl -LO https://github.com/FISCO-BCOS/LargeFiles/raw/master/tools/asset-app.
↪tar.gz
    # 解压得到Java工程项目asset-app目录
    $ tar -zxf asset-app.tar.gz
```

注解:

- 如果因为网络问题导致长时间无法下载，请尝试 *curl -LO https://www.fisco.com.cn/cdn/deps/tools/asset-app.tar.gz*

asset-app项目的目录结构如下：

```
|-- build.gradle // gradle配置文件
|-- gradle
|   |-- wrapper
|       |-- gradle-wrapper.jar // 用于下载Gradle的相关代码实现
|       |-- gradle-wrapper.properties // wrapper所使用的配置信息，比如gradle的版本等信息
|-- gradlew // Linux或者Unix下用于执行wrapper命令的Shell脚本
|-- gradlew.bat // Windows下用于执行wrapper命令的批处理脚本
|-- src
|   |-- main
|   |   |-- java
|   |         |-- org
|   |             |-- fisco
|   |                   |-- bcos
|   |                         |-- asset
|   |                               |-- client // 放置客户端调用类
|   |                                     |-- AssetClient.java
|   |                               |-- contract // 放置Java合约类
|   |                                     |-- Asset.java
|   |-- test
|       |-- resources // 存放代码资源文件
|           |-- applicationContext.xml // 项目配置文件
|           |-- contract.properties // 存储部署合约地址的文件
|           |-- log4j.properties // 日志配置文件
```

(continues on next page)

(续上页)

```
|            |-- contract //存放solidity约文件
|                    |-- Asset.sol
|                    |-- Table.sol
|
|-- tool
    |-- asset_run.sh // 项目运行脚本
```

项目引入Web3SDK

项目的`build.gradle`文件已引入Web3SDK，不需修改。其引入方法介绍如下：

- Web3SDK引入了以太坊的solidity编译器相关jar包，因此在`build.gradle`文件需要添加以太坊的远程仓库：

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

- 引入Web3SDK jar包

```
compile ('org.fisco-bcos:web3sdk:2.5.0')
```

证书与配置文件

- 区块链节点证书配置

拷贝区块链节点对应的SDK证书

```
# 进入~目录
# 拷贝节点证书到项目的资源目录
$ cd ~
$ cp fisco/nodes/127.0.0.1/sdk/* asset-app/src/test/resources/
```

- applicationContext.xml

注意：如果搭链时设置的jsonrpc_listen_ip为127.0.0.1或者0.0.0.0，channel_port为20200，则`applicationContext.xml`配置不用修改。若区块链节点配置有改动，需要同样修改配置`applicationContext.xml`，具体请参考SDK使用文档。

4.2.5 业务开发

我们已经介绍了如何在自己的项目中引入以及配置Web3SDK，本节介绍如何通过Java程序调用合约，同样以示例的资产管理说明。asset-app项目已经包含示例的完整源码，用户可以直接使用，现在介绍核心类`AssetClient`的设计与实现。

`AssetClient.java`：通过调用`Asset.java`实现对合约的部署与调用，路径`/src/main/java/org/fisco/bcos/asset/client`，初始化以及调用流程都在该类中进行。

- 初始化

初始化代码的主要功能为构造Web3j与Credentials对象，这两个对象在创建对应的合约类对象(调用合约类的deploy或者load函数)时需要使用。

```
// 函数initialize中进行初始化
ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
Service service = context.getBean(Service.class);
service.run();

ChannelEthereumService channelEthereumService = new ChannelEthereumService();
channelEthereumService.setChannelService(service);
// 初始化Web3j对象
Web3j web3j = Web3j.build(channelEthereumService, 1);
// 初始化Credentials对象
Credentials credentials = Credentials.create(Keys.createEcKeyPair());
```

- 构造合约类对象

可以使用deploy或者load函数初始化合约对象，两者使用场景不同，前者适用于初次部署合约，后者在合约已经部署并且已知合约地址时使用。

```
// 部署合约
Asset asset = Asset.deploy(web3j, credentials, new StaticGasProvider(gasPrice,␣
↪gasLimit)).send();
// 加载合约地址
Asset asset = Asset.load(contractAddress, web3j, credentials, new␣
↪StaticGasProvider(gasPrice, gasLimit));
```

- 接口调用

使用合约对象调用对应的接口，处理返回结果。

```
// select接口调用
Tuple2<BigInteger, BigInteger> result = asset.select(assetAccount).send();
// register接口调用
TransactionReceipt receipt = asset.register(assetAccount, amount).send();
// transfer接口
TransactionReceipt receipt = asset.transfer(fromAssetAccount, toAssetAccount,␣
↪amount).send();
```

4.2.6 运行

至此我们已经介绍使用区块链开发资产管理应用的所有流程并实现了功能，接下来可以运行项目，测试功能是否正常。

- 编译

```
# 切换到项目目录
$ cd ~/asset-app
# 编译项目
$ ./gradlew build
```

编译成功之后，将在项目根目录下生成dist目录。dist目录下有一个asset_run.sh脚本，简化项目运行。现在开始一一验证本文开始定下的需求。

- 部署Asset.sol合约

```
# 进入dist目录
$ cd dist
$ bash asset_run.sh deploy
Deploy Asset successfully, contract address is␣
↪0xd09ad04220e40bb8666e885730c8c460091a4775
```

- 注册资产

```
$ bash asset_run.sh register Alice 100000
Register account successfully => account: Alice, value: 100000
$ bash asset_run.sh register Bob 100000
Register account successfully => account: Bob, value: 100000
```

- 查询资产

```
$ bash asset_run.sh query Alice
account Alice, value 100000
$ bash asset_run.sh query Bob
account Bob, value 100000
```

- 资产转移

```
$ bash asset_run.sh transfer Alice Bob  50000
Transfer successfully => from_account: Alice, to_account: Bob, amount: 50000
$ bash asset_run.sh query Alice
account Alice, value 50000
$ bash asset_run.sh query Bob
account Bob, value 150000
```

总结： 至此，我们通过合约开发，合约编译，SDK配置与业务开发构建了一个基于FISCO BCOS联盟区块链的应用。

CHAPTER 5

使用手册

本章提供了FISCO BCOS平台的使用手册，使用手册介绍FISCO BCOS平台各种功能使用方式。

5.1 获取可执行程序

用户可以自由选择以下任一方式获取FISCO BCOS可执行程序。推荐从GitHub下载预编译二进制。

- 官方提供的静态链接的预编译文件，可以在Ubuntu 16.04和CentOS 7.2以上版本运行。
- 官方提供docker镜像，欢迎使用。docker-hub地址
- 源码编译获取可执行程序，参考源码编译。

5.1.1 下载预编译fisco-bcos

我们提供静态链接的预编译程序，在Ubuntu 16.04和CentOS 7经过测试。请从Release页面下载最新发布的**预编译程序**。

5.1.2 docker镜像

从v2.0.0版本开始，我们提供对应版本tag的docker镜像。对应于master分支，我们提供`lastest`标签的镜像，更多的docker标签请参考这里。

build_chain.sh脚本增加了`-d`选项，提供docker模式建链的选择，方便开发者部署。详情请参考这里。

注解： build_chain.sh脚本为了简单易用，启动docker使用了 `--network=host` 网络模式，实际使用中用户可能需要根据自己的网络场景定制改造。

5.1.3 源码编译

注解： 源码编译适合于有丰富开发经验的用户，编译过程中需要下载依赖库，请保持网络畅通。受网络和机器配置影响，编译用时5-20分钟不等。

FSICO-BCOS使用通用CMake构建系统生成特定平台的构建文件，这意味着无论您使用什么操作系统工作流都非常相似：

1. 安装构建工具和依赖包（依赖于平台）。
2. 从FISCO BCOS克隆代码。
3. 运行cmake生成构建文件并编译。

安装依赖

- Ubuntu

推荐Ubuntu 16.04以上版本，16.04以下的版本没有经过测试，源码编译时依赖于编译工具和libssl。

```
sudo apt install -y g++ libssl-dev openssl cmake git build-essential autoconf␣
↪texinfo flex patch bison libgmp-dev
```

- CentOS

推荐使用CentOS7以上版本。

```
$ sudo yum install -y epel-release
$ sudo yum install -y openssl-devel openssl cmake3 gcc-c++ git flex patch bison␣
↪gmp-static
```

- macOS

推荐xcode10以上版本。macOS依赖包安装依赖于Homebrew。

```
brew install openssl git flex patch bison gmp
```

克隆代码

```
git clone https://github.com/FISCO-BCOS/FISCO-BCOS.git
```

编译

编译完成后二进制文件位于FISCO-BCOS/build/bin/fisco-bcos。

```
$ cd FISCO-BCOS
$ git checkout master
$ mkdir -p build && cd build
# CentOS请使用cmake3
$ cmake ..
# 高性能机器可添加-j4使用4核加速编译
$ make
```

注解：

- 如果因为网络问题导致长时间无法下载依赖库，请尝试从 *https://gitee.com/FISCO-BCOS/LargeFiles/tree/master/libs* 下载，放在FISCO-BCOS/deps/src/

编译选项介绍

- BUILD_GM，默认off，国密编译开关。通过`cmake -DBUILD_GM=on ..`打开国密开关。
- TESTS，默认off，单元测试编译开关。通过`cmake -DTESTS=on ..`打开单元测试开关。
- DEMO，默认off，测试程序编译开关。通过`cmake -DDEMO=on ..`打开单元测试开关。
- TOOL，默认off，工具程序编译开关。通过`cmake -DTOOL=on ..`打开工具开关，提供FISCO节点的rocksdb读取工具。
- BUILD_STATIC，默认off，静态编译开关，只支持Ubuntu。通过`cmake -DBUILD_STATIC=on ..`打开静态编译开关。
- 生成源码文档。

```
# 安装Doxygen
$ sudo apt install -y doxygen graphviz
# 生成源码文档 生成的源码文档位于build/doc
$ make doc
```

5.2 硬件要求

注解： 由于节点多群组共享网络带宽、CPU和内存资源，因此为了保证服务的稳定性，一台机器上不推荐配置过多节点。

下表是单群组单节点推荐的配置，节点耗费资源与群组个数呈线性关系，您可根据实际的业务需求和机器资源，合理地配置节点数目。

配置	最低配置	推荐配置
CPU	1.5GHz	2.4GHz
内存	1GB	8GB
核心	1核	8核
带宽	1Mb	10Mb

5.3 支持的平台

- CentOS 7.2+
- Ubuntu 16.04
- macOS 10.14+

5.4 开发部署工具

重要： 开发部署工具 build_chain脚本目标是让用户最快的使用FISCO BCOS，对于企业级应用部署FISCO BCOS请参考 运维部署工具 。

FISCO BCOS提供了`build_chain.sh`脚本帮助用户快速搭建FISCO BCOS联盟链，该脚本默认从GitHub下载`master`分支最新版本预编译可执行程序进行相关环境的搭建。

5.4.1 脚本功能简介

- build_chain.sh脚本用于快速生成一条链中节点的配置文件，脚本依赖于openssl请根据自己的操作系统安装openssl 1.0.2以上版本。脚本的源码位于FISCO-BCOS/tools/build_chain.sh。
- 快速体验可以使用-l选项指定节点IP和数目。-f选项通过使用一个指定格式的配置文件，支持创建各种复杂业务场景FISCO BCOS链。**-l和-f选项必须指定一个且不可共存**。
- 建议测试时使用-T，-T开启log级别到DEBUG，**p2p模块默认监听 0.0.0.0**。

注解： 为便于开发和体验，p2p模块默认监听IP是 *0.0.0.0* ，出于安全考虑，请根据实际业务网络情况，修改为安全的监听地址，如内网IP或特定的外网IP

5.4.2 帮助

```
Usage:
    -l <IP list>                        [Required] "ip1:nodeNum1,ip2:nodeNum2" e.g:
↪"192.168.0.1:2,192.168.0.2:3"
    -f <IP list file>                   [Optional] split by line, every line␣
↪should be "ip:nodeNum agencyName groupList p2p_port,channel_port,jsonrpc_port".␣
↪eg "127.0.0.1:4 agency1 1,2 30300,20200,8545"
    -e <FISCO-BCOS binary path>         Default download fisco-bcos from GitHub.␣
↪If set -e, use the binary at the specified location
    -o <Output Dir>                     Default ./nodes/
    -p <Start Port>                     Default 30300,20200,8545 means p2p_port␣
↪start from 30300, channel_port from 20200, jsonrpc_port from 8545
    -i <Host ip>                        Default 127.0.0.1. If set -i, listen 0.0.0.
↪0
    -v <FISCO-BCOS binary version>      Default get version from https://github.
↪com/FISCO-BCOS/FISCO-BCOS/releases. If set use specificd version binary
    -s <DB type>                        Default rocksdb. Options can be rocksdb /␣
↪mysql / scalable, rocksdb is recommended
    -d <docker mode>                    Default off. If set -d, build with docker
    -c <Consensus Algorithm>            Default PBFT. Options can be pbft / raft /
↪rpbft, pbft is recommended
    -C <Chain id>                       Default 1. Can set uint.
    -g <Generate guomi nodes>           Default no
    -z <Generate tar packet>            Default no
    -t <Cert config file>               Default auto generate
    -k <The path of ca root>            Default auto generate, the ca.crt and ca.
↪key must in the path, if use intermediate the root.crt must in the path
    -K <The path of sm crypto ca root>  Default auto generate, the gmca.crt and␣
↪gmca.key must in the path, if use intermediate the gmroot.crt must in the path
    -D <Use Deployment mode>            Default false, If set -D, use deploy mode␣
↪directory struct and make tar
    -G <channel use sm crypto ssl>      Default false, only works for guomi mode
    -X <Certificate expiration time>    Default 36500 days
    -T <Enable debug log>               Default off. If set -T, enable debug log
    -S <Enable statistics>              Default off. If set -S, enable statistics
    -F <Disable log auto flush>         Default on. If set -F, disable log auto␣
↪flush
    -E <Enable free_storage_evm>        Default off. If set -E, enable free_
↪storage_evm
    -h Help
e.g
    ./build_chain.sh -l "127.0.0.1:4"
```

5.4.3 选项介绍

l选项:

用于指定要生成的链的IP列表以及每个IP下的节点数，以逗号分隔。脚本根据输入的参数生成对应的节点配置文件，其中每个节点的端口号默认从30300开始递增，所有节点属于同一个机构和群组。

f选项

```
+ 用于根据配置文件生成节点，相比于`l`选项支持更多的定制。
+ 按行分割，每一行表示一个服务器，格式为`IP:NUM AgencyName GroupList`，每行内的项使用空格分割，**不可有空行**。
+ `IP:NUM`表示机器的IP地址以及该机器上的节点数。`AgencyName`表示机构名，用于指定使用的机构证书。`GroupList`表示该行生成的节点所属的组，以`,`分割。例如`192.168.0.1:2 agency1 1,2`表示`ip`为`192.168.0.1`的机器上有两个节点，这两个节点属于机构`agency1`，属于group1和group2。
```

下面是一个配置文件的例子，每个配置项以空格分隔。

```
192.168.0.1:1 agency1 1,2 30300,20200,8545
192.168.0.2:1 agency1 1,2 30300,20200,8545
192.168.0.3:2 agency1 1,3 30300,20200,8545
192.168.0.4:1 agency2 1   30300,20200,8545
192.168.0.5:1 agency3 2,3 30300,20200,8545
192.168.0.6:1 agency2 3   30300,20200,8545
```

假设上述文件名为ipconf，则使用下列命令建链，表示使用配置文件，设置日志级别为DEBUG。

```
bash build_chain.sh -f ipconf -T
```

e选项[Optional]

用于指定fisco-bcos二进制所在的**完整路径**，脚本会将fisco-bcos拷贝以IP为名的目录下。不指定时，默认从GitHub下载master分支最新的二进制程序。

```
# 从GitHub下载下载最新release二进制，生成本机4节点
$ bash build_chain.sh -l "127.0.0.1:4"
# 使用 bin/fisco-bcos 二进制，生成本机4节点
$ bash build_chain.sh -l "127.0.0.1:4" -e bin/fisco-bcos
```

o选项[Optional]

指定生成的配置所在的目录。

p选项[Optional]

指定节点的起始端口，每个节点占用三个端口，分别是p2p,channel,jsonrpc使用,分割端口，必须指定三个端口。同一个IP下的不同节点所使用端口从起始端口递增。

```
# 两个节点分别占用`30300,20200,8545`和`30301,20201,8546`。
$ bash build_chain.sh -l 127.0.0.1:2 -p 30300,20200,8545
```

v选项[Optional]

用于指定搭建FISCO BCOS时使用的二进制版本。build_chain默认下载Release页面最新版本，设置该选项时下载参数指定version版本并设置config.ini配置文件中的[compatibility].supported_version=${version}。如果同时使用-e选项，则配置[compatibility].supported_version=${version}为Release页面最新版本号。

d选项[Optional]

使用docker模式搭建FISCO BCOS，使用该选项时不再拉取二进制，但要求用户启动节点机器安装docker且账户有docker权限，即用户加入docker群组。 在节点目录下执行如下命令启动节点

```
$ ./start.sh
```

该模式下 start.sh 脚本启动节点的命令如下

```
$ docker run -d --rm --name ${nodePath} -v ${nodePath}:/data --network=host -w=/
↪data fiscoorg/fiscobcos:latest -c config.ini
```

s选项[Optional]

有参数选项，参数为db名，目前支持rocksdb、mysql、external、scalable。默认使用RocksDB。

- RocksDB模式，使用RocksDB作为后端数据库。
- MySQL模式，使用MySQL作为后端数据库，节点直连MySQL，需要在群组ini文件中配置数据库相关信息。
- External模式，使用MySQL作为后端数据库，节点使用`amdb-proxy`连接数据库，代理和节点通过amop协议通信，需要在群组ini文件中配置topic信息。
- scalable模式，区块数据和状态数据存储在不同的数据库中，区块数据根据配置存储在以块高命名的RocksDB实例中。如需使用裁剪数据的功能，必须使用scalable模式。

c选项[Optional]

有参数选项，参数为共识算法类型，目前支持PBFT、Raft、rPBFT。默认共识算法是PBFT。

- `PBFT`：设置节点共识算法为PBFT。
- `Raft`：设置节点共识算法为Raft。
- `rPBFT`：设置节点共识算法为rPBFT。

C选项[Optional]

用于指定搭建FISCO BCOS时的链标识。设置该选项时将使用参数设置`config.ini`配置文件中的`[chain].id`，参数范围为正整数，默认设置为1。

```
# 该链标识为2。
$ bash build_chain.sh -l 127.0.0.1:2 -C 2
```

g选项[Optional]

无参数选项，设置该选项时，搭建国密版本的FISCO BCOS。**使用g选项时要求二进制fisco-bcos为国密版本。**

z选项[Optional]

无参数选项，设置该选项时，生成节点的tar包。

t选项[Optional]

该选项用于指定生成证书时的证书配置文件。

T选项[Optional]

无参数选项，设置该选项时，设置节点的log级别为DEBUG。log相关配置参考这里。

k选项[Optional]

使用用户指定的链证书和私钥签发机构和节点的证书，参数指定路径，路径下必须包括ca.crt/ca.key，如果所指定的私钥和证书是中间ca，那么此文件夹下还需要包括root.crt，用于存放上级证书链。

K选项[Optional]

国密模式使用用户指定的链证书和私钥签发机构和节点的证书，参数指定路径，路径下必须包括gmca.crt/gmca.key，如果所指定的私钥和证书是中间ca，那么此文件夹下还需要包括gmroot.crt，用于存放上级证书链。

G选项[Optional]

从2.5.0开始，国密模式下，用户可以配置节点与SDK连接是否使用国密SSL，设置此选项则`chain.sm_crypto_channel=true`。默认节点与SDK的channel连接使用secp256k1的证书。

D选项[Optional]

无参数选项，设置该选项时，生成节点的目录名为IP_P2P端口，默认为节点从0开始的编号。

E选项[Optional]

无参数选项，设置该选项时，启用Free Storage Gas模式，默认关闭`Free Storage` Gas模式

5.4.4 节点文件组织结构

- cert文件夹下存放链的根证书和机构证书。
- 以IP命名的文件夹下存储该服务器所有节点相关配置、`fisco-bcos`可执行程序、SDK所需的证书文件。
- 每个IP文件夹下的`node*`文件夹下存储节点所需的配置文件。其中`config.ini`为节点的主配置，`conf`目录下存储证书文件和群组相关配置。配置文件详情，请参考这里。每个节点中还提供`start.sh`和`stop.sh`脚本，用于启动和停止节点。
- 每个IP文件夹下的提供`start_all.sh`和`stop_all.sh`两个脚本用于启动和停止所有节点。

```
nodes/
├── 127.0.0.1
│   ├── fisco-bcos # 二进制程序
│   ├── node0 # 节点0文件夹
│   │   ├── conf # 配置文件夹
│   │   │   ├── ca.crt # 链根证书
│   │   │   ├── group.1.genesis # 群组1初始化配置，该文件不可更改
│   │   │   ├── group.1.ini # 群组1配置文件
│   │   │   ├── node.crt # 节点证书
│   │   │   ├── node.key # 节点私钥
│   │   │   └── node.nodeid # 节点id，公钥的16进制表示
│   │   ├── config.ini # 节点主配置文件，配置监听IP、端口等
│   │   ├── start.sh # 启动脚本，用于启动节点
│   │   └── stop.sh # 停止脚本，用于停止节点
```

(continues on next page)

(续上页)

```
|   ├── node1  # 节点1文件夹
|   ·····
|   ├── node2  # 节点2文件夹
|   ·····
|   ├── node3  # 节点3文件夹
|   ·····
|   ├── sdk  # SDK与节点SSL连接配置
|   |   ├── ca.crt  # SSL连接根证书
|   |   ├── sdk.crt  # SSL连接证书
|   |   └── sdk.key  # SSL连接证书私钥
|   |   ├── gm  # SDK与节点国密SSL连接配置，注意：生成国密区块链环境时才会生成该目录，用于节点
与SDK的国密SSL连接
|   |   |   ├── gmca.crt  # 国密SSL连接根证书
|   |   |   ├── gmensdk.crt  # 国密SSL连接加密证书
|   |   |   ├── gmensdk.key  # 国密SSL连接加密证书私钥
|   |   |   ├── gmsdk.crt  # 国密SSL连接签名证书
|   |   |   └── gmsdk.key  # 国密SSL连接签名证书私钥
├── cert  # 证书文件夹
|   ├── agency  # 机构证书文件夹
|   |   ├── agency.crt  # 机构证书
|   |   ├── agency.key  # 机构私钥
|   |   ├── agency.srl
|   |   ├── ca-agency.crt
|   |   ├── ca.crt
|   |   └── cert.cnf
|   ├── ca.crt  # 链证书
|   ├── ca.key  # 链私钥
|   ├── ca.srl
|   └── cert.cnf
```

5.4.5 使用举例

无外网条件的单群组

最简单的操作方式是在有外网的Linux机器上使用build_chain建好链，借助-z选项打包，然后拷贝到无外网的机器上运行。

1. 针对某下场景下无外网条件下建链，请从发布页面下载最新的目标操作系统的二进制，例如对于Linux系统下载fisco-bcos.tar.gz。
2. 请从发布页面下载最新版本的build_chain脚本。
3. 上传fisco-bcos.tar.gz和build_chain.sh到目标服务器，需要注意目标服务器要求64位，要求安装有openssl 1.0.2以上版本。
4. 解压fisco-bcos.tar.gz得到fisco-bcos可执行文件，作为-e选项的参数。
5. 构建本机上4节点的FISCO BCOS联盟链，使用默认起始端口30300,20200,8545（4个节点会占用30300-30303,20200-20203,8545-8548）。
6. 执行下面的指令，假设最新版本是2.2.0，则将2.2.0作为-v选项参数。

```
# 构建FISCO BCOS联盟链
$ bash build_chain.sh -l "127.0.0.1:4" -p 30300,20200,8545 -e ./fisco-bcos -v 2.2.0
# 生成成功后，输出`All completed`提示
Generating CA key...
================================================================
Generating keys ...
Processing IP:127.0.0.1 Total:4 Agency:agency Groups:1
================================================================
Generating configurations...
```

(continues on next page)

(续上页)

```
Processing IP:127.0.0.1 Total:4 Agency:agency Groups:1
================================================================
[INFO] FISCO-BCOS Path   : bin/fisco-bcos
[INFO] Start Port        : 30300 20200 8545
[INFO] Server IP         : 127.0.0.1:4
[INFO] State Type        : storage
[INFO] RPC listen IP     : 127.0.0.1
[INFO] Output Dir        : /Users/fisco/WorkSpace/FISCO-BCOS/tools/nodes
[INFO] CA Key Path       : /Users/fisco/WorkSpace/FISCO-BCOS/tools/nodes/cert/ca.
↪key
================================================================
[INFO] All completed. Files in /Users/fisco/WorkSpace/FISCO-BCOS/tools/nodes
```

群组新增节点

本节以为上一小节生成的群组1新增一个共识节点为例操作。

为新节点生成私钥证书

接下来的操作，都在上一节生成的nodes/127.0.0.1目录下进行

1. 获取证书生成脚本

```
curl -LO https://raw.githubusercontent.com/FISCO-BCOS/FISCO-BCOS/master/tools/gen_
↪node_cert.sh
```

注解:

- 如果因为网络问题导致长时间无法下载，请尝试 *curl -LO https://gitee.com/FISCO-BCOS/FISCO-BCOS/raw/master/tools/gen_node_cert.sh*

1. 生成新节点私钥证书

```
# -c指定机构证书及私钥所在路径
# -o输出到指定文件夹，其中newNode/conf中会存在机构agency新签发的证书和私钥
bash gen_node_cert.sh -c ../cert/agency -o newNode
```

国密版本请执行下面的指令生成证书。

```
bash gen_node_cert.sh -c ../cert/agency -o newNodeGm -g ../gmcert/agency/
```

准备配置文件

1. 拷贝群组1中节点node0配置文件与工具脚本

   ```
   cp node0/config.ini newNode/config.ini
   cp node0/conf/group.1.genesis newNode/conf/group.1.genesis
   cp node0/conf/group.1.ini newNode/conf/group.1.ini
   cp node0/*.sh newNode/
   cp -r node0/scripts newNode/
   ```

2. 更新newNode/config.ini中监听的IP和端口，对于[rpc]模块，修改listen_ip、channel_listen_port和jsonrpc_listen_port；对于[p2p]模块，修改listen_port

3. 将新节点的P2P配置中的IP和Port加入原有节点的config.ini中的[p2p]字段。假设新节点IP:Port为127.0.0.1:30304则，修改后的[P2P]配置为

注解： 为便于开发和体验，p2p模块默认监听IP是 *0.0.0.0* ，出于安全考虑，请根据实际业务网络情况，修改为安全的监听地址，如：内网IP或特定的外网IP

```
```bash
[p2p]
    listen_ip=0.0.0.0
    listen_port=30300
    ;enable_compress=true
    ; nodes to connect
    node.0=127.0.0.1:30300
    node.1=127.0.0.1:30301
    node.2=127.0.0.1:30302
    node.3=127.0.0.1:30303
    node.4=127.0.0.1:30304
```
```

1. 启动新节点，执行newNode/start.sh
2. 通过console将新节点加入群组1，请参考这里和这里，nodeID可以通过命令cat newNode/conf/node.nodeid来获取
3. 检查连接和共识

生成新机构证书

1. 获取机构证书生成脚本

```
curl -LO https://raw.githubusercontent.com/FISCO-BCOS/FISCO-BCOS/master/tools/gen_
↪agency_cert.sh
```

1. 生成新机构私钥和证书

```
# -c 指定链证书及私钥所在路径，目录下必须有ca.crt 和 ca.key， 如果ca.crt是二级CA，则还需
要root.crt(根证书)
# -g 指定国密链证书及私钥所在路径，目录下必须有gmca.crt 和 gmca.key，如果gmca.crt是二级CA，则
还需要gmroot.crt(根证书)
# -a 新机构的机构名
bash gen_agency_cert.sh -c nodes/cert/ -a newAgencyName
```

国密版本请执行下面的指令。

```
bash gen_agency_cert.sh -c nodes/cert/ -a newAgencyName -g nodes/gmcert/
```

多服务器多群组

使用build_chain脚本构建多服务器多群组的FISCO BCOS联盟链需要借助脚本配置文件，详细使用方式可以参考这里。

5.5 证书说明

FISCO BCOS网络采用面向CA的准入机制，支持任意多级的证书结构，保障信息保密性、认证性、完整性、不可抵赖性。

FISCO BCOS使用x509协议的证书格式，根据现有业务场景，默认采用三级的证书结构，自上而下分别为链证书、机构证书、节点证书。

在多群组架构中，一条链拥有一个链证书及对应的链私钥，链私钥由联盟链委员会共同管理。联盟链委员会可以使用机构的证书请求文件`agency.csr`，签发机构证书`agency.crt`。

机构私钥由机构管理员持有，可以对机构下属节点签发节点证书。

节点证书是节点身份的凭证，用于与其他持有合法证书的节点间建立SSL连接，并进行加密通讯。

sdk证书是sdk与节点通信的凭证，机构生成sdk证书，允许sdk与节点进行通信。

FISCO BCOS节点运行时的文件后缀介绍如下：

5.5.1 角色定义

FISCO BCOS的证书结构中，共有四种角色，分别是联盟链委员会管理员、机构、节点和SDK。

联盟链委员会

- 联盟链委员会管理链的私钥，并根据机构的证书请求文件`agency.csr`为机构颁发机构证书。

```
ca.crt 链证书
ca.key 链私钥
```

FISCO BCOS进行SSL加密通信时，拥有相同链证书`ca.crt`的节点才可建立连接。

机构

- 机构管理员管理机构私钥，可以颁发节点证书和sdk证书。

```
ca.crt 链证书
agency.crt 机构证书
agency.csr 机构证书请求文件
agency.key 机构私钥
```

节点/SDK

- FISCO BCOS节点包括节点证书和私钥，用于建立节点间SSL加密连接；
- SDK包括SDK证书和私钥，用于与区块链节点建立SSL加密连接。

```
ca.crt #链证书
node.crt #节点证书
node.key #节点私钥
sdk.crt #SDK证书
sdk.key #SDK私钥
```

节点证书`node.crt`包括节点证书和机构证书信息，节点与其他节点/SDK通信验证时会用自己的私钥`node.key`对消息进行签名，并发送自己的`node.crt`至对方进行验证

5.5.2 证书生成流程

FISCO BCOS的证书生成流程如下，用户也可以使用企业部署工具生成相应证书

生成链证书

- 联盟链委员会使用openssl命令请求链私钥ca.key，根据ca.key生成链证书ca.crt

生成机构证书

- 机构使用openssl命令生成机构私钥agency.key
- 机构使用机构私钥agency.key得到机构证书请求文件agency.csr，发送agency.csr给联盟链委员会
- 联盟链委员会使用链私钥ca.key，根据得到机构证书请求文件agency.csr生成机构证书agency.crt，并将机构证书agency.crt发送给对应机构

生成节点/SDK证书

- 节点生成私钥node.key和证书请求文件node.csr，机构管理员使用私钥agency.key和证书请求文件node.csr为节点颁发证书。同理，可用相同的方式为SDK生成证书

5.5.3 节点证书续期操作

完成证书续期前推荐使用证书检测脚本对证书进行检测。

当证书过期时，需要用户使用对当前节点私钥重新签发证书，操作如下：

假设用户证书过期的节点目录为~/mynode，节点目录如下：

```
mynode
├── conf
│   ├── ca.crt
│   ├── group.1.genesis
│   ├── group.1.ini
│   ├── node.crt  #节点证书过期，需要替换
│   ├── node.key  #节点私钥，证书续期需要使用
│   └── node.nodeid
├── config.ini
├── scripts
│   ├── load_new_groups.sh
│   └── reload_whitelist.sh
├── start.sh
└── stop.sh
```

设用户机构证书目录为~/myagency，目录如下：

```
agency
├── agency.crt  #机构证书，证书续期需要使用
├── agency.key  #机构私钥，证书续期需要使用
├── agency.srl
├── ca.crt
└── cert.cnf
```

续期操作如下：

- 使用节点私钥生成证书请求文件，请将~/mynode/node/conf/node.key修改为你自己的节点私钥，将~/myagency/cert.cnf替换为自己的证书配置文件

```
openssl req -new -sha256 -subj "/CN=RenewalNode/O=fisco-bcos/OU=node" -key ~/
↪mynode/node/conf/node.key -config ~/myagency/cert.cnf -out node.csr
```

操作完成后会在当前目录下生成证书请求文件node.csr。

- 查看证书请求文件

```
cat node.csr
```

操作完成后显示如下：

```
-----BEGIN CERTIFICATE REQUEST-----
MIIBGzCBwgIBADA6MRQwEgYDVQQDDAtSZW5ld2FsTm9kZTETMBEGA1UECgwKZmlz
Y28tYmNvczENMAsGA1UECwwEbm9kZTBWMBAGByqGSM49AgEGBSuBBAAKA0IABICU
KLP9GFRF6bBz+pfHCl1ifqzqrPiVoSPtwubXx+NRAI502EENMpnLqaXWm+OyadKz
PqUneVDQ6U+CvgY2IPygKTAnBgkqhkiG9w0BCQ4xGjAYMAkGA1UdEwQCMAAwCwYD
VR0PBAQDAgXgMAoGCCqGSM49BAMCA0gAMEUCIQDa8PzS1sCdk+rWgEsaOdvBnY+z
NDw6LU44WHCtrW6iNQIgY7Ne4EpAvPGmMOXalJsvYm2Xy6Bm9MlL7NEIP9Y0ai0=
-----END CERTIFICATE REQUEST-----
```

- 使用机构私钥和机构证书对证书请求文件node.csr签发新证书，请将~/myagency/agency.key修改为你自己的机构私钥，请将~/myagency/agency.crt修改为你自己的机构证书

```
openssl x509 -req -days 3650 -sha256 -in node.csr -CAkey ~/myagency/agency.key -CA␣
↪~/myagency/agency.crt -out node.crt -CAcreateserial -extensions v3_req -extfile ~
↪/myagency/cert.cnf
```

成功会有如下显示

```
Signature ok
subject=/CN=RenewalNode/O=fisco-bcos/OU=node
Getting CA Private Key
```

操作完成后会在当前目录下生成续期后的证书node.crt。

- 查看节点新证书

```
cat ./node.crt
```

操作完成后显示如下：

```
-----BEGIN CERTIFICATE-----
MIICQDCCASigAwIBAgIJALm++fKF6UmXMA0GCSqGSIb3DQEBCwUAMDcxDzANBgNV
BAMMBmFnZW5jeTETMBEGA1UECgwKZmlzY28tYmNvczEPMA0GA1UECwwGYWdlbmN5
MB4XDTE5MDkyNjEwMjEyNVoXDTI5MDkyMzEwMjEyNVowOjEUMBIGA1UEAwwLUmVu
ZXdhbE5vZGUxEzARBgNVBAoMCmZpc2NvLWJjb3MxDTALBgNVBAsMBG5vZGUwVjAQ
BgcqhkjOPQIBBgUrgQQACgNCAASAlCiz/RhURemwc/qXxwpdYn6s6qz4laEj7cLm
18fjUQCOdNhBDTKZy6ml1pvjsmnSsz6lJ3lQ0OlPgr4GNiD8oxowGDAJBgNVHRME
AjAAMAsGA1UdDwQEAwIF4DANBgkqhkiG9w0BAQsFAAOCAQEAVvLUYeOJBfr1bbwp
E2H2QTb4phgcFGvrW5tqfvDvKaVGrSjJowZPKX+ruWFRQAZJBCc3/4M0Q1PYlWpB
R5a9Tpc7ebmUVltY7/GqASlDExdt2nqSvLxOKWgE++FveCdJzOEGuuttTZxjWFhQ
Yr9rPlKhzhEo2jM0lFIxdoCrG/WkcKmzJEyHdVwxLr2FOF9q9e9O9xyUkt2QRBGD
T4dIOeLRK6V1pnNkbBNRYG+tGMq2nBUPCAKJbV1LnhaNNRRbE5z7I4JkRnLHea6P
1VIiwnmbv9a3aM7lsnisPAz8PY5Ddmflo87UiL02J2UnQmq+gtAB9C9DUROGbSH5
Q6CXDA==
-----END CERTIFICATE-----
```

- 将机构证书添加到节点证书末尾

由于fisco-bcos使用三级证书结构，需要将机构证书和节点证书合并

```
cat ~/myagency/agency.crt >> ./node.crt
```

- 查看合并后的节点新证书

```
cat ./node.crt
```

操作完成后显示如下：

```
-----BEGIN CERTIFICATE-----
MIICQDCCASigAwIBAgIJALm++fKF6UmXMA0GCSqGSIb3DQEBCwUAMDcxDzANBgNV
BAMMBmFnZW5jeTETMBEGA1UECgwKZmlzY28tYmNvczEPMA0GA1UECwwGYWdlbmN5
MB4XDTE5MDkyNjEwMjEyNVoXDTI5MDkyMzEwMjEyNVowOjEUMBIGA1UEAwwLUmVu
ZXdhbE5vZGUxEzARBgNVBAoMCmZpc2NvLWJjb3MxDTALBgNVBAsMBG5vZGUwVjAQ
BgcqhkjOPQIBBgUrgQQACgNCAASAlCiz/RhURemwc/qXxwpdYn6s6qz4laEj7cLm
18fjUQCOdNhBDTKZy6ml1pvjsmnSsz6lJ3lQ0OlPgr4GNiD8oxowGDAJBgNVHRME
AjAAMAsGA1UdDwQEAwIF4DANBgkqhkiG9w0BAQsFAAOCAQEAVvLUYeOJBfr1bbwp
E2H2QTb4phgcFGvrW5tqfvDvKaVGrSjJowZPKX+ruWFRQAZJBCc3/4M0Q1PYlWpB
R5a9Tpc7ebmUVltY7/GqASlDExdt2nqSvLxOKWgE++FveCdJzOEGuuttTZxjWFhQ
Yr9rPlKhzhEo2jM0lFIxdoCrG/WkcKmzJEyHdVwxLr2FOF9q9e9O9xyUkt2QRBGD
T4dIOeLRK6V1pnNkbBNRYG+tGMq2nBUPCAKJbV1LnhaNNRRbE5z7I4JkRnLHea6P
1VIiwnmbv9a3aM7lsnisPAz8PY5Ddmflo87UiL02J2UnQmq+gtAB9C9DUROGbSH5
Q6CXDA==
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIC/zCCAeegAwIBAgIJAKK0/dNnUmlqMA0GCSqGSIb3DQEBCwUAMDUxDjAMBgNV
BAMMBWNoYWluMRMwEQYDVQQKDApmaXNjby1iY29zMQ4wDAYDVQQLDAVjaGFpbjAe
Fw0xOTA5MjYwOTU4NDFaFw0yOTA5MjMwOTU4NDFaMDcxDzANBgNVBAMMBmFnZW5j
eTETMBEGA1UECgwKZmlzY28tYmNvczEPMA0GA1UECwwGYWdlbmN5MIIBIjANBgkq
hkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAuPqz154aXw4t+dcRl+aOz3X7yy0PUymm
DqMq3O7OeWXWYa8MWss5GBGWa2SL6puX/uryZJUUYcmSDwAo7Rsrf8zmbiHqouEC
liy01IqM+9jE7/IywRpRZO7W/QNrv9vRXxDJsr120vs760aMRKWD6UCd7bOQ/m/H
N8VC66r3cvcqey1q49idwOnhh5g80921MFlvu30Rire8kzckzUDr/SV3yt036tZs
D+9l/jHRc/tWo38nkiPy3DIm2oOlrNeJ4+IHnXOfxQxOwsiAeFluxtCq/ZFh4pTL
5lJZTo7bzRcORLOdz40svwDxJKyrMflhue0kGDC0WMExzzvx2oT14wIDAQABoxAw
DjAMBgNVHRMEBTADAQH/MA0GCSqGSIb3DQEBCwUAA4IBAQBlXrFIPQPlKosm2q/O
KktQA04Qh/y6w94Z4bHve0AqzTZn3/tf5q0e9C4f8F/Da+D+nV0GETLtEqRSHT+r
CCAAm78qN9oXmfkt3LvK/YXLNCVB6SSXw8fQx+bfDbIVRB5ivkG1+pmmnh3po1zU
zbrnfdSQi0ZV9MjIPsArjWwkE1i0GkXeiXov305iEX6J5pgu3AMe2RRMwyJiJ6ud
PRPCsF5BN6QrtMubwEnyvyrrX0/drBMtHLMCgecLd/nYMyJ4P15L6UnxC8taQSjM
rAtP3RZrBvBTwXKED0ge/hGIzrO9I1vjfCEuxV3DLlKfGVewuuboW2tYFWGfmrEX
MB7w
-----END CERTIFICATE-----
```

- 将生成的节点证书node.crt替换至节点的conf文件夹下

```
cp -f ./node.crt ~/mynode/node/conf
```

- 启动节点

```
bash ~/mynode/node/start.sh
```

- 查看节点共识

```
tail -f ~/mynode/log/log*  | grep +++
```

正常情况会不停输出++++Generating seal，表示共识正常。

通过上述操作，完成了证书续期的操作。

5.5.4 四类证书续期简易流程

当整条链的证书均已过期时，需要重新对整条链的证书进行续期操作，续期证书的OpenSSL命令与节点续期操作基本相同，或查阅build_chain.sh脚本签发证书的操作，简要步骤如下：

- 使用链私钥ca.key重新签发链证书ca.crt
- 使用机构私钥agency.key生成证书请求文件agency.csr
- 使用链私钥ca.key和链证书ca.crt对证书请求文件agency.csr签发得到机构证书agency.crt

- 使用节点私钥node.key生成证书请求文件node.csr
- 使用机构私钥agency.key和机构证书agency.crt对证书请求文件node.csr签发得到节点证书node.crt
- 将节点证书和机构证书拼接得到node.crt，拼接操作可以参考节点证书续期操作
- SDK证书sdk.crt签发步骤同节点证书签发
- 使用新生成的链证书ca.crt，节点证书node.crt替换所有节点conf目录下的证书

5.6 配置文件与配置项

FISCO BCOS支持多账本，每条链包括多个独立账本，账本间数据相互隔离，群组间交易处理相互隔离，每个节点包括一个主配置config.ini和多个账本配置group.group_id.genesis、group.group_id.ini。

- config.ini：主配置文件，主要配置RPC、P2P、SSL证书、账本配置文件路径、兼容性等信息。
- group.group_id.genesis：群组配置文件，群组内所有节点一致，节点启动后，不可手动更改该配置。主要包括群组共识算法、存储类型、最大gas限制等配置项。
- group.group_id.ini：群组可变配置文件，包括交易池大小等，配置后重启节点生效。

5.6.1 主配置文件config.ini

config.ini采用ini格式，主要包括 **rpc**、**p2p**、**group**、**network_security**和**log** 配置项。

重要:

- 云主机的公网IP均为虚拟IP，若listen_ip/jsonrpc_listen_ip/channel_listen_ip填写外网IP，会绑定失败，须填写0.0.0.0
- RPC/P2P/Channel监听端口必须位于1024-65535范围内，且不能与机器上其他应用监听端口冲突
- 为便于开发和体验，listen_ip/channel_listen_ip参考配置是 *0.0.0.0* ，出于安全考虑，请根据实际业务网络情况，修改为安全的监听地址，如：内网IP或特定的外网IP

配置RPC

- channel_listen_ip: Channel监听IP，为方便节点和SDK跨机器部署，默认设置为0.0.0.0；
- jsonrpc_listen_ip：RPC监听IP，安全考虑，默认设置为127.0.0.1，若有外网访问需求，请监听**节点外网IP**或0.0.0.0；
- channel_listen_port: Channel端口，对应到Web3SDK配置中的channel_listen_port；
- jsonrpc_listen_port: JSON-RPC端口。

注解: 出于安全性和易用性考虑，v2.3.0版本最新配置将listen_ip拆分成jsonrpc_listen_ip和channel_listen_ip，但仍保留对listen_ip的解析功能：

- 配置中仅包含listen_ip：RPC和Channel的监听IP均为配置的listen_ip
- 配置中同时包含listen_ip、channel_listen_ip或jsonrpc_listen_ip：优先解析channel_listen_ip和jsonrpc_listen_ip，没有配置的配置项用listen_ip的值替代

RPC配置示例如下：

```
[rpc]
    channel_listen_ip=0.0.0.0
    jsonrpc_listen_ip=127.0.0.1
    channel_listen_port=30301
    jsonrpc_listen_port=30302
```

配置P2P

当前版本FISCO BCOS必须在config.ini配置中配置连接节点的IP和Port，P2P相关配置包括：

注解： 为便于开发和体验，listen_ip参考配置是 *0.0.0.0* ，出于安全考虑，请根据实际业务网络情况，修改为安全的监听地址，如：内网IP或特定的外网IP

- listen_ip：P2P监听IP，默认设置为0.0.0.0。
- listen_port：节点P2P监听端口。
- node.*: 节点需连接的所有节点IP:Port或DomainName:Port。该选项支持域名，但建议需要使用的用户手动**编译源码**。
- enable_compress：开启网络压缩的配置选项，配置为true，表明开启网络压缩功能，配置为false，表明关闭网络压缩功能，网络压缩详细介绍请参考这里。

P2P配置示例如下：

```
[p2p]
    listen_ip=0.0.0.0
    listen_port=30300
    node.0=127.0.0.1:30300
    node.1=127.0.0.1:30304
    node.2=127.0.0.1:30308
    node.3=127.0.0.1:30312
```

配置账本文件路径

[group]配置本节点所属的所有群组配置路径：

- group_data_path: 群组数据存储路径。
- group_config_path: 群组配置文件路径。

  节点根据group_config_path路径下的所有.genesis后缀文件启动群组。

```
[group]
    ; 所有群组数据放置于节点的data子目录
    group_data_path=data/
    ; 程序自动加载该路径下的所有.genesis文件
    group_config_path=conf/
```

配置证书信息

基于安全考虑，FISCO BCOS节点间采用SSL加密通信，[network_security]配置SSL连接的证书信息：

- data_path：证书和私钥文件所在目录。
- key: 节点私钥相对于data_path的路径。
- cert: 证书node.crt相对于data_path的路径。

- ca_cert: ca证书文件路径。
- ca_path: ca证书文件夹，多ca时需要。
- check_cert_issuer: 设置SDK是否只能连本机构节点，默认为开启（check_cert_issuer=true）。

```
[network_security]
    data_path=conf/
    key=node.key
    cert=node.crt
    ca_cert=ca.crt
    ;ca_path=
```

配置黑名单列表

基于防作恶考虑，FISCO BCOS允许节点将不受信任的节点加入到黑名单列表，并拒绝与这些黑名单节点建立连接，通过[certificate_blacklist]配置:

crl.idx: 黑名单节点的Node ID, 节点Node ID可通过node.nodeid文件获取; idx是黑名单节点的索引。

黑名单的详细信息还可参考CA黑名单

黑名单列表配置示例如下:

```
; 证书黑名单
[certificate_blacklist]
    crl.
↪0=4d9752efbb1de1253d1d463a934d34230398e787b3112805728525ed5b9d2ba29e4ad92c6fcde5156ede8baa5aca3
3787c338a4
```

配置日志信息

FISCO BCOS支持功能强大的boostlog，日志配置主要位于config.ini的[log]配置项中。

日志通用配置

FISCO BCOS通用日志配置项如下:

- enable: 启用/禁用日志，设置为true表示启用日志；设置为false表示禁用日志，**默认设置为true，性能测试可将该选项设置为false，降低打印日志对测试结果的影响**
- log_path:日志文件路径。
- level: 日志级别，当前主要包括trace、debug、info、warning、error五种日志级别，设置某种日志级别后，日志文件中会输大于等于该级别的日志，日志级别从大到小排序error > warning > info > debug > trace。
- max_log_file_size：每个日志文件最大容量，**计量单位为MB，默认为200MB**。
- flush：boostlog默认开启日志自动刷新，若需提升系统性能，建议将该值设置为false。

boostlog示例配置如下:

```
[log]
    ; 是否启用日志，默认为true
    enable=true
    log_path=./log
    level=info
    ; 每个日志文件最大容量，默认为200MB
```

(continues on next page)

(续上页)

```
    max_log_file_size=200
    flush=true
```

统计日志配置

考虑到实时监控系统资源使用情况在实际生产系统中非常重要，FISCO BCOS v2.4.0引入了统计日志，统计日志配置项位于config.ini中。

配置统计日志开关

考虑到并非所有场景都需要网络流量和Gas统计功能，FISCO BCOS在config.ini中提供了enable_statistic选项来开启和关闭该功能，默认关闭该功能。

- log.enable_statistic配置成true，开启网络流量和Gas统计功能
- log.enable_statistic配置成false，关闭网络流量和Gas统计功能

配置示例如下：

```
[log]
    ; enable/disable the statistics function
    enable_statistic=false
```

配置网络统计日志输出间隔

由于网络统计日志周期性输出，引入了log.stat_flush_interval来控制统计间隔和日志输出频率，单位是秒，默认为60s，配置示例如下：

```
[log]
    ; network statistics interval, unit is second, default is 60s
    stat_flush_interval=60
```

配置链属性

可通过config.ini中的[chain]配置节点的链属性。此配置项建链时工具会自动生成，用户不需修改。

- id，链ID，默认为1；
- sm_crypto，2.5.0版本以后，节点支持以国密模式或非国密模式启动，true表示节点使用国密模式，false表示节点使用非国密模式，默认为false；
- sm_crypto_channel，2.5.0版本以后，节点支持与SDK连接使用国密SSL，此选项用于配置是否使用国密SSL与SDK连接，默认为false。

配置节点兼容性

FISCO BCOS 2.0+所有版本向前兼容，可通过config.ini中的[compatibility]配置节点的兼容性，此配置项建链时工具会自动生成，用户不需修改。

- supported_version：当前节点运行的版本

重要:

- 可通过 ‘./fisco-bcos –version | grep “Version” ‘ 命令查看FISCO BCOS的当前支持的最高版本

- build_chain.sh生成的区块链节点配置中，supported_version配置为FISCO BCOS当前的最高版本
- 旧节点升级为新节点时，直接将旧的FISCO BCOS二进制替换为最新FISCO BCOS二进制即可，千万不可修改supported_version

FISCO BCOS 2.2.0节点的[compatibility]配置如下：

```
[compatibility]
    supported_version=2.2.0
```

可选配置：落盘加密

为了保障节点数据机密性，FISCO BCOS引入落盘加密保障节点数据的机密性，**落盘加密**操作手册请参考这里。

config.ini中的storage_security用于配置落盘加密，主要包括：

- enable： 是否开启落盘加密，默认不开启；
- key_manager_ip：Key Manager服务的部署IP；
- key_manager_port：Key Manager服务的监听端口；
- cipher_data_key: 节点数据加密密钥的密文，cipher_data_key的产生参考落盘加密操作手册。

落盘加密节点配置示例如下：

```
[storage_security]
enable=true
key_manager_ip=127.0.0.1
key_manager_port=31443
cipher_data_key=ed157f4588b86d61a2e1745efe71e6ea
```

可选配置：流量控制

为实现区块链系统柔性服务，防止多群组间资源相互影响，FISCO BCOS v2.5.0引入了流量控制功能，主要包括SDK到节点请求速率的限制以及节点间流量限制，配置项位于config.ini的[flow_control]，默认关闭，流控的详细设计请参考这里。

SDK请求速率限制配置

SDK请求速率限制位于配置项[flow_control].limit_req中，用于限制SDK每秒到节点的最大请求数目，当每秒到节点的请求超过配置项的值时，请求会被拒绝，**SDK请求速率限制默认关闭，若要开启该功能，需要将limit_req配置项前面的;去掉**，打开SDK请求速率限制并设计节点每秒可接受2000个SDK请求的示例如下：

```
[flow_control]
    ; restrict QPS of the node
    limit_req=2000
```

节点间流量限制配置

为了防止区块同步、AMOP消息传输占用过多的网络流量，并影响共识模块的消息包传输，FISCO BCOS v2.5.0引入了节点间流量限制的功能，该配置项用于配置节点平均出带宽的上限，但不限制区块共识、交易同步的流量，当节点平均出带宽超过配置值时，会暂缓区块发送、AMOP消息传输。

- `[flow_control].outgoing_bandwidth_limit`：节点出带宽限制，单位为`Mbit/s`，当节点出带宽超过该值时，会暂缓区块发送，也会拒绝客户端发送的AMOP请求，但不会限制区块共识和交易广播的流量，**该配置项默认关闭，若要打开流量限制功能，请将`outgoing_bandwidth_limit`配置项前面的;去掉**。

打开节点出带宽流量限制，并将其设置为`5MBit/s`的配置示例如下：

```
[flow_control]
    ; Mb, can be a decimal
    ; when the outgoing bandwidth exceeds the limit, the block synchronization␣
↪operation will not proceed
    outgoing_bandwidth_limit=5
```

5.6.2 群组系统配置说明

每个群组都有单独的配置文件，按照启动后是否可更改，可分为**群组系统配置**和**群组可变配置**。群组系统配置一般位于节点的`conf`目录下`.genesis`后缀配置文件中。

如：`group1`的系统配置一般命名为`group.1.genesis`，群组系统配置主要包括**群组ID、共识、存储和gas**相关的配置。

重要：配置系统配置时，需注意：

- **配置群组内一致**：群组系统配置用于产生创世块(第0块)，因此必须保证群组内所有节点的该配置一致
- **节点启动后不可更改**：系统配置已经作为创世块写入了系统表，链初始化后不可更改
- 链初始化后，即使更改了genesis配置，新的配置不会生效，系统仍然使用初始化链时的genesis配置
- 由于genesis配置要求群组内所有节点一致，建议使用 开发部署工具 build_chain 生成该配置

群组配置

`[group]`配置**群组ID**，节点根据该ID初始化群组。

群组2的群组配置示例如下：

```
[group]
id=2
```

共识配置

`[consensus]`涉及共识相关配置，包括：

- `consensus_type`：共识算法类型，目前支持PBFT，Raft和rPBFT，默认使用PBFT共识算法；
- `max_trans_num`：一个区块可打包的最大交易数，默认是1000，链初始化后，可通过控制台动态调整该参数；
- `node.idx`：共识节点列表，配置了参与共识节点的Node ID，节点的Node ID可通过`${data_path}/node.nodeid`文件获取(其中`${data_path}`可通过主配置`config.ini`的`[network_security].data_path`配置项获取)

FISCO BCOS v2.3.0引入了rPBFT共识算法，具体可参考这里，rPBFT相关配置如下：

- `epoch_sealer_num`：一个共识周期内选择参与共识的节点数目，默认是所有共识节点总数，链初始化后可通过控制台动态调整该参数；
- `epoch_block_num`：一个共识周期出块数目，默认为1000，可通过控制台动态调整该参数；

注解： rPBFT配置对其他共识算法不生效。

配置节点使用PBFT共识算法如下：

```
; 共识协议配置
[consensus]
    ; 共识算法，目前支持PBFT(consensus_type=pbft), Raft(consensus_
↪type=raft)和rPBFT(consensus_type=rpbft)
    consensus_type=pbft
    ; 单个块最大交易数
    max_trans_num=1000
    ; 一个共识周期内选取参与共识的节点数，rPBFT配置项，对其他共识算法不生效
    epoch_sealer_num=4
    ; 一个共识周期出块数，rPBFT配置项，对其他共识算法不生效
    epoch_block_num=1000
    ; leader节点的ID列表
    node.
↪0=123d24a998b54b31f7602972b83d899b5176add03369395e53a5f60c303acb719ec0718ef1ed51feb7e9cf4836f26
    node.
↪1=70ee8e4bf85eccda9529a8daf5689410ff771ec72fc4322c431d67689efbd6fbd474cb7dc7435f63fa592b98f22b1
    node.
↪2=7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b
    node.
↪3=fd6e0bfe509078e273c0b3e23639374f0552b512c2bea1b2d3743012b7fed8a9dec7b47c57090fa6dcc5341922c32
```

配置节点开启rPBFT共识算法如下：

```
; 共识协议配置
[consensus]
    ; 共识算法，目前支持PBFT(consensus_type=pbft), Raft(consensus_
↪type=raft)和rPBFT(consensus_type=rpbft)
    consensus_type=rpbft
    ; 单个块最大交易数
    max_trans_num=1000
    ; 一个共识周期内选取参与共识的节点数，rPBFT配置项，对其他共识算法不生效
    epoch_sealer_num=4
    ; 一个共识周期出块数，rPBFT配置项，对其他共识算法不生效
    epoch_block_num=1000
    ; leader节点的ID列表
    node.
↪0=123d24a998b54b31f7602972b83d899b5176add03369395e53a5f60c303acb719ec0718ef1ed51feb7e9cf4836f26
    node.
↪1=70ee8e4bf85eccda9529a8daf5689410ff771ec72fc4322c431d67689efbd6fbd474cb7dc7435f63fa592b98f22b1
    node.
↪2=7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b
    node.
↪3=fd6e0bfe509078e273c0b3e23639374f0552b512c2bea1b2d3743012b7fed8a9dec7b47c57090fa6dcc5341922c32
```

状态模式配置

state用于存储区块链状态信息，位于genesis文件中`[state]`：

- `type`：state类型，目前支持storage state和MPT state，**默认为storage state**，storage state将交易执行结果存储在系统表中，效率较高，MPT state将交易执行结果存储在MPT树中，效率较低，但包含完整的历史信息。

重要： 推荐使用 **storage state**

```
[state]
    type=storage
```

gas配置

FISCO BCOS兼容以太坊虚拟机(EVM)，为了防止针对EVM的DOS攻击，EVM在执行交易时，引入了gas概念，用来度量智能合约执行过程中消耗的计算和存储资源，包括交易最大gas限制和区块最大gas限制，若交易或区块执行消耗的gas超过限制(gas limit)，则丢弃交易或区块。FISCO BCOS是联盟链，简化了gas设计，**仅保留交易最大gas限制，区块最大gas通过共识配置的max_trans_num和交易最大gas限制一起约束**。FISCO BCOS通过genesis的`[tx].gas_limit`来配置交易最大gas限制，默认是300000000，链初始化完毕后，可通过控制台指令动态调整gas限制。

```
[tx]
    gas_limit=300000000
```

EVM配置

FISCO BCOS v2.4.0引入`Free Storage` Gas衡量模式，提升CPU和内存在Gas消耗中的占比，详细可参考这里。`Free Storage` Gas模式的开启和关闭通过`genesis`文件的`evm.enable_free_storage`配置项控制。

注解:

- `evm.enable_free_storage` v2.4.0开始支持，当 `supported_version` 小于v2.4.0，或者旧链直接替换二进制升级时，不支持该特性
- 链初始化时，`evm.enable_free_storage` 写入创世块中；链初始化后，节点从创世块中读取 `evm.enable_free_storage` 配置项，手动修改 `genesis` 配置项不会生效
- `evm.enable_free_storage` 默认设置为false

- `evm.enable_free_storage`设置为true：开启`Free Storage` Gas模式
- `evm.enable_free_storage`设置为false：关闭`Free Storage` Gas模式

配置示例如下：

```
[evm]
    enable_free_storage=false
```

5.6.3 账本可变配置说明

账本可变配置位于节点`conf`目录下`.ini`后缀的文件中。

如：`group1`可变配置一般命名为`group.1.ini`，可变配置主要包括交易池大小、PBFT共识消息转发的TTL、PBFT共识打包时间设置、PBFT交易打包动态调整设置、并行交易设置等。

配置storage

存储目前支持RocksDB、MySQL、External三种模式，用户可以根据需要选择使用的DB，其中RocksDB性能最高；MySQL支持用户使用MySQL数据库，方便数据的查看；External通过数据代理访问mysql，用户需要在启动并配置数据代理。设计文档参考AMDB存储设计。RC3版本起我们使用RocksDB替代LevelDB以获得更好的性能表现，仍支持旧版本LevelDB。

注解:

- v2.3.0版本开始，为便于链的维护，推荐使用 *MySQL* 存储模式替代 *External* 存储模式
- 若要使用 *External*，请将 *supported_version* 配置成v2.2.0或其以下版本

公共配置项

重要： 推荐使用Mysql直连模式，配置type为MySQL。

- `type`：存储的DB类型，支持`RocksDB`、`MySQL`、`External`和`scalable`，不区分大小写。DB类型为RocksDB时，区块链系统所有数据存储于RocksDB本地数据库中；type为`MySQL`时，节点根据配置访问mysql数据库；type为`external`时，节点通过数据代理访问mysql数据库，AMDB代理配置请参考这里；type为`scalable`时，需要设置`binary_log=true`，此时状态数据和区块数据分别存储在不同的RocksDB实例中，存储区块数据的RocksDB实例根据配置项`scroll_threshold_multiple`*1000切换实例，实例以存储的起始区块高度命名。
- `max_capacity`：配置允许节点用于内存缓存的空间大小。
- `max_forward_block`：配置允许节点用于内存区块的大小，当节点出的区块超出该数值时，节点停止共识等待区块写入数据库。
- `binary_log`：当设置为`true`时打开binary_log，此时关闭RocksDB的WAL。
- `cached_storage`：控制是否使用缓存，默认`true`。

数据库相关配置项

- `topic`：当type为`External`时，需要配置该字段，表示区块链系统关注的AMDB代理topic，详细请参考这里。
- `max_retry`：当type为`External`时，需要配置该字段，表示写入失败时的重试次数，详细请参考这里。
- `scroll_threshold_multiple`：当type为`Scalable`时，此配置项用于配置区块数据库的切换阈值，按`scroll_threshold_multiple*1000`。默认为2，区块数据按每2000块存储在不同的RocksDB实例中。
- `db_ip`：当type为`MySQL`时，需要配置该字段，表示MySQL的IP地址。
- `db_port`：当type为`MySQL`时，需要配置该字段，表示MySQL的端口号。
- `db_username`：当type为`MySQL`时，需要配置该字段，表示MySQL的用户名。
- `db_passwd`：当type为`MySQL`时，需要配置该字段，表示MySQL用户对应的密码。
- `db_name`：当type为`MySQL`时，需要配置该字段，表示MySQL中使用的数据库名。
- `init_connections`：当type为`MySQL`时，可选配置该字段，表示与MySQL建立的初始连接数，默认15。使用默认值即可。
- `max_connections`：当type为`MySQL`时，可选配置该字段，表示与MySQL建立的最大连接数，默认20。使用默认值即可。

下面是[storage]的配置示例：

```
[storage]
    ; storage db type, rocksdb / mysql / external, rocksdb is recommended
    type=RocksDB
    max_capacity=256
```

(continues on next page)

(续上页)

```
    max_forward_block=10
    ; only for external
    max_retry=100
    topic=DB
    ; only for mysql
    db_ip=127.0.0.1
    db_port=3306
    db_username=
    db_passwd=
    db_name=
```

交易池配置

FISCO BCOS将交易池容量配置开放给用户，用户可根据自己的业务规模需求、稳定性需求以及节点的硬件配置动态调整交易池配置。

交易池容量限制

为防止过多交易堆积在交易池内占用太多内存，FISCO BCOS提供了[tx_pool].limit和[tx_pool].memory_limit两个配置项来限制交易池容量：

- [tx_pool].limit: 限制交易池内可以容纳的最大交易数目，默认为150000，超过该限制后，客户端发到节点的交易会被拒绝。
- [tx_pool].memory_limit: 交易池内交易占用的内存大小限制，默认为512MB，超过该限制后，客户端发到节点的交易会被拒绝。

交易池容量配置如下：

```
[tx_pool]
    limit=150000
    ; transaction pool memory size limit, MB
    memory_limit=512
```

交易池推送线程数配置

为提升区块链系统性能，FISCO BCOS采用了交易回执异步推送逻辑，当交易上链后，交易池内的推送线程会把交易上链的回执异步推送给客户端，为防止推送线程过多占用较多的系统资源，也为了防止推送线程过少影响交易推送的时效性，FISCO BCOS提供了[tx_pool].notify_worker_num配置项来配置异步推送线程数目：

- [tx_pool].notify_worker_num：异步推送线程数目，默认为2，建议该值不超过8

交易池推送线程数配置如下：

```
[tx_pool]
    ; number of threads responsible for transaction notification,
    ; default is 2, not recommended for more than 8
    notify_worker_num=2
```

PBFT共识配置

为提升PBFT算法的性能、可用性、网络效率，FISCO BCOS针对区块打包算法和网络做了一系列优化，包括PBFT区块打包动态调整策略、PBFT消息转发优化、PBFT Prepare包结构优化等。

注解： 因协议和算法一致性要求，建议保证所有节点PBFT共识配置一致。

PBFT共识消息转发配置

PBFT共识算法为了保证共识过程最大网络容错性，每个共识节点收到有效的共识消息后，会向其他节点广播该消息，在网络较好的环境下，共识消息转发机制会造成额外的网络带宽浪费，因此在群组可变配置项中引入了`ttl`来控制消息最大转发次数，消息最大转发次数为`ttl-1`，**该配置项仅对PBFT有效**。

设置共识消息最多转发一次，配置示例如下：

```
; the ttl for broadcasting pbft message
[consensus]
ttl=2
```

PBFT共识打包时间配置

考虑到PBFT模块打包太快会导致某些区块中仅打包1到2个很少的交易，浪费存储空间，FISCO BCOS v2.0.0-rc2在群组可变配置`group.group_id.ini`的`[consensus]`下引入`min_block_generation_time`配置项来控制PBFT共识打包的最短时间，即：共识节点打包时间超过`min_block_generation_time`且打包的交易数大于0才会开始共识流程，处理打包生成的新区块。

重要：

- `min_block_generation_time` 默认为500ms
- 共识节点最长打包时间为1000ms，若超过1000ms新区块中打包到的交易数仍为0，共识模块会进入出空块逻辑，空块并不落盘；
- `min_block_generation_time` 不可超过出空块时间1000ms，若设置值超过1000ms，系统默认min_block_generation_time为500ms

```
[consensus]
;min block generation time(ms), the max block generation time is 1000 ms
min_block_generation_time=500
```

PBFT交易打包动态调整

考虑到CPU负载和网络延迟对系统处理能力的影响，PBFT提供了动态调整一个区块内可打包最大交易数的算法，该算法会根据历史交易处理情况动态调整区块内可打包的最大交易数，默认开启，也可通过将可变配置`group.group_id.ini`的`[consensus].enable_dynamic_block_size`配置项修改为`false`来关闭该算法，此时区块内可打包的最大交易数为`group.group_id.genesis`的`[consensus].max_trans_num`。

关闭区块打包交易数动态调整算法的配置如下：

```
[consensus]
    enable_dynamic_block_size=false
```

PBFT消息转发配置

FISCO BCOS v2.2.0优化了PBFT消息转发机制，保证网络断连场景下PBFT消息包能尽量到达每个共识节点的同时，降低网络中冗余的PBFT消息包，PBFT消息转发优化策略请参考这里。可通过`group.group_id.ini`的`[consensus].enable_ttl_optimization`配置项开启或关闭PBFT消息转发优化策略。

- `[consensus].enable_ttl_optimization`配置为`true`：打开PBFT消息转发优化策略
- `[consensus].enable_ttl_optimization`配置为`false`：关闭PBFT消息转发优化策略
- `supported_version`不小于v2.2.0时，默认打开PBFT消息转发策略；`supported_version`小于v2.2.0时，默认关闭PBFT消息转发优化策略

关闭PBFT消息转发优化策略配置如下：

```
[consensus]
    enable_ttl_optimization=false
```

PBFT Prepare包结构优化

考虑到PBFT算法中，Leader广播的Prepare包内区块的交易有极大概率在其他共识节点的交易池中命中，为了节省网络带宽，FISCO BCOS v2.2.0优化了Prepare包结构：Prepare包内的区块仅包含交易哈希列表，其他共识节点收到Prepare包后，优先从本地交易池获取命中的交易，并向Leader请求缺失的交易，详细设计请参考这里。可通过`group.group_id.ini`的`[consensus].enable_prepare_with_txsHash`配置项开启或关闭该策略。

- `[consensus].enable_prepare_with_txsHash`配置为`true`：打开Prepare包结构优化，Prepare消息包内区块仅包含交易哈希列表
- `[consensus].enable_prepare_with_txsHash`配置为`false`：关闭Prepare包结构优化，Prepare消息包内区块包含全量的交易
- `supported_version`不小于v2.2.0时，`[consensus].enable_prepare_with_txsHash`默认为`true`；`supported_version`小于v2.2.0时，`[consensus].enable_prepare_with_txsHash`默认为`false`

注解： 因协议一致性要求，须保证所有节点 *enable_prepare_with_txsHash* 配置一致

关闭PBFT Prepare包结构优化配置如下：

```
[consensus]
    enable_prepare_with_txsHash=false
```

rPBFT共识配置

FISCO BCOS v2.3.0引入rPBFT共识算法，具体可参考这里，为保证rPBFT算法网络流量负载均衡，引入了Prepare包树状广播策略以及该策略相对应的容错方案。

- `[consensus].broadcast_prepare_by_tree`：Prepare包树状广播策略开启/关闭开关，设置为`true`，开启Prepare包树状广播策略；设置为`false`，关闭Prepare包树状广播策略，默认为`true`

下面为开启Prepare包树状广播策略后的容错配置：

- `[consensus].prepare_status_broadcast_percent`：Prepare状态包随机广播的节点占共识节点总数的百分比，取值在25到100之间，默认为33

- `[consensus].max_request_prepare_waitTime`：节点Prepare缓存缺失时，等待父节点发送Prepare包的最长时延，默认为100ms，超过这个时延后，节点会向其他拥有该Prepare包的节点请求

下面为rPBFT模式下开启Prepare包结构优化后，负载均衡相关配置：

- `[consensus].max_request_missedTxs_waitTime`：节点Prepare包内交易缺失后，等待父节点或其他非leader节点同步Prepare包状态的最长时延，默认为100ms，若在等待时延窗口内同步到父节点或非leader节点Prepare包状态，则会随机选取一个节点请求缺失交易；若等待超时，直接向leader请求缺失交易。

rPBFT默认配置如下：

```
; 默认开启Prepare包树状广播策略
broadcast_prepare_by_tree=true
; 仅在开启prepare包树状广播时生效
; 每个节点随机选取33%共识节点同步prepare包状态
prepare_status_broadcast_percent=33
; prepare包树状广播策略下，缺失prepare包的节点超过100ms没等到父节点转发的prepare包，会向其他节点请求缺失的prepare包
max_request_prepare_waitTime=100
; 节点等待父节点或其他非leader节点同步prepare包最长时延为100ms
max_request_missedTxs_waitTime=100
```

同步配置

同步模块是"网络消耗大户"，包括区块同步和交易同步，FISCO BCOS秉着负载均衡的原则优化了共识模块网络使用效率。

注解： 因协议一致性要求，建议保证所有节点PBFT共识配置一致。

区块同步优化配置

为了增强区块链系统在网络带宽受限情况下的可扩展性，FISCO BCOS v2.2.0对区块同步进行了优化，详细的优化策略请参考这里。可通过`group.group_id.ini`的`[sync].sync_block_by_tree`开启或关闭区块同步优化策略。

- `[sync].sync_block_by_tree`配置为`true`：打开区块同步优化策略
- `[sync].sync_block_by_tree`配置为`false`：关闭区块同步优化策略
- `supported_version`不小于v2.2.0时，`[sync].sync_block_by_tree`默认为`true`；`supported_version`小于v2.2.0时，`[sync].sync_block_by_tree`默认为`false`

此外，为了保障树状拓扑区块同步的健壮性，FISCO BCOS v2.2.0还引入了gossip协议定期同步区块状态，gossip协议相关配置项均位于`group.group_id.ini`的`[sync]`中，具体如下：

- `gossip_interval_ms`：gossip协议同步区块状态周期，默认为1000ms
- `gossip_peers_number`：节点每次同步区块状态时，随机选取的邻居节点数目，默认为3

注解：

1. gossip协议配置项，仅在开启区块树状广播优化时生效
2. 必须保证所有节点 *sync_block_by_tree* 配置一致

开启区块树状广播优化配置如下：

```
[sync]
    ; 默认开启区块树状同步策略
    sync_block_by_tree=true
    ; 每个节点每隔1000ms同步一次最新区块状态
    gossip_interval_ms=1000
    ; 每个节点每次随机选择3个邻居节点同步最新区块状态
    gossip_peers_number=3
```

交易树状广播优化配置

为了降低SDK直连节点的峰值出带宽，提升区块链系统可扩展性，FISCO BCOS v2.2.0引入了交易树状广播优化策略，详细设计请参考这里。可通过group.group_id.ini的[sync].send_txs_by_tree开启或关闭交易树状广播策略，详细配置如下：

- [sync].sync_block_by_tree：设置为true，打开交易树状广播策略；设置为false，关闭交易树状广播优化策略

关闭交易树状广播策略的配置如下：

```
[sync]
    ; 默认开启交易树状广播策略
    send_txs_by_tree=false
```

注解：

- 由于协议一致性需求，须保证所有节点交易树状广播开关'send_txs_by_tree'配置一致
- *supported_version* 不小于v2.2.0时，默认打开交易树状广播优化策略；*supported_version* 小于v2.2.0时，默认关闭交易树状广播策略

交易转发优化配置

为降低交易转发导致的流量开销，FISCO BCOS v2.2.0引入基于状态包的交易转发策略，具体设计可参考这里。可通过group.group_id.ini的[sync].txs_max_gossip_peers_num配置交易状态最多转发节点数目，默认为5。

注解： 为保障交易到达每个节点的同时，尽量降低交易状态转发引入的流量开销，不建议将 *txs_max_gossip_peers_num* 设置太小或太大，直接使用默认配置即可

交易状态转发最大节点数配置如下：

```
[sync]
    ; 每个节点每轮最多随机选择5个邻居节点同步最新交易状态
    txs_max_gossip_peers_num=5
```

并行交易配置

FISCO BCOS支持交易的并行执行。开启交易并行执行开关，能够让区块内的交易被并行的执行，提高吞吐量，**交易并行执行仅在storage state模式下生效**。

注解：

为简化系统配置，v2.3.0去除了 *enable_parallel* 配置项，该配置项仅在 *supported_version < v2.3.0* 时生效，v2.3.0版本中

- storageState模式：开启并行交易
- mptState模式: 关闭并行交易

```
[tx_execute]
    enable_parallel=true
```

可选配置：群组流量控制

为了防止多群组间资源相互影响，FISCO BCOS v2.5.0引入了流量控制功能，支持群组级别的SDK请求速率限制以及流量限制，配置位于`group.{group_id}.ini`的`[flow_control]`，默认关闭，流控的详细设计请参考这里。

SDK到群组的请求速率限制配置

群组内的SDK请求速率限制位于配置项`[flow_control].limit_req`中，用于限制SDK每秒到群组的最大请求数目，当每秒到节点的请求超过配置项的值时，请求会被拒绝，**SDK到群组请求速率限制默认关闭，若要开启该功能，需要将`limit_req`配置项前面的;去掉**，打开SDK请求速率限制并配置群组每秒可接受1000个SDK请求的示例如下：

```
[flow_control]
    ; restrict QPS of the group
    limit_req=1000
```

群组间流量限制配置

为了防止区块同步占用过多的网络流量影响到共识模块的消息包传输，FISCO BCOS v2.5.0引入了群组级流量限制的功能，其配置了群组平均出带宽的上限，但不限制区块共识、交易同步的流量，当群组平均出带宽超过配置值时，会暂缓区块发送。

- `[flow_control].outgoing_bandwidth_limit`：群组出带宽限制，单位为`Mbit/s`，当群组出带宽超过该值时，会暂缓发送区块，但不会限制区块共识和交易广播的流量，**该配置项默认关闭，若要打开流量限制功能，请将`outgoing_bandwidth_limit`配置项前面的;去掉**。

打开群组出带宽流量限制，并将其设置为`2MBit/s`的配置示例如下：

```
[flow_control]
    ; Mb, can be a decimal
    ; when the outgoing bandwidth exceeds the limit, the block synchronization␣
↪operation will not proceed
    outgoing_bandwidth_limit=2
```

5.6.4 动态配置系统参数

FISCO BCOS系统目前主要包括如下系统参数(未来会扩展其他系统参数)：

控制台提供 **setSystemConfigByKey** 命令来修改这些系统参数，**getSystemConfigByKey** 命令可查看系统参数的当前值：

重要：不建议随意修改tx_count_limit和tx_gas_limit，如下情况可修改这些参数：

- 机器网络或CPU等硬件性能有限：调小tx_count_limit，或降低业务压力；
- 业务逻辑太复杂，执行交易时gas不足：调大tx_gas_limit。

rpbft_epoch_sealer_num 和 *rpbft_epoch_block_num* 仅对rPBFT共识算法生效，为了保障共识性能，不建议频繁动态切换共识列表，即不建议 *rpbft_epoch_block_num* 配置值太小

```
# 设置一个区块可打包最大交易数为500
[group:1]> setSystemConfigByKey tx_count_limit 500
# 查询tx_count_limit
[group:1]> getSystemConfigByKey tx_count_limit
[500]

# 设置交易gas限制为400000000
[group:1]> setSystemConfigByKey tx_gas_limit 400000000
[group:1]> getSystemConfigByKey tx_gas_limit
[400000000]

# rPBFT共识算法下，设置一个共识周期选取参与共识的节点数目为4
[group:1]> setSystemConfigByKey rpbft_epoch_sealer_num 4
Note: rpbft_epoch_sealer_num only takes effect when rPBFT is used
{
    "code":0,
    "msg":"success"
}
# 查询rpbft_epoch_sealer_num
[group:1]> getSystemConfigByKey rpbft_epoch_sealer_num
Note: rpbft_epoch_sealer_num only takes effect when rPBFT is used
4

# rPBFT共识算法下，设置一个共识周期出块数目为10000
[group:1]> setSystemConfigByKey rpbft_epoch_block_num 10000
Note: rpbft_epoch_block_num only takes effect when rPBFT is used
{
    "code":0,
    "msg":"success"
}
# 查询rpbft_epoch_block_num
[group:1]> getSystemConfigByKey rpbft_epoch_block_num
Note: rpbft_epoch_block_num only takes effect when rPBFT is used
10000
```

5.7 多群组部署

本章主要以星形组网和并行多组组网拓扑为例，指导您了解如下内容：

- 了解如何使用build_chain.sh创建多群组区块链安装包；
- 了解build_chain.sh创建的多群组区块链安装包目录组织形式；
- 学习如何启动该区块链节点，并通过日志查看各群组共识状态；
- 学习如何向各群组发送交易，并通过日志查看群组出块状态；
- 了解群组内节点管理，包括节点入网、退网等；
- 了解如何新建群组。

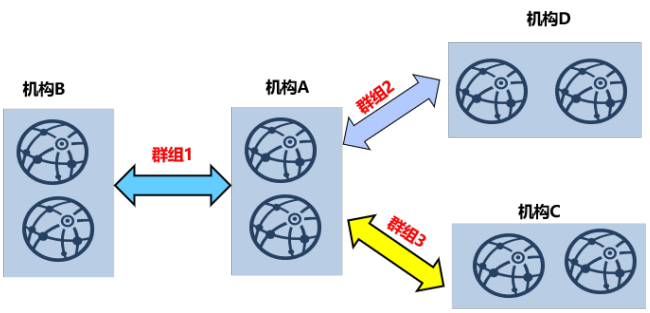
重要：

- build_chain.sh适用于开发者和体验者快速搭链使用
- 搭建企业级业务链，推荐使用 企业搭链工具

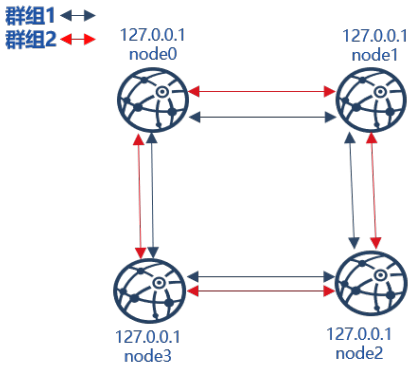
5.7.1 星形拓扑和并行多组

如下图，星形组网拓扑和并行多组组网拓扑是区块链应用中使用较广泛的两种组网方式。

- **星形拓扑**：中心机构节点同时属于多个群组，运行多家机构应用，其他每家机构属于不同群组，运行各自应用；
- **并行多组**：区块链中每个节点均属于多个群组，可用于多方不同业务的横向扩展，或者同一业务的纵向扩展。



星形拓扑



并行多组

下面以构建八节点星形拓扑和四节点并行多组区块链为例，详细介绍多群组操作方法。

5.7.2 安装依赖

部署FISCO BCOS区块链节点前，需安装`openssl`, `curl`等依赖软件，具体命令如下：

```
# CentOS
$ sudo yum install -y openssl curl

# Ubuntu
$ sudo apt install -y openssl curl

# Mac OS
$ brew install openssl curl
```

5.7.3 星形拓扑

本章以构建上图所示的**单机、四机构、三群组、八节点的星形组网拓扑**为例，介绍多群组使用方法。

星形区块链组网如下：

- `agencyA`：在`127.0.0.1`上有2个节点，同时属于`group1`、`group2`、`group3`；
- `agencyB`：在`127.0.0.1`上有2个节点，属于`group1`；
- `agencyC`：在`127.0.0.1`上有2个节点，属于`group2`；
- `agencyD`：在`127.0.0.1`上有2个节点，属于`group3`。

重要:

- 实际应用场景中，**不建议将多个节点部署在同一台机器**，建议根据 **机器负载** 选择部署节点数目，请参考 硬件配置
- **星形网络拓扑** 中，核心节点(本例中agencyA节点)属于所有群组，负载较高，**建议单独部署于性能较好的机器**

- **在不同机器操作时，请将生成的对应IP的文件夹拷贝到对应机器启动，建链操作只需要执行一次！**

构建星形区块链节点配置文件夹

build_chain.sh支持任意拓扑多群组区块链构建，可使用该脚本构建星形拓扑区块链节点配置文件夹：

准备依赖

- 创建操作目录

```
mkdir -p ~/fisco && cd ~/fisco
```

- 获取build_chain.sh脚本

```
curl -LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.5.0/build_
↪chain.sh && chmod u+x build_chain.sh
```

生成星形区块链系统配置文件

```
# 生成区块链配置文件ip_list
$ cat > ipconf << EOF
127.0.0.1:2 agencyA 1,2,3
127.0.0.1:2 agencyB 1
127.0.0.1:2 agencyC 2
127.0.0.1:2 agencyD 3
EOF

# 查看配置文件ip_list内容
$ cat ipconf
# 空格分隔的参数分别表示如下含义:
# ip:num: 物理机IP以及物理机上的节点数目
# agency_name: 机构名称
# group_list: 节点所属的群组列表，不同群组以逗号分隔
127.0.0.1:2 agencyA 1,2,3
127.0.0.1:2 agencyB 1
127.0.0.1:2 agencyC 2
127.0.0.1:2 agencyD 3
```

使用build_chain脚本构建星形区块链节点配置文件夹

build_chain更多参数说明请参考这里。

```
# 根据配置生成星形区块链 需要保证机器的30300~30301，20200~20201，8545~8546端口没有被占用
$ bash build_chain.sh -f ipconf -p 30300,20200,8545
Generating CA key...
================================================================
Generating keys ...
Processing IP:127.0.0.1 Total:2 Agency:agencyA Groups:1,2,3
Processing IP:127.0.0.1 Total:2 Agency:agencyB Groups:1
Processing IP:127.0.0.1 Total:2 Agency:agencyC Groups:2
Processing IP:127.0.0.1 Total:2 Agency:agencyD Groups:3
================================================================
......此处省略其他输出......
================================================================
[INFO] FISCO-BCOS Path   : ./bin/fisco-bcos
[INFO] IP List File      : ipconf
[INFO] Start Port        : 30300 20200 8545
[INFO] Server IP         : 127.0.0.1:2 127.0.0.1:2 127.0.0.1:2 127.0.0.1:2
[INFO] State Type        : storage
[INFO] RPC listen IP     : 127.0.0.1
```

(continues on next page)

(续上页)

```
[INFO] Output Dir        : /home/ubuntu16/fisco/nodes
[INFO] CA Key Path       : /home/ubuntu16/fisco/nodes/cert/ca.key
============================================================
[INFO] All completed. Files in /home/ubuntu16/fisco/nodes

# 生成的节点文件如下
nodes
|-- 127.0.0.1
|   |-- fisco-bcos
|   |-- node0
|   |   |-- conf  #节点配置目录
|   |   |   |-- ca.crt
|   |   |   |-- group.1.genesis
|   |   |   |-- group.1.ini
|   |   |   |-- group.2.genesis
|   |   |   |-- group.2.ini
|   |   |   |-- group.3.genesis
|   |   |   |-- group.3.ini
|   |   |   |-- node.crt
|   |   |   |-- node.key
|   |   |   `-- node.nodeid # 记录节点Node ID信息
|   |   |-- config.ini #节点配置文件
|   |   |-- start.sh  #节点启动脚本
|   |   `-- stop.sh   #节点停止脚本
|   |-- node1
|   |   |-- conf
......此处省略其他输出......
```

注解： 若生成的区块链节点属于不同物理机，需要将区块链节点拷贝到相应的物理机

启动节点

节点提供start_all.sh和stop_all.sh脚本启动和停止节点。

```
# 进入节点目录
$ cd ~/fisco/nodes/127.0.0.1

# 启动节点
$ bash start_all.sh

# 查看节点进程
$ ps aux | grep fisco-bcos
ubuntu16          301  0.8  0.0 986644  7452 pts/0    Sl   15:21   0:00 /home/
↪ubuntu16/fisco/nodes/127.0.0.1/node5/../fisco-bcos -c config.ini
ubuntu16          306  0.9  0.0 986644  6928 pts/0    Sl   15:21   0:00 /home/
↪ubuntu16/fisco/nodes/127.0.0.1/node6/../fisco-bcos -c config.ini
ubuntu16          311  0.9  0.0 986644  7184 pts/0    Sl   15:21   0:00 /home/
↪ubuntu16/fisco/nodes/127.0.0.1/node7/../fisco-bcos -c config.ini
ubuntu16       131048  2.1  0.0 1429036 7452 pts/0    Sl   15:21   0:00 /home/
↪ubuntu16/fisco/nodes/127.0.0.1/node0/../fisco-bcos -c config.ini
ubuntu16       131053  2.1  0.0 1429032 7180 pts/0    Sl   15:21   0:00 /home/
↪ubuntu16/fisco/nodes/127.0.0.1/node1/../fisco-bcos -c config.ini
ubuntu16       131058  0.8  0.0 986644  7928 pts/0    Sl   15:21   0:00 /home/
↪ubuntu16/fisco/nodes/127.0.0.1/node2/../fisco-bcos -c config.ini
ubuntu16       131063  0.8  0.0 986644  7452 pts/0    Sl   15:21   0:00 /home/
↪ubuntu16/fisco/nodes/127.0.0.1/node3/../fisco-bcos -c config.ini
ubuntu16       131068  0.8  0.0 986644  7672 pts/0    Sl   15:21   0:00 /home/
↪ubuntu16/fisco/nodes/127.0.0.1/node4/../fisco-bcos -c config.ini
```

查看群组共识状态

不发交易时，共识正常的节点会输出+++日志，本例中，node0、node1同时属于group1、group2和group3；node2、node3属于group1；node4、node5属于group2；node6、node7属于group3，可通过tail -f node*/log/* | grep "++"查看各节点是否正常。

重要:

节点正常共识打印 +++ 日志， +++ 日志字段含义:

- g:: 群组ID
- blkNum: Leader节点产生的新区块高度;
- tx: 新区块中包含的交易数目;
- nodeIdx: 本节点索引;
- hash: 共识节点产生的最新区块哈希。

```
# 查看node0 group1是否正常共识（Ctrl+c退回命令行）
$ tail -f node0/log/* | grep "g:1.*++"
info|2019-02-11 15:33:09.914042| [g:1][p:264][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=2,hash=72254a42....

# 查看node0 group2是否正常共识
$ tail -f node0/log/* | grep "g:2.*++"
info|2019-02-11 15:33:31.021697| [g:2][p:520][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=3,hash=ef59cf17...

# ... 查看node1, node2节点每个群组是否正常可参考以上操作方法...

# 查看node3 group1是否正常共识
$ tail -f node3/log/*| grep "g:1.*++"
info|2019-02-11 15:39:43.927167| [g:1][p:264][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=3,hash=5e94bf63...

# 查看node5 group2是否正常共识
$ tail -f node5/log/* | grep "g:2.*++"
info|2019-02-11 15:39:42.922510| [g:2][p:520][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=2,hash=b80a724d...
```

配置控制台

控制台通过Web3SDK链接FISCO BCOS节点，实现查询区块链状态、部署调用合约等功能，能够快速获取到所需要的信息。控制台指令详细介绍参考这里。

重要: 控制台依赖于Java 8以上版本，Ubuntu 16.04系统安装openjdk 8即可。CentOS请安装Oracle Java 8以上版本。如果因为网络问题导致长时间无法下载，请尝试 *curl -LO https://gitee.com/FISCO-BCOS/console/raw/master/tools/download_console.sh && bash download_console.sh*

```
#回到fisco目录
$ cd ~/fisco

# 获取控制台
$ curl -LO https://github.com/FISCO-BCOS/console/releases/download/v1.0.9/download_
↪console.sh && bash download_console.sh

# 进入控制台操作目录
$ cd console
```

(continues on next page)

(续上页)

```
# 拷贝group2节点证书到控制台配置目录
$ cp ~/fisco/nodes/127.0.0.1/sdk/* conf/

# 获取node0的channel_listen_port
$ grep "channel_listen_port" ~/fisco/nodes/127.0.0.1/node*/config.ini
/home/ubuntu16/fisco/nodes/127.0.0.1/node0/config.ini:    channel_listen_port=20200
/home/ubuntu16/fisco/nodes/127.0.0.1/node1/config.ini:    channel_listen_port=20201
/home/ubuntu16/fisco/nodes/127.0.0.1/node2/config.ini:    channel_listen_port=20202
/home/ubuntu16/fisco/nodes/127.0.0.1/node3/config.ini:    channel_listen_port=20203
/home/ubuntu16/fisco/nodes/127.0.0.1/node4/config.ini:    channel_listen_port=20204
/home/ubuntu16/fisco/nodes/127.0.0.1/node5/config.ini:    channel_listen_port=20205
/home/ubuntu16/fisco/nodes/127.0.0.1/node6/config.ini:    channel_listen_port=20206
/home/ubuntu16/fisco/nodes/127.0.0.1/node7/config.ini:    channel_listen_port=20207
```

重要： 使用控制台连接节点时，控制台连接的节点必须在控制台配置的组中

创建控制台配置文件conf/applicationContext.xml的配置如下，控制台从node0(127.0.0.1:20200)分别接入三个group中，控制台配置方法请参考这里。

```
cat > ./conf/applicationContext.xml << EOF
<?xml version="1.0" encoding="UTF-8" ?>

<beans xmlns="http://www.springframework.org/schema/beans"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://
↪www.springframework.org/schema/p"
           xmlns:tx="http://www.springframework.org/schema/tx" xmlns:aop="http://
↪www.springframework.org/schema/aop"
           xmlns:context="http://www.springframework.org/schema/context"
           xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
         http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
         http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">


        <bean id="encryptType" class="org.fisco.bcos.web3j.crypto.EncryptType">
                <constructor-arg value="0"/> <!-- 0:standard 1:guomi -->
        </bean>

      <bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.
↪handler.GroupChannelConnectionsConfig">
        <property name="allChannelConnections">
        <list>
            <bean id="group1"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                <property name="groupId" value="1" />
                    <property name="connectionsStr">
                    <list>
                    <value>127.0.0.1:20200</value>
                    </list>
                    </property>
            </bean>
            <bean id="group2"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                <property name="groupId" value="2" />
                    <property name="connectionsStr">
                    <list>
```

(continues on next page)

(续上页)

```
                    <value>127.0.0.1:20200</value>
                    </list>
                    </property>
            </bean>
            <bean id="group3"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                <property name="groupId" value="3" />
                    <property name="connectionsStr">
                    <list>
                    <value>127.0.0.1:20200</value>
                    </list>
                    </property>
            </bean>
        </list>
        </property>
        </bean>

        <bean id="channelService" class="org.fisco.bcos.channel.client.Service"␣
↪depends-on="groupChannelConnectionsConfig">
                <property name="groupId" value="1" />
                <property name="orgID" value="fisco" />
                <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>
        </bean>
</beans>
EOF
```

启动控制台

```
$ bash start.sh
# 输出下述信息表明启动成功 否则请检查conf/applicationContext.xml中节点端口配置是否正确
=====================================================================================================
Welcome to FISCO BCOS console(1.0.3)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ _______  _______   _______  _______       ________  _______  _______  _______
|        |       \/       \ /       \ /       \     |        \ /       \ /       \ /       ␣
↪\
| $$$$$$$$\$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
| $$__      | $$ | $$___\$| $$    \$| $$  | $$    | $$__/ $| $$    \$| $$  | $| $$___\
↪$$
| $$  \     | $$  \$$     \| $$      | $$  | $$    | $$    $| $$      | $$  | $$\$$    ␣
↪\
| $$$$$     | $$  _\$$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$\
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__|
↪$$
| $$      |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$
↪$


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

向群组发交易

上节配置了控制台，本节通过控制台向各群组发交易。

重要： 多群组架构中，群组间账本相互独立，向某个群组发交易仅会导致本群组区块高度增加，不会增

加其他群组区块高度

控制台发送交易

```
# ... 向group1发交易...
$ [group:1]> deploy HelloWorld
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
# 查看group1当前块高，块高增加为1表明出块正常，否则请检查group1是否共识正常
$ [group:1]> getBlockNumber
1

# ... 向group2发交易...
# 切换到group2
$ [group:1]> switch 2
Switched to group 2.
# 向group2发交易，返回交易哈希表明交易部署成功，否则请检查group2是否共识正常
$ [group:2]> deploy HelloWorld
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
# 查看group2当前块高，块高增加为1表明出块正常，否则请检查group2是否共识正常
$ [group:2]> getBlockNumber
1

# ... 向group3发交易...
# 切换到group3
$ [group:2]> switch 3
Switched to group 3.
# 向group3发交易，返回交易哈希表明交易部署成功
$ [group:3]> deploy HelloWorld
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
# 查看group3当前块高，块高为1表明出块正常，否则请检查group3是否共识正常
$ [group:3]> getBlockNumber
1

# ... 切换到不存在的组4，控制台提示group4不存在，并输出当前的group列表 ...
$ [group:3]> switch 4
Group 4 does not exist. The group list is [1, 2, 3].

# 退出控制台
$ [group:3]> exit
```

查看日志

节点出块后，会输出Report日志，日志各个字段含义如下：

重要:

节点每出一个新块，会打印一条Report日志，Report日志中各字段含义如下：

- g:：群组ID
- num：出块高度；
- sealerIdx：共识节点索引；
- hash：区块哈希；
- next：下一个区块高度；
- tx：区块包含的交易数；
- nodeIdx：当前节点索引。

```
# 进入节点目录
$ cd ~/fisco/nodes/127.0.0.1

# 查看group1出块情况：有新区块产生
$ cat node0/log/* |grep "g:1.*Report"
info|2019-02-11 16:08:45.077484| [g:1][p:264][CONSENSUS][PBFT]^^^^^^^^Report,num=1,
↪sealerIdx=1,hash=9b5487a6...,next=2,tx=1,nodeIdx=2

# 查看group2出块情况：有新区块产生
$ cat node0/log/* |grep "g:2.*Report"
info|2019-02-11 16:11:55.354881| [g:2][p:520][CONSENSUS][PBFT]^^^^^^^^Report,num=1,
↪sealerIdx=0,hash=434b6e07...,next=2,tx=1,nodeIdx=0

# 查看group3出块情况：有新区块产生
$ cat node0/log/* |grep "g:3.*Report"
info|2019-02-11 16:14:33.930978| [g:3][p:776][CONSENSUS][PBFT]^^^^^^^^Report,num=1,
↪sealerIdx=1,hash=3a42fcd1...,next=2,tx=1,nodeIdx=2
```

节点加入群组

通过控制台，FISCO BCOS可将指定节点加入到指定群组，也可将节点从指定群组删除，详细介绍请参考节点准入管理手册，控制台配置参考控制台操作手册。

本章以将node2加入group2为例，介绍如何在已有的群组中，加入新节点。

重要： 新节点加入群组前，请确保：

- 新加入NodeID存在
- 群组内节点正常共识：正常共识的节点会输出+++日志

拷贝group2群组配置到node2

```
# 进入节点目录
$ cd ~/fisco/nodes/127.0.0.1

# ... 从node0拷贝group2的配置到node2...
$ cp node0/conf/group.2.* node2/conf

# ...重启node2(重启后请确定节点正常共识)...
$ cd node2 && bash stop.sh && bash start.sh
```

获取node2的节点ID

```
# 请记住node2的node ID，将node2加入到group2需用到该node ID
$ cat conf/node.nodeid
6dc585319e4cf7d73ede73819c6966ea4bed74aadbbcba1bbb777132f63d355965c3502bed7a04425d99cdcfb7694a1c1
```

通过控制台向group2发送命令，将node2加入到group2

```
# ...回到控制台目录，并启动控制台（直接启动到group2）...
$ cd ~/fisco/console && bash start.sh 2

# ...通过控制台将node2加入为共识节点...
# 1. 查看当前共识节点列表
$ [group:2]> getSealerList
[

↪9217e87c6b76184cf70a5a77930ad5886ea68aefbcce1909bdb799e45b520baa53d5bb9a5edddeab94751df179d54d4
↪
```

(continues on next page)

(续上页)

```
↪227c600c2e52d8ec37aa9f8de8db016ddc1c8a30bb77ec7608b99ee2233480d4c06337d2461e24c26617b6fd53acfa6
↪

↪7a50b646fcd9ac7dd0b87299f79ccaa2a4b3af875bd0947221ba6dec1c1ba4add7f7f690c95cf3e796296cf4adc989f
↪

↪8b2c4204982d2a2937261e648c20fe80d256dfb47bda27b420e76697897b0b0ebb42c140b4e8bf0f27dfee64c946039
]
# 2. 将node2加入到共识节点
# addSealer后面的参数是上步获取的node ID
$ [group:2]> addSealer
↪6dc585319e4cf7d73ede73819c6966ea4bed74aadbbcba1bbb777132f63d355965c3502bed7a04425d99cdcfb7694a1
{
    "code":0,
    "msg":"success"
}
# 3. 查看共识节点列表
$ [group:2]> getSealerList
[

↪9217e87c6b76184cf70a5a77930ad5886ea68aefbcce1909bdb799e45b520baa53d5bb9a5edddeab94751df179d54d4
↪

↪227c600c2e52d8ec37aa9f8de8db016ddc1c8a30bb77ec7608b99ee2233480d4c06337d2461e24c26617b6fd53acfa6
↪

↪7a50b646fcd9ac7dd0b87299f79ccaa2a4b3af875bd0947221ba6dec1c1ba4add7f7f690c95cf3e796296cf4adc989f
↪

↪8b2c4204982d2a2937261e648c20fe80d256dfb47bda27b420e76697897b0b0ebb42c140b4e8bf0f27dfee64c946039
↪

↪6dc585319e4cf7d73ede73819c6966ea4bed74aadbbcba1bbb777132f63d355965c3502bed7a04425d99cdcfb7694a1
↪# 新加入节点
]
# 获取group2当前块高
$ [group:2]> getBlockNumber
2

#... 向group2发交易
# 部署HelloWorld合约，输出合约地址，若合约部署失败，请检查group2共识情况
$ [group:2] deploy HelloWorld
contract address:0xdfdd3ada340d7346c40254600ae4bb7a6cd8e660

# 获取group2当前块高，块高增加为3，若块高不变，请检查group2共识情况
$ [group:2]> getBlockNumber
3

# 退出控制台
$ [group:2]> exit
```

通过日志查看新加入节点出块情况

```
# 进入节点所在目录
cd ~/fisco/nodes/127.0.0.1
# 查看节点共识情况（Ctrl+c退回命令行）
$ tail -f node2/log/* | grep "g:2.*++"
info|2019-02-11 18:41:31.625599| [g:2][p:520][CONSENSUS][SEALER]++++++++Generating
↪seal on,blkNum=4,tx=0,nodeIdx=1,hash=c8a1ed9c...
......此处省略其他输出......
```

(continues on next page)

(续上页)

```
# 查看node2 group2出块情况：有新区块产生
$ cat node2/log/* | grep "g:2.*Report"
info|2019-02-11 18:53:20.708366| [g:2][p:520][CONSENSUS][PBFT]^^^^^Report:,num=3,
↪idx=3,hash=80c98d31...,next=10,tx=1,nodeIdx=1
# node2也Report了块高为3的区块，说明node2已经加入group2
```

停止节点

```
# 回到节点目录 && 停止节点
$ cd ~/fisco/nodes/127.0.0.1 && bash stop_all.sh
```

5.7.4 并行多组

并行多组区块链搭建方法与星形拓扑区块链搭建方法类似，以搭建四节点两群组并行多链系统为例：

- 群组1：包括四个节点，节点IP均为127.0.0.1；
- 群组2：包括四个节点，节点IP均为127.0.0.1。

重要:

- 真实应用场景中，**不建议将多个节点部署在同一台机器**，建议根据 **机器负载** 选择部署节点数目
- 为演示并行多组扩容流程，这里仅先创建group1
- 并行多组场景中，节点加入和退出群组操作与星形组网拓扑类似

构建单群组四节点区块链

用build_chain.sh脚本生成单群组四节点区块链节点配置文件夹

```
$ mkdir -p ~/fisco && cd ~/fisco
# 获取build_chain.sh脚本
$ curl -LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.5.0/build_
↪chain.sh && chmod u+x build_chain.sh
# 构建本机单群组四节点区块链(生产环境中，建议每个节点部署在不同物理机上)
$ bash build_chain.sh -l "127.0.0.1:4" -o multi_nodes -p 20000,20100,7545
Generating CA key...
================================================================
Generating keys ...
Processing IP:127.0.0.1 Total:4 Agency:agency Groups:1
================================================================
Generating configurations...
Processing IP:127.0.0.1 Total:4 Agency:agency Groups:1
================================================================
[INFO] FISCO-BCOS Path   : bin/fisco-bcos
[INFO] Start Port        : 20000 20100 7545
[INFO] Server IP         : 127.0.0.1:4
[INFO] State Type        : storage
[INFO] RPC listen IP     : 127.0.0.1
[INFO] Output Dir        : /home/ubuntu16/fisco/multi_nodes
[INFO] CA Key Path       : /home/ubuntu16/fisco/multi_nodes/cert/ca.key
================================================================
[INFO] All completed. Files in /home/ubuntu16/fisco/multi_nodes
```

启动所有节点

```
# 进入节点目录
$ cd ~/fisco/multi_nodes/127.0.0.1
$ bash start_all.sh

# 查看进程情况
$ ps aux | grep fisco-bcos
ubuntu16       55028  0.9  0.0 986384  6624 pts/2    Sl   20:59   0:00 /home/
↪ubuntu16/fisco/multi_nodes/127.0.0.1/node0/../fisco-bcos -c config.ini
ubuntu16       55034  0.8  0.0 986104  6872 pts/2    Sl   20:59   0:00 /home/
↪ubuntu16/fisco/multi_nodes/127.0.0.1/node1/../fisco-bcos -c config.ini
ubuntu16       55041  0.8  0.0 986384  6584 pts/2    Sl   20:59   0:00 /home/
↪ubuntu16/fisco/multi_nodes/127.0.0.1/node2/../fisco-bcos -c config.ini
ubuntu16       55047  0.8  0.0 986396  6656 pts/2    Sl   20:59   0:00 /home/
↪ubuntu16/fisco/multi_nodes/127.0.0.1/node3/../fisco-bcos -c config.ini
```

查看节点共识情况

```
# 查看node0共识情况（Ctrl+c退回命令行）
$ tail -f node0/log/* | grep "g:1.*++"
info|2019-02-11 20:59:52.065958| [g:1][p:264][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=2,hash=da72649e...

# 查看node1共识情况
$ tail -f node1/log/* | grep "g:1.*++"
info|2019-02-11 20:59:54.070297| [g:1][p:264][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=0,hash=11c9354d...

# 查看node2共识情况
$ tail -f node2/log/* | grep "g:1.*++"
info|2019-02-11 20:59:55.073124| [g:1][p:264][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=1,hash=b65cbac8...

# 查看node3共识情况
$ tail -f node3/log/* | grep "g:1.*++"
info|2019-02-11 20:59:53.067702| [g:1][p:264][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=3,hash=0467e5c4...
```

将group2加入区块链

并行多组区块链每个群组的genesis配置文件几乎相同，但[group].id不同，为群组号。

```
# 进入节点目录
$ cd ~/fisco/multi_nodes/127.0.0.1

# 拷贝group1的配置
$ cp node0/conf/group.1.genesis node0/conf/group.2.genesis
$ cp node0/conf/group.1.ini node0/conf/group.2.ini

# 修改群组ID
$ sed -i "s/id=1/id=2/g"  node0/conf/group.2.genesis
$ cat node0/conf/group.2.genesis | grep "id"
# 已修改到     id=2

# 更新group.2.genesis文件中的共识节点列表，剔除已废弃的共识节点。

# 将配置拷贝到各个节点
$ cp node0/conf/group.2.genesis node1/conf/group.2.genesis
$ cp node0/conf/group.2.genesis node2/conf/group.2.genesis
$ cp node0/conf/group.2.genesis node3/conf/group.2.genesis
$ cp node0/conf/group.2.ini node1/conf/group.2.ini
```

(continues on next page)

(续上页)

```
$ cp node0/conf/group.2.ini node2/conf/group.2.ini
$ cp node0/conf/group.2.ini node3/conf/group.2.ini

# 重启各个节点
$ bash stop_all.sh
$ bash start_all.sh
```

查看群组共识情况

```
# 查看node0 group2共识情况（Ctrl+c退回命令行）
$ tail -f node0/log/* | grep "g:2.*++"
info|2019-02-11 21:13:28.541596| [g:2][p:520][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=2,hash=f3562664...

# 查看node1 group2共识情况
$ tail -f node1/log/* | grep "g:2.*++"
info|2019-02-11 21:13:30.546011| [g:2][p:520][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=0,hash=4b17e74f...

# 查看node2 group2共识情况
$ tail -f node2/log/* | grep "g:2.*++"
info|2019-02-11 21:13:59.653615| [g:2][p:520][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=1,hash=90cbd225...

# 查看node3 group2共识情况
$ tail -f node3/log/* | grep "g:2.*++"
info|2019-02-11 21:14:01.657428| [g:2][p:520][CONSENSUS][SEALER]++++++++Generating␣
↪seal on,blkNum=1,tx=0,nodeIdx=3,hash=d7dcb462...
```

向群组发交易

获取控制台

```
# 若从未下载控制台，请进行下面操作下载控制台，否则将控制台拷贝到~/fisco目录：
$ cd ~/fisco
# 获取控制台
$ curl -LO https://github.com/FISCO-BCOS/console/releases/download/v1.0.9/download_
↪console.sh && bash download_console.sh
```

配置控制台

```
# 获取channel_port
$ grep "channel_listen_port" multi_nodes/127.0.0.1/node0/config.ini
multi_nodes/127.0.0.1/node0/config.ini:    channel_listen_port=20100

# 进入控制台目录
$ cd console
# 拷贝节点证书
$ cp ~/fisco/multi_nodes/127.0.0.1/sdk/* conf
```

创建控制台配置文件conf/applicationContext.xml的配置如下，在node0（127.0.0.1:20100）上配置了两个group（group1和group2）：

```
cat > ./conf/applicationContext.xml << EOF
<?xml version="1.0" encoding="UTF-8" ?>

<beans xmlns="http://www.springframework.org/schema/beans"
```

(continues on next page)

(续上页)

```
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://
↪www.springframework.org/schema/p"
          xmlns:tx="http://www.springframework.org/schema/tx" xmlns:aop="http://
↪www.springframework.org/schema/aop"
          xmlns:context="http://www.springframework.org/schema/context"
          xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
         http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
         http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">


        <bean id="encryptType" class="org.fisco.bcos.web3j.crypto.EncryptType">
                <constructor-arg value="0"/> <!-- 0:standard 1:guomi -->
        </bean>

      <bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.
↪handler.GroupChannelConnectionsConfig">
        <property name="allChannelConnections">
        <list>
            <bean id="group1"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                <property name="groupId" value="1" />
                    <property name="connectionsStr">
                    <list>
                    <value>127.0.0.1:20100</value>
                    </list>
                    </property>
            </bean>
            <bean id="group2"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                <property name="groupId" value="2" />
                    <property name="connectionsStr">
                    <list>
                    <value>127.0.0.1:20100</value>
                    </list>
                    </property>
            </bean>
        </list>
        </property>
        </bean>

        <bean id="channelService" class="org.fisco.bcos.channel.client.Service"␣
↪depends-on="groupChannelConnectionsConfig">
                <property name="groupId" value="1" />
                <property name="orgID" value="fisco" />
                <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>
        </bean>
</beans>
EOF
```

通过控制台向群组发交易

```
# ... 启动控制台 ...
$ bash start.sh
# 输出如下信息表明控制台启动成功，若启动失败，请检查是否配置证书、channel listen port配置是否正
确
=====================================================================================
Welcome to FISCO BCOS console(1.0.3)!
```

(continues on next page)

(续上页)

```
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ ______  ______   ______   ______       _______   ______   ______   ______
|        |      \/      \ /      \ /      \     |       \ /      \ /      \ /      ␣
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
| $$__     | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \    | $$  \$$    \| $$     | $$  | $$    | $$    $| $$     | $$  | $$\$$   ␣
↪\
| $$$$$    | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$\
| $$      _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__|
↪$$
| $$     |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$      \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$
↪$

==================================================================================
# ... 向group1发交易...
# 获取当前块高
$ [group:1]> getBlockNumber
0
# 向group1部署HelloWorld合约，若部署失败，请检查group1共识是否正常
$ [group:1]> deploy HelloWorld
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
# 获取当前块高，若块高没有增加，请检查group1共识是否正常
$ [group:1]> getBlockNumber
1

# ... 向group2发交易...
# 切换到group2
$ [group:1]> switch 2
Switched to group 2.
# 获取当前块高
$ [group:2]> getBlockNumber
0
# 向group2部署HelloWorld合约
$ [group:2]> deploy HelloWorld
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
# 获取当前块高，若块高没有增加，请检查group2共识是否正常
$ [group:2]> getBlockNumber
1
# 退出控制台
$[group:2]> exit
```

通过日志查看节点出块状态

```
# 切换到节点目录
$ cd ~/fisco/multi_nodes/127.0.0.1/

# 查看group1出块情况，看到Report了属于group1的块高为1的块
$ cat node0/log/* | grep "g:1.*Report"
info|2019-02-11 21:14:57.216548| [g:1][p:264][CONSENSUS][PBFT]^^^^^Report:,num=1,
↪sealerIdx=3,hash=be961c98...,next=2,tx=1,nodeIdx=2

# 查看group2出块情况，看到Report了属于group2的块高为1的块
$ cat node0/log/* | grep "g:2.*Report"
info|2019-02-11 21:15:25.310565| [g:2][p:520][CONSENSUS][PBFT]^^^^^Report:,num=1,
↪sealerIdx=3,hash=5d006230...,next=2,tx=1,nodeIdx=2
```

停止节点

```
# 回到节点目录 && 停止节点
$ cd ~/fisco/multi_nodes/127.0.0.1 && bash stop_all.sh
```

5.8 分布式存储

5.8.1 安装MySQL

当前支持的分布式数据库是MySQL,在使用分布式存储之前，需要先搭建MySQL服务，在Ubuntu和CentOS服务器上的配置方式如下：

Ubuntu：执行下面三条命令，安装过程中，配置 root 账户密码。

```
sudo apt install -y mysql-server mysql-client libmysqlclient-dev
```

启动 MySQL 服务并登陆: root 账户密码。

```
service msyql start
mysql -uroot -p
```

CentOS: 执行下面两条命令进行安装。

```
yum install mysql*
#某些版本的linux，需要安装mariadb，mariadb是mysql的一个分支
yum install mariadb*
```

启动 MySQL 服务，登陆并为 root 用户设置密码。

```
service mysqld start
#若安装了mariadb，则使用下面的命令启动
service mariadb start
mysql -uroot -p
mysql> set password for root@localhost = password('123456');
```

5.8.2 配置MySQL参数

查看配置文件my.cnf

```
mysql --help | grep 'Default options' -A 1
```

执行之后可以看到如下如下数据

```
Default options are read from the following files in the given order:
/etc/mysql/my.cnf /etc/my.cnf ~/.my.cnf
```

配置my.cnf

mysql依次从/etc/mysql/my.cnf，/etc/my.cnf，~/.my.cnf中加载配置。依次查找这几个文件，找到第一个存在的文件，在[mysqld]段中新增如下内容（如果存在则修改值）。

```
max_allowed_packet = 1024M
sql_mode =STRICT_TRANS_TABLES
```

重启mysql-sever，验证参数。

Ubuntu：执行如下命令重启

```
service mysql restart
```

CentOS：执行如下命令重启

```
service mysqld start
#若安装了mariadb，则使用下面的命令启动
service mariadb start
```

验证参数过程

```
mysql -uroot -p
#执行下面命令，查看max_allowed_packet的值
MariaDB [(none)]>  show variables like 'max_allowed_packet%';
+--------------------+------------+
| Variable_name      | Value      |
+--------------------+------------+
| max_allowed_packet | 1073741824 |
+--------------------+------------+
1 row in set (0.00 sec)

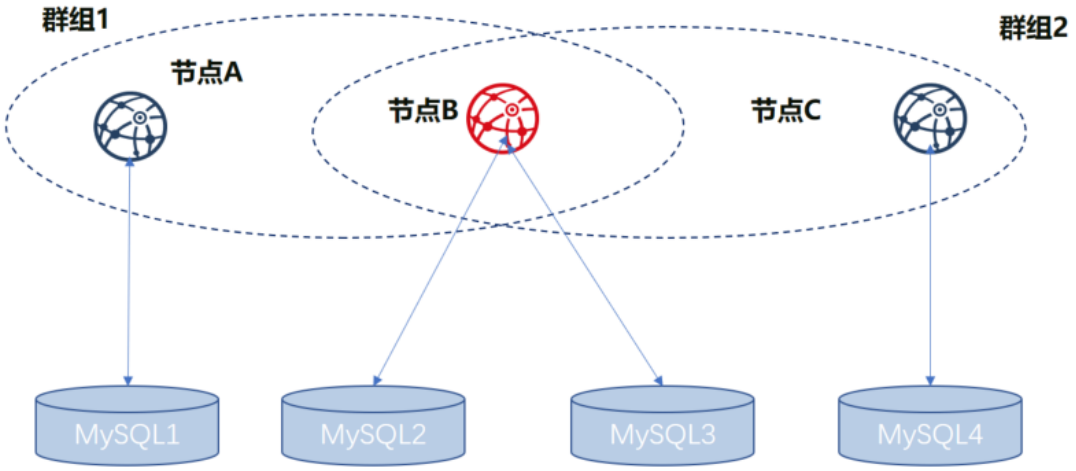
#执行下面命令，查看sql_mode的值
MariaDB [(none)]>  show variables like 'sql_mode%';
+---------------+----------------------+
| Variable_name | Value                |
+---------------+----------------------+
| sql_mode      | STRICT_TRANS_TABLES |
+---------------+----------------------+
1 row in set (0.00 sec)
```

5.8.3 节点直连MySQL

FISCO BCOS在2.0.0-rc3之后，支持节点通过连接池直连MySQL，相对于代理访问MySQL方式，配置简单，不需要手动创建数据库。配置方法请参考:

逻辑架构图

多群组架构是指区块链节点支持启动多个群组，群组间交易处理、数据存储、区块共识相互隔离的。因此群组下的每一个节点对应一个数据库实例，例如，区块链网络中，有三个节点A,B,C，其中A,B属于Group1,B,C属于Group2。节点A和C分别对应1个数据库实例，B节点对应了2个数据库实例，逻辑架构图如下



如上图所示，节点B属于多个群组，不同群组下的同一个节点，对应的数据库实例是分开的，为了区分不同群组下的同一个节点，将A,B,C三个节点，分别用Group1_A（Group1下的A节点，下同），Group1_B，Group2_B，Group2_C表示。

下面以上图为例，描述搭建配置过程。

节点搭建

使用分布式存储之前，需要完成联盟链的搭建和多群组的配置，具体参考如下步骤。

准备依赖

```
mkdir -p ~/fisco && cd ~/fisco
# 获取build_chain.sh脚本
curl -LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.5.0/build_
↪chain.sh && chmod u+x build_chain.sh
```

生成配置文件

```
# 生成区块链配置文件ipconf
cat > ipconf << EOF
127.0.0.1:1 agencyA 1
127.0.0.1:1 agencyB 1,2
127.0.0.1:1 agencyC 2
EOF

# 查看配置文件
cat ipconf
127.0.0.1:1 agencyA 1
127.0.0.1:1 agencyB 1,2
127.0.0.1:1 agencyC 2
```

使用build_chain搭建区块链

```
### 搭建区块链（请先确认30300~30302，20200~20202，8545~8547端口没有被占用）
### 这里区别是在命令后面追加了参数"-s MySQL"以及换了端口。
bash build_chain.sh -f ipconf -p 30300,20200,8545 -s MySQL
=============================================================
Generating CA key...
=============================================================
Generating keys ...
Processing IP:127.0.0.1 Total:1 Agency:agencyA Groups:1
Processing IP:127.0.0.1 Total:1 Agency:agencyB Groups:1,2
Processing IP:127.0.0.1 Total:1 Agency:agencyC Groups:2
=============================================================
Generating configurations...
Processing IP:127.0.0.1 Total:1 Agency:agencyA Groups:1
Processing IP:127.0.0.1 Total:1 Agency:agencyB Groups:1,2
Processing IP:127.0.0.1 Total:1 Agency:agencyC Groups:2
=============================================================
Group:1 has 2 nodes
Group:2 has 2 nodes
```

修改节点ini文件

group.[群组].ini配置文件中，和本特性相关的是MySQL的配置信息。假设MySQL的配置信息如下：

```
|节点|db_ip|db_port|db_username|db_passwd|db_name|
|Group1_A|127.0.0.1|3306|root|123456|db_Group1_A|
|Group1_B|127.0.0.1|3306|root|123456|db_Group1_B|
|Group2_B|127.0.0.1|3306|root|123456|db_Group2_B|
|Group2_C|127.0.0.1|3306|root|123456|db_Group2_C|
```

修改node0下的group.1.ini配置

修改~/fisco/nodes/127.0.0.1/node0/conf/group.1.ini[storage]段的内容，配置如下内容。db_passwd为对应的密码。

```
            db_ip=127.0.0.1
            db_port=3306
            db_username=root
            db_name=db_Group1_A
            db_passwd=
```

修改node1下的group.1.ini配置

修改~/fisco/nodes/127.0.0.1/node0/conf/group.1.ini[storage]段的内容，新增如下内容。db_passwd为对应的密码。

```
            db_ip=127.0.0.1
            db_port=3306
            db_username=root
            db_name=db_Group1_B
            db_passwd=
```

修改node1下的group.2.ini配置

修改~/fisco/nodes/127.0.0.1/node1/conf/group.2.ini[storage]段的内容，新增如下内容。db_passwd为对应的密码。

```
            db_ip=127.0.0.1
            db_port=3306
            db_username=root
            db_name=db_Group2_B
            db_passwd=
```

修改node2下的group.2.ini配置

修改~/fisco/nodes/127.0.0.1/node2/conf/group.2.ini[storage]段的内容，新增如下内容。db_passwd为对应的密码。

```
            db_ip=127.0.0.1
            db_port=3306
            db_username=root
            db_name=db_Group2_C
            db_passwd=
```

启动节点

```
cd ~/fisco/nodes/127.0.0.1;sh start_all.sh
```

检查进程

```
ps -ef|grep fisco-bcos|grep -v grep
fisco   111061      1  0 16:22 pts/0    00:00:04 /data/home/fisco/nodes/127.0.0.1/
↪node2/../fisco-bcos -c config.ini
fisco   111065      1  0 16:22 pts/0    00:00:04 /data/home/fisco/nodes/127.0.0.1/
↪node0/../fisco-bcos -c config.ini
fisco   122910      1  1 16:22 pts/0    00:00:02 /data/home/fisco/nodes/127.0.0.1/
↪node1/../fisco-bcos -c config.ini
```

启动成功，3个fisco-bcos进程。不成功的话请参考日志确认配置是否正确。

检查日志输出

执行下面指令，查看节点node0链接的节点数（其他节点类似）

```
tail -f nodes/127.0.0.1/node0/log/log*  | grep connected
```

正常情况会看到类似下面的输出，从输出可以看出node0与另外2个节点有连接。

```
info|2019-05-28 16:28:57.267770|[P2P][Service] heartBeat,connected count=2
info|2019-05-28 16:29:07.267935|[P2P][Service] heartBeat,connected count=2
info|2019-05-28 16:29:17.268163|[P2P][Service] heartBeat,connected count=2
info|2019-05-28 16:29:27.268284|[P2P][Service] heartBeat,connected count=2
info|2019-05-28 16:29:37.268467|[P2P][Service] heartBeat,connected count=2
```

执行下面指令，检查是否在共识

```
tail -f nodes/127.0.0.1/node0/log/log*  | grep +++
```

正常情况会不停输出++++Generating seal表示共识正常。

```
info|2019-05-28 16:26:32.454059|[g:1][CONSENSUS][SEALER]++++++++++++++++
 ↪Generating seal on,blkNum=28,tx=0,nodeIdx=3,hash=c9c859d5...
info|2019-05-28 16:26:36.473543|[g:1][CONSENSUS][SEALER]++++++++++++++++
 ↪Generating seal on,blkNum=28,tx=0,nodeIdx=3,hash=6b319fa7...
info|2019-05-28 16:26:40.498838|[g:1][CONSENSUS][SEALER]++++++++++++++++
 ↪Generating seal on,blkNum=28,tx=0,nodeIdx=3,hash=2164360f...
```

使用控制台发送交易

准备依赖

```
cd ~/fisco;
curl -LO https://github.com/FISCO-BCOS/console/releases/download/v1.0.9/download_
 ↪console.sh && bash download_console.sh
cp -n console/conf/applicationContext-sample.xml console/conf/applicationContext.
 ↪xml
cp nodes/127.0.0.1/sdk/* console/conf/
```

修改配置文件

将~/fisco/console/conf/applicationContext.xml修改为如下配置(部分信息)

```
<bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.handler.
 ↪GroupChannelConnectionsConfig">
        <property name="allChannelConnections">
                <list>
                        <bean id="group1"  class="org.fisco.bcos.channel.handler.
 ↪ChannelConnections">
                                <property name="groupId" value="1" />
                                        <property name="connectionsStr">
                                        <list>
                                                <value>127.0.0.1:20200</value>
                                        </list>
                                </property>
                        </bean>
                </list>
        </property>
</bean>
```

启用控制台

```
cd ~/fisco/console
sh start.sh 1
#部署TableTest合约
[group:1]> deploy TableTest
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
```

查看数据库中的表情况

```
MySQL -uroot -p123456 -A db_Group1_A
use db_Group1_A;
show tables;
------------------------------------------------------------+
| Tables_in_db_Group1_A                                      |
```

(continues on next page)

(续上页)

```
+-----------------------------------------------------------+
| c_8c17cf316c1063ab6c89df875e96c9f0f5b2f744 |
| c_f69a2fa2eca49820218062164837c6eecc909abd |
| _sys_block_2_nonces_                                      |
| _sys_cns_                                                 |
| _sys_config_                                              |
| _sys_consensus_                                           |
| _sys_current_state_                                       |
| _sys_hash_2_block_                                        |
| _sys_number_2_hash_                                       |
| _sys_table_access_                                        |
| _sys_tables_                                              |
| _sys_tx_hash_2_block_                                     |
+-----------------------------------------------------------+
12 rows in set (0.02 sec)
```

查看数据库中的表情况

```
show tables;
+-----------------------------------------------------------+
| Tables_in_db_Group1_A                                     |
+-----------------------------------------------------------+
| c_8c17cf316c1063ab6c89df875e96c9f0f5b2f744 |
| c_f69a2fa2eca49820218062164837c6eecc909abd |
| _sys_block_2_nonces_                                      |
| _sys_cns_                                                 |
| _sys_config_                                              |
| _sys_consensus_                                           |
| _sys_current_state_                                       |
| _sys_hash_2_block_                                        |
| _sys_number_2_hash_                                       |
| _sys_table_access_                                        |
| _sys_tables_                                              |
| _sys_tx_hash_2_block_                                     |
| u_t_test                                                  |
+-----------------------------------------------------------+
```

往表里面插入一条数据

```
#往表里插入数据
call TableTest 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744 insert "fruit" 100 "apple
↪"
0x082ca6a5a292f1f7b20abeb3fb03f45e0c6f48b5a79cc65d1246bfe57be358d1
```

打开MySQL客户端，查询u_t_test表数据

```
#查看用户表中的数据
select * from u_t_test\G;
*************************** 1. row ***************************
     _id_: 31
   _hash_: 0a0ed3b2b0a227a6276114863ef3e8aa34f44e31567a5909d1da0aece31e575e
    _num_: 3
 _status_: 0
     name: fruit
  item_id: 100
item_name: apple
1 row in set (0.00 sec)
```

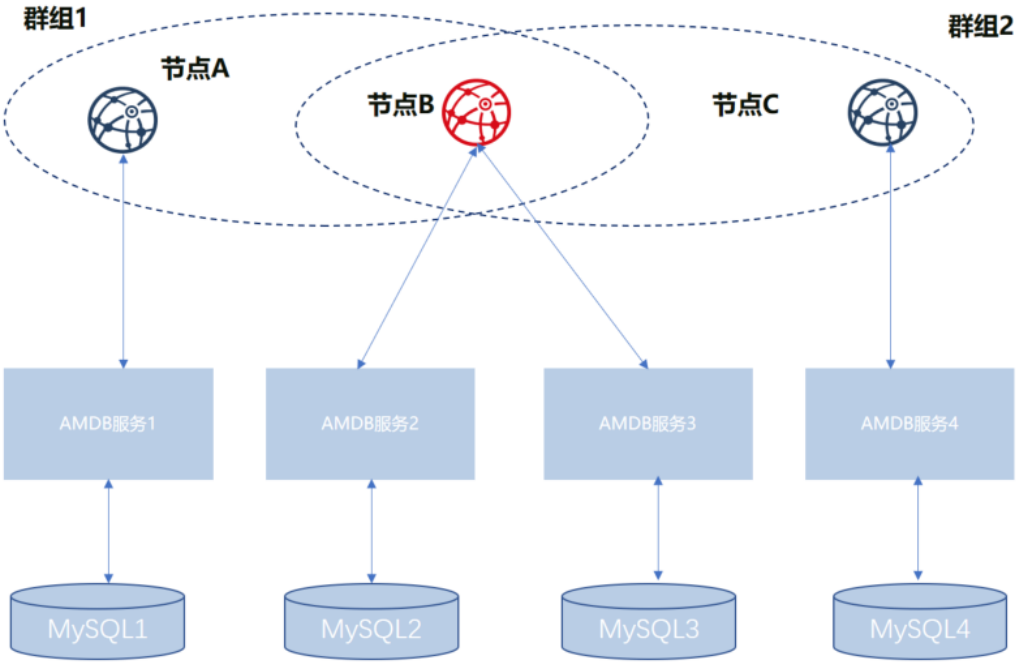
5.8.4 通过代理访问MySQL

本使用手册仅对节点版本2.1.0以及以后的版本有效，需要在2.0.0-rc3或者2.0.0中使用“通过代理访问MySQL”的访问方式去搭建分布式存储环境。请参考文档分布式存储搭建方法。需要在2.0.0-rc2中使用“通过代理访问MySQL”的访问方式去搭建分布式存储环境。请参考文档分布式存储搭建方法

重要： 推荐使用MySQL模式。

逻辑架构图

多群组架构是指区块链节点支持启动多个群组，群组间交易处理、数据存储、区块共识相互隔离的。因此群组下的每一个节点对应一个amdb-proxy实例，例如，区块链网络中，有三个节点A,B,C，其中A,B属于群组1,B,C属于群组2。节点A和C分别对应1个数据库实例，B节点对应了2个数据库实例，逻辑架构图如下：



如上图所示，节点B属于多个群组，不同群组下的同一个节点，对应的amdb-proxy服务和MySQL是分开的，为了区分不同群组下的同一个节点，将A,B,C三个节点，分别用Group1_A（Group1下的A节点，下同），Group1_B，Group2_B，Group2_C表示。 下面以上图为例，描述搭建配置过程。

节点搭建

配置amdb-proxy服务之前，需要完成联盟链的搭建和多群组的配置，具体参考如下步骤。

准备依赖

- 创建文件夹

```
mkdir -p ~/fisco && cd ~/fisco
```

- 获取build_chain脚本

```
curl -LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.5.0/build_
↪chain.sh && chmod u+x build_chain.sh
```

生成配置文件

```
# 生成区块链配置文件ipconf
cat > ipconf << EOF
127.0.0.1:1 agencyA 1
127.0.0.1:1 agencyB 1,2
127.0.0.1:1 agencyC 2
EOF

# 查看配置文件
cat ipconf
127.0.0.1:1 agencyA 1
127.0.0.1:1 agencyB 1,2
127.0.0.1:1 agencyC 2
```

使用build_chain搭建区块链

```
### 搭建区块链（请先确认30300~30302，20200~20202，8545~8547端口没有被占用）
bash build_chain.sh -f ipconf -p 30300,20200,8545
================================================================
Generating CA key...
================================================================
Generating keys ...
Processing IP:127.0.0.1 Total:1 Agency:agencyA Groups:1
Processing IP:127.0.0.1 Total:1 Agency:agencyB Groups:1,2
Processing IP:127.0.0.1 Total:1 Agency:agencyC Groups:2
================================================================
Generating configurations...
Processing IP:127.0.0.1 Total:1 Agency:agencyA Groups:1
Processing IP:127.0.0.1 Total:1 Agency:agencyB Groups:1,2
Processing IP:127.0.0.1 Total:1 Agency:agencyC Groups:2
================================================================
Group:1 has 2 nodes
Group:2 has 2 nodes
```

修改节点ini文件

修改node0下的group.1.ini配置

修改~/fisco/nodes/127.0.0.1/node0/conf/group.1.ini文件中[storage]段的内容，设置为如下内容

```
[storage]
            type=external
            topic=DB_Group1_A
            max_retry=100
```

修改node1下的group.1.ini配置

修改~/fisco/nodes/127.0.0.1/node1/conf/group.1.ini文件中[storage]段的内容，设置为如下内容

```
[storage]
            type=external
            topic=DB_Group1_B
            max_retry=100
```

修改node1下的group.2.ini配置

修改~/fisco/nodes/127.0.0.1/node1/conf/group.2.ini文件中[storage]段的内容，设置为如下内容

```
[storage]
            type=external
            topic=DB_Group2_B
            max_retry=100
```

修改node2下的group.2.ini配置

修改~/fisco/nodes/127.0.0.1/node2/conf/group.2.ini文件中[storage]段的内容，设置为如下内容

```
[storage]
            type=external
            topic=DB_Group2_C
            max_retry=100
```

准备amdb代理

源码获取

```
cd ~/fisco;
git clone https://github.com/FISCO-BCOS/amdb-proxy.git
```

源码编译

```
cd AMDB;gradle build
```

编译完成之后，会生成一个dist目录，文件结构如下：

```
├── apps
│   └── AMDB.jar
├── conf
│   ├── applicationContext.xml
│   ├── contracts
│   │   ├── Table.sol
│   │   └── TableTest.sol
│   ├── doc
│   │   ├── amop.png
│   │   ├── leveldb.png
│   │   └── README.md
│   ├── log4j2.xml
│   └── mappers
│       └── data_mapper.xml
├── lib
├── log
└── start.sh
```

配置amdb-proxy

amdb-proxy与节点连接过程，amdb-proxy是client,节点是server，启动过程是amdb-proxy服务主动连接节点，节点只需要配置amdb-proxy关注的topic即可，关于topic的介绍请参考AMOP，amdb-proxy需要通过证书准入。

证书配置

```
cp ~/fisco/nodes/127.0.0.1/sdk/* ~/fisco/AMDB/dist/conf/
```

amdb实例拷贝

```
cd ~/fisco;
###dist_Group1_A是节点Group1_A对应的amdb实例
cp AMDB/dist/ dist_Group1_A -R
###dist_Group1_B是节点Group1_B对应的amdb实例
cp AMDB/dist/ dist_Group1_B -R
###dist_Group2_B是节点Group2_B对应的amdb实例
cp AMDB/dist/ dist_Group2_B -R
###dist_Group2_C是节点Group2_C对应的amdb实例
cp AMDB/dist/ dist_Group2_C -R
```

经过上述步骤，可以看到~/fisco目录的文件结构如下：

```
drwxrwxr-x 8 fisco fisco  4096 May  7 15:53 AMDB
-rwxrw-r-- 1 fisco fisco 37539 May  7 14:58 build_chain.sh
drwxrwxr-x 5 fisco fisco  4096 May  7 15:58 dist_Group1_A
drwxrwxr-x 5 fisco fisco  4096 May  7 15:58 dist_Group1_B
drwxrwxr-x 5 fisco fisco  4096 May  7 15:59 dist_Group2_B
drwxrwxr-x 5 fisco fisco  4096 May  7 15:59 dist_Group2_C
-rw-rw-r-- 1 fisco fisco    68 May  7 14:59 ipconf
drwxrwxr-x 4 fisco fisco  4096 May  7 15:08 nodes
```

DB创建

```
        MySQL -uroot -p123456
        CREATE DATABASE `bcos_Group1_A`;
        CREATE DATABASE `bcos_Group1_B`;
        CREATE DATABASE `bcos_Group2_B`;
        CREATE DATABASE `bcos_Group2_C`;
```

配置文件配置

这里假设MySQL的配置信息如下：

```
|节点|db_ip|db_port|db_username|db_passwd|db_name|
|Group1_A|127.0.0.1|3306|root|123456|bcos_Group1_A|
|Group1_B|127.0.0.1|3306|root|123456|bcos_Group1_B|
|Group2_B|127.0.0.1|3306|root|123456|bcos_Group2_B|
|Group2_C|127.0.0.1|3306|root|123456|bcos_Group2_C|
```

配置过程需要修改applicationContext.xml文件，需要修改的配置项包括topic配置**node.topic**,MySQL配置信息配置包括**db.ip**、**db.port**、**db.database**、**db.user**和**db.password**。

为Group1的A节点配置amdb-proxy

将~/fisco/dist_Group1_A/conf/applicationContext.xml修改为如下配置(部分信息)

```
<bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.handler.
↪GroupChannelConnectionsConfig">
        <property name="allChannelConnections">
                <list>
                        <bean id="group1"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                                <property name="groupId" value="1" />
                                        <property name="connectionsStr">
                                        <list>
                                                <value>127.0.0.1:20200</value>
                                        </list>
                                </property>
                        </bean>
                </list>
        </property>
</bean>

<bean id="DBChannelService" class="org.fisco.bcos.channel.client.Service">
                <property name="groupId" value="1" />
                <property name="orgID" value="fisco" />
                <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>
                <property name="topics">
                        <list>
                                <value>DB_Group1_A</value>
                        </list>
                </property>
        <property name="pushCallback" ref="DBHandler"/>
</bean>
<!-- database connection configuration -->
        <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!-- please configure db connection here-->
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/bcos_Group1_A?
↪characterEncoding=UTF-8&amp;zeroDateTimeBehavior=convertToNull" />
        <property name="username" value="root" />
        <property name="password" value="123456" />
</bean>
```

为Group1的B节点配置amdb-proxy

将~/fisco/dist_Group1_B/conf/applicationContext.xml修改为如下配置(部分信息)

```
<bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.handler.
↪GroupChannelConnectionsConfig">
        <property name="allChannelConnections">
                <list>
                        <bean id="group1"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                                <property name="groupId" value="1" />
                                        <property name="connectionsStr">
                                        <list>
                                                <value>127.0.0.1:20201</value>
                                        </list>
                                </property>
                        </bean>
```

(continues on next page)

(续上页)

```
                </list>
        </property>
</bean>

<bean id="DBChannelService" class="org.fisco.bcos.channel.client.Service">
                <property name="groupId" value="1" />
                <property name="orgID" value="fisco" />
                <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>
                <property name="topics">
                        <list>
                                <value>DB_Group1_B</value>
                        </list>
                </property>
        <property name="pushCallback" ref="DBHandler"/>
</bean>

<!-- database connection configuration -->
        <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!-- please configure db connection here-->
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/bcos_Group1_B?
↪characterEncoding=UTF-8&amp;zeroDateTimeBehavior=convertToNull" />
        <property name="username" value="root" />
        <property name="password" value="123456" />
</bean>
```

为Group2的B节点配置amdb-proxy

将~/fisco/dist_Group2_B/conf/applicationContext.xml修改为如下配置(部分信息)

```
<bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.handler.
↪GroupChannelConnectionsConfig">
        <property name="allChannelConnections">
                <list>
                        <bean id="group2"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                                <property name="groupId" value="2" />
                                        <property name="connectionsStr">
                                        <list>
                                                <value>127.0.0.1:20201</value>
                                        </list>
                                </property>
                        </bean>
                </list>
        </property>
</bean>

<bean id="DBChannelService" class="org.fisco.bcos.channel.client.Service">
                <property name="groupId" value="2" />
                <property name="orgID" value="fisco" />
                <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>

                <!-- communication topic configuration of the node -->
                <property name="topics">
                        <list>
                                <value>DB_Group2_B</value>
                        </list>
```

(continues on next page)

(续上页)

```
                </property>
                <property name="pushCallback" ref="DBHandler"/>
        </bean>
<!-- database connection configuration -->
        <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!-- please configure db connection here-->
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/bcos_Group2_B?
↪characterEncoding=UTF-8&amp;zeroDateTimeBehavior=convertToNull" />
        <property name="username" value="root" />
        <property name="password" value="123456" />
</bean>
```

为Group2的C节点配置amdb-proxy

将~/fisco/dist_Group2_C/conf/applicationContext.xml修改为如下配置(部分信息)

```
<bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.handler.
↪GroupChannelConnectionsConfig">
        <property name="allChannelConnections">
                <list>
                        <bean id="group2"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                                <property name="groupId" value="2" />
                                        <property name="connectionsStr">
                                        <list>
                                                <value>127.0.0.1:20202</value>
                                        </list>
                                </property>
                        </bean>
                </list>
        </property>
</bean>

<bean id="DBChannelService" class="org.fisco.bcos.channel.client.Service">
                <property name="groupId" value="2" />
                <property name="orgID" value="fisco" />
                <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>

        <!-- communication topic configuration of the node -->
        <property name="topics">
                        <list>
                                <value>DB_Group2_C</value>
                        </list>
                </property>
        <property name="pushCallback" ref="DBHandler"/>
</bean>

<!-- database connection configuration -->
        <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!-- please configure db connection here-->
        <property name="url" value="jdbc:mysql://jdbc:mysql://127.0.0.1:3306/bcos_
↪Group2_C?characterEncoding=UTF-8&amp;zeroDateTimeBehavior=convertToNull" />
        <property name="username" value="root" />
        <property name="password" value="123456" />
</bean>
```

启动amdb-proxy

```
cd ~/fisco/dist_Group1_A;sh start.sh
cd ~/fisco/dist_Group1_B;sh start.sh
cd ~/fisco/dist_Group2_B;sh start.sh
cd ~/fisco/dist_Group2_C;sh start.sh
```

启动节点

```
cd ~/fisco/nodes/127.0.0.1;sh start_all.sh
```

检查进程

```
ps -ef|grep org.bcos.amdb.server.Main|grep -v grep
fisco   110734      1  1 17:25 ?        00:00:10 java -cp conf/:apps/*:lib/* org.
↪bcos.amdb.server.Main
fisco   110778      1  1 17:25 ?        00:00:11 java -cp conf/:apps/*:lib/* org.
↪bcos.amdb.server.Main
fisco   110803      1  1 17:25 ?        00:00:10 java -cp conf/:apps/*:lib/* org.
↪bcos.amdb.server.Main
fisco   122676      1 16 17:38 ?        00:00:08 java -cp conf/:apps/*:lib/* org.
↪bcos.amdb.server.Main

ps -ef|grep fisco-bcos|grep -v grep
fisco   111061      1  0 17:25 pts/0    00:00:04 /data/home/fisco/nodes/127.0.0.1/
↪node2/../fisco-bcos -c config.ini
fisco   111065      1  0 17:25 pts/0    00:00:04 /data/home/fisco/nodes/127.0.0.1/
↪node0/../fisco-bcos -c config.ini
fisco   122910      1  1 17:38 pts/0    00:00:02 /data/home/fisco/nodes/127.0.0.1/
↪node1/../fisco-bcos -c config.ini
```

启动成功，会看到有4个java进程，3个fisco-bcos进程。不成功的话请参考日志确认配置是否正确。

检查日志输出

执行下面指令，查看节点node0链接的节点数（其他节点类似）

```
tail -f nodes/127.0.0.1/node0/log/log*  | grep connected
```

正常情况会看到类似下面的输出，从输出可以看出node0与另外2个节点有连接。

```
info|2019-05-07 21:47:22.849910| [P2P][Service] heartBeat connected count,size=2
info|2019-05-07 21:47:32.849970| [P2P][Service] heartBeat connected count,size=2
info|2019-05-07 21:47:42.850024| [P2P][Service] heartBeat connected count,size=2
```

执行下面指令，检查是否在共识

```
tail -f nodes/127.0.0.1/node0/log/log*  | grep +++
```

正常情况会不停输出++++Generating seal表示共识正常。

```
info|2019-05-07 21:48:54.942111| [g:1][p:65544][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=6,tx=0,nodeIdx=1,hash=355790f7...
info|2019-05-07 21:48:56.946022| [g:1][p:65544][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=6,tx=0,nodeIdx=1,hash=4ef772bb...
info|2019-05-07 21:48:58.950222| [g:1][p:65544][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=6,tx=0,nodeIdx=1,hash=48341ee5...
```

使用控制台发送交易

请参考“节点直连MySQL”中的使用控制台发送交易章节。

5.9 控制台

控制台是FISCO BCOS 2.0重要的交互式客户端工具，它通过Web3SDK与区块链节点建立连接，实现对区块链节点数据的读写访问请求。控制台拥有丰富的命令，包括查询区块链状态、管理区块链节点、部署并调用合约等。此外，控制台提供一个合约编译工具，用户可以方便快捷的将Solidity合约文件编译为Java合约文件。

5.9.1 控制台命令

控制台命令由两部分组成，即指令和指令相关的参数：

- **指令**: 指令是执行的操作命令，包括查询区块链相关信息，部署合约和调用合约的指令等，其中部分指令调用JSON-RPC接口，因此与JSON-RPC接口同名。 **使用提示： 指令可以使用tab键补全，并且支持按上下键显示历史输入指令。**
- **指令相关的参数**: 指令调用接口需要的参数，指令与参数以及参数与参数之间均用空格分隔，与JSON-RPC接口同名命令的输入参数和获取信息字段的详细解释参考JSON-RPC API。

5.9.2 常用命令链接

合约相关命令

- 利用CNS部署和调用合约(**推荐**)
  - 部署合约: deployByCNS
  - 调用合约: callByCNS
  - 查询CNS部署合约信息: queryCNS
- 普通部署和调用合约
  - 部署合约: deploy
  - 调用合约: call

其他命令

- 查询区块高度：getBlockNumber
- 查询共识节点列表：getSealerList
- 查询交易回执信息: getTransactionReceipt
- 切换群组: switch

5.9.3 快捷键

- `Ctrl+A`：光标移动到行首
- `Ctrl+D`：退出控制台
- `Ctrl+E`：光标移动到行尾
- `Ctrl+R`：搜索输入的历史命令

- ↑：向前浏览历史命令
- ↓：向后浏览历史命令

5.9.4 控制台响应

当发起一个控制台命令时，控制台会获取命令执行的结果，并且在终端展示执行结果，执行结果分为2类：

- **正确结果:** 命令返回正确的执行结果，以字符串或是json的形式返回。
- **错误结果:** 命令返回错误的执行结果，以字符串或是json的形式返回。
  - 控制台的命令调用JSON-RPC接口时，错误码参考这里。
  - 控制台的命令调用Precompiled Service接口时，错误码参考这里。

5.9.5 控制台配置与运行

重要: 前置条件：搭建FISCO BCOS区块链请参考 开发部署工具 或 企业工具。

获取控制台

```
cd ~ && mkdir -p fisco && cd fisco
# 获取控制台
curl -LO https://github.com/FISCO-BCOS/console/releases/download/v1.0.9/download_
↪console.sh && bash download_console.sh
```

注解:

- 如果因为网络问题导致长时间无法下载，请尝试 *curl -LO https://gitee.com/FISCO-BCOS/console/raw/master/tools/download_console.sh && bash download_console.sh*

目录结构如下：

```
|-- apps # 控制台jar包目录
|   -- console.jar
|-- lib # 相关依赖的jar包目录
|-- conf
|   |-- applicationContext-sample.xml # 配置文件
|   |-- log4j.properties  # 日志配置文件
|-- contracts # 合约所在目录
|   -- solidity  # solidity合约存放目录
|       -- HelloWorld.sol # 普通合约：HelloWorld合约，可部署和调用
|       -- TableTest.sol # 使用CRUD接口的合约：TableTest合约，可部署和调用
|       -- Table.sol # 提供CRUD操作的接口合约
|   -- console  # 控制台部署合约时编译的合约abi, bin, java文件目录
|   -- sdk      # sol2java.sh脚本编译的合约abi, bin, java文件目录
|-- start.sh # 控制台启动脚本
|-- get_account.sh # 账户生成脚本
|-- sol2java.sh # solidity合约文件编译为java合约文件的开发工具脚本
|-- replace_solc_jar.sh # 编译jar包替换脚本
```

配置控制台

- 区块链节点和证书的配置:
  - 将节点sdk目录下的ca.crt、sdk.crt和sdk.key文件拷贝到conf目录下。
  - 将conf目录下的applicationContext-sample.xml文件重命名为applicationContext.xml文件。配置applicationContext.xml文件，其中添加注释的内容根据区块链节点配置做相应修改。**提示：如果搭链时设置的channel_listen_ip(若节点版本小于v2.3.0，查看配置项listen_ip)为127.0.0.1或者0.0.0.0，channel_port为20200，则applicationContext.xml配置不用修改。**

```
<?xml version="1.0" encoding="UTF-8" ?>

<beans xmlns="http://www.springframework.org/schema/beans"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://
↪www.springframework.org/schema/p"
           xmlns:tx="http://www.springframework.org/schema/tx" xmlns:aop="http://
↪www.springframework.org/schema/aop"
           xmlns:context="http://www.springframework.org/schema/context"
           xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
         http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
         http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">


        <bean id="encryptType" class="org.fisco.bcos.web3j.crypto.EncryptType">
                <constructor-arg value="0"/> <!-- 0:standard 1:guomi -->
        </bean>

        <bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.
↪handler.GroupChannelConnectionsConfig">
                <property name="allChannelConnections">
                        <list>  <!-- 每个群组需要配置一个bean -->
                                <bean id="group1"  class="org.fisco.bcos.channel.
↪handler.ChannelConnections">
                                        <property name="groupId" value="1" /> <!--␣
↪群组的groupID -->
                                        <property name="connectionsStr">
                                                <list>
                                                        <value>127.0.0.1:20200</
↪value>  <!-- IP:channel_port -->
                                                </list>
                                        </property>
                                </bean>
                        </list>
                </property>
        </bean>

        <bean id="channelService" class="org.fisco.bcos.channel.client.Service"␣
↪depends-on="groupChannelConnectionsConfig">
                <property name="groupId" value="1" /> <!-- 连接ID为1的群组 -->
                <property name="agencyName" value="fisco" />
                <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>
        </bean>

</beans>
```

配置项详细说明参考这里。

重要： 控制台说明

- 控制台启动失败

  参考，附录：JavaSDK启动失败场景。

- 当控制台配置文件在一个群组内配置多个节点连接时，由于群组内的某些节点在操作过程中可能退出群组，因此控制台轮询节点查询时，其返回信息可能不一致，属于正常现象。建议使用控制台时，配置一个节点或者保证配置的节点始终在群组中，这样在同步时间内查询的群组内信息保持一致。

配置国密版控制台

国密版的控制台配置与非国密版控制台的配置流程有一些区别，流程如下：

- 区块链节点和证书的配置：
  - 将节点sdk目录下的ca.crt、sdk.crt和sdk.key文件拷贝到conf目录下。
  - 将conf目录下的applicationContext-sample.xml文件重命名为applicationContext.xml文件。配置applicationContext.xml文件，其中添加注释的内容根据区块链节点配置做相应修改。**提示：如果搭链时设置的channel_listen_ip(若节点版本小于v2.3.0，查看配置项listen_ip)为127.0.0.1或者0.0.0.0，channel_port为20200，则applicationContext.xml配置不用修改。**
- 打开国密开关

```
<bean id="encryptType" class="org.fisco.bcos.web3j.crypto.EncryptType">
    <!-- encryptType值设置为1，打开国密开关 -->
    <constructor-arg value="1"/> <!-- 0:standard 1:guomi -->
</bean>
```

- 替换国密编译包

```
# 下载solcJ-all-0.4.25-gm.jar放在console目录下
$ curl -LO https://github.com/FISCO-BCOS/LargeFiles/raw/master/tools/solcj/solcJ-
↪all-0.4.25-gm.jar
# 替换Jar包
$ bash replace_solc_jar.sh solcJ-all-0.4.25-gm.jar
```

注解：

- 如果因为网络问题导致长时间无法下载，请尝试 *curl -LO https://www.fisco.com.cn/cdn/deps/tools/solcj/solcJ-all-0.4.25-gm.jar*

合约编译工具

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。 使用该工具，分为两步：

- 将solidity合约文件放在contracts/solidity目录下。
- 通过运行sol2java.sh脚本(**需要指定一个java的包名**)完成编译合约任务。例如，contracts/solidity目录下已有HelloWorld.sol、TableTest.sol、Table.sol合约，指定包名为org.com.fisco，命令如下：

```
$ cd ~/fisco/console
$ ./sol2java.sh org.com.fisco
```

运行成功之后，将会在console/contracts/sdk目录生成java、abi和bin目录，如下所示。

```
|-- abi # 编译生成的abi目录，存放solidity合约编译的abi文件
|   |-- HelloWorld.abi
|   |-- Table.abi
|   |-- TableTest.abi
|-- bin # 编译生成的bin目录，存放solidity合约编译的bin文件
|   |-- HelloWorld.bin
|   |-- Table.bin
|   |-- TableTest.bin
|-- java  # 存放编译的包路径及Java合约文件
|   |-- org
|       |-- com
|           |-- fisco
|               |-- HelloWorld.java # 编译的HelloWorld Java文件
|               |-- Table.java  # 编译的CRUD接口合约 Java文件
|               |-- TableTest.java  # 编译的TableTest Java文件
```

java目录下生成了org/com/fisco/包路径目录。包路径目录下将会生成java合约文件HelloWorld.java、TableTest.java和Table.java。其中HelloWorld.java和TableTest.java是java应用所需要的java合约文件。

注：下载的控制台其console/lib目录下包含solcJ-all-0.4.25.jar，因此支持0.4版本的合约编译。如果使用0.5版本合约编译器或国密合约编译器，请下载相关合约编译器jar包，然后替换console/lib目录下的solcJ-all-0.4.25.jar。可以通过./replace_solc_jar.sh脚本进行替换，指定下载的编译器jar包路径，命令如下：

```
# 下载solcJ-all-0.5.2.jar放在console目录下，示例用法如下
$ ./replace_solc_jar.sh solcJ-all-0.5.2.jar
```

下载合约编译jar包

0.4版本合约编译jar包

```
$ curl -LO https://github.com/FISCO-BCOS/LargeFiles/raw/master/tools/solcj/solcJ-
↪all-0.4.25.jar
```

0.5版本合约编译jar包

```
$ curl -LO https://github.com/FISCO-BCOS/LargeFiles/raw/master/tools/solcj/solcJ-
↪all-0.5.2.jar
```

国密0.4版本合约编译jar包

```
$ curl -LO https://github.com/FISCO-BCOS/LargeFiles/raw/master/tools/solcj/solcJ-
↪all-0.4.25-gm.jar
```

国密0.5版本合约编译jar包

```
$ curl -LO https://github.com/FISCO-BCOS/LargeFiles/raw/master/tools/solcj/solcJ-
↪all-0.5.2-gm.jar
```

启动控制台

在节点正在运行的情况下，启动控制台：

```
$ ./start.sh
# 输出下述信息表明启动成功
========================================================================================
```

(continues on next page)

(续上页)

```
Welcome to FISCO BCOS console(1.0.4)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ _______  _______   _______   _______       ________   _______   _______   _______
|        |       \/       \ /       \ /       \     |        \ /       \ /       \ /       ␣
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$$\
| $$__      | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \     | $$  \$$    \| $$      | $$  | $$    | $$    $| $$      | $$  | $$\$$   ␣
↪\
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$$\
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__|
↪$$
| $$      |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$
↪$

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

启动脚本说明

查看当前控制台版本：

```
./start.sh --version
console version: 1.0.4
```

账户使用方式

控制台加载私钥

控制台提供账户生成脚本get_account.sh(脚本用法请参考账户管理文档，生成的的账户文件在accounts目录下，控制台加载的账户文件必须放置在该目录下。 控制台启动方式有如下几种：

```
./start.sh
./start.sh groupID
./start.sh groupID -pem pemName
./start.sh groupID -p12 p12Name
```

默认启动

控制台随机生成一个账户，使用控制台配置文件指定的群组号启动。

```
./start.sh
```

指定群组号启动

控制台随机生成一个账户，使用命令行指定的群组号启动。

```
./start.sh 2
```

- 注意：指定的群组在控制台配置文件中需要配置bean。

使用PEM格式私钥文件启动

- 使用指定的pem文件的账户启动，输入参数：群组号、-pem、pem文件路径

```
./start.sh 1 -pem accounts/0xebb824a1122e587b17701ed2e512d8638dfb9c88.pem
```

使用PKCS12格式私钥文件启动

- 使用指定的p12文件的账户，需要输入密码，输入参数：群组号、-p12、p12文件路径

```
./start.sh 1 -p12 accounts/0x5ef4df1b156bc9f077ee992a283c2dbb0bf045c0.p12
Enter Export Password:
```

注意： 控制台启动时加载p12文件出现下面报错：

```
exception unwrapping private key - java.security.InvalidKeyException: Illegal key␣
↪size
```

可能是Java版本的原因，参考解决方案：https://stackoverflow.com/questions/3862800/invalidkeyexception-illegal-key-size

5.9.6 控制台命令

help

输入help或者h，查看控制台所有的命令。

```
[group:1]> help
--------------------------------------------------------------------------------
↪--
addObserver                              Add an observer node.
addSealer                                Add a sealer node.
call                                     Call a contract by a function and␣
↪paramters.
callByCNS                                Call a contract by a function and␣
↪paramters by CNS.
deploy                                   Deploy a contract on blockchain.
deployByCNS                              Deploy a contract on blockchain by CNS.
desc                                     Description table information.
exit                                     Quit console.
getBlockByHash                           Query information about a block by hash.
getBlockByNumber                         Query information about a block by block␣
↪number.
getBlockHashByNumber                     Query block hash by block number.
getBlockNumber                           Query the number of most recent block.
getCode                                  Query code at a given address.
getConsensusStatus                       Query consensus status.
getDeployLog                             Query the log of deployed contracts.
getGroupList                             Query group list.
getGroupPeers                            Query nodeId list for sealer and observer␣
↪nodes.
getNodeIDList                            Query nodeId list for all connected nodes.
getNodeVersion                           Query the current node version.
getObserverList                          Query nodeId list for observer nodes.
getPbftView                              Query the pbft view of node.
```

(continues on next page)

(续上页)

```
getPeers                                Query peers currently connected to the␣
↪client.
getPendingTransactions                  Query pending transactions.
getPendingTxSize                        Query pending transactions size.
getSealerList                           Query nodeId list for sealer nodes.
getSyncStatus                           Query sync status.
getSystemConfigByKey                    Query a system config value by key.
setSystemConfigByKey                    Set a system config value by key.
getTotalTransactionCount                Query total transaction count.
getTransactionByBlockHashAndIndex       Query information about a transaction by␣
↪block hash and transaction index position.
getTransactionByBlockNumberAndIndex     Query information about a transaction by␣
↪block number and transaction index position.
getTransactionByHash                    Query information about a transaction␣
↪requested by transaction hash.
getTransactionReceipt                   Query the receipt of a transaction by␣
↪transaction hash.
getTransactionByHashWithProof           Query the transaction and transaction␣
↪proof by transaction hash.
getTransactionReceiptByHashWithProof    Query the receipt and transaction receipt␣
↪proof by transaction hash.
grantCNSManager                         Grant permission for CNS by address.
grantDeployAndCreateManager             Grant permission for deploy contract and␣
↪create user table by address.
grantNodeManager                        Grant permission for node configuration␣
↪by address.
grantSysConfigManager                   Grant permission for system configuration␣
↪by address.
grantUserTableManager                   Grant permission for user table by table␣
↪name and address.
help(h)                                 Provide help information.
listCNSManager                          Query permission information for CNS.
listDeployAndCreateManager              Query permission information for deploy␣
↪contract and create user table.
listNodeManager                         Query permission information for node␣
↪configuration.
listSysConfigManager                    Query permission information for system␣
↪configuration.
listUserTableManager                    Query permission for user table␣
↪information.
queryCNS                                Query CNS information by contract name␣
↪and contract version.
quit(q)                                 Quit console.
removeNode                              Remove a node.
revokeCNSManager                        Revoke permission for CNS by address.
revokeDeployAndCreateManager            Revoke permission for deploy contract and␣
↪create user table by address.
revokeNodeManager                       Revoke permission for node configuration␣
↪by address.
revokeSysConfigManager                  Revoke permission for system␣
↪configuration by address.
revokeUserTableManager                  Revoke permission for user table by table␣
↪name and address.
listContractWritePermission             Query the account list which have write␣
↪permission of the contract.
grantContractWritePermission            Grant the account the contract write␣
↪permission.
revokeContractWritePermission           Revoke the account the contract write␣
↪permission.
grantContractStatusManager              Grant contract authorization to the user.
getContractStatus                       Get the status of the contract.
```

(continues on next page)

(续上页)

```
listContractStatusManager                List the authorization of the contract.
grantCommitteeMember                     Grant the account committee member
revokeCommitteeMember                    Revoke the account from committee member
listCommitteeMembers                     List all committee members
grantOperator                            Grant the account operator
revokeOperator                           Revoke the operator
listOperators                            List all operators
updateThreshold                          Update the threshold
queryThreshold                           Query the threshold
updateCommitteeMemberWeight              Update the committee member weight
queryCommitteeMemberWeight               Query the committee member weight
freezeAccount                            Freeze the account.
unfreezeAccount                          Unfreeze the account.
getAccountStatus                         GetAccountStatus of the account.
freezeContract                           Freeze the contract.
unfreezeContract                         Unfreeze the contract.
switch(s)                                Switch to a specific group by group ID.
[create sql]                             Create table by sql.
[delete sql]                             Remove records by sql.
[insert sql]                             Insert records by sql.
[select sql]                             Select records by sql.
[update sql]                             Update records by sql.
--------------------------------------------------------------------------------
↪--
```

注：

- help显示每条命令的含义是：命令 命令功能描述
- 查看具体命令的使用介绍说明，输入命令 -h或–help查看。例如：

```
[group:1]> getBlockByNumber -h
Query information about a block by block number.
Usage: getBlockByNumber blockNumber [boolean]
blockNumber -- Integer of a block number, from 0 to 2147483647.
boolean -- (optional) If true it returns the full transaction objects, if false␣
↪only the hashes of the transactions.
```

switch

运行switch或者s，切换到指定群组。群组号显示在命令提示符前面。

```
[group:1]> switch 2
Switched to group 2.

[group:2]>
```

注： 需要切换的群组，请确保在`console/conf`目录下的`applicationContext.xml`(该配置文件初始状态只提供群组1的配置)文件中配置了该群组的信息，并且该群组中配置的节点ip和端口正确，该节点正常运行。

getBlockNumber

运行getBlockNumber，查看区块高度。

```
[group:1]> getBlockNumber
90
```

getSealerList

运行getSealerList，查看共识节点列表。

```
[group:1]> getSealerList
[
    ↵
↪0c0bbd25152d40969d3d3cee3431fa28287e07cff2330df3258782d3008b876d146ddab97eab42796495bfbb281591f
↪
    ↵
↪10b3a2d4b775ec7f3c2c9e8dc97fa52beb8caab9c34d026db9b95a72ac1d1c1ad551c67c2b7fdc34177857eada75836
↪
    ↵
↪622af37b2bd29c60ae8f15d467b67c0a7fe5eb3e5c63fdc27a0ee8066707a25afa3aa0eb5a3b802d3a8e5e26de9d5af
]
```

getObserverList

运行getObserverList，查看观察节点列表。

```
[group:1]> getObserverList
[
    ↵
↪037c255c06161711b6234b8c0960a6979ef039374ccc8b723afea2107cba3432dbbc837a714b7da20111f74d5a24e91
]
```

getNodeIDList

运行getNodeIDList，查看节点及连接p2p节点的nodeId列表。

```
[group:1]> getNodeIDList
[
    ↵
↪41285429582cbfe6eed501806391d2825894b3696f801e945176c7eb2379a1ecf03b36b027d72f480e89d15bacd4346
↪
    ↵
↪87774114e4a496c68f2482b30d221fa2f7b5278876da72f3d0a75695b81e2591c1939fc0d3fadb15cc359c997bafc9e
↪
    ↵
↪29c34347a190c1ec0c4507c6eed6a5bcd4d7a8f9f54ef26da616e81185c0af11a8cea4eacb74cf6f61820292b24bc5d
↪
    ↵
↪d5b3a9782c6aca271c9642aea391415d8b258e3a6d92082e59cc5b813ca123745440792ae0b29f4962df568f8ad58b7
]
```

getPbftView

运行getPbftView，查看pbft视图。

```
[group:1]> getPbftView
2730
```

getConsensusStatus

运行getConsensusStatus，查看共识状态。

```
[group:1]> getConsensusStatus
[
    {
  "id": 1,
  "jsonrpc": "2.0",
  "result": [
    {
      "accountType": 1,
      "allowFutureBlocks": true,
      "cfgErr": false,
      "connectedNodes": 3,
      "consensusedBlockNumber": 38207,
      "currentView": 54477,
      "groupId": 1,
      "highestblockHash":
↪"0x19a16e8833e671aa11431de589c866a6442ca6c8548ba40a44f50889cd785069",
      "highestblockNumber": 38206,
      "leaderFailed": false,
      "max_faulty_leader": 1,
      "nodeId":
↪"f72648fe165da17a889bece08ca0e57862cb979c4e3661d6a77bcc2de85cb766af5d299fec8a4337eedd142dca026a
↪",
      "nodeNum": 4,
      "node_index": 3,
      "omitEmptyBlock": true,
      "protocolId": 65544,
      "sealer.0":
↪"6a99f357ecf8a001e03b68aba66f68398ee08f3ce0f0147e777ec77995369aac470b8c9f0f85f91ebb58a98475764b
↪",
      "sealer.1":
↪"8a453f1328c80b908b2d02ba25adca6341b16b16846d84f903c4f4912728c6aae1050ce4f24cd9c13e010ce922d339
↪",
      "sealer.2":
↪"ed483837e73ee1b56073b178f5ac0896fa328fc0ed418ae3e268d9e9109721421ec48d68f28d6525642868b40dd265
↪",
      "sealer.3":
↪"f72648fe165da17a889bece08ca0e57862cb979c4e3661d6a77bcc2de85cb766af5d299fec8a4337eedd142dca026a
↪",
      "toView": 54477
    },
    [
      {
        "nodeId":
↪"6a99f357ecf8a001e03b68aba66f68398ee08f3ce0f0147e777ec77995369aac470b8c9f0f85f91ebb58a98475764b
↪",
        "view": 54474
      },
      {
        "nodeId":
↪"8a453f1328c80b908b2d02ba25adca6341b16b16846d84f903c4f4912728c6aae1050ce4f24cd9c13e010ce922d339
↪",
        "view": 54475
      },
      {
        "nodeId":
↪"ed483837e73ee1b56073b178f5ac0896fa328fc0ed418ae3e268d9e9109721421ec48d68f28d6525642868b40dd265
↪",
        "view": 54476
      },
      {
        "nodeId":
↪"f72648fe165da17a889bece08ca0e57862cb979c4e3661d6a77bcc2de85cb766af5d299fec8a4337eedd142dca026a
↪",
```

(continues on next page)

(续上页)

```
        "view": 54477
      }
    ]
  ]
}
]
```

getSyncStatus

运行getSyncStatus，查看同步状态。

```
[group:1]> getSyncStatus
{
    "blockNumber":5,
    "genesisHash":
↪"0xeccad5274949b9d25996f7a96b89c0ac5c099eb9b72cc00d65bc6ef09f7bd10b",
    "isSyncing":false,
    "latestHash":
↪"0xb99703130e24702d3b580111b0cf4e39ff60ac530561dd9eb0678d03d7acce1d",
    "nodeId":
↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bc
↪",
    "peers":[
        {
            "blockNumber":5,
            "genesisHash":
↪"0xeccad5274949b9d25996f7a96b89c0ac5c099eb9b72cc00d65bc6ef09f7bd10b",
            "latestHash":
↪"0xb99703130e24702d3b580111b0cf4e39ff60ac530561dd9eb0678d03d7acce1d",
            "nodeId":
↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bc
↪"
        },
        {
            "blockNumber":5,
            "genesisHash":
↪"0xeccad5274949b9d25996f7a96b89c0ac5c099eb9b72cc00d65bc6ef09f7bd10b",
            "latestHash":
↪"0xb99703130e24702d3b580111b0cf4e39ff60ac530561dd9eb0678d03d7acce1d",
            "nodeId":
↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966
↪"
        },
        {
            "blockNumber":5,
            "genesisHash":
↪"0xeccad5274949b9d25996f7a96b89c0ac5c099eb9b72cc00d65bc6ef09f7bd10b",
            "latestHash":
↪"0xb99703130e24702d3b580111b0cf4e39ff60ac530561dd9eb0678d03d7acce1d",
            "nodeId":
↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54
↪"
        }
    ],
    "protocolId":265,
    "txPoolSize":"0"
}
```

getNodeVersion

运行getNodeVersion，查看节点的版本。

```
[group:1]> getNodeVersion
{
        "Build Time":"20190107 10:15:23",
        "Build Type":"Linux/g++/RelWithDebInfo",
        "FISCO-BCOS Version":"2.0.0",
        "Git Branch":"master",
        "Git Commit Hash":"be95a6e3e85b621860b101c3baeee8be68f5f450"
}
```

getPeers

运行getPeers，查看节点的peers。

```
[group:1]> getPeers
[
        {
                "IPAndPort":"127.0.0.1:50723",
                "nodeId":
↪"8718579e9a6fee647b3d7404d59d66749862aeddef22e6b5abaafe1af6fc128fc33ed5a9a105abddab51e12004c6bf
↪",
                "Topic":[

                ]
        },
        {
                "IPAndPort":"127.0.0.1:50719",
                "nodeId":
↪"697e81e512cffc55fc9c506104fb888a9ecf4e29eabfef6bb334b0ebb6fc4ef8fab60eb614a0f2be178d0b5993464c
↪",
                "Topic":[

                ]
        },
        {
                "IPAndPort":"127.0.0.1:30304",
                "nodeId":
↪"8fc9661baa057034f10efacfd8be3b7984e2f2e902f83c5c4e0e8a60804341426ace51492ffae087d96c0b968bd5e9
↪",
                "Topic":[

                ]
        }
]
```

getGroupPeers

运行getGroupPeers，查看节点所在group的共识节点和观察节点列表。

```
[group:1]> getGroupPeers
[

↪cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bcc
↪

↪ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54b
↪
```

(continues on next page)

(续上页)

```
↪0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bca
↪
↪2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966a
]
```

getGroupList

运行getGroupList，查看群组列表:

```
[group:1]> getGroupList
[1]
```

getBlockByHash

运行getBlockByHash，根据区块哈希查询区块信息。 参数:

- 区块哈希：0x开头的区块哈希值。
- 交易标志：默认false，区块中的交易只显示交易哈希，设置为true，显示交易具体信息。

```
[group:1]> getBlockByHash␣
↪0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855
{
    "extraData":[

    ],
    "gasLimit":"0x0",
    "gasUsed":"0x0",
    "hash":"0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855",
    "logsBloom":
↪"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "number":"0x1",
    "parentHash":
↪"0xeccad5274949b9d25996f7a96b89c0ac5c099eb9b72cc00d65bc6ef09f7bd10b",
    "sealer":"0x0",
    "sealerList":[

↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bc
↪",

↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966
↪",

↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bc
↪",

↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54
↪"
    ],
    "stateRoot":"0x9711819153f7397ec66a78b02624f70a343b49c60bc2f21a77b977b0ed91cef9
↪",
    "timestamp":"0x1692f119c84",
    "transactions":[
        "0xa14638d47cc679cf6eeb7f36a6d2a30ea56cb8dcf0938719ff45023a7a8edb5d"
    ],
    "transactionsRoot":
↪"0x516787f85980a86fd04b0e9ce82a1a75950db866e8cdf543c2cae3e4a51d91b7
```

(continues on next page)

(续上页)

```
}
[group:1]> getBlockByHash␣
↪0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855 true
{
    "extraData":[

    ],
    "gasLimit":"0x0",
    "gasUsed":"0x0",
    "hash":"0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855",
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "number":"0x1",
    "parentHash":
↪"0xeccad5274949b9d25996f7a96b89c0ac5c099eb9b72cc00d65bc6ef09f7bd10b",
    "sealer":"0x0",
    "sealerList":[

↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bca
↪",

↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966a
↪",

↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bc
↪",

↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54
↪"
    ],
    "stateRoot":"0x9711819153f7397ec66a78b02624f70a343b49c60bc2f21a77b977b0ed91cef9
↪",
    "timestamp":"0x1692f119c84",
    "transactions":[
        {
            "blockHash":
↪"0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855",
            "blockNumber":"0x1",
            "from":"0x7234c32327795e4e612164e3442cfae0d445b9ad",
            "gas":"0x1c9c380",
            "gasPrice":"0x1",
            "hash":
↪"0xa14638d47cc679cf6eeb7f36a6d2a30ea56cb8dcf0938719ff45023a7a8edb5d",
            "input":
↪"0x6080604052348015610010576000080fd5b506040805190810160405280600d81526020017f48656c6c6f2c20576f
↪",
            "nonce":
↪"0x3443a1391c9c29f751e8350304efb310850b8afbaa7738f5e89ddfce79b1d6",
            "to":null,
            "transactionIndex":"0x0",
            "value":"0x0"
        }
    ],
    "transactionsRoot":
↪"0x516787f85980a86fd04b0e9ce82a1a75950db866e8cdf543c2cae3e4a51d91b7"
}
```

getBlockByNumber

运行getBlockByNumber，根据区块高度查询区块信息。 参数：

- 区块高度：十进制整数。
- 交易标志：默认false，区块中的交易只显示交易哈希，设置为true，显示交易具体信息。

```
[group:1]> getBlockByNumber 1
{
    "extraData":[

    ],
    "gasLimit":"0x0",
    "gasUsed":"0x0",
    "hash":"0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855",
    "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "number":"0x1",
    "parentHash":
↪"0xeccad5274949b9d25996f7a96b89c0ac5c099eb9b72cc00d65bc6ef09f7bd10b",
    "sealer":"0x0",
    "sealerList":[

↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bca
↪",

↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966a
↪",

↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bcc
↪",

↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54b
↪"
    ],
    "stateRoot":"0x9711819153f7397ec66a78b02624f70a343b49c60bc2f21a77b977b0ed91cef9
↪",
    "timestamp":"0x1692f119c84",
    "transactions":[
        "0xa14638d47cc679cf6eeb7f36a6d2a30ea56cb8dcf0938719ff45023a7a8edb5d"
    ],
    "transactionsRoot":
↪"0x516787f85980a86fd04b0e9ce82a1a75950db866e8cdf543c2cae3e4a51d91b7"
}
```

getBlockHashByNumber

运行getBlockHashByNumber，通过区块高度获得区块哈希。 参数：

- 区块高度：十进制整数。

```
[group:1]> getBlockHashByNumber 1
0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855
```

getTransactionByHash

运行getTransactionByHash，通过交易哈希查询交易信息。 参数：

- 交易哈希：0x开头的交易哈希值。

- 合约名：可选，发送交易产生该交易的合约名称，使用该参数可以将交易中的input解析并输出。如果是部署合约交易则不解析。

```
[group:1]> getTransactionByHash␣
↪0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gas":"0x11e1a300",
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000
↪",
    "nonce":"0x1aec6e447da49b9a140bf39a91a4d75fd19ea77f7dc38ccf940d8d510d78bd0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionIndex":"0x0",
    "value":"0x0"
}
# input字段是合约接口的编码，解析后的内容包括接口签名，输入参数值。
[group:1]> getTransactionByHash␣
↪0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5 TableTest
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gas":"0x11e1a300",
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000
↪",
    "nonce":"0x1aec6e447da49b9a140bf39a91a4d75fd19ea77f7dc38ccf940d8d510d78bd0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionIndex":"0x0",
    "value":"0x0"
}
----------------------------------------------------------------------------------------
↪----------
Input
function: insert(string,int256,string)
input value: (fruit, 1, apple)
----------------------------------------------------------------------------------------
↪----------
```

getTransactionByBlockHashAndIndex

运行getTransactionByBlockHashAndIndex，通过区块哈希和交易索引查询交易信息。 参数:

- 区块哈希：0x开头的区块哈希值。
- 交易索引：十进制整数。
- 合约名：可选，发送交易产生该交易的合约名称，使用该参数可以将交易中的input解析并输出。如果是部署合约交易则不解析。

```
[group:1]> getTransactionByBlockHashAndIndex␣
↪0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b 0
{
```

(continues on next page)

(续上页)

```
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gas":"0x11e1a300",
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000000
↪",
    "nonce":"0x1aec6e447da49b9a140bf39a91a4d75fd19ea77f7dc38ccf940d8d510d78bd0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionIndex":"0x0",
    "value":"0x0"
}

[group:1]> getTransactionByBlockHashAndIndex␣
↪0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b 0 TableTest
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gas":"0x11e1a300",
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000000
↪",
    "nonce":"0x1aec6e447da49b9a140bf39a91a4d75fd19ea77f7dc38ccf940d8d510d78bd0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionIndex":"0x0",
    "value":"0x0"
}
----------------------------------------------------------------------------------
↪----------
Input
function: insert(string,int256,string)
input value: (fruit, 1, apple)
----------------------------------------------------------------------------------
↪----------
```

getTransactionByBlockNumberAndIndex

运行getTransactionByBlockNumberAndIndex，通过区块高度和交易索引查询交易信息。 参数：

- 区块高度：十进制整数。
- 交易索引：十进制整数。
- 合约名：可选，发送交易产生该交易的合约名称，使用该参数可以将交易中的input解析并输出。如果是部署合约交易则不解析。

```
[group:1]> getTransactionByBlockNumberAndIndex 8 0
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gas":"0x11e1a300",
```

(continues on next page)

(续上页)

```
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000000
↪",
    "nonce":"0x1aec6e447da49b9a140bf39a91a4d75fd19ea77f7dc38ccf940d8d510d78bd0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionIndex":"0x0",
    "value":"0x0"
}

[group:1]> getTransactionByBlockNumberAndIndex 8 0 TableTest
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gas":"0x11e1a300",
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000000
↪",
    "nonce":"0x1aec6e447da49b9a140bf39a91a4d75fd19ea77f7dc38ccf940d8d510d78bd0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionIndex":"0x0",
    "value":"0x0"
}
----------------------------------------------------------------------------------------
↪----------
Input
function: insert(string,int256,string)
input value: (fruit, 1, apple)
----------------------------------------------------------------------------------------
↪----------
```

getTransactionReceipt

运行getTransactionReceipt，通过交易哈希查询交易回执。 参数:

- 交易哈希：0x开头的交易哈希值。
- 合约名：可选，发送交易产生该交易回执的合约名称，使用该参数可以将交易回执中的input、output和event log解析并输出。（注：input字段在web3sdk 2.0.4版本中新增加的字段，之前版本无该字段则只解析output和event log。）

```
[group:1]> getTransactionReceipt␣
↪0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "contractAddress":"0x0000000000000000000000000000000000000000",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gasUsed":"0x94f5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000000
↪",
    "logs":[
        {
```

(continues on next page)

(续上页)

```
            "address":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
            "data":
↪"0x0000000000000000000000000000000000000000000000000000000000000001",
            "topics":[
                "0xc57b01fa77f41df77eaab79a0e2623fab2e7ae3e9530d9b1cab225ad65f2b7ce
↪"
            ]
        }
    ],
    "logsBloom":
↪"0x00000000000000800000000000000000000000000000000000000000000000000000000000000000000000000000000000000000020000000000
↪",
    "output":"0x0000000000000000000000000000000000000000000000000000000000000001",
    "status":"0x0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionHash":
↪"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "transactionIndex":"0x0"
}

[group:1]> getTransactionReceipt␣
↪0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5 TableTest
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "contractAddress":"0x0000000000000000000000000000000000000000",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gasUsed":"0x94f5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000
↪",
    "logs":[
        {
            "address":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
            "data":
↪"0x0000000000000000000000000000000000000000000000000000000000000001",
            "topics":[
                "0xc57b01fa77f41df77eaab79a0e2623fab2e7ae3e9530d9b1cab225ad65f2b7ce
↪"
            ]
        }
    ],
    "logsBloom":
↪"0x00000000000000800000000000000000000000000000000000000000000000000000000000000000000000000000000000000000020000000000
↪",
    "output":"0x0000000000000000000000000000000000000000000000000000000000000001",
    "status":"0x0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionHash":
↪"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "transactionIndex":"0x0"
}
-------------------------------------------------------------------------------
↪----------
Input
function: insert(string,int256,string)
input value: (fruit, 1, apple)
-------------------------------------------------------------------------------
↪----------
-------------------------------------------------------------------------------
↪----------
```

(continues on next page)

(续上页)

```
Output
function: insert(string,int256,string)
return type: (int256)
return value: (1)
-------------------------------------------------------------------------------------------
↪----------
Event logs
event signature: InsertResult(int256) index: 0
event value: (1)
-------------------------------------------------------------------------------------------
↪----------
```

getPendingTransactions

运行getPendingTransactions，查询等待处理的交易。

```
[group:1]> getPendingTransactions
[]
```

getPendingTxSize

运行getPendingTxSize，查询等待处理的交易数量（交易池中的交易数量）。

```
[group:1]> getPendingTxSize
0
```

getCode

运行getCode，根据合约地址查询合约二进制代码。 参数:

- 合约地址：0x的合约地址(部署合约可以获得合约地址)。

```
[group:1]> getCode 0x97b8c19598fd781aaeb53a1957ef9c8acc59f705
0x60606040526000357c0100000000000000000000000000000000000000000000000000000000900463ffffffff16806
```

getTotalTransactionCount

运行getTotalTransactionCount，查询当前块高和累计交易数（从块高为0开始）。

```
[group:1]> getTotalTransactionCount
{
        "blockNumber":1,
        "txSum":1,
        "failedTxSum":0
}
```

deploy

运行deploy，部署合约。(默认提供HelloWorld合约和TableTest.sol进行示例使用) 参数:

- 合约名称：部署的合约名称(可以带.sol后缀)，即HelloWorld或者HelloWorld.sol均可。

```
# 部署HelloWorld合约
[group:1]> deploy HelloWorld.sol
contract address:0xc0ce097a5757e2b6e189aa70c7d55770ace47767

# 部署TableTest合约
[group:1]> deploy TableTest.sol
contract address:0xd653139b9abffc3fe07573e7bacdfd35210b5576
```

注：

- 部署用户编写的合约，只需要将solidity合约文件放到控制台根目录的`contracts/solidity/`目录下，然后进行部署即可。按tab键可以搜索`contracts/solidity/`目录下的合约名称。
- 若需要部署的合约引用了其他其他合约或library库，引用格式为`import "./XXX.sol";`。其相关引入的合约和library库均放在`contracts/solidity/`目录。
- 如果合约引用了library库，library库文件的名称必须以`Lib`字符串开始，以便于区分是普通合约与library库文件。library库文件不能单独部署和调用。
- **由于FISCO BCOS已去除以太币的转账支付逻辑，因此solidity合约的方法不支持使用`payable`关键字，该关键字会导致solidity合约转换成的java合约文件在编译时失败。**

getDeployLog

运行getDeployLog，查询群组内**由当前控制台**部署合约的日志信息。日志信息包括部署合约的时间，群组ID，合约名称和合约地址。参数：

- 日志行数，可选，根据输入的期望值返回最新的日志信息，当实际条数少于期望值时，按照实际数量返回。当期望值未给出时，默认按照20条返回最新的日志信息。

```
[group:1]> getDeployLog 2

2019-05-26 08:37:03  [group:1]  HelloWorld           ␣
↪0xc0ce097a5757e2b6e189aa70c7d55770ace47767
2019-05-26 08:37:45  [group:1]  TableTest            ␣
↪0xd653139b9abffc3fe07573e7bacdfd35210b5576

[group:1]> getDeployLog 1

2019-05-26 08:37:45  [group:1]  TableTest            ␣
↪0xd653139b9abffc3fe07573e7bacdfd35210b5576
```

注：如果要查看所有的部署合约日志信息，请查看`console`目录下的`deploylog.txt`文件。该文件只存储最近10000条部署合约的日志记录。

call

运行call，调用合约。参数：

- 合约名称：部署的合约名称(可以带.sol后缀)。
- 合约地址: 部署合约获取的地址，合约地址可以省略前缀0，例如，0x000ac78可以简写成0xac78。
- 合约接口名：调用的合约接口名。
- 参数：由合约接口参数决定。**参数由空格分隔，其中字符串、字节类型参数需要加上双引号；数组参数需要加上中括号，比如[1,2,3]，数组中是字符串或字节类型，加双引号，例如["alice","bob"]，注意数组参数中不要有空格；布尔类型为true或者false。**

```
# 调用HelloWorld的get接口获取name字符串
[group:1]> call HelloWorld.sol 0xc0ce097a5757e2b6e189aa70c7d55770ace47767 get
Hello, World!
```

(continues on next page)

(续上页)

```
# 调用HelloWorld的set接口设置name字符串
[group:1]> call HelloWorld.sol 0xc0ce097a5757e2b6e189aa70c7d55770ace47767 set
↪"Hello, FISCO BCOS"
transaction hash:0xa7c7d5ef8d9205ce1b228be1fe90f8ad70eeb6a5d93d3f526f30d8f431cb1e70

# 调用HelloWorld的get接口获取name字符串，检查设置是否生效
[group:1]> call HelloWorld.sol 0xc0ce097a5757e2b6e189aa70c7d55770ace47767 get
Hello, FISCO BCOS

# 调用TableTest的insert接口插入记录，字段为name, item_id, item_name
[group:1]> call TableTest.sol 0xd653139b9abffc3fe07573e7bacdfd35210b5576 insert
↪"fruit" 1 "apple"
transaction hash:0x6393c74681f14ca3972575188c2d2c60d7f3fb08623315dbf6820fc9dcc119c1
----------------------------------------------------------------------------------------
↪----------
Output
function: insert(string,int256,string)
return type: (int256)
return value: (1)
----------------------------------------------------------------------------------------
↪----------
Event logs
event signature: InsertResult(int256) index: 0
event value: (1)
----------------------------------------------------------------------------------------
↪----------

# 调用TableTest的select接口查询记录
[group:1]> call TableTest.sol 0xd653139b9abffc3fe07573e7bacdfd35210b5576 select
↪"fruit"
[[fruit], [1], [apple]]
```

注： TableTest.sol合约代码参考这里。

deployByCNS

运行deployByCNS，采用CNS部署合约。用CNS部署的合约，可用合约名直接调用。 参数:

- 合约名称：部署的合约名称。
- 合约版本号：部署的合约版本号(长度不能超过40)。

```
# 部署HelloWorld合约1.0版
[group:1]> deployByCNS HelloWorld.sol 1.0
contract address:0x3554a56ea2905f366c345bd44fa374757fb4696a

# 部署HelloWorld合约2.0版
[group:1]> deployByCNS HelloWorld.sol 2.0
contract address:0x07625453fb4a6459cbf14f5aa4d574cae0f17d92

# 部署TableTest合约
[group:1]> deployByCNS TableTest.sol 1.0
contract address:0x0b33d383e8e93c7c8083963a4ac4a58b214684a8
```

注：

- 部署用户编写的合约，只需要将solidity合约文件放到控制台根目录的`contracts/solidity/`目录下，然后进行部署即可。按tab键可以搜索`contracts/solidity/`目录下的合约名称。
- 若需要部署的合约引用了其他其他合约或library库，引用格式为`import "./XXX.sol";`。其相关引入的合约和library库均放在`contracts/solidity/`目录。

- 如果合约引用了library库，library库文件的名称必须以Lib字符串开始，以便于区分是普通合约与library库文件。library库文件不能单独部署和调用。
- **由于FISCO BCOS已去除以太币的转账支付逻辑，因此solidity合约的方法不支持使用payable关键字，该关键字会导致solidity合约转换成的java合约文件在编译时失败。**

queryCNS

运行queryCNS，根据合约名称和合约版本号（可选参数）查询CNS表记录信息（合约名和合约地址的映射）。参数：

- 合约名称：部署的合约名称。
- 合约版本号：(可选)部署的合约版本号。

```
[group:1]> queryCNS HelloWorld.sol
--------------------------------------------------------------------------------------
↪----------
|                    version                    |                    address
↪          |
|                      1.0                      |
↪0x3554a56ea2905f366c345bd44fa374757fb4696a  |
--------------------------------------------------------------------------------------
↪----------

[group:1]> queryCNS HelloWorld 1.0
--------------------------------------------------------------------------------------
↪----------
|                    version                    |                    address
↪          |
|                      1.0                      |
↪0x3554a56ea2905f366c345bd44fa374757fb4696a  |
--------------------------------------------------------------------------------------
↪----------
```

callByCNS

运行callByCNS，采用CNS调用合约，即用合约名直接调用合约。参数：

- 合约名称与合约版本号：合约名称与版本号用英文冒号分隔，例如HelloWorld:1.0或HelloWorld.sol:1.0。当省略合约版本号时，例如HelloWorld或HelloWorld.sol，则调用最新版本的合约。
- 合约接口名：调用的合约接口名。
- 参数：由合约接口参数决定。**参数由空格分隔，其中字符串、字节类型参数需要加上双引号；数组参数需要加上中括号，比如[1,2,3]，数组中是字符串或字节类型，加双引号，例如["alice","bob"]；布尔类型为true或者false。**

```
# 调用HelloWorld合约1.0版，通过set接口设置name字符串
[group:1]> callByCNS HelloWorld:1.0 set "Hello,CNS"
transaction hash:0x80bb37cc8de2e25f6a1cdcb6b4a01ab5b5628082f8da4c48ef1bbc1fb1d28b2d

# 调用HelloWorld合约2.0版，通过set接口设置name字符串
[group:1]> callByCNS HelloWorld:2.0 set "Hello,CNS2"
transaction hash:0x43000d14040f0c67ac080d0179b9499b6885d4a1495d3cfd1a79ffb5f2945f64

# 调用HelloWorld合约1.0版，通过get接口获取name字符串
[group:1]> callByCNS HelloWorld:1.0 get
Hello,CNS
```

(continues on next page)

(续上页)

```
# 调用HelloWorld合约最新版(即2.0版)，通过get接口获取name字符串
[group:1]> callByCNS HelloWorld get
Hello,CNS2
```

addSealer

运行addSealer，将节点添加为共识节点。 参数：

- 节点nodeId

```
[group:1]> addSealer␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
        "code":0,
        "msg":"success"
}
```

addObserver

运行addObserver，将节点添加为观察节点。 参数：

- 节点nodeId

```
[group:1]> addObserver␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
        "code":0,
        "msg":"success"
}
```

removeNode

运行removeNode，节点退出。通过addSealer命令可以将退出的节点添加为共识节点，通过addObserver命令将节点添加为观察节点。 参数：

- 节点nodeId

```
[group:1]> removeNode␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
        "code":0,
        "msg":"success"
}
```

setSystemConfigByKey

运行setSystemConfigByKey，以键值对方式设置系统参数。目前设置的系统参数支持tx_count_limit,tx_gas_limit, rpbft_epoch_sealer_num和rpbft_epoch_block_num。这些系统参数的键名可以通过tab键补全：

- tx_count_limit：区块最大打包交易数
- tx_gas_limit：交易执行允许消耗的最大gas数
- rpbft_epoch_sealer_num: rPBFT系统配置，一个共识周期内选取的共识节点数目
- rpbft_epoch_block_num: rPBFT系统配置，一个共识周期出块数目

参数：

- key
- value

```
[group:1]> setSystemConfigByKey tx_count_limit 100
{
        "code":0,
        "msg":"success"
}
```

getSystemConfigByKey

运行getSystemConfigByKey，根据键查询系统参数的值。 参数：

- key

```
[group:1]> getSystemConfigByKey tx_count_limit
100
```

grantPermissionManager

运行grantPermissionManager，授权账户的链管理员权限。参数：

- 账户地址

```
[group:1]> grantPermissionManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

注：权限控制相关命令的示例使用可以参考权限控制使用文档。

listPermissionManager

运行listPermissionManager，查询拥有链管理员权限的账户列表。

```
[group:1]> listPermissionManager
-------------------------------------------------------------------------------------------
↪----------
|                    address                    |                 enable_num
↪          |
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                      2
↪          |
-------------------------------------------------------------------------------------------
↪----------
```

revokePermissionManager

运行revokePermissionManager，撤销账户的链管理员权限。 参数：

- 账户地址

```
[group:1]> revokePermissionManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

grantUserTableManager

运行grantUserTableManager，授权账户对用户表的写权限。 参数：

- 表名
- 账户地址

```
[group:1]> grantUserTableManager t_test 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

listUserTableManager

运行listUserTableManager，查询拥有对用户表写权限的账号列表。 参数：

- 表名

```
[group:1]> listUserTableManager t_test
----------------------------------------------------------------------------------------
↪----------
|                    address                    |                 enable_num
↪          |
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                      2
↪          |
----------------------------------------------------------------------------------------
↪----------
```

revokeUserTableManager

运行revokeUserTableManager，撤销账户对用户表的写权限。 参数：

- 表名
- 账户地址

```
[group:1]> revokeUserTableManager t_test 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

grantDeployAndCreateManager

运行grantDeployAndCreateManager，授权账户的部署合约和创建用户表权限。

参数：

- 账户地址

```
[group:1]> grantDeployAndCreateManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

listDeployAndCreateManager

运行listDeployAndCreateManager，查询拥有部署合约和创建用户表权限的账户列表。

```
[group:1]> listDeployAndCreateManager
----------------------------------------------------------------------------------------
----------
|                    address                    |                  enable_num
|
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                       2
|
----------------------------------------------------------------------------------------
----------
```

revokeDeployAndCreateManager

运行revokeDeployAndCreateManager，撤销账户的部署合约和创建用户表权限。 参数：

- 账户地址

```
[group:1]> revokeDeployAndCreateManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

grantNodeManager

运行grantNodeManager，授权账户的节点管理权限。参数：

- 账户地址

```
[group:1]> grantNodeManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

listNodeManager

运行listNodeManager，查询拥有节点管理的账户列表。

```
[group:1]> listNodeManager
----------------------------------------------------------------------------------------
----------
|                    address                    |                  enable_num
|
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                       2
|
----------------------------------------------------------------------------------------
----------
```

revokeNodeManager

运行revokeNodeManager，撤销账户的节点管理权限。 参数：

- 账户地址

```
[group:1]> revokeNodeManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

grantCNSManager

运行grantCNSManager，授权账户的使用CNS权限。参数：

- 账户地址

```
[group:1]> grantCNSManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

listCNSManager

运行listCNSManager，查询拥有使用CNS的账户列表。

```
[group:1]> listCNSManager
--------------------------------------------------------------------------------
↪----------
|                   address                   |                 enable_num        ␣
↪          |
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                      2            ␣
↪          |
--------------------------------------------------------------------------------
↪----------
```

revokeCNSManager

运行revokeCNSManager，撤销账户的使用CNS权限。参数：

- 账户地址

```
[group:1]> revokeCNSManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

grantSysConfigManager

运行grantSysConfigManager，授权账户的修改系统参数权限。参数：

- 账户地址

```
[group:1]> grantSysConfigManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

listSysConfigManager

运行listSysConfigManager，查询拥有修改系统参数的账户列表。

```
[group:1]> listSysConfigManager
--------------------------------------------------------------------------------
↪----------
|                    address                    |                 enable_num        ␣
↪          |
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                      2            ␣
↪          |
--------------------------------------------------------------------------------
↪----------
```

revokeSysConfigManager

运行revokeSysConfigManager，撤销账户的修改系统参数权限。参数：

- 账户地址

```
[group:1]> revokeSysConfigManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

grantContractWritePermission

运行grantContractWritePermission，添加账户对合约写接口的调用权限。参数：

- 合约地址
- 账户地址

```
[group:1]> grantContractWritePermission 0xc0ce097a5757e2b6e189aa70c7d55770ace47767␣
↪0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

listContractWritePermission

运行listContractWritePermission，显示对某个合约的写接口有调用权限的账户。参数：

- 合约地址

```
[group:1]> listContractWritePermission 0xc0ce097a5757e2b6e189aa70c7d55770ace47767
--------------------------------------------------------------------------------
↪----------
|                    address                    |                 enable_num        ␣
↪          |
```

(continues on next page)

(续上页)

```
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                    11
↪          |
--------------------------------------------------------------------------------
↪----------
```

revokeContractWritePermission

运行revokeContractWritePermission，撤销账户对合约的写接口调用权限。参数：

- 合约地址
- 账户地址

```
[group:1]> revokeContractWritePermission
↪0xc0ce097a5757e2b6e189aa70c7d55770ace47767
↪0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

quit

运行quit、q或exit，退出控制台。

```
quit
```

[create sql]

运行create sql语句创建用户表，使用mysql语句形式。

```
# 创建用户表t_demo，其主键为name，其他字段为item_id和item_name
[group:1]> create table t_demo(name varchar, item_id varchar, item_name varchar,
↪primary key(name))
Create 't_demo' Ok.
```

注意：

- 创建表的字段类型均为字符串类型，即使提供数据库其他字段类型，也按照字符串类型设置。
- 必须指定主键字段。例如创建t_demo表，主键字段为name。
- 表的主键与关系型数据库中的主键不是相同概念，这里主键取值不唯一，区块链底层处理记录时需要传入主键值。
- 可以指定字段为主键，但设置的字段自增，非空，索引等修饰关键字不起作用。

desc

运行desc语句查询表的字段信息，使用mysql语句形式。

```
# 查询t_demo表的字段信息，可以查看表的主键名和其他字段名
[group:1]> desc t_demo
{
    "key":"name",
    "valueFields":"item_id,item_name"
}
```

[insert sql]

运行insert sql语句插入记录，使用mysql语句形式。

```
[group:1]> insert into t_demo (name, item_id, item_name) values (fruit, 1, apple1)
Insert OK, 1 row affected.
```

注意：

- 插入记录sql语句必须插入表的主键字段值。
- 输入的值带标点符号、空格或者以数字开头的包含字母的字符串，需要加上双引号，双引号中不允许再用双引号。

[select sql]

运行select sql语句查询记录，使用mysql语句形式。

```
# 查询包含所有字段的记录
select * from t_demo where name = fruit
{item_id=1, item_name=apple1, name=fruit}
1 row in set.

# 查询包含指定字段的记录
[group:1]> select name, item_id, item_name from t_demo where name = fruit
{name=fruit, item_id=1, item_name=apple1}
1 row in set.

# 插入一条新记录
[group:1]> insert into t_demo values (fruit, 2, apple2)
Insert OK, 1 row affected.

# 使用and关键字连接多个查询条件
[group:1]> select * from t_demo where name = fruit and item_name = apple2
{item_id=2, item_name=apple2, name=fruit}
1 row in set.

# 使用limit字段，查询第1行记录，没有提供偏移量默认为0
[group:1]> select * from t_demo where name = fruit limit 1
{item_id=1, item_name=apple1, name=fruit}
1 row in set.

# 使用limit字段，查询第2行记录，偏移量为1
[group:1]> select * from t_demo where name = fruit limit 1,1
{item_id=2, item_name=apple2, name=fruit}
1 rows in set.
```

注意：

- 查询记录sql语句必须在where子句中提供表的主键字段值。
- 关系型数据库中的limit字段可以使用，提供两个参数，分别offset(偏移量)和记录数(count)。
- where条件子句只支持and关键字，其他or、in、like、inner、join，union以及子查询、多表联合查询等均不支持。
- 输入的值带标点符号、空格或者以数字开头的包含字母的字符串，需要加上双引号，双引号中不允许再用双引号。

[update sql]

运行update sql语句更新记录，使用mysql语句形式。

```
[group:1]> update t_demo set item_name = orange where name = fruit and item_id = 1
Update OK, 1 row affected.
```

注意：

- 更新记录sql语句的where子句必须提供表的主键字段值。
- 输入的值带标点符号、空格或者以数字开头的包含字母的字符串，需要加上双引号，双引号中不允许再用双引号。

[delete sql]

运行delete sql语句删除记录，使用mysql语句形式。

```
[group:1]> delete from t_demo where name = fruit and item_id = 1
Remove OK, 1 row affected.
```

注意：

- 删除记录sql语句的where子句必须提供表的主键字段值。
- 输入的值带标点符号、空格或者以数字开头的包含字母的字符串，需要加上双引号，双引号中不允许再用双引号。

重要： 执行‘freezeContract‘、‘unfreezeContract‘和‘grantContractStatusManager‘三个合约管理的控制台命令，需指定私钥启动控制台，用于进行操作权限判断。该私钥为部署指定合约时所用的账号私钥，即部署合约时也许指定私钥启动控制台。

---

freezeContract

运行freezeContract，对指定合约进行冻结操作。参数：

- 合约地址：部署合约可以获得合约地址，其中0x前缀非必须。

```
[group:1]> freezeContract 0xcc5fc5abe347b7f81d9833f4d84a356e34488845
{
    "code":0,
    "msg":"success"
}
```

unfreezeContract

运行unfreezeContract，对指定合约进行解冻操作。参数：

- 合约地址：部署合约可以获得合约地址，其中0x前缀非必须。

```
[group:1]> unfreezeContract 0xcc5fc5abe347b7f81d9833f4d84a356e34488845
{
    "code":0,
    "msg":"success"
}
```

grantContractStatusManager

运行grantContractStatusManager，用于已有权限账号给其他账号授予指定合约的合约管理权限。参数：

- 合约地址：部署合约可以获得合约地址，其中0x前缀非必须。

- 账号地址：tx.origin，其中0x前缀非必须。

```
[group:1]> grantContractStatusManager 0x30d2a17b6819f0d77f26dd3a9711ae75c291f7f1␣
↪0x965ebffc38b309fa706b809017f360d4f6de909a
{
    "code":0,
    "msg":"success"
}
```

getContractStatus

运行getContractStatus，查询指定合约的状态。参数：

- 合约地址：部署合约可以获得合约地址，其中0x前缀非必须。

```
[group:1]> getContractStatus 0xcc5fc5abe347b7f81d9833f4d84a356e34488845
The contract is available.
```

listContractStatusManager

运行listContractStatusManager，查询能管理指定合约的权限账号列表。参数：

- 合约地址：部署合约可以获得合约地址，其中0x前缀非必须。

```
[group:1]> listContractStatusManager 0x30d2a17b6819f0d77f26dd3a9711ae75c291f7f1
[
    "0x0cc9b73b960323816ac5f52806257184c08b5ac0",
    "0x965ebffc38b309fa706b809017f360d4f6de909a"
]
```

grantCommitteeMember

添加治理委员会委员，如果当前没有委员，则直接添加成功，否则判断投票账号是否有权限投票，如有则记录投票并检查投票是否生效。委员账号可以添加运维、管理链系统配置、节点增删等，详情参考这里。参数：

- 账号地址：投票添加该账号为委员

```
[group:1]> grantCommitteeMember 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a
{
    "code":0,
    "msg":"success"
}
```

revokeCommitteeMember

撤销治理委员会委员判断投票账号是否有权限投票，如有则记录投票并检查投票是否生效。参数：

- 账号地址：投票撤销该账号的委员权限

```
[group:1]> revokeCommitteeMember 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a
{
    "code":0,
    "msg":"success"
}
```

listCommitteeMembers

查询所有治理委员会委员。

```
[group:1]> listCommitteeMembers
--------------------------------------------------------------------------------------
↪----------
|                   address                    |                 enable_num
↪          |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a   |                      1
↪          |
| 0x85961172229aec21694d742a5bd577bedffcfec3   |                      2
↪          |
--------------------------------------------------------------------------------------
↪----------
```

updateThreshold

投票更新生效阈值，判断投票账号是否是委员，是则计票并判断是否生效。参数：

- 生效阈值：取值范围[0,99]

```
[group:1]> updateThreshold 75
{
    "code":0,
    "msg":"success"
}
```

queryThreshold

查询生效阈值。

```
[group:1]> queryThreshold
Effective threshold : 50%
```

queryCommitteeMemberWeight

查询治理委员会委员的票数。

```
[group:1]> queryCommitteeMemberWeight 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a
Account: 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a Weight: 1
```

updateCommitteeMemberWeight

投票更新委员账号的票数，检查投票账号是否有权限，有则计票并检查是否生效，生效后该委员账号投票操作相当于设置的票数。参数：

- 委员账号：被投票修改票数的委员账号
- 投票权重：希望修改的权重

```
[group:1]> updateCommitteeMemberWeight 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a 2
{
    "code":0,
    "msg":"success"
}
```

grantOperator

添加运维账号，运维角色拥有部署合约、创建用户表和管理CNS的权限，治理委员会委员可以添加运维，如果当前没有委员，则不限制。参数：

- 账号地址：添加该账号为运维

```
[group:1]> grantOperator 0x283f5b859e34f7fd2cf136c07579dcc72423b1b2
{
    "code":0,
    "msg":"success"
}
```

revokeOperator

撤销账号的运维权限，委员可以操作，如果当前没有委员，则不限制。参数：

- 账号地址：撤销该账号的运维权限

```
[group:1]> revokeOperator 0x283f5b859e34f7fd2cf136c07579dcc72423b1b2
{
    "code":0,
    "msg":"success"
}
```

listOperators

查询有运维权限的账号。

```
[group:1]> listOperators
--------------------------------------------------------------------------------------------
↪----------
|                    address                    |                  enable_num
↪          |
| 0x283f5b859e34f7fd2cf136c07579dcc72423b1b2  |                           1
↪          |
--------------------------------------------------------------------------------------------
↪----------
```

freezeAccount

运行freezeAccount，对指定账号进行冻结操作。对没有发送过交易的账号，冻结操作将提示该账号地址不存在。参数：

- 账号地址：tx.origin，其中0x前缀必须。

```
[group:1]> freezeAccount 0xcc5fc5abe347b7f81d9833f4d84a356e34488845
{
    "code":0,
    "msg":"success"
}
```

unfreezeAccount

运行unfreezeAccount，对指定账号进行解冻操作。对没有发送过交易的账号，解冻操作将提示该账号地址不存在。参数：

- 账号地址：tx.origin，其中0x前缀必须。

```
[group:1]> unfreezeAccount 0xcc5fc5abe347b7f81d9833f4d84a356e34488845
{
    "code":0,
    "msg":"success"
}
```

getAccountStatus

运行getAccountStatus，查询指定账号的状态。对没有发送过交易的账号，查询操作将提示该账号地址不存在。参数：

- 账号地址：tx.origin，其中0x前缀必须。

```
[group:1]> getAccountStatus 0xcc5fc5abe347b7f81d9833f4d84a356e34488845
The account is available.
```

5.9.7 附录：Java环境配置

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
```

CentOS环境安装Java

注意：CentOS下OpenJDK无法正常工作，需要安装OracleJDK下载链接。

```
# 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
$ 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   #这是一个文件目录，非文件
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
```

5.10 账户管理

FISCO BCOS使用账户来标识和区分每一个独立的用户。在采用公私钥体系的区块链系统里，每一个账户对应着一对公钥和私钥。其中，由公钥经哈希等安全的单向性算法计算后得到地址字符串被用作该账户的账户名，即**账户地址**，为了与智能合约的地址相区别和一些其他的历史原因，账户地址也常被称之**外部账户地址**。而仅有用户知晓的私钥则对应着传统认证模型中的密码。用户需要通过安全的密码学协议证明其知道对应账户的私钥，来声明其对于该账户的所有权，以及进行敏感的账户操作。

重要: 在之前的其他教程中，为了简化操作，使用了工具提供的默认的账户进行操作。但在实际应用部署中，用户需要创建自己的账户，并妥善保存账户私钥，避免账户私钥泄露等严重的安全问题。

本文将具体介绍账户的创建、存储和使用方式，要求阅读者有一定的Linux操作基础。

FISCO BCOS提供了脚本和Web3SDK用以创建账户，同时也提供了Web3SDK和控制台来存储账户私钥。用户可以根据需求选择将账户私钥存储为PEM或者PKCS12格式的文件。其中，PEM格式使用明文存储私钥，而PKCS12使用用户提供的口令加密存储私钥。

5.10.1 账户的创建

使用脚本创建账户

1. 获取脚本

```
curl -LO https://raw.githubusercontent.com/FISCO-BCOS/console/master/tools/get_
↪account.sh && chmod u+x get_account.sh && bash get_account.sh -h
```

注解:

- 如果因为网络问题导致长时间无法下载，请尝试 *curl -LO https://gitee.com/FISCO-BCOS/console/raw/master/tools/get_account.sh && chmod u+x get_account.sh && bash get_account.sh -h*

国密版本请使用下面的指令获取脚本

```
curl -LO https://raw.githubusercontent.com/FISCO-BCOS/console/master/tools/get_gm_
↪account.sh && chmod u+x get_gm_account.sh && bash get_gm_account.sh -h
```

注解:

- 如果因为网络问题导致长时间无法下载，请尝试 *curl -LO https://gitee.com/FISCO-BCOS/console/raw/master/tools/get_gm_account.sh && chmod u+x get_gm_account.sh && bash get_gm_account.sh -h*
- get_gm_account需要下载tassl，如果无法下载，请尝试 *curl -LO https://www.fisco.com.cn/cdn/deps/tools/tassl.tar.gz* ，解压放在~/.fisco/tassl，1.0.9及以下版本放在~/.tassl

执行上面的指令，看到如下输出则下载到了正确的脚本，否则请重试。

```
Usage: ./get_account.sh
    default       generate account and store private key in PEM format file
    -p            generate account and store private key in PKCS12 format file
    -k [FILE]     calculate address of PEM format [FILE]
    -P [FILE]     calculate address of PKCS12 format [FILE]
    -h Help
```

2. 使用脚本生成PEM格式私钥

- 生成私钥与地址

```
bash get_account.sh
```

执行上面的命令，可以得到类似下面的输出，包括账户地址和以账户地址为文件名的私钥PEM文件。

```
[INFO] Account Address   : 0xee5fffba2da55a763198e361c7dd627795906ead
[INFO] Private Key (pem) : accounts/0xee5fffba2da55a763198e361c7dd627795906ead.pem
```

- 指定PEM私钥文件计算账户地址

```
bash get_account.sh -k accounts/0xee5fffba2da55a763198e361c7dd627795906ead.pem
```

执行上面的命令，结果如下

```
[INFO] Account Address   : 0xee5fffba2da55a763198e361c7dd627795906ead
```

3. 使用脚本生成PKCS12格式私钥

- 生成私钥与地址

```
bash get_account.sh -p
```

执行上面的命令，可以得到类似下面的输出，按照提示输入密码，生成对应的p12文件。

```
Enter Export Password:
Verifying - Enter Export Password:
[INFO] Account Address   : 0x02f1b23310ac8e28cb6084763d16b25a2cc7f5e1
[INFO] Private Key (p12) : accounts/0x02f1b23310ac8e28cb6084763d16b25a2cc7f5e1.p12
```

- 指定p12私钥文件计算账户地址，**按提示输入p12文件密码**

```
bash get_account.sh -P accounts/0x02f1b23310ac8e28cb6084763d16b25a2cc7f5e1.p12
```

执行上面的命令，结果如下

```
Enter Import Password:
MAC verified OK
[INFO] Account Address   : 0x02f1b23310ac8e28cb6084763d16b25a2cc7f5e1
```

调用Web3SDK创建账户

```
//创建普通账户
EncryptType.encryptType = 0;
//创建国密账户，向国密区块链节点发送交易需要使用国密账户
// EncryptType.encryptType = 1;
Credentials credentials = GenCredential.create();
//账户地址
String address = credentials.getAddress();
//账户私钥
String privateKey = credentials.getEcKeyPair().getPrivateKey().toString(16);
//账户公钥
String publicKey = credentials.getEcKeyPair().getPublicKey().toString(16);
```

更多操作详情，请参见创建并使用指定外部账号。

5.10.2 账户的存储

- web3SDK支持通过私钥字符串或者文件加载，所以账户的私钥可以存储在数据库中或者本地文件。
- 本地文件支持两种存储格式，其中PKCS12加密存储，而PEM格式明文存储。

- 开发业务时可以根据实际业务场景选择私钥的存储管理方式。

5.10.3 账户的使用

控制台加载私钥文件

控制台提供账户生成脚本get_account.sh，生成的的账户文件在accounts目录下，控制台加载的账户文件必须放置在该目录下。 控制台启动方式有如下几种：

```
./start.sh
./start.sh groupID
./start.sh groupID -pem pemName
./start.sh groupID -p12 p12Name
```

默认启动

控制台随机生成一个账户，使用控制台配置文件指定的群组号启动。

```
./start.sh
```

指定群组号启动

控制台随机生成一个账户，使用命令行指定的群组号启动。

```
./start.sh 2
```

- 注意：指定的群组在控制台配置文件中需要配置bean。

使用PEM格式私钥文件启动

- 使用指定的pem文件的账户启动，输入参数：群组号、-pem、pem文件路径

```
./start.sh 1 -pem accounts/0xebb824a1122e587b17701ed2e512d8638dfb9c88.pem
```

使用PKCS12格式私钥文件启动

- 使用指定的p12文件的账户，需要输入密码，输入参数：群组号、-p12、p12文件路径

```
./start.sh 1 -p12 accounts/0x5ef4df1b156bc9f077ee992a283c2dbb0bf045c0.p12
Enter Export Password:
```

Web3SDK加载私钥文件

如果通过账户生成脚本get_accounts.sh生成了PEM或PKCS12格式的账户私钥文件，则可以通过加载PEM或PKCS12账户私钥文件使用账户。加载私钥有两个类：P12Manager和PEMManager，其中，P12Manager用于加载PKCS12格式的私钥文件，PEMManager用于加载PEM格式的私钥文件。

- P12Manager用法举例： 在applicationContext.xml中配置PKCS12账户的私钥文件路径和密码

```
<bean id="p12" class="org.fisco.bcos.channel.client.P12Manager" init-method="load"
↪>
        <property name="password" value="123456" />
        <property name="p12File" value=
↪"classpath:0x0fc3c4bb89bd90299db4c62be0174c4966286c00.p12" />
</bean>
```

开发代码

```
//加载Bean
ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
P12Manager p12 = context.getBean(P12Manager.class);
//提供密码获取ECKeyPair，密码在生产p12账户文件时指定
ECKeyPair p12KeyPair = p12.getECKeyPair(p12.getPassword());

//以十六进制串输出私钥和公钥
System.out.println("p12 privateKey: " + p12KeyPair.getPrivateKey().toString(16));
System.out.println("p12 publicKey: " + p12KeyPair.getPublicKey().toString(16));

//生成web3sdk使用的Credentials
Credentials credentials =  GenCredential.create(p12KeyPair.getPrivateKey().
↪toString(16));
System.out.println("p12 Address: " + credentials.getAddress());
```

- PEMManager使用举例

在applicationContext.xml中配置PEM账户的私钥文件路径

```
<bean id="pem" class="org.fisco.bcos.channel.client.PEMManager" init-method="load"
↪>
        <property name="pemFile" value=
↪"classpath:0x0fc3c4bb89bd90299db4c62be0174c4966286c00.pem" />
</bean>
```

使用代码加载

```
//加载Bean
ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext-keystore-sample.xml");
PEMManager pem = context.getBean(PEMManager.class);
ECKeyPair pemKeyPair = pem.getECKeyPair();

//以十六进制串输出私钥和公钥
System.out.println("PEM privateKey: " + pemKeyPair.getPrivateKey().toString(16));
System.out.println("PEM publicKey: " + pemKeyPair.getPublicKey().toString(16));

//生成web3sdk使用的Credentials
Credentials credentialsPEM = GenCredential.create(pemKeyPair.getPrivateKey().
↪toString(16));
System.out.println("PEM Address: " + credentialsPEM.getAddress());
```

5.10.4 账户地址的计算

FISCO BCOS的账户地址由ECDSA公钥计算得来，对ECDSA公钥的16进制表示计算keccak-256sum哈希，取计算结果的后20字节的16进制表示作为账户地址，每个字节需要两个16进制数表示，所以账户地址长度为40。FISCO BCOS的账户地址与以太坊兼容。 注意keccak-256sum与SHA3**不相同**，详情参考这里。

以太坊地址生成

1. 生成ECDSA私钥

首先，我们使用OpenSSL生成椭圆曲线私钥，椭圆曲线的参数使用secp256k1。执行下面的命令，生成PEM格式的私钥并保存在ecprivkey.pem文件中。

```
openssl ecparam -name secp256k1 -genkey -noout -out ecprivkey.pem
```

执行下面的指令，查看文件内容。

```
cat ecprivkey.pem
```

可以看到类似下面的输出。

```
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEINHaCmLhw9S9+vD0IOSUd9IhHO9bBVJXTbbBeTyFNvesoAcGBSuBBAAK
oUQDQgAEjSUbQAZn4tzHnsbeahQ2J0AeMu0iNOxpdpyPo3j9Diq3qdljrv07wvjx
zOzLpUNRcJCC5hnU500MD+4+Zxc8zQ==
-----END EC PRIVATE KEY-----
```

接下来根据私钥计算公钥，执行下面的指令

```
openssl ec -in ecprivkey.pem -text -noout 2>/dev/null| sed -n '7,11p' | tr -d ": \n
↪" | awk '{print substr($0,3);}'
```

可以得到类似下面的输出

```
8d251b400667e2dcc79ec6de6a143627401e32ed2234ec69769c8fa378fd0e2ab7a9d963aefd3bc2f8f1cceccba543517
```

2. 根据公钥计算地址

本节我们根据公钥计算对应的账户地址。我们需要获取keccak-256sum工具，可以从这里下载。

```
openssl ec -in ecprivkey.pem -text -noout 2>/dev/null| sed -n '7,11p' | tr -d ": \n
↪" | awk '{print substr($0,3);}' | ./keccak-256sum -x -l | tr -d ' -' | tail -c 41
```

得到类似下面的输出，就是计算得出的账户地址。

```
dcc703c0e500b653ca82273b7bfad8045d85a470
```

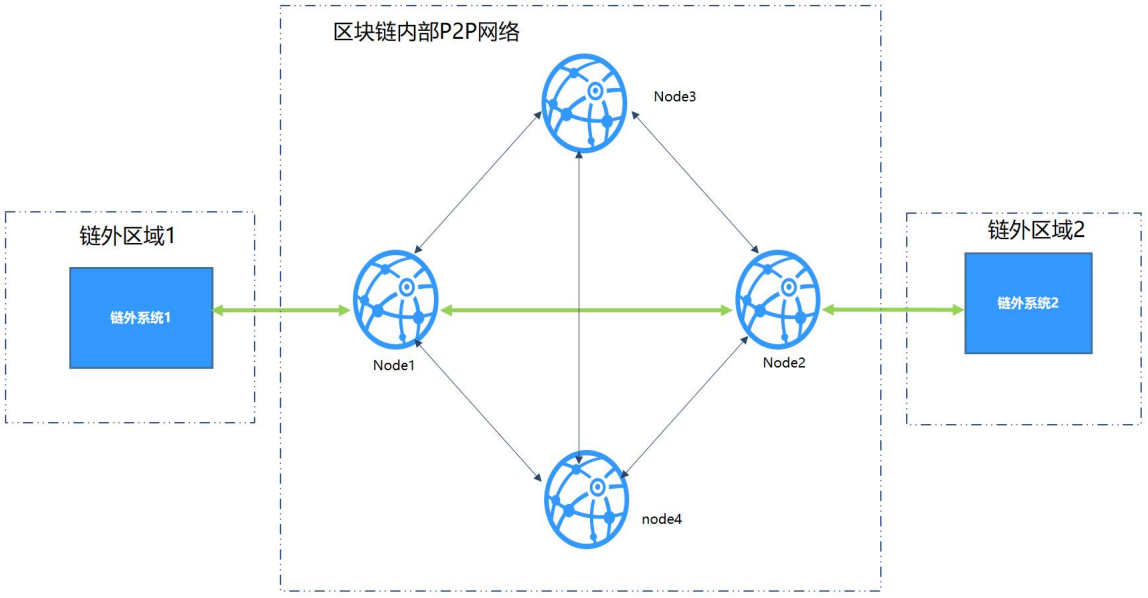
5.11 链上信使协议

5.11.1 介绍

链上信使协议AMOP（Advanced Messages Onchain Protocol）系统旨在为联盟链提供一个安全高效的消息信道，联盟链中的各个机构，只要部署了区块链节点，无论是共识节点还是观察节点，均可使用AMOP进行通讯，AMOP有如下优势：

- 实时：AMOP消息不依赖区块链交易和共识，消息在节点间实时传输，延时在毫秒级。
- 可靠：AMOP消息传输时，自动寻找区块链网络中所有可行的链路进行通讯，只要收发双方至少有一个链路可用，消息就保证可达。
- 高效：AMOP消息结构简洁、处理逻辑高效，仅需少量cpu占用，能充分利用网络带宽。
- 安全：AMOP的所有通讯链路使用SSL加密，加密算法可配置,支持身份认证机制。
- 易用：使用AMOP时，无需在SDK做任何额外配置。

5.11.2 逻辑架构



以银行典型IDC架构为例，各区域概述：

- 链外区域：机构内部的业务服务区,此区域内的业务子系统使用区块链SDK，连接到区块链节点。
- 区块链P2P网络：此区域部署各机构的区块链节点，此区域为逻辑区域，区块链节点也可部署在机构内部。

5.11.3 配置

AMOP无需任何额外配置，以下为Web3SDK的配置案例。 SDK配置（Spring Bean）：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.
↪springframework.org/schema/p"
  xmlns:tx="http://www.springframework.org/schema/tx" xmlns:aop="http://www.
↪springframework.org/schema/aop"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
          http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
          http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

<!-- AMOP消息处理线程池配置，根据实际需要配置 -->
<bean id="pool" class="org.springframework.scheduling.concurrent.
↪ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="50" />
  <property name="maxPoolSize" value="100" />
  <property name="queueCapacity" value="500" />
  <property name="keepAliveSeconds" value="60" />
  <property name="rejectedExecutionHandler">
    <bean class="java.util.concurrent.ThreadPoolExecutor.AbortPolicy" />
  </property>
</bean>
```

(continues on next page)

(续上页)

```
<!-- 群组信息配置 -->
  <bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.handler.
↪GroupChannelConnectionsConfig">
      <property name="allChannelConnections">
        <list>
          <bean id="group1"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
            <property name="groupId" value="1" />
            <property name="connectionsStr">
              <list>
                <value>127.0.0.1:20200</value> <!-- 格式: IP:端口 -->
                <value>127.0.0.1:20201</value>
              </list>
            </property>
          </bean>
        </list>
      </property>
    </bean>

  <!-- 区块链节点信息配置 -->
    <bean id="channelService" class="org.fisco.bcos.channel.client.Service"␣
↪depends-on="groupChannelConnectionsConfig">
      <property name="groupId" value="1" />
      <property name="orgID" value="fisco" />
      <property name="allChannelConnections" ref="groupChannelConnectionsConfig"></
↪property>
      <!-- 如果需要使用topic认证功能，请将下面的注释去除 -->
      <!-- <property name="topic2KeyInfo" ref="amopVerifyTopicToKeyInfo"></
↪property>-->
    </bean>

  <!--  这里配置的是topic到公私钥配置信息的映射关系，这里只配置了一个topic，可以通过新增entry的
方式来新增映射关系。-->
  <!--
    <bean class="org.fisco.bcos.channel.handler.AMOPVerifyTopicToKeyInfo" id=
↪"amopVerifyTopicToKeyInfo">
                <property name="topicToKeyInfo">
                        <map>
                                <entry key="${topicname}" value-ref=
↪"AMOPVerifyKeyInfo_${topicname}" />
                        </map>
                </property>
        </bean>
  -->

  <!--  在topic的生产者端，请将如下的注释打开，并配置公钥文件，
        每个需要身份验证的消费者都拥有不同的公私钥对，请列出所有需要身份验证的消费者的公钥文件。
  -->
  <!--
        <bean class="org.fisco.bcos.channel.handler.AMOPVerifyKeyInfo" id=
↪"AMOPVerifyKeyInfo_${topicname}">
                <property name="publicKey">
                        <list>
                                <value>classpath:$consumer_public_key_1.pem$</
↪value>
                                <value>classpath:$consumer_public_key_2.pem$</
↪value>
                        </list>
                </property>
        </bean>
        -->
```

(continues on next page)

(续上页)

```
  <!--  在topic的消费者端，请将如下的注释打开，并配置私钥文件，程序使用私钥向相应的主题生产者验证您的身份。-->
  <!--
        <bean class="org.fisco.bcos.channel.handler.AMOPVerifyKeyInfo" id="AMOPVerifyKeyInfo_${topicname}">
                <property name="privateKey" value="classpath:$consumer_private_key.pem$"></property>
        </bean>
        -->
```

5.11.4 SDK使用

AMOP的消息收发基于topic（主题）机制，服务端首先设置一个topic，客户端往该topic发送消息，服务端即可收到。

AMOP支持在同一个区块链网络中有多个topic收发消息，topic支持任意数量的服务端和客户端，当有多个服务端关注同一个topic时，该topic的消息将随机下发到其中一个可用的服务端。

服务端代码案例：

```
package org.fisco.bcos.channel.test.amop;

import org.fisco.bcos.channel.client.Service;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.HashSet;
import java.util.Set;

public class Channel2Server {
    static Logger logger = LoggerFactory.getLogger(Channel2Server.class);

    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
          System.out.println("Param: topic");
          return;
        }

        String topic = args[0];

        logger.debug("init Server");

        ApplicationContext context = new ClassPathXmlApplicationContext("classpath:applicationContext.xml");
        Service service = context.getBean(Service.class);

        // 设置topic，支持多个topic
        Set<String> topics = new HashSet<String>();
        topics.add(topic);
        service.setTopics(topics);

        // 处理消息的PushCallback类，参见Callback代码
        PushCallback cb = new PushCallback();
        service.setPushCallback(cb);

        System.out.println("3s...");
        Thread.sleep(1000);
```

(continues on next page)

(续上页)

```
        System.out.println("2s...");
        Thread.sleep(1000);
        System.out.println("1s...");
        Thread.sleep(1000);

        System.out.println("start test");
        System.out.println(
↪"======================================================================");

        // 启动服务
        service.run();
    }
}
```

服务端的PushCallback类案例:

```
package org.fisco.bcos.channel.test.amop;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import org.fisco.bcos.channel.client.ChannelPushCallback;
import org.fisco.bcos.channel.dto.ChannelPush;
import org.fisco.bcos.channel.dto.ChannelResponse;

class PushCallback extends ChannelPushCallback {
    static Logger logger = LoggerFactory.getLogger(PushCallback.class);

    // onPush方法，在收到AMOP消息时被调用
    @Override
    public void onPush(ChannelPush push) {
        DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        logger.debug("push:" + push.getContent());

        System.out.println(df.format(LocalDateTime.now()) + "server:push:" + push.
↪getContent());

        // 回包消息
        ChannelResponse response = new ChannelResponse();
        response.setContent("receive request seq:" + String.valueOf(push.
↪getMessageID()));
        response.setErrorCode(0);

        push.sendResponse(response);
    }
}
```

客户端案例:

```
package org.fisco.bcos.channel.test.amop;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

(continues on next page)

(续上页)

```
import org.fisco.bcos.channel.client.Service;
import org.fisco.bcos.channel.dto.ChannelRequest;
import org.fisco.bcos.channel.dto.ChannelResponse;

public class Channel2Client {
    static Logger logger = LoggerFactory.getLogger(Channel2Client.class);

    public static void main(String[] args) throws Exception {
        if(args.length < 2) {
            System.out.println("param: target topic total number of request");
            return;
        }

        String topic = args[0];
        Integer count = Integer.parseInt(args[1]);
        DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

        ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");

        Service service = context.getBean(Service.class);
        service.run();

        System.out.println("3s ...");
        Thread.sleep(1000);
        System.out.println("2s ...");
        Thread.sleep(1000);
        System.out.println("1s ...");
        Thread.sleep(1000);

        System.out.println("start test");
        System.out.println(
↪"======================================================================");
        for (Integer i = 0; i < count; ++i) {
          Thread.sleep(2000);  // 建立连接需要一点时间，如果立即发送消息会失败

          ChannelRequest request = new ChannelRequest();
          request.setToTopic(topic);  // 设置消息topic
          request.setMessageID(service.newSeq());  // 消息序列号，唯一标识某条消息，可
用newSeq()随机生成
          request.setTimeout(5000);  // 消息的超时时间

          request.setContent("request seq:" + request.getMessageID());  // 发送的消息
内容
          System.out.println(df.format(LocalDateTime.now()) + " request seq:" +␣
↪String.valueOf(request.getMessageID())
          + ", Content:" + request.getContent());

          ChannelResponse response = service.sendChannelMessage2(request);  // 发送
消息

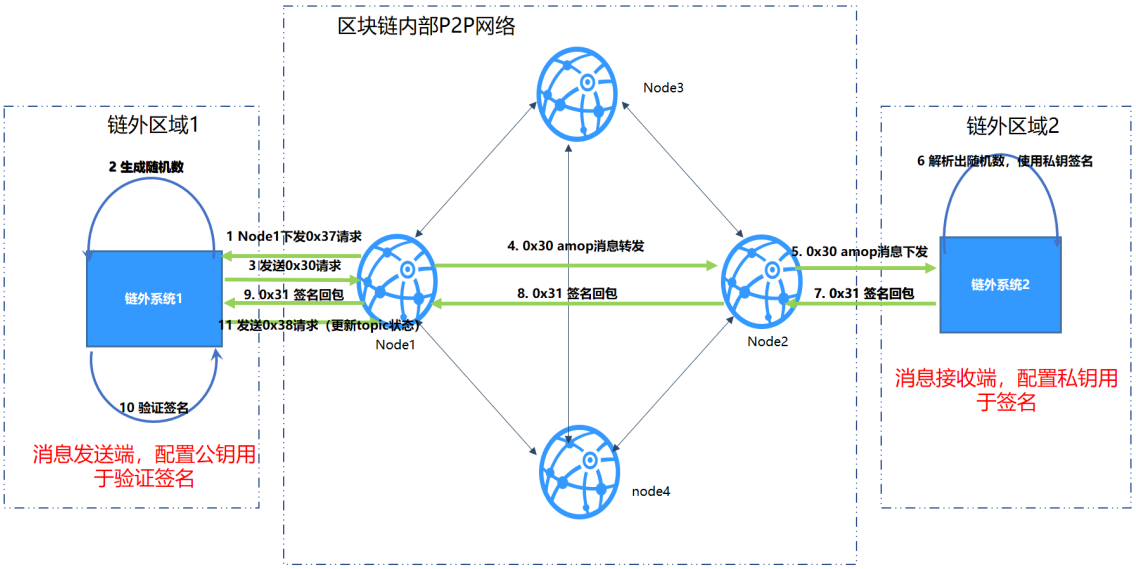
          System.out.println(df.format(LocalDateTime.now()) + "response seq:" +␣
↪String.valueOf(response.getMessageID())
                  + ", ErrorCode:" + response.getErrorCode() + ", Content:" +␣
↪response.getContent());
        }
    }
}
```

5.11.5 Topic认证功能

在普通的配置下，任何一个监听了某topic的接收者都能接受到发送者推送的消息。但在某些场景下，发送者只希望特定的接收者能接收到消息，不希望无关的接收者能任意的监听此topic。在此场景下，需要使用Topic认证功能。 认证功能是指对于特定的topic消息，允许通过认证的接收者接收消息。 2.1.0及之后的sdk和节点版本新增了topic认证功能，默认的配置没有开启认证功能，需要用到认证功能的话请参考配置文件配置配置好公私钥，公私钥的生成方式请参考生成公私钥脚本。使用过程如下：

- 1：接收者使用生成公私钥脚本生成公私钥文件，私钥保留，公钥给生产者。
- 2：参考配置案例将配置文件配好。启动接收端和发送端进行收发消息。

假定链外系统1是消息发送者，链外系统2是消息接收者。链外系统2宣称监听topic T1的消息，topic认证流程图如下：



- 1：链外系统2连接Node2,宣称监听T1,Node2将T1加入到topic列表，并将seq加1。同时每5秒同步seq到其他节点。
- 2：Node1收到seq之后，对比本地seq和同步过来的seq，不一致从Node2获取topic列表，并将topic列表更新到p2p的topic列表，对于需要认证且还没认证的topic，状态置为待认证。Node1遍历列表。针对每一个待认证的topic,进行如下操作：
  - 2.1：Node1往Node1推送消息(消息类型0x37)，请求链外系统1发起topic认证流程。
  - 2.2：链外系统1接收到消息之后，生成随机数，并使用amop消息(消息类型0x30)将消息发送出去，并监听回包。
  - 2.3：消息经过 链外系统1–>Node1–>Node2–>链外系统2的路由，链外系统2接收到消息后解析出随机数并使用私钥签名随机数。
  - 2.4：签名包(消息类型0x31)经过 链外系统2–>Node2–>Node1->链外系统1的路由，链外系统1接收到签名包之后，解析出签名并使用公钥验证签名。
  - 2.5：链外系统1验证签名后，发送消息(消息类型0x38)，请求节点更新topic状态（认证成功或者认证失败）。
- 3：如果认证成功，链外系统的一条消息到达Node1之后，Node1会将这条消息转发给Node2,Node2会将消息推送给链外系统2。

5.11.6 topic认证功能配置

默认提供的配置文件不包括认证功能，需要使用认证功能，请参考如下配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.
↪springframework.org/schema/p"
  xmlns:tx="http://www.springframework.org/schema/tx" xmlns:aop="http://www.
↪springframework.org/schema/aop"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
          http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
          http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

<!-- AMOP消息处理线程池配置，根据实际需要配置 -->
<bean id="pool" class="org.springframework.scheduling.concurrent.
↪ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="50" />
  <property name="maxPoolSize" value="100" />
  <property name="queueCapacity" value="500" />
  <property name="keepAliveSeconds" value="60" />
  <property name="rejectedExecutionHandler">
    <bean class="java.util.concurrent.ThreadPoolExecutor.AbortPolicy" />
  </property>
</bean>

<!-- 群组信息配置 -->
  <bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.handler.
↪GroupChannelConnectionsConfig">
      <property name="allChannelConnections">
        <list>
          <bean id="group1"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
            <property name="groupId" value="1" />
            <property name="connectionsStr">
              <list>
                <value>127.0.0.1:20200</value> <!-- 格式: IP:端口 -->
                <value>127.0.0.1:20201</value>
              </list>
            </property>
          </bean>
        </list>
      </property>
    </bean>

  <!-- 区块链节点信息配置 -->
    <bean id="channelService" class="org.fisco.bcos.channel.client.Service"␣
↪depends-on="groupChannelConnectionsConfig">
      <property name="groupId" value="1" />
      <property name="orgID" value="fisco" />
      <property name="allChannelConnections" ref="groupChannelConnectionsConfig"></
↪property>
        <!--  topic认证功能的配置项 -->
      <property name="topic2KeyInfo" ref="amopVerifyTopicToKeyInfo"></property>>
    </bean>

  <!--  这里配置的是topic到公私钥配置信息的映射关系，这里只配置了一个topic，可以通过新增entry的
方式来新增映射关系。-->
    <bean class="org.fisco.bcos.channel.handler.AMOPVerifyTopicToKeyInfo" id=
↪"amopVerifyTopicToKeyInfo">
                <property name="topicToKeyInfo">
                        <map>
```

(continues on next page)

(续上页)

```
                                <entry key="${topicname}" value-ref=
↪"AMOPVerifyKeyInfo_${topicname}" />
                        </map>
                </property>
        </bean>

  <!--  在topic的生产者端，请在这里配置公钥文件，每个需要身份验证的消费者 都拥有不同的公私钥对，
        请列出所有需要身份验证的消费者的公钥文件。 程序启动前请确保所有的公钥文件都存在
于web3sdk的conf目录下，
        文件名分别为$consumer_public_key_1.pem$，$consumer_public_key_2.pem$(请将这2个
变量替换为实际文件名)，如果不需要两个公钥文件，请将其中一行删除并替换变量名，可以通过新增行的方式来
增加公钥文件配置。-->
        <bean class="org.fisco.bcos.channel.handler.AMOPVerifyKeyInfo" id=
↪"AMOPVerifyKeyInfo_${topicname}">
                <property name="publicKey">
                        <list>
                                <value>classpath:$consumer_public_key_1.pem$</
↪value>
                                <value>classpath:$consumer_public_key_2.pem$</
↪value>
                        </list>
                </property>
        </bean>

  <!--  在topic的消费者端，请在这里配置私钥文件，程序使用私钥向相应的主题生产者验证您的身份。
        程序启动前请确保私钥文件存在于web3sdk的conf目录下，文件名为$consumer_private_key.
↪pem$(请将变量替换为实际文件名)。-->
        <bean class="org.fisco.bcos.channel.handler.AMOPVerifyKeyInfo" id=
↪"AMOPVerifyKeyInfo_${topicname}">
                <property name="privateKey" value="classpath:$consumer_private_key.
↪pem$"></property>
        </bean>
```

配置需要重启才可以生效，配置修改完成后，请重启基于web3sdk的应用程序。

5.11.7 测试

按上述说明配置好后，用户指定一个主题：topic，执行以下两个命令可以进行测试。

单播文本

启动AMOP服务端：

```
java -cp 'conf/:apps/*:lib/*' org.fisco.bcos.channel.test.amop.Channel2Server␣
↪[topic]
```

启动AMOP客户端：

```
java -cp 'conf/:apps/*:lib/*' org.fisco.bcos.channel.test.amop.Channel2Client␣
↪[topic] [消息条数]
```

AMOP除了支持单播文本，还支持发送二进制，多播以及身份认证机制。相应的测试命令如下：

单播二进制，多播文本，多播二进制

启动AMOP服务端：

```
java -cp 'conf/:apps/*:lib/*' org.fisco.bcos.channel.test.amop.Channel2Server␣
↪[topic]
```

启动AMOP客户端：

```
#单播二进制
java -cp 'conf/:apps/*:lib/*' org.fisco.bcos.channel.test.amop.Channel2ClientBin␣
↪[topic] [filename]
#多播文本
java -cp 'conf/:lib/*:apps/*' org.fisco.bcos.channel.test.amop.Channel2ClientMulti␣
↪[topic]  [消息条数]
#多播二进制
java -cp 'conf/:lib/*:apps/*' org.fisco.bcos.channel.test.amop.
↪Channel2ClientMultiBin [topic]  [filename]
```

带认证机制的单播文本，单播二进制，多播文本，多播二进制

在使用认证机制前，请确保参考配置文件配置好了用于认证的公私钥对。 启动AMOP服务端：

```
java -cp 'conf/:apps/*:lib/*' org.fisco.bcos.channel.test.amop.
↪Channel2ServerNeedVerify [topic]
```

启动AMOP客户端：

```
#带认证机制的单播文本
java -cp 'conf/:apps/*:lib/*' org.fisco.bcos.channel.test.amop.
↪Channel2ClientNeedVerify [topic] [消息条数]
#带认证机制的单播二进制
java -cp 'conf/:apps/*:lib/*' org.fisco.bcos.channel.test.amop.
↪Channel2ClientBinNeedVerify [topic] [filename]
#带认证机制的多播文本
java -cp 'conf/:lib/*:apps/*' org.fisco.bcos.channel.test.amop.
↪Channel2ClientMultiNeedVerify [topic]  [消息条数]
#带认证机制的多播二进制
java -cp 'conf/:lib/*:apps/*' org.fisco.bcos.channel.test.amop.
↪Channel2ClientMultiBinNeedVerify [topic]  [filename]
```

5.11.8 错误码

- 99：发送消息失败，AMOP经由所有链路的尝试后，消息未能发到服务端，建议使用发送时生成的seq，检查链路上各个节点的处理情况。
- 100：区块链节点之间经由所有链路的尝试后，消息未能发送到可以接收该消息的节点，和错误码‘99’一样，建议使用发送时生成的‘seq’，检查链路上各个节点的处理情况。
- 101：区块链节点往Sdk推送消息，经由所有链路的尝试后，未能到达Sdk端，和错误码‘99’一样，建议使用发送时生成的‘seq’，检查链路上各个节点以及Sdk的处理情况。
- 102：消息超时，建议检查服务端是否正确处理了消息，带宽是否足够。
- 103：因节点出带宽限制，SDK发到节点的AMOP请求被拒绝。

5.12 智能合约开发

FISCO BCOS平台目前支持Solidity及Precompiled两类合约形式。

- Solidity合约与以太坊相同，用Solidity语法实现，最高支持0.5.2版本。

- KVTable合约的读写接口与Table合约的CRUD接口通过在Solidity合约中支持分布式存储预编译合约，可以实现将Solidity合约中数据存储在FISCO BCOS平台AMDB的表结构中，实现合约逻辑与数据的分离。
- 预编译（Precompiled）合约使用C++开发，内置于FISCO BCOS平台，相比于Solidity合约具有更好的性能，其合约接口需要在编译时预先确定，适用于逻辑固定但需要共识的场景，例如群组配置。关于预编译合约的开发将在下一节进行介绍。

5.12.1 Solidity合约开发

- Solidity官方文档
- Remix在线IDE

5.12.2 使用KVTable合约读写接口

注解： 为实现AMDB创建的表可被多个合约共享访问，其表名是群组内全局可见且唯一的，所以无法在同一条链上的同一个群组中，创建多个表名相同的表 KVTable功能在2.3.0版本添加，2.3.0以上版本的链可以使用此功能。

KVTable合约实现键值型读写数据的方式，KVTable合约接口声明如下:

```
pragma solidity ^0.4.24;

contract KVTableFactory {
    function openTable(string) public view returns (KVTable);
    // 创建KVTable，参数分别是表名、主键列名、以逗号分割的字段名，字段可以有多个
    function createTable(string, string, string) public returns (int256);
}

//一条记录
contract Entry {
    function getInt(string) public constant returns (int256);
    function getUInt(string) public constant returns (int256);
    function getAddress(string) public constant returns (address);
    function getBytes64(string) public constant returns (bytes1[64]);
    function getBytes32(string) public constant returns (bytes32);
    function getString(string) public constant returns (string);

    function set(string, int256) public;
    function set(string, uint256) public;
    function set(string, string) public;
    function set(string, address) public;
}

//KVTable 每个键对应一条entry
contract KVTable {
    function get(string) public view returns (bool, Entry);
    function set(string, Entry) public returns (int256);
    function newEntry() public view returns (Entry);
}
```

提供一个合约案例KVTableTest.sol，代码如下:

```
pragma solidity ^0.4.24;
import "./Table.sol";

contract KVTableTest {
```

(continues on next page)

(续上页)

```
    event SetResult(int256 count);

    KVTableFactory tableFactory;
    string constant TABLE_NAME = "t_kvtest";

    constructor() public {
        //The fixed address is 0x1010 for KVTableFactory
        tableFactory = KVTableFactory(0x1010);
        tableFactory.createTable(TABLE_NAME, "id", "item_price,item_name");
    }

    //get record
    function get(string id) public view returns (bool, int256, string) {
        KVTable table = tableFactory.openTable(TABLE_NAME);
        bool ok = false;
        Entry entry;
        (ok, entry) = table.get(id);
        int256 item_price;
        string memory item_name;
        if (ok) {
            item_price = entry.getInt("item_price");
            item_name = entry.getString("item_name");
        }
        return (ok, item_price, item_name);
    }

    //set record
    function set(string id, int256 item_price, string item_name)
        public
        returns (int256)
    {
        KVTable table = tableFactory.openTable(TABLE_NAME);
        Entry entry = table.newEntry();
        // the length of entry's field value should < 16MB
        entry.set("id", id);
        entry.set("item_price", item_price);
        entry.set("item_name", item_name);
        // the first parameter length of set should <= 255B
        int256 count = table.set(id, entry);
        emit SetResult(count);
        return count;
    }
}
```

KVTableTest.sol调用了KVTable合约，实现的是创建用户表t_kvtest，并对t_kvtest表进行读写的功能。t_kvtest表结构如下，该表记录某公司仓库中物资，以唯一的物资编号作为主key，保存物资的名称和价格。

重要： 客户端需要调用转换为Java文件的合约代码，需要将KVTableTest.sol和Table.sol放入控制台的contracts/solidity目录下，通过控制台的编译脚本sol2java.sh生成SImpleTableTest.java。

5.12.3 使用Table合约CRUD接口

访问 AMDB 需要使用Table合约CRUD接口，Table合约声明于Table.sol，该接口是数据库合约，可以创建表，并对表进行增删改查操作。

注解： 为实现AMDB创建的表可被多个合约共享访问，其表名是群组内全局可见且唯一的，所以无法在

同一条链上的同一个群组中，创建多个表名相同的表。 Table的CRUD接口一个key下可以有多条记录，使用时会进行数据批量操作，包括批量写入和范围查询。对应此特性，推荐使用关系型数据库MySQL作为后端数据库。 使用KVTtable的get/set接口时，推荐使用RocksDB作为后端数据库，因RocksDB是Key-Value存储的非关系型数据库，使用KVTable接口时单key操作效率更高。

Table.sol文件代码如下:

```
pragma solidity ^0.4.24;

contract TableFactory {
    function openTable(string) public constant returns (Table);  // 打开表
    function createTable(string,string,string) public returns(int);  // 创建表
}

// 查询条件
contract Condition {
    //等于
    function EQ(string, int) public;
    function EQ(string, string) public;

    //不等于
    function NE(string, int) public;
    function NE(string, string)  public;

    //大于
    function GT(string, int) public;
    //大于或等于
    function GE(string, int) public;

    //小于
    function LT(string, int) public;
    //小于或等于
    function LE(string, int) public;

    //限制返回记录条数
    function limit(int) public;
    function limit(int, int) public;
}

// 单条数据记录
contract Entry {
    function getInt(string) public constant returns(int);
    function getAddress(string) public constant returns(address);
    function getBytes64(string) public constant returns(byte[64]);
    function getBytes32(string) public constant returns(bytes32);
    function getString(string) public constant returns(string);

    function set(string, int) public;
    function set(string, string) public;
    function set(string, address) public;
}

// 数据记录集
contract Entries {
    function get(int) public constant returns(Entry);
    function size() public constant returns(int);
}

// Table主类
contract Table {
    // 查询接口
    function select(string, Condition) public constant returns(Entries);
```

(continues on next page)

(续上页)

```
    // 插入接口
    function insert(string, Entry) public returns(int);
    // 更新接口
    function update(string, Entry, Condition) public returns(int);
    // 删除接口
    function remove(string, Condition) public returns(int);

    function newEntry() public constant returns(Entry);
    function newCondition() public constant returns(Condition);
}
```

注解:

- Table合约的insert、remove、update和select函数中key的类型为string，其长度最大支持255字符。
- Entry的get/set接口的key的类型为string，其长度最大支持255字符，value支持的类型有int256(int)、address和string，其中string的不能超过16MB。

提供一个合约案例TableTest.sol，代码如下：

```
pragma solidity ^0.4.24;

import "./Table.sol";

contract TableTest {
    event CreateResult(int count);
    event InsertResult(int count);
    event UpdateResult(int count);
    event RemoveResult(int count);

    // 创建表
    function create() public returns(int){
        TableFactory tf = TableFactory(0x1001);  // TableFactory的地址固定为0x1001
        // 创建t_test表，表的key_field为name，value_field为item_id,item_name
        // key_field表示AMDB主key value_field表示表中的列，可以有多列，以逗号分隔
        int count = tf.createTable("t_test", "name", "item_id,item_name");
        emit CreateResult(count);

        return count;
    }

    // 查询数据
    function select(string name) public constant returns(bytes32[], int[],␣
↪bytes32[]){
        TableFactory tf = TableFactory(0x1001);
        Table table = tf.openTable("t_test");

        // 条件为空表示不筛选 也可以根据需要使用条件筛选
        Condition condition = table.newCondition();

        Entries entries = table.select(name, condition);
        bytes32[] memory user_name_bytes_list = new bytes32[](uint256(entries.
↪size()));
        int[] memory item_id_list = new int[](uint256(entries.size()));
        bytes32[] memory item_name_bytes_list = new bytes32[](uint256(entries.
↪size()));

        for(int i=0; i<entries.size(); ++i) {
            Entry entry = entries.get(i);
```

(continues on next page)

(续上页)

```
            user_name_bytes_list[uint256(i)] = entry.getBytes32("name");
            item_id_list[uint256(i)] = entry.getInt("item_id");
            item_name_bytes_list[uint256(i)] = entry.getBytes32("item_name");
        }

        return (user_name_bytes_list, item_id_list, item_name_bytes_list);
    }
    // 插入数据
    function insert(string name, int item_id, string item_name) public␣
↪returns(int) {
        TableFactory tf = TableFactory(0x1001);
        Table table = tf.openTable("t_test");

        Entry entry = table.newEntry();
        entry.set("name", name);
        entry.set("item_id", item_id);
        entry.set("item_name", item_name);

        int count = table.insert(name, entry);
        emit InsertResult(count);

        return count;
    }
    // 更新数据
    function update(string name, int item_id, string item_name) public␣
↪returns(int) {
        TableFactory tf = TableFactory(0x1001);
        Table table = tf.openTable("t_test");

        Entry entry = table.newEntry();
        entry.set("item_name", item_name);

        Condition condition = table.newCondition();
        condition.EQ("name", name);
        condition.EQ("item_id", item_id);

        int count = table.update(name, entry, condition);
        emit UpdateResult(count);

        return count;
    }
    // 删除数据
    function remove(string name, int item_id) public returns(int){
        TableFactory tf = TableFactory(0x1001);
        Table table = tf.openTable("t_test");

        Condition condition = table.newCondition();
        condition.EQ("name", name);
        condition.EQ("item_id", item_id);

        int count = table.remove(name, condition);
        emit RemoveResult(count);

        return count;
    }
}
```

TableTest.sol调用了 AMDB 专用的智能合约Table.sol，实现的是创建用户表t_test，并对t_test表进行增删改查的功能。t_test表结构如下，该表记录某公司员工领用物资和编号。

重要: 客户端需要调用转换为Java文件的合约代码，需要将TableTest.sol和Table.sol放入控制台

的contracts/solidity目录下，通过控制台的编译脚本sol2java.sh生成TableTest.java。

5.12.4 预编译合约开发

一. 简介

预编译（precompiled）合约是一项以太坊原生支持的功能：在底层使用c++代码实现特定功能的合约，提供给EVM模块调用。FISCO BCOS继承并且拓展了这种特性，在此基础上发展了一套功能强大并易于拓展的框架precompiled设计原理。 本文作为一篇入门指导，旨在指引用户如何实现自己的precompiled合约,并实现precompiled合约的调用。

二. 实现预编译合约

2.1 流程

实现预编译合约的流程：

- **分配合约地址**

调用solidity合约或者预编译合约需要根据合约地址来区分，地址空间划分：

用户分配地址空间为0x5001-0xffff,用户需要为新添加的预编译合约分配一个未使用的地址，**预编译合约地址必须唯一， 不可冲突**。

FISCO BCOS中实现的precompild合约列表以及地址分配：

- **定义合约接口**

同solidity合约，设计合约时需要首先确定合约的ABI接口， precomipiled合约的ABI接口规则与solidity完全相同，solidity ABI链接。

> 定义预编译合约接口时，通常需要定义一个有相同接口的solidity合约，并且将所有的接口的函数体置空，这个合约我们称为预编译合约的**接口合约**，接口合约在调用预编译合约时需要使用。

```
pragma solidity ^0.4.24;
contract Contract_Name {
    function interface0(parameters ... ) {}
    ....
    function interfaceN(parameters ... ) {}
}
```

- **设计存储结构**

预编译合约涉及存储操作时，需要确定存储的表信息(表名与表结构,存储数据在FISCO BCOS中会统一抽象为表结构)， 存储结构。

注解： 不涉及存储操作可以省略该流程。

- **实现调用逻辑**

实现新增合约的调用逻辑，需要新实现一个c++类，该类需要继承Precompiled, 重载call函数， 在call函数中实现各个接口的调用行为。

```
// libprecompiled/Precompiled.h
class Precompiled
{
    virtual bytes call(std::shared_ptr<ExecutiveContext> _context,␣
↪bytesConstRef _param,
```

(continues on next page)

(续上页)

```
            Address const& _origin = Address()) = 0;
    };
```

call函数有三个参数：

`std::shared_ptr<ExecutiveContext> _context` ：保存交易执行的上下文

`bytesConstRef _param` ：调用合约的参数信息，本次调用对应合约接口以及接口的参数可以从_param解析获取

`Address const& _origin` ：交易发送者，用来进行权限控制

如何实现一个Precompiled类在下面的sample中会详细说明。

- **注册合约**

最后需要将合约的地址与对应的类注册到合约的执行上下文，这样通过地址调用precompiled合约时合约的执行逻辑才能被正确识别执行， 查看注册的预编译合约列表。 注册路径：

```
    file        libblockverifier/ExecutiveContextFactory.cpp
    function    initExecutiveContext
```

2.2 示例合约开发

```
// HelloWorld.sol
pragma solidity ^0.4.24;

contract HelloWorld{
    string name;

    function HelloWorld(){
       name = "Hello, World!";
    }

    function get()constant returns(string){
        return name;
    }

    function set(string n){
            name = n;
    }
}
```

上述源码为solidity编写的HelloWorld合约， 本章节会实现一个相同功能的预编译合约，通过step by step使用户对预编译合约编写有直观的认识。 示例的c++源码路径：

```
    libprecompiled/extension/HelloWorldPrecompiled.h
    libprecompiled/extension/HelloWorldPrecompiled.cpp
```

2.2.1 分配合约地址

参照地址分配空间，HelloWorld预编译合约的地址分配为：

```
0x5001
```

2.2.2 定义合约接口

需要实现HelloWorld合约的功能，接口与HelloWorld接口相同， HelloWorldPrecompiled的接口合约：

```
pragma solidity ^0.4.24;

contract HelloWorldPrecompiled {
    function get() public constant returns(string) {}
    function set(string _m) {}
}
```

2.2.3 设计存储结构

HelloWorldPrecompiled需要存储set的字符串值，所以涉及到存储操作，需要设计存储的表结构。

表名：_ext_hello_world_

表结构：

该表只存储一对键值对，key字段为hello_key，value字段为hello_value 存储对应的字符串值，可以通过set(string)接口修改，通过get()接口获取。

2.2.4 实现调用逻辑

添加HelloWorldPrecompiled类，重载call函数，实现所有接口的调用行为，call函数源码。

用户自定义的Precompiled合约需要新增一个类，在类中定义合约的调用行为，在示例中添加HelloWorldPrecompiled类，然后主要需要完成以下工作：

- 接口注册

```
// 定义类中所有的接口
const char* const HELLO_WORLD_METHOD_GET = "get()";
const char* const HELLO_WORLD_METHOD_SET = "set(string)";

// 在构造函数进行接口注册
HelloWorldPrecompiled::HelloWorldPrecompiled()
{
    // name2Selector是基类Precompiled类中成员，保存接口调用的映射关系
    name2Selector[HELLO_WORLD_METHOD_GET] = getFuncSelector(HELLO_WORLD_METHOD_
↪GET);
    name2Selector[HELLO_WORLD_METHOD_SET] = getFuncSelector(HELLO_WORLD_METHOD_
↪SET);
}
```

- 创建表

定义表名，表的字段结构

```
// 定义表名
const std::string HELLO_WORLD_TABLE_NAME = "_ext_hello_world_";
// 主键字段
const std::string HELLOWORLD_KEY_FIELD = "key";
// 其他字段字段，多个字段使用逗号分割，比如 "field0,field1,field2"
const std::string HELLOWORLD_VALUE_FIELD = "value";
```

```
// call函数中，表存在时打开，否则首先创建表
Table::Ptr table = openTable(_context, HELLO_WORLD_TABLE_NAME);
if (!table)
{
    // 表不存在，首先创建
    table = createTable(_context, HELLO_WORLD_TABLE_NAME, HELLOWORLD_KEY_FIELD,
        HELLOWORLD_VALUE_FIELD, _origin);
    if (!table)
```

(continues on next page)

(续上页)

```
    {
        // 创建表失败，返回错误码
    }
}
```

获取表的操作句柄之后，用户可以实现对表操作的具体逻辑。

- 区分调用接口

通过getParamFunc解析_param可以区分调用的接口。 **注意：合约接口一定要先在构造函数中注册**

```
uint32_t func = getParamFunc(_param);
if (func == name2Selector[HELLO_WORLD_METHOD_GET])
{
    // get() 接口调用逻辑
}
else if (func == name2Selector[HELLO_WORLD_METHOD_SET])
{
    // set(string) 接口调用逻辑
}
else
{
    // 未知接口，调用错误，返回错误码
}
```

- 参数解析与结果返回

调用合约时的参数包含在call函数的_param参数中，是按照Solidity ABI格式进行编码，使用dev::eth::ContractABI工具类可以进行参数的解析，同样接口返回时返回值也需要按照该编码格编码。Solidity ABI。

dev::eth::ContractABI类中我们需要使用abiIn abiOut两个接口，前者用户参数的序列化，后者可以从序列化的数据中解析参数

```
// 序列化ABI数据, c++类型数据序列化为evm使用的格式
// _id : 函数接口声明对应的字符串, 一般默认为""即可。
template <class... T> bytes abiIn(std::string _id, T const&... _t)
// 将序列化数据解析为c++类型数据
template <class... T> void  abiOut(bytesConstRef _data, T&... _t)
```

下面的示例代码说明接口如何使用:

```
// 对于transfer接口 :  transfer(string,string,uint256)

// 参数1
std::string str1 = "fromAccount";
// 参数2
std::string str2 = "toAccount";
// 参数3
uint256 transferAmoumt = 11111;

dev::eth::ContractABI abi;
// 序列化, abiIn第一个string参数默认""
bytes out = abi.abiIn("", str1, str2, transferAmoumt);

std::string strOut1;
std::string strOut2;
uint256 amoumt;

// 解析参数
abi.abiOut(out, strOut1, strOut2, amount);
// 解析之后
```

(continues on next page)

(续上页)

```
// strOut1 = "fromAccount";
// strOut2 = "toAccount"
// amoumt = 11111
```

最后，给出HelloWorldPrecompiled call函数的完整实现源码链接。

```
bytes HelloWorldPrecompiled::call(dev::blockverifier::ExecutiveContext::Ptr _
↪context,
    bytesConstRef _param, Address const& _origin)
{
    // 解析函数接口
    uint32_t func = getParamFunc(_param);
    //
    bytesConstRef data = getParamData(_param);
    bytes out;
    dev::eth::ContractABI abi;

    // 打开表
    Table::Ptr table = openTable(_context, HELLO_WORLD_TABLE_NAME);
    if (!table)
    {
        // 表不存在，首先创建
        table = createTable(_context, HELLO_WORLD_TABLE_NAME, HELLOWORLD_KEY_FIELD,
            HELLOWORLD_VALUE_FIELD, _origin);
        if (!table)
        {
            // 创建表失败，无权限？
            out = abi.abiIn("", CODE_NO_AUTHORIZED);
            return out;
        }
    }

    // 区分调用接口，各个接口的具体调用逻辑
    if (func == name2Selector[HELLO_WORLD_METHOD_GET])
    {  // get() 接口调用
        // 默认返回值
        std::string retValue = "Hello World!";
        auto entries = table->select(HELLOWORLD_KEY_FIELD_NAME, table->
↪newCondition());
        if (0u != entries->size())
        {
            auto entry = entries->get(0);
            retValue = entry->getField(HELLOWORLD_VALUE_FIELD);
        }
        out = abi.abiIn("", retValue);
    }
    else if (func == name2Selector[HELLO_WORLD_METHOD_SET])
    {  // set(string) 接口调用

        std::string strValue;
        abi.abiOut(data, strValue);
        auto entries = table->select(HELLOWORLD_KEY_FIELD_NAME, table->
↪newCondition());
        auto entry = table->newEntry();
        entry->setField(HELLOWORLD_KEY_FIELD, HELLOWORLD_KEY_FIELD_NAME);
        entry->setField(HELLOWORLD_VALUE_FIELD, strValue);

        int count = 0;
        if (0u != entries->size())
        {  // 值存在，更新
            count = table->update(HELLOWORLD_KEY_FIELD_NAME, entry, table->
↪newCondition(),
```

(continues on next page)

(续上页)

```
                std::make_shared<AccessOptions>(_origin));
        }
        else
        {  // 值不存在，插入
            count = table->insert(
                HELLOWORLD_KEY_FIELD_NAME, entry, std::make_shared<AccessOptions>(_
↪origin));
        }

        if (count == CODE_NO_AUTHORIZED)
        {  // 没有表操作权限
            PRECOMPILED_LOG(ERROR) << LOG_BADGE("HelloWorldPrecompiled") << LOG_
↪DESC("set")
                                   << LOG_DESC("non-authorized");
        }
        out = abi.abiIn("", u256(count));
    }
    else
    {  // 参数错误，未知的接口调用
        PRECOMPILED_LOG(ERROR) << LOG_BADGE("HelloWorldPrecompiled") << LOG_DESC("␣
↪unkown func ")
                               << LOG_KV("func", func);
        out = abi.abiIn("", u256(CODE_UNKNOW_FUNCTION_CALL));
    }

    return out;
}
```

2.2.5 注册合约并编译源码

- 注册开发的预编译合约。修改FISCO-BCOS/cmake/templates/UserPrecompiled.h.in，在下面的函数中注册HelloWorldPrecompiled合约的地址。默认已有，取消注释即可。

```
void␣
↪dev::blockverifier::ExecutiveContextFactory::registerUserPrecompiled(dev::blockverifier::Executi
↪context)
{
    // Address should in [0x5001,0xffff]
    context->setAddress2Precompiled(Address(0x5001), std::make_shared
↪<dev::precompiled::HelloWorldPrecompiled>());
}
```

- 编译源码。请参考这里，安装依赖并编译源码。

注意：实现的HelloWorldPrecompiled.cpp和头文件需要放置于FISCO-BCOS/libprecompiled/extension目录下。

- 搭建FISCO BCOS联盟链。假设当前位于FISCO-BCOS/build目录下，则使用下面的指令搭建本机4节点的链指令如下。更多选项参考这里。

```
bash ../tools/build_chain.sh -l "127.0.0.1:4" -e bin/fisco-bcos
```

三 调用

从用户角度，预编译合约与solidity合约的调用方式基本相同，唯一的区别是solidity合约在部署之后才能获取到调用的合约地址，预编译合约的地址为预分配，不用部署，可以直接使用。

3.1 使用控制台调用HelloWorld预编译合约

在控制台contracts/solidity创建HelloWorldPrecompiled.sol文件，文件内容是HelloWorld预编译合约的接口声明，如下

```
pragma solidity ^0.4.24;
contract HelloWorldPrecompiled{
    function get() public constant returns(string);
    function set(string n);
}
```

使用编译出的二进制搭建节点后，部署控制台v1.0.2以上版本，然后执行下面语句即可调用

```
[group:1]> call HelloWorldPrecompiled.sol 0x5001 get
Hello World!

[group:1]> call HelloWorldPrecompiled.sol 0x5001 set "Hello, FISCO BCOS"
0xb0542ffab97f93b8cebadb39d54825b1f709c2f185c093e8ed39ce74b5391b83

[group:1]> call HelloWorldPrecompiled.sol 0x5001 get
Hello, FISCO BCOS

[group:1]>
```

3.2 solidity调用

我们尝试在Solidity合约中创建预编译合约对象并调用其接口。在控制台contracts/solidity创建HelloWorldHelper.sol文件，文件内容如下

```
pragma solidity ^0.4.24;
import "./HelloWorldPrecompiled.sol";

contract HelloWorldHelper {
    HelloWorldPrecompiled hello;
    function HelloWorldHelper() {
        // 调用HelloWorld预编译合约
        hello = HelloWorldPrecompiled(0x5001);
    }
    function get() public constant returns(string) {
        return hello.get();
    }
    function set(string m) {
        hello.set(m);
    }
}
```

部署HelloWorldHelper合约，然后调用HelloWorldHelper合约的接口，结果如下

```
[group:1]> deploy HelloWorldHelper.sol
0x6096966a7c06006385ec0eb774f6dc783a8ee4f0

[group:1]> call HelloWorldHelper.sol 0x6096966a7c06006385ec0eb774f6dc783a8ee4f0 get
Hello, FISCO BCOS

[group:1]> call HelloWorldHelper.sol 0x6096966a7c06006385ec0eb774f6dc783a8ee4f0 set "Hello World"
0x62b0277f4b265cb40c64a05f4c5ca52307013dcbb678ab9092c4fec512b40c79

[group:1]> call HelloWorldHelper.sol 0x6096966a7c06006385ec0eb774f6dc783a8ee4f0 get
Hello World

[group:1]>
```

5.13 并行合约

FISCO BCOS提供了可并行合约开发框架，开发者按照框架规范编写的合约，能够被FISCO BCOS节点并行地执行。并行合约的优势有：

- 高吞吐：多笔独立交易同时被执行，能最大限度利用机器的CPU资源，从而拥有较高的TPS
- 可拓展：可以通过提高机器的配置来提升交易执行的性能，以支持不断扩大业务规模

接下来，我将介绍如何编写FISCO BCOS并行合约，以及如何部署和执行并行合约。

5.13.1 预备知识

并行互斥

两笔交易是否能被并行执行，依赖于这两笔交易是否存在**互斥**。互斥，是指两笔交易各自**操作合约存储变量的集合存在交集**。

例如，在转账场景中，交易是用户间的转账操作。用transfer(X, Y) 表示从X用户转到Y用户的转账接口，则互斥情况如下。

此处给出更具体的定义：

- **互斥参数**：合约**接口**中，与合约存储变量的“读/写”操作相关的参数。例如转账的接口transfer(X, Y)，X和Y都是互斥参数。
- **互斥对象**：一笔**交易**中，根据互斥参数提取出来的、具体的互斥内容。例如转账的接口transfer(X, Y), 一笔调用此接口的交易中，具体的参数是transfer(A, B)，则这笔操作的互斥对象是[A, B]；另外一笔交易，调用的参数是transfer(A, C)，则这笔操作的互斥对象是[A, C]。

判断同一时刻两笔交易是否能并行执行，就是判断两笔交易的互斥对象是否有交集。相互之间交集为空的交易可并行执行。

5.13.2 编写并行合约

FISCO BCOS提供了**可并行合约开发框架**，开发者只需按照框架的规范开发合约，定义好每个合约接口的互斥参数，即可实现能被并行执行的合约。当合约被部署后，FISCO BCOS会在执行交易前，自动解析互斥对象，在同一时刻尽可能让无依赖关系的交易并行执行。

目前，FISCO BCOS提供了solidity与预编译合约两种可并行合约开发框架。

solidity合约并行框架

编写并行的solidity合约，开发流程与开发普通的solidity合约的流程相同。在基础上，只需要将`ParallelContract`作为需要并行的合约的基类，并调用`registerParallelFunction()`，注册可以并行的接口即可。（ParallelContract.sol合约代码参考这里）

先给出完整的举例，例子中的ParallelOk合约实现了并行转账的功能

```
pragma solidity ^0.4.25;

import "./ParallelContract.sol";  // 引入ParallelContract.sol

contract ParallelOk is ParallelContract // 将ParallelContract 作为基类
{
    // 合约实现
    mapping (string => uint256) _balance;

    function transfer(string from, string to, uint256 num) public
    {
```

(continues on next page)

(续上页)

```
        // 此处为简单举例，实际生产中请用SafeMath代替直接加减
        _balance[from] -= num;
        _balance[to] += num;
    }

    function set(string name, uint256 num) public
    {
        _balance[name] = num;
    }

    function balanceOf(string name) public view returns (uint256)
    {
        return _balance[name];
    }

    // 注册可以并行的合约接口
    function enableParallel() public
    {
        // 函数定义字符串（注意",”后不能有空格）,参数的前几个是互斥参数（设计函数时互斥参数必须放在前面
        registerParallelFunction("transfer(string,string,uint256)", 2); // critical: string string
        registerParallelFunction("set(string,uint256)", 1); // critical: string
    }

    // 注销并行合约接口
    function disableParallel() public
    {
        unregisterParallelFunction("transfer(string,string,uint256)");
        unregisterParallelFunction("set(string,uint256)");
    }
}
```

具体步骤如下：

（1）将ParallelContract作为合约的基类

```
pragma solidity ^0.4.25;

import "./ParallelContract.sol"; // 引入ParallelContract.sol

contract ParallelOk is ParallelContract // 将ParallelContract 作为基类
{
   // 合约实现

   // 注册可以并行的合约接口
   function enableParallel() public;

   // 注销并行合约接口
   function disableParallel() public;
}
```

（2）编写可并行的合约接口

合约中的public函数，是合约的接口。编写可并行的合约接口，是根据一定的规则，实现一个合约中的public函数。

确定接口是否可并行

可并行的合约接口，必须满足：

- 无调用外部合约
- 无调用其它函数接口

确定互斥参数

在编写接口前，先确定接口的互斥参数，接口的互斥即是对全局变量的互斥，互斥参数的确定规则为：

- 接口访问了全局mapping，mapping的key是互斥参数
- 接口访问了全局数组，数组的下标是互斥参数
- 接口访问了简单类型的全局变量，所有简单类型的全局变量共用一个互斥参数，用不同的变量名作为互斥对象

  例如：合约里有多个简单类型的全局变量，不同接口访问了不同的全局变量。如要将不同接口并行，则需要在修改了全局变量的接口参数中定义一个互斥参数，用来调用时指明使用了哪个全局变量。在调用时，主动给互斥参数传递相应修改的全局变量的“变量名”，用以标明此笔交易的互斥对象。如：`setA(int x)`函数中修改了全局参数`globalA`，则需要将`setA`函数定义为`set(string aflag, int x)`，在调用时，传入`setA("globalA", 10)`，用变量名“`globalA`”来指明此交易的互斥对象是`globalA`。

确定参数类型和顺序

确定互斥参数后，根据规则确定参数类型和顺序，规则为：

- 接口参数仅限：**string**、**address**、**uint256**、**int256**（未来会支持更多类型）
- 互斥参数必须全部出现在接口参数中
- 所有互斥参数排列在接口参数的最前

```
mapping (string => uint256) _balance; // 全局mapping

// 互斥变量from、to排在最前，作为transfer()开头的两个参数
function transfer(string from, string to, uint256 num) public
{
    _balance[from] -= num;  // from 是全局mapping的key，是互斥参数
    _balance[to] += num; // to 是全局mapping的key，是互斥参数
}

// 互斥变量name排在最前，作为set()开头的参数
function set(string name, uint256 num) public
{
    _balance[name] = num;
}
```

（3）在框架中注册可并行的合约接口

在合约中实现 enableParallel() 函数，调用registerParallelFunction()注册可并行的合约接口。同时也需要实现disableParallel()函数，使合约具备取消并行执行的能力。

```
// 注册可以并行的合约接口
function enableParallel() public
{
    // 函数定义字符串（注意","后不能有空格）,参数的前几个是互斥参数
    registerParallelFunction("transfer(string,string,uint256)", 2); // transfer接口，前2个是互斥参数
    registerParallelFunction("set(string,uint256)", 1); // set接口，前1个是互斥参数
}

// 注销并行合约接口
function disableParallel() public
{
    unregisterParallelFunction("transfer(string,string,uint256)");
    unregisterParallelFunction("set(string,uint256)");
}
```

（4）部署/执行并行合约

用控制台或Web3SDK编译和部署合约，此处以控制台为例。

部署合约

```
[group:1]> deploy ParallelOk.sol
```

调用 enableParallel()接口，让ParallelOk能并行执行

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744␣
↪enableParallel
```

发送并行交易 set()

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744 set
↪"jimmyshi" 100000
```

发送并行交易 transfer()

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744 transfer
↪"jimmyshi" "jinny" 80000
```

查看交易执行结果 balanceOf()

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744␣
↪balanceOf "jinny"
80000
```

用SDK发送大量交易的例子，将在下文的举例中给出。

预编译并行合约框架

编写并行的预编译合约，开发流程与开发普通预编译合约的流程相同。普通的预编译合约以Precompile为基类，在这之上实现合约逻辑。基于此，Precompile的基类还为并行提供了两个虚函数，继续实现这两个函数，即可实现并行的预编译合约。

（1）将合约定义成支持并行

```
bool isParallelPrecompiled() override { return true; }
```

（2）定义并行接口和互斥参数

注意，一旦定义成支持并行，所有的接口都需要进行定义。若返回空，表示此接口无任何互斥对象。互斥参数与预编译合约的实现相关，此处涉及对FISCO BCOS存储的理解，具体的实现可直接阅读代码或询问相关有经验的程序员。

```
// 根据并行接口，从参数中取出互斥对象，返回互斥对象
std::vector<std::string> getParallelTag(bytesConstRef param) override
{
    // 获取被调用的函数名（func）和参数（data）
    uint32_t func = getParamFunc(param);
    bytesConstRef data = getParamData(param);

    std::vector<std::string> results;
    if (func == name2Selector[DAG_TRANSFER_METHOD_TRS_STR2_UINT]) // 函数是并行接口
    {
        // 接口为：userTransfer(string,string,uint256)
        // 从data中取出互斥对象
        std::string fromUser, toUser;
        dev::u256 amount;
        abi.abiOut(data, fromUser, toUser, amount);

        if (!invalidUserName(fromUser) && !invalidUserName(toUser) && (amount > 0))
        {
```

(continues on next page)

(续上页)

```
            // 将互斥对象写到results中
            results.push_back(fromUser);
            results.push_back(toUser);
        }
    }
    else if ... // 所有的接口都需要给出互斥对象，返回空表示无任何互斥对象

        return results;  //返回互斥
}
```

（3）编译，重启节点

手动编译节点的方法，参考：这里

编译之后，关闭节点，替换掉原来的节点二进制文件，再重启节点即可。

5.13.3 举例：并行转账

此处分别给出solidity合约和预编译合约的并行举例。

配置环境

该举例需要以下执行环境：

- Web3SDK客户端
- 一条FISCO BCOS链

Web3SDK用来发送并行交易，FISCO BCOS链用来执行并行交易。相关配置，可参考：

- Web3SDK的配置
- 搭链

若需要压测最大的性能，至少需要：

- 3个Web3SDK，才能产生足够多的交易
- 4个节点，且所有Web3SDK都配置了链上所有的节点信息，让交易均匀的发送到每个节点上，才能让链能接收足够多的交易

并行Solidity合约：ParallelOk

基于账户模型的转账，是一种典型的业务操作。ParallelOk合约，是账户模型的一个举例，能实现并行的转账功能。ParallelOk合约已在上文中给出。

FISCO BCOS在Web3SDK中内置了ParallelOk合约，此处给出用Web3SDK来发送大量并行交易的操作方法。

（1）用SDK部署合约、新建用户、开启合约的并行能力

```
# 参数: <groupID> add <创建的用户数量> <此创建操作请求的TPS> <生成的用户信息文件名>
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.parallelok.
↪PerformanceDT 1 add 10000 2500 user
# 在group1上创建了 10000个用户，创建操作以2500TPS发送的，生成的用户信息保存在user中
```

执行成功后，ParallelOk被部署到区块链上，创建的用户信息保存在user文件中，同时开启了ParallelOk的并行能力。

（2）批量发送并行转账交易

注意：在批量发送前，请将SDK的日志等级请调整为ERROR，才能有足够的发送能力。

```
# 参数：<groupID> transfer <总交易数量> <此转账操作请求的TPS上限> <需要的用户信息文件> <交易互斥百分比：0~10>
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.parallelok.PerformanceDT 1 transfer 100000 4000 user 2

# 向group1发送了 100000比交易，发送的TPS上限是4000，用的之前创建的user文件里的用户，发送的交易间有20%的互斥。
```

（3）验证并行正确性

并行交易执行完成后，Web3SDK会打印出执行结果。TPS 是此SDK发送的交易在节点上执行的TPS。validation 是转账交易执行结果的检查。

```
Total transactions:  100000
Total time: 34412ms
TPS: 2905.9630361501804
Avg time cost: 4027ms
Error rate: 0%
Return Error rate: 0%
Time area:
0    < time <  50ms   : 0   : 0.0%
50   < time <  100ms  : 44  : 0.044000000000000004%
100  < time <  200ms  : 2617  : 2.617%
200  < time <  400ms  : 6214  : 6.214%
400  < time <  1000ms : 14190  : 14.19%
1000 < time <  2000ms : 9224  : 9.224%
2000 < time           : 67711  : 67.711%
validation:
         user count is 10000
         verify_success count is 10000
         verify_failed count is 0
```

可以看出，本次交易执行的TPS是2905。执行结果校验后，无任何错误(verify_failed count is 0)。

（4）计算总TPS

单个Web3SDK无法发送足够多的交易以达到节点并行执行能力的上限。需要多个Web3SDK同时发送交易。在多个Web3SDK同时发送交易后，单纯的将结果中的TPS加和得到的TPS不够准确，需要直接从节点处获取TPS。

用脚本从日志文件中计算TPS

```
cd tools
sh get_tps.sh log/log_2019031821.00.log 21:26:24 21:26:59 # 参数：<日志文件> <计算开始时间> <计算结束时间>
```

得到TPS（2 SDK、4节点，8核，16G内存）

```
statistic_end = 21:26:58.631195
statistic_start = 21:26:24.051715
total transactions = 193332, execute_time = 34580ms, tps = 5590 (tx/s)
```

并行预编译合约：DagTransferPrecompiled

与ParallelOk合约的功能一样，FISCO BCOS内置了一个并行预编译合约的例子（DagTransferPrecompiled），实现了简单的基于账户模型的转账功能。该合约能够管理多个用户的存款，并提供一个支持并行的transfer接口，实现对用户间转账操作的并行处理。

注意：DagTransferPrecompiled为并行交易的举例，功能较为简单，请勿用于线上业务。

（1）生成用户

用Web3SDK发送创建用户的操作，创建的用户信息保存在user文件中。命令参数与parallelOk相同，不同的仅仅是命令所调用的对象是precompile。

```
# 参数: <groupID> add <创建的用户数量> <此创建操作请求的TPS> <生成的用户信息文件名>
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.precompile.
↪PerformanceDT 1 add 10000 2500 user
# 在group1上创建了 10000个用户，创建操作以2500TPS发送的，生成的用户信息保存在user中
```

（2）批量发送并行转账交易

用Web3SDK发送并行转账交易

注意：在批量发送前，请将SDK的日志等级请调整为ERROR，才能有足够的发送能力。

```
# 参数: <groupID> transfer <总交易数量> <此转账操作请求的TPS上限> <需要的用户信息文件> <交易
互斥百分比: 0~10>
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.precompile.
↪PerformanceDT 1 transfer 100000 4000 user 2
# 向group1发送了 100000比交易，发送的TPS上限是4000，用的之前创建的user文件里的用户，发送的交易
间有20%的互斥。
```

（3）验证并行正确性

并行交易执行完成后，Web3SDK会打印出执行结果。TPS 是此SDK发送的交易在节点上执行的TPS。validation 是转账交易执行结果的检查。

```
Total transactions:  80000
Total time: 25451ms
TPS: 3143.2949589407094
Avg time cost: 5203ms
Error rate: 0%
Return Error rate: 0%
Time area:
0    < time <  50ms   : 0   : 0.0%
50   < time <  100ms  : 0   : 0.0%
100  < time <  200ms  : 0   : 0.0%
200  < time <  400ms  : 0   : 0.0%
400  < time <  1000ms : 403   : 0.50375%
1000 < time <  2000ms : 5274   : 6.592499999999999%
2000 < time           : 74323   : 92.90375%
validation:
         user count is 10000
         verify_success count is 10000
         verify_failed count is 0
```

从图中可看出，本次交易执行的TPS是3143。执行结果校验后，无任何错误(verify_failed count is 0)。

（4）计算总TPS

单个Web3SDK无法发送足够多的交易以达到节点并行执行能力的上限。需要多个Web3SDK同时发送交易。在多个Web3SDK同时发送交易后，单纯的将结果中的TPS加和得到的TPS不够准确，需要直接从节点处获取TPS。

用脚本从日志文件中计算TPS

```
cd tools
sh get_tps.sh log/log_2019031311.17.log 11:25 11:30 # 参数: <日志文件> <计算开始时间>
↪<计算结束时间>
```

得到TPS（3 SDK、4节点，8核，16G内存）

```
statistic_end = 11:29:59.587145
statistic_start = 11:25:00.642866
total transactions = 3340000, execute_time = 298945ms, tps = 11172 (tx/s)
```

结果说明

本文举例中的性能结果，是在3SDK、4节点、8核、16G内存、1G网络下测得。每个SDK和节点都部署在不同的VPS中，硬盘为云硬盘。实际TPS会根据你的硬件配置、操作系统和网络带宽有所变化。

5.14 组员管理

FISCO BCOS引入了游离节点、观察者节点和共识节点，这三种节点类型可通过控制台相互转换。

- 组员
  - 共识节点：参与共识的节点，拥有群组的所有数据（搭链时默认都生成共识节点）。
  - 观察者节点：不参与共识，但能实时同步链上数据的节点。
- 非组员
  - 游离节点：已启动，待等待加入群组的节点。处在一种暂时的节点状态，不能获取链上的数据。

5.14.1 操作命令

控制台提供了 **addSealer** 、**addObserver** 和 **removeNode** 三类命令将指定节点转换为共识节点、观察者节点和游离节点，并可使用 **getSealerList**、**getObserverList** 和 **getNodeIDList** 查看当前组的共识节点列表、观察者节点列表和组内所有节点列表。

- addSealer：根据节点NodeID设置对应节点为共识节点；
- addObserver：根据节点NodeID设置对应节点为观察节点；
- removeNode：根据节点NodeID设置对应节点为游离节点；
- getSealerList：查看群组中共识节点列表；
- getObserverList：查看群组中观察节点列表；
- getNodeIDList：查看节点已连接的所有其他节点的NodeID。

例： 将指定节点分别转换成共识节点、观察者节点、游离节点，主要操作命令如下：

重要： 节点准入操作前，请确保：

- 操作节点Node ID存在，节点Node ID可在节点目录下执行 cat conf/node.nodeid获取
- 节点加入的区块链所有节点共识正常：正常共识的节点会输出+++日志

```
# 获取节点Node ID（设节点目录为~/nodes/192.168.0.1/node0/）
$ cat ~/fisco/nodes/192.168.0.1/node0/conf/node.nodeid
7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b9695

# 连接控制台(设控制台位于~/fisco/console目录)
$ cd ~/fisco/console

$ bash start.sh

# 将指定节点转换为共识节点
[group:1]> addSealer␣
↪7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b96
# 查询共识节点列表
[group:1]> getSealerList
[
```

(continues on next page)

(续上页)

```
        7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff
]

# 将指定节点转换为观察者节点
[group:1]> addObserver␣
↪7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b96

# 查询观察者节点列表
[group:1]> getObserverList
[
        7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff
]

# 将指定节点转换为游离节点
[group:1]> removeNode␣
↪7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b96

# 查询节点列表
[group:1]> getNodeIDList
[
        7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff
]
[group:1]> getSealerList
[]
[group:1]> getObserverList
[]
```

5.14.2 操作案例

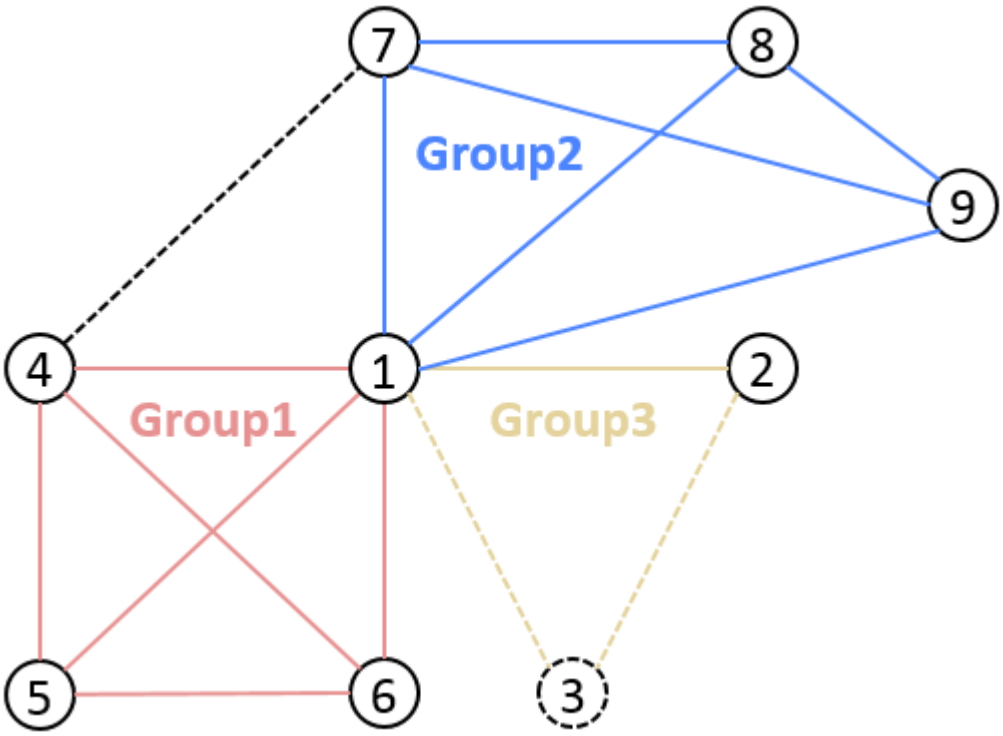
下面结合具体操作案例详细阐述群组扩容操作及节点退网操作。扩容操作分两个阶段， 分别为**将节点加入网络**、**将节点加入群组**。退网操作也分为两个阶段，为**将节点退出群组**、**将节点退出网络**。

操作方式

- 修改节点配置：节点修改自身配置后重启生效，涉及的操作项目包括**网络的加入/退出**、**CA黑名单的列入/移除**。
- 交易共识上链：节点发送上链交易修改需群组共识的配置项，涉及的操作项目包括**节点类型的修改**。目前提供的发送交易途径为控制台、SDK提供的precompiled service接口。
- RPC查询：使用curl命令查询链上信息，涉及的操作项目包括**群组节点的查询**。

操作步骤

本节将以下图为例对上述扩容操作及退网操作进行描述。虚线表示节点间能进行网络通信，实线表示节点间在可通信的基础上具备群组关系，不同颜色区分不同的群组关系。下图有一个网络，包含三个群组，其中群组Group3有三个节点。Group3是否与其他群组存在交集节点，不影响以下操作过程的通用性。



节点1的目录名为node0，IP端口为127.0.0.1:30400，nodeID前四个字节为b231b309...

节点2的目录名为node1，IP端口为127.0.0.1:30401，nodeID前四个字节为aab37e73...

节点3的目录名为node2，IP端口为127.0.0.1:30402，nodeID前四个字节为d6b01a96...

A节点加入网络

场景描述：

节点3原先不在网络中，现在加入网络。

操作顺序：

1 . 进入nodes同级目录，在该目录下拉取并执行gen_node_cert.sh生成节点目录，目录名以node2为例，node2内有conf/目录；

```
# 获取脚本
$ curl -LO https://raw.githubusercontent.com/FISCO-BCOS/FISCO-BCOS/master/tools/
↪gen_node_cert.sh && chmod u+x gen_node_cert.sh
# 执行，-c为生成节点所提供的ca路径，agency为机构名，-o为将生成的节点目录名
$ ./gen_node_cert.sh -c nodes/cert/agency -o node2
```

注解：

- 如果因为网络问题导致长时间无法下载，请尝试 *curl -LO https://gitee.com/FISCO-BCOS/FISCO-BCOS/raw/master/tools/gen_node_cert.sh*

2 . 拷贝node2到nodes/127.0.0.1/下，与其他节点目录（node0、node1）同级；

```
$ cp -r ./node2/ nodes/127.0.0.1/
```

3 . 进入nodes/127.0.0.1/，拷贝node0/config.ini、node0/start.sh和node0/stop.sh到node2目录；

```
$ cd nodes/127.0.0.1/
$ cp node0/config.ini node0/start.sh node0/stop.sh node2/
```

4 . 修改node2/config.ini。对于[rpc]模块，修改channel_listen_port和jsonrpc_listen_port；对于[p2p]模块，修改listen_port并在node.中增加自身节点信息；

```
$ vim node2/config.ini
[rpc]
    ;rpc listen ip
    listen_ip=127.0.0.1
    ;channelserver listen port
    channel_listen_port=20302
    ;jsonrpc listen port
    jsonrpc_listen_port=8647
[p2p]
    ;p2p listen ip
    listen_ip=0.0.0.0
    ;p2p listen port
    listen_port=30402
    ;nodes to connect
    node.0=127.0.0.1:30400
    node.1=127.0.0.1:30401
    node.2=127.0.0.1:30402
```

5 . 节点3拷贝节点1的node1/conf/group.3.genesis（内含**群组节点初始列表**）和node1/conf/group.3.ini到node2/conf目录下，不需改动；

```
$ cp node1/conf/group.3.genesis node2/conf/
$ cp node1/conf/group.3.ini node2/conf/
```

6 . 执行node2/start.sh启动节点3；

```
$ ./node2/start.sh
```

7 . 确认节点3与节点1和节点2的连接已经建立，加入网络操作完成。

```
# 在打开DEBUG级别日志前提下，查看自身节点（node2）连接的节点数及所连接的节点信息（nodeID）
# 以下日志表明节点node2与两个节点（节点的nodeID前4个字节为b231b309、aab37e73）建立了连接
$ tail -f node2/log/log*  | grep P2P
debug|2019-02-21 10:30:18.694258| [P2P][Service] heartBeat ignore connected,
 ↪endpoint=127.0.0.1:30400,nodeID=b231b309...
debug|2019-02-21 10:30:18.694277| [P2P][Service] heartBeat ignore connected,
 ↪endpoint=127.0.0.1:30401,nodeID=aab37e73...
info|2019-02-21 10:30:18.694294| [P2P][Service] heartBeat connected count,size=2
```

注解:

- 若启用了白名单，需确保所有节点的config.ini中的白名单都已配置了所有的节点，并正确的将白名单配置刷新入节点中。参考《CA黑白名单》；
- 从节点1拷贝过来的config.ini的其余配置可保持不变；
- 理论上，节点1和2不需修改自身的P2P节点连接列表，即可完成扩容节点3的操作；
- 步骤5中所选择的群组建议为节点3后续需加入的群组；
- 建议用户在节点1和2的config.ini的P2P节点连接列表中加入节点3的信息并重启节点1和2，保持全网节点的全互联状态。

A节点退出网络

场景描述：

节点3已在网络中，与节点1和节点2通信，现在退出网络。

操作顺序：

1 . 对于节点3，将自身的**P2P节点连接列表**内容清空，重启节点3；

```
# 在node2目录下执行
$ ./stop.sh
$ ./start.sh
nohup: appending output to 'nohup.out'
```

2 . 对于节点1和2，将节点3从自身的**P2P节点连接列表**中移除（如有），重启节点1和2；

3 . 确认节点3与节点1（和2）的原有连接已经断开，退出网络操作完成。

注解：

- **节点3需先退出群组再退出网络，退出顺序由用户保证，系统不再作校验**；
- 网络连接由节点主动发起，如缺少第2步，节点3仍可感知节点1和节点2发起的P2P连接请求，并建立连接，可使用CA黑名单避免这种情况。
- 若启用了白名单，需将退出节点的从所有节点的config.ini的白名单配置中删除，并正确的将新的白名单配置刷入节点中。参考《CA黑白名单》。

A节点加入群组

场景描述：

群组Group3原有节点1和节点2，两节点轮流出块，现在将节点3加入群组。

操作顺序：

1. 节点3加入网络；
2. 使用控制台addSealer根据节点3的nodeID设置节点3为**共识节点**；
3. 使用控制台getSealerList查询group3的共识节点中是否包含节点3的nodeID，如存在，加入群组操作完成。

注解：

- 节点3的NodeID可以使用‘cat nodes/127.0.0.1/node2/conf/node.nodeid‘获取；
- 节点3首次启动会将配置的群组节点初始列表内容写入群组节点系统表，区块同步结束后，**群组各节点的群组节点系统表均一致**；
- **节点3需先完成网络准入后，再执行加入群组的操作，系统将校验操作顺序**；
- **节点3的群组固定配置文件需与节点1和2的一致**。

A节点退出群组

场景描述：

群组Group3原有节点1、节点2和节点3，三节点轮流出块，现在将节点3退出群组。

操作顺序：

1. 使用控制台removeNode根据节点3的NodeID设置节点3为**游离节点**；
2. 使用控制台getSealerList查询group3的共识节点中是否包含节点3的nodeID，如已消失，退出群组操作完成。

补充说明：

注解：

- 节点3可以共识节点或观察节点的身份执行退出操作。

5.15 权限控制

5.15.1 基于角色的权限控制

本节描述角色权限控制的操作，使用前请先阅读设计文档角色权限控制设计文档。2.5.0版本开始，提供一种基于角色的权限控制模型，原来的链管理员相当于当前的治理委员会委员角色，拥有链治理相关的操作权限。用户不需要去具体关注底层系统表对应的权限，只需要关注角色的权限即可。

权限与角色

1. 链治理委员会委员简称为委员
2. 权限使用白名单机制，默认不检查，当存在至少一个角色的账号时，角色对应的权限检查生效
3. 委员可以冻结解冻任意合约，同时合约的部署账号也可以冻结解冻合约
4. 委员可以冻结解冻账号，被冻结的账号无法发送交易

操作内容

- 委员新增、撤销与查询
- 委员权重修改
- 委员投票生效阈值修改
- 委员权限
- 运维新增、撤销与查询

环境配置

配置并启动FISCO BCOS 2.0区块链节点和控制台，请参考安装文档。

权限控制示例账户

控制台提供账户生成脚本`get_account.sh`，生成的账户文件在`accounts`目录下。控制台可以指定账户启动，具体用法参考控制台手册。因此，通过控制台可以指定账户，体验权限控制功能。在控制台根目录下通过`get_account.sh`脚本生成三个PEM格式的账户文件如下：

```
# 账户1
0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a.pem
# 账户2
0x85961172229aec21694d742a5bd577bedffcfec3.pem
# 账户3
0x0b6f526d797425540ea70becd7adac7d50f4a7c0.pem
```

现在可以打开三个连接Linux的终端，分别以三个账户登录控制台。

指定账户1登录控制台：

```
./start.sh 1 -pem accounts/0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a.pem
```

指定账户2登录控制台：

```
./start.sh 1 -pem accounts/0x85961172229aec21694d742a5bd577bedffcfec3.pem
```

指定账户3登录控制台：

```
./start.sh 1 -pem accounts/0x0b6f526d797425540ea70becd7adac7d50f4a7c0.pem
```

委员新增、撤销与查询

我们添加账户1、账户2为委员，账户3为普通用户。链初始状态，没有任何权限账户记录。测试账户3不能操作委员权限，而账户1、2可以。

委员的权限包括治理投票、增删节点、冻结解冻合约、冻结解冻账号、修改链配置和增删运维账号。

添加账户1为委员

添加账户1为委员成功。

```
[group:1]> grantCommitteeMember 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a
{
    "code":0,
    "msg":"success"
}
[group:1]> listCommitteeMembers
------------------------------------------------------------------------------------------
↪----------
|                    address                     |                 enable_num
↪          |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a  |                      1
↪          |
------------------------------------------------------------------------------------------
↪----------
```

使用账户1添加账户2为委员

增加委员需要链治理委员会投票，有效票大于阈值才可以生效。此处由于只有账号1是委员，所以账号1投票即可生效。

```
[group:1]> grantCommitteeMember 0x85961172229aec21694d742a5bd577bedffcfec3
{
    "code":0,
    "msg":"success"
}

[group:1]> listCommitteeMembers
------------------------------------------------------------------------------------------
↪----------
|                    address                     |                 enable_num
↪          |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a  |                      1
↪          |
```

(continues on next page)

(续上页)

```
| 0x85961172229aec21694d742a5bd577bedffcfec3  |                    2
↪          |
--------------------------------------------------------------------------------
↪----------
```

注意这里使用账户1对应的控制台操作。

验证账号3无权限执行委员操作

在账号3的控制台种操作，此处以链配置操作为例。

```
[group:1]> setSystemConfigByKey tx_count_limit 100
{
    "code":-50000,
    "msg":"permission denied"
}
```

撤销账号2的委员权限

此时系统中有两个委员，默认投票生效阈值50%，所以需要两个委员都投票撤销账号2的委员权限，有效票/总票数=2/2=1>0.5才满足条件。

1. 账号1投票撤销账号2的委员权限

```
[group:1]> revokeCommitteeMember 0x85961172229aec21694d742a5bd577bedffcfec3
{
    "code":0,
    "msg":"success"
}

[group:1]> listCommitteeMembers
--------------------------------------------------------------------------------
↪----------
|                   address                    |                enable_num
↪          |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a  |                    1
↪          |
| 0x85961172229aec21694d742a5bd577bedffcfec3  |                    2
↪          |
--------------------------------------------------------------------------------
↪----------
```

1. 账号2投票撤销账号2的委员权限

```
[group:1]> revokeCommitteeMember 0x85961172229aec21694d742a5bd577bedffcfec3
{
    "code":0,
    "msg":"success"
}

[group:1]> listCommitteeMembers
--------------------------------------------------------------------------------
↪----------
|                   address                    |                enable_num
↪          |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a  |                    1
↪          |
--------------------------------------------------------------------------------
↪----------
```

(continues on next page)

(续上页)

委员权重修改

先添加账户1、账户3为委员。然后更新委员1的票数为2。

1. 使用账号1的控制台添加账号3为委员

```
[group:1]> grantCommitteeMember 0x0b6f526d797425540ea70becd7adac7d50f4a7c0
{
    "code":0,
    "msg":"success"
}

[group:1]> listCommitteeMembers
------------------------------------------------------------------------------------------
↪----------
|                    address                     |                 enable_num
↪          |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a  |                      1
↪          |
| 0x0b6f526d797425540ea70becd7adac7d50f4a7c0  |                      9
↪          |
------------------------------------------------------------------------------------------
↪----------
```

1. 使用账号1的控制台投票更新账号1的票数为2

```
[group:1]> updateCommitteeMemberWeight 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a 2
{
    "code":0,
    "msg":"success"
}

[group:1]> queryCommitteeMemberWeight 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a
Account: 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a Weight: 1
```

1. 使用账号3的控制台投票更新账号1的票数为2

```
[group:1]> updateCommitteeMemberWeight 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a 2
{
    "code":0,
    "msg":"success"
}

[group:1]> queryCommitteeMemberWeight 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a
Account: 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a Weight: 2
```

委员投票生效阈值修改

账户1和账户3为委员，账号1有2票，账号3有1票，使用账号1添加账号2为委员，由于2/3>0.5所以直接生效。使用账号1和账号2，更新生效阈值为75%。

1. 账户1添加账户2为委员

```
[group:1]> grantCommitteeMember 0x85961172229aec21694d742a5bd577bedffcfec3
{
    "code":0,
    "msg":"success"
```

(continues on next page)

(续上页)

```
}

[group:1]> listCommitteeMembers
-------------------------------------------------------------------------------
↪----------
|                  address                   |                enable_num        ␣
↪          |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a  |                     1            ␣
↪          |
| 0x0b6f526d797425540ea70becd7adac7d50f4a7c0  |                     9            ␣
↪          |
| 0x85961172229aec21694d742a5bd577bedffcfec3  |                    12            ␣
↪          |
-------------------------------------------------------------------------------
↪----------
```

1. 使用账户1控制台投票更新生效阈值为75%

```
[group:1]> updateThreshold 75
{
    "code":0,
    "msg":"success"
}

[group:1]> queryThreshold
Effective threshold : 50%
```

1. 使用账户2控制台投票更新生效阈值为75%

```
[group:1]> updateThreshold 75
{
    "code":0,
    "msg":"success"
}

[group:1]> queryThreshold
Effective threshold : 75%
```

运维新增、撤销与查询

委员可以添加运维，运维角色的权限包括部署合约、创建表、冻结解冻所部署的合约、使用CNS服务。

基于职责权限分离的设计，委员角色不能兼有运维的权限，生成一个新的账号40x283f5b859e34f7fd2cf136c07579dcc72423b1b2.pem。

1. 添加账号4为运维角色

```
[group:1]> grantOperator 0x283f5b859e34f7fd2cf136c07579dcc72423b1b2
{
    "code":0,
    "msg":"success"
}

[group:1]> listOperators
-------------------------------------------------------------------------------
↪----------
|                  address                   |                enable_num        ␣
↪          |
| 0x283f5b859e34f7fd2cf136c07579dcc72423b1b2  |                    15            ␣
↪          |
```

(continues on next page)

(续上页)

```
--------------------------------------------------------------------------------
↪----------
```

1. 使用运维账号部署HelloWorld

```
[group:1]> deploy HelloWorld
contract address: 0xac1e28ad93e0b7f9108fa1167a8a06585f663726
```

1. 使用账号1部署HelloWorld失败

```
[group:1]> deploy HelloWorld
permission denied
```

1. 使用账号1控制台撤销账号4的运维权限

```
[group:1]> revokeOperator 0x283f5b859e34f7fd2cf136c07579dcc72423b1b2
{
    "code":0,
    "msg":"success"
}

[group:1]> listOperators
Empty set.
```

5.15.2 基于表的权限控制

本文档描述权限控制的实践操作，有关权限控制的详细设计请参考权限控制设计文档。

重要： 推荐管理员机制：由于系统默认无权限设置记录，因此任何账户均可以使用权限设置功能。例如当账户1设置账户1有权限部署合约，但是账户2也可以设置账户2有权限部署合约。那么账户1的设置将失去控制的意义，因为其他账户可以自由添加权限。因此，搭建联盟链之前，推荐确定权限使用规则。可以使用grantPermissionManager指令设置链管理员账户，即指定特定账户可以使用权限分配功能，非链管理员账户无权限分配功能。

操作内容

本文档分别对以下功能进行权限控制的操作介绍:

- 授权账户为链管理员
- 授权账户为系统管理员
- 授权部署合约和创建用户表
- 授权利用CNS部署合约
- 授权管理节点
- 授权修改系统参数
- 授权账户写用户表

环境配置

配置并启动FISCO BCOS 2.0区块链节点和控制台，请参考安装文档。

权限控制工具

FISCO BCOS提供控制台命令使用权限功能（针对开发者，可以调用SDK API的PermissionService接口使用权限功能），其中涉及的权限控制命令如下:

权限控制示例账户

控制台提供账户生成脚本get_account.sh，生成的账户文件在accounts目录下。控制台可以指定账户启动，具体用法参考控制台手册。因此，通过控制台可以指定账户，体验权限控制功能。为了账户安全起见，我们可以在控制台根目录下通过get_account.sh脚本生成三个PKCS12格式的账户文件，生成过程中输入的密码需要牢记。生成的三个PKCS12格式的账户文件如下:

```
# 账户1
0x2c7f31d22974d5b1b2d6d5c359e81e91ee656252.p12
# 账户2
0x7fc8335fec9da5f84e60236029bb4a64a469a021.p12
# 账户3
0xd86572ad4c92d4598852e2f34720a865dd4fc3dd.p12
```

现在可以打开三个连接Linux的终端，分别以三个账户登录控制台。

指定账户1登录控制台:

```
$ ./start.sh 1 -p12 accounts/0x2c7f31d22974d5b1b2d6d5c359e81e91ee656252.p12
```

指定账户2登录控制台:

```
$ ./start.sh 1 -p12 accounts/0x7fc8335fec9da5f84e60236029bb4a64a469a021.p12
```

指定账户3登录控制台:

```
$ ./start.sh 1 -p12 accounts/0xd86572ad4c92d4598852e2f34720a865dd4fc3dd.p12
```

授权账户为链管理员

提供的三个账户设为三种角色，设定账户1为链管理员账户，账户2为系统管理员账户，账户3为普通账户。链管理员账户拥有权限管理的权限，即能分配权限。系统管理员账户可以管理系统相关功能的权限，每一种系统功能权限都需要单独分配，具体包括部署合约和创建用户表的权限、管理节点的权限、利用CNS部署合约的权限以及修改系统参数的权限。链管理员账户可以授权其他账户为链管理员账户或系统管理员账户，也可以授权指定账号可以写指定的用户表，即普通账户。

链初始状态，没有任何权限账户记录。现在，可以进入账户1的控制台，设置账户1成为链管理员账户，则其他账户为非链管理员账户。

```
[group:1]> grantPermissionManager 0x2c7f31d22974d5b1b2d6d5c359e81e91ee656252
{
    "code":0,
    "msg":"success"
}

[group:1]> listPermissionManager
--------------------------------------------------------------------------------
↪----------
|                    address                    |                 enable_num        ␣
↪          |
| 0x2c7f31d22974d5b1b2d6d5c359e81e91ee656252  |                      1            ␣
↪          |
--------------------------------------------------------------------------------
↪----------
```

设置账户1为链管理员成功。

授权账户为系统管理员

授权部署合约和创建用户表

通过账户1授权账户2为系统管理员账户，首先授权账户2可以部署合约和创建用户表。

```
[group:1]> grantDeployAndCreateManager 0x7fc8335fec9da5f84e60236029bb4a64a469a021
{
    "code":0,
    "msg":"success"
}

[group:1]> listDeployAndCreateManager
-------------------------------------------------------------------------------------------
↪----------
|                   address                    |                 enable_num
↪          |
| 0x7fc8335fec9da5f84e60236029bb4a64a469a021   |                      2
↪          |
-------------------------------------------------------------------------------------------
↪----------
```

登录账户2的控制台，部署控制台提供的TableTest合约。TableTest.sol合约代码参考这里。其提供创建用户表t_test和相关增删改查的方法。

```
[group:1]> deploy TableTest.sol
contract address:0xfe649f510e0ca41f716e7935caee74db993e9de8
```

调用TableTest的create接口创建用户表t_test。

```
[group:1]> call TableTest.sol 0xfe649f510e0ca41f716e7935caee74db993e9de8 create
transaction hash:0x67ef80cf04d24c488d5f25cc3dc7681035defc82d07ad983fbac820d7db31b5b
-------------------------------------------------------------------------------------------
↪----------
Event logs
-------------------------------------------------------------------------------------------
↪----------
createResult index: 0
count = 0
-------------------------------------------------------------------------------------------
↪----------
```

用户表t_test创建成功。

登录账户3的控制台，部署TableTest合约。

```
[group:1]> deploy TableTest.sol
{
    "code":-50000,
    "msg":"permission denied"
}
```

账户3没有部署合约的权限，部署合约失败。

- **注意：** 其中部署合约和创建用户表是“二合一”的控制项，在使用Table合约（CRUD接口合约）时，我们建议部署合约的时候一起把合约里用到的表创建了（在合约的构造函数中创建表），否则接下来读写表的交易可能会遇到“缺表”错误。如果业务流程需要动态创建表，动态建表的权限也应该只分配给少数账户，否则链上可能会出现各种废表。

授权利用CNS部署合约

控制台提供3个涉及CNS的命令，如下所示：

注意： 其中deployByCNS命令受权限可以控制，**且同时需要部署合约和使用CNS的权限**，callByCNS和queryCNS命令不受权限控制。

登录账户1的控制台，授权账户2拥有利用CNS部署合约的权限。

```
[group:1]> grantCNSManager 0x7fc8335fec9da5f84e60236029bb4a64a469a021
{
    "code":0,
    "msg":"success"
}

[group:1]> listCNSManager
--------------------------------------------------------------------------------------------
----------
|                   address                    |                 enable_num
|
| 0x7fc8335fec9da5f84e60236029bb4a64a469a021   |                     13
|
--------------------------------------------------------------------------------------------
----------
```

登录账户2的控制台，利用CNS部署合约。

```
[group:1]> deployByCNS TableTest.sol 1.0
contract address:0x24f902ff362a01335db94b693edc769ba6226ff7

[group:1]> queryCNS TableTest.sol
--------------------------------------------------------------------------------------------
----------
|                   version                    |                   address
|
|                     1.0                      |
0x24f902ff362a01335db94b693edc769ba6226ff7   |
--------------------------------------------------------------------------------------------
----------
```

登录账户3的控制台，利用CNS部署合约。

```
[group:1]> deployByCNS TableTest.sol 2.0
{
    "code":-50000,
    "msg":"permission denied"
}

[group:1]> queryCNS TableTest.sol
--------------------------------------------------------------------------------------------
----------
|                   version                    |                   address
|
|                     1.0                      |
0x24f902ff362a01335db94b693edc769ba6226ff7   |
--------------------------------------------------------------------------------------------
----------
```

部署失败，账户3无权限利用CNS部署合约。

授权管理节点

控制台提供5个有关节点类型操作的命令，如下表所示：

- **注　意　：**　其　中addSealer、addObserver和removeNode命　令　受　权　限　控制，getSealerList和getObserverList命令不受权限控制。

登录账户1的控制台，授权账户2拥有管理节点的权限。

```
[group:1]> grantNodeManager 0x7fc8335fec9da5f84e60236029bb4a64a469a021
{
    "code":0,
    "msg":"success"
}

[group:1]> listNodeManager
------------------------------------------------------------------------------------------
----------
|                    address                     |                 enable_num
↪          |
| 0x7fc8335fec9da5f84e60236029bb4a64a469a021  |                     20
↪          |
------------------------------------------------------------------------------------------
↪----------
```

登录账户2的控制台，查看共识节点列表。

```
[group:1]> getSealerList
[
    "01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d95
↪
    "279c4adfd1e51e15e7fbd3fca37407db84bd60a6dd36813708479f31646b7480d776b84df5fea2f3157da6df9cad078
↪
    "320b8f3c485c42d2bfd88bb6bb62504a9433c13d377d69e9901242f76abe2eae3c1ca053d35026160d86db1a563ab2a
↪
    "c26dc878c4ff109f81915accaa056ba206893145a7125d17dc534c0ec41c6a10f33790ff38855df008aeca3a27ae7d9
]
```

查看观察节点列表：

```
[group:1]> getObserverList
[]
```

将第一个nodeID对应的节点设置为观察节点：

```
[group:1]> addObserver
↪01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d95
{
    "code":0,
    "msg":"success"
}

[group:1]> getObserverList
[
    "01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d95
]
```

(continues on next page)

(续上页)

```
[group:1]> getSealerList
[
    ␣
↪279c4adfd1e51e15e7fbd3fca37407db84bd60a6dd36813708479f31646b7480d776b84df5fea2f3157da6df9cad078
↪
    ␣
↪320b8f3c485c42d2bfd88bb6bb62504a9433c13d377d69e9901242f76abe2eae3c1ca053d35026160d86db1a563ab2a
↪
    ␣
↪c26dc878c4ff109f81915accaa056ba206893145a7125d17dc534c0ec41c6a10f33790ff38855df008aeca3a27ae7d9
]
```

登录账户3的控制台，将观察节点加入共识节点列表。

```
[group:1]> addSealer␣
↪01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d95
{
    "code":-50000,
    "msg":"permission denied"
}

[group:1]> getSealerList
[
    ␣
↪279c4adfd1e51e15e7fbd3fca37407db84bd60a6dd36813708479f31646b7480d776b84df5fea2f3157da6df9cad078
↪
    ␣
↪320b8f3c485c42d2bfd88bb6bb62504a9433c13d377d69e9901242f76abe2eae3c1ca053d35026160d86db1a563ab2a
↪
    ␣
↪c26dc878c4ff109f81915accaa056ba206893145a7125d17dc534c0ec41c6a10f33790ff38855df008aeca3a27ae7d9
]

[group:1]> getObserverList
[
    ␣
↪01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d95
]
```

添加共识节点失败，账户3没有权限管理节点。现在只有账户2有权限将观察节点加入共识节点列表。

授权修改系统参数

控制台提供2个关于修改系统参数的命令，如下表所示：

- **注意：** 目前支持键为tx_count_limit和tx_gas_limit的系统参数设置。其中setSystemConfigByKey命令受权限控制，getSystemConfigByKey命令不受权限控制。

登录账户1的控制台，授权账户2拥有修改系统参数的权限。

```
[group:1]> grantSysConfigManager 0x7fc8335fec9da5f84e60236029bb4a64a469a021
{
    "code":0,
    "msg":"success"
}

[group:1]> listSysConfigManager
-------------------------------------------------------------------------------
↪----------
|                    address                    |                    enable_num        ␣
↪          |
```

(continues on next page)

(续上页)

```
| 0x7fc8335fec9da5f84e60236029bb4a64a469a021  |                    23
↪          |
-------------------------------------------------------------------------------
↪----------
```

登录账户2的控制台，修改系统参数tx_count_limit的值为2000。

```
[group:1]> getSystemConfigByKey tx_count_limit
1000

[group:1]> setSystemConfigByKey tx_count_limit 2000
{
    "code":0,
    "msg":"success"
}

[group:1]> getSystemConfigByKey tx_count_limit
2000
```

登录账户3的控制台，修改系统参数tx_count_limit的值为3000。

```
[group:1]> setSystemConfigByKey tx_count_limit 3000
{
    "code":-50000,
    "msg":"permission denied"
}

[group:1]> getSystemConfigByKey tx_count_limit
2000
```

设置失败，账户3没有修改系统参数的权限。

授权账户写用户表

通过账户1授权账户3可以写用户表t_test的权限。

```
[group:1]> grantUserTableManager t_test 0xd86572ad4c92d4598852e2f34720a865dd4fc3dd
{
    "code":0,
    "msg":"success"
}
[group:1]> listUserTableManager t_test
-------------------------------------------------------------------------------
↪----------
|                  address                   |                enable_num
↪          |
| 0xd86572ad4c92d4598852e2f34720a865dd4fc3dd  |                     6
↪          |
-------------------------------------------------------------------------------
↪----------
```

登录账户3的控制台，在用户表t_test插入一条记录，然后查询该表的记录。

```
[group:1]> call TableTest.sol 0xfe649f510e0ca41f716e7935caee74db993e9de8 insert
↪"fruit" 1 "apple"

transaction hash:0xc4d261026851c3338f1a64ecd4712e5fc2a028c108363181725f07448b986f7e
-------------------------------------------------------------------------------
↪----------
Event logs
```

(continues on next page)

(续上页)

```
----------------------------------------------------------------------------------------
↪----------
InsertResult index: 0
count = 1
----------------------------------------------------------------------------------------
↪----------

[group:1]> call TableTest.sol 0xfe649f510e0ca41f716e7935caee74db993e9de8 select
↪"fruit"
[[fruit], [1], [apple]]
```

登录账户2的控制台，更新账户3插入的记录，并查询该表的记录。

```
[group:1]> call TableTest.sol 0xfe649f510e0ca41f716e7935caee74db993e9de8 update
↪"fruit" 1 "orange"
{
    "code":-50000,
    "msg":"permission denied"
}
[group:1]> call TableTest.sol 0xfe649f510e0ca41f716e7935caee74db993e9de8 select
↪"fruit"
[[fruit], [1], [apple]]
```

更新失败，账户2没有权限更新用户表t_test。

- 通过账户1撤销账户3写用户表t_test的权限。

```
[group:1]> revokeUserTableManager t_test 0xd86572ad4c92d4598852e2f34720a865dd4fc3dd
{
    "code":0,
    "msg":"success"
}

[group:1]> listUserTableManager t_test
Empty set.
```

撤销成功。

- **注意：** 此时没有账户拥有对用户表t_test的写权限，因此对该表的写权限恢复了初始状态，即所有账户均拥有对该表的写权限。如果让账户1没有对该表的写权限，则可以通过账号1授权另外一个账号，比如账号2拥有该表的写权限实现。

5.16 CA黑白名单

本文档描述CA黑、白名单的实践操作，建议阅读本操作文档前请先行了解《CA黑白名单介绍》。

5.16.1 黑名单

通过配置黑名单，能够拒绝与指定的节点连接。

配置方法

编辑config.ini

```
[certificate_blacklist]
    ; crl.0 should be nodeid, nodeid's length is 128
    ;crl.0=
```

重启节点生效

```
$ bash stop.sh && bash start.sh
```

查看节点连接

```
$ curl -X POST --data '{"jsonrpc":"2.0","method":"getPeers","params":[1],"id":1}'␣
↪http://127.0.0.1:8545 |jq
```

5.16.2 白名单

通过配置白名单，能够只与指定的节点连接，拒绝与白名单之外的节点连接。

配置方法

编辑config.ini，**不配置表示白名单关闭，可与任意节点建立连接。**

```
[certificate_whitelist]
    ; cal.0 should be nodeid, nodeid's length is 128
    cal.
↪0=7718df20f0f7e27fdab97b3d69deebb6e289b07eb7799c7ba92fe2f43d2efb4c1250dd1f11fa5b5ce687c8283d650
    cal.
↪1=f306eb1066ceb9d46e3b77d2833a1bde2a9899cfc4d0433d64b01d03e79927aa60a40507c5739591b8122ee609cf5
```

若节点未启动，则直接启动节点，若节点已启动，可直接用脚本reload_whitelist.sh刷新白名单配置即可（暂不支持动态刷新黑名单）。

```
# 若节点未启动
$ bash start.sh
# 若节点已启动
$ cd scripts
$ bash reload_whitelist.sh
node_127.0.0.1_30300 is not running, use start.sh to start and enable whitelist␣
↪directlly.
```

查看节点连接

```
$ curl -X POST --data '{"jsonrpc":"2.0","method":"getPeers","params":[1],"id":1}'␣
↪http://127.0.0.1:8545 |jq
```

5.16.3 使用场景：公共CA

所有用CFCA颁发证书搭的链，链的CA都是CFCA。此CA是共用的。必须启用白名单功能。使用公共CA搭的链，会存在两条链共用同一个CA的情况，造成无关的两条链的节点能彼此建立连接。此时需要配置白名单，拒绝与无关的链的节点建立连接。

搭链操作步骤

1. 用工具搭链
2. 查询所有节点的NodeID
3. 将所有NodeID配置入**每个**节点的白名单中
4. 启动节点或用脚本reload_whitelist.sh刷新节点白名单配置

扩容操作步骤

1. 用工具扩容一个节点
2. 查询此扩容节点的NodeID
3. 将此NodeID**追加**到入所有节点的白名单配置中
4. 将其他节点的白名单配置拷贝到新扩容的节点上

5. 用脚本`reload_whitelist.sh`刷新已启动的所有节点的白名单配置
6. 启动扩容节点
7. 将扩容节点加成组员（addSealer 或 addObserver）

5.16.4 黑白名单操作举例

准备

搭一个四个节点的链

```
bash build_chain.sh -l "127.0.0.1:4"
```

查看四个节点的NodeID

```
$ cat node*/conf/node.nodeid
219b319ba7b2b3a1ecfa7130ea314410a52c537e6e7dda9da46dec492102aa5a43bad81679b6af0cd5b9feb7cfdc0b395
7718df20f0f7e27fdab97b3d69deebb6e289b07eb7799c7ba92fe2f43d2efb4c1250dd1f11fa5b5ce687c8283d65030aa
f306eb1066ceb9d46e3b77d2833a1bde2a9899cfc4d0433d64b01d03e79927aa60a40507c5739591b8122ee609cf5636
38158ef34eb2d58ce1d31c8f3ef9f1fa829d0eb8ed1657f4b2a3ebd3265d44b243c69ffee0519c143dd67e91572ea8cb4
```

可得四个节点的NodeID:

- **node0**: 219b319b....
- **node1**: 7718df20....
- **node2**: f306eb10....
- **node3**: 38158ef3....

启动所有节点

```
$ cd node/127.0.0.1/
$ bash start_all.sh
```

查看连接，以node0为例。（8545是node0的rpc端口）

```
$ curl -X POST --data '{"jsonrpc":"2.0","method":"getPeers","params":[1],"id":1}'␣
↪http://127.0.0.1:8545 |jq
```

可看到连接信息，node0连接了除自身之外的其它三个节点。

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": [
    {
      "Agency": "agency",
      "IPAndPort": "127.0.0.1:62774",
      "Node": "node3",
      "NodeID":
↪"38158ef34eb2d58ce1d31c8f3ef9f1fa829d0eb8ed1657f4b2a3ebd3265d44b243c69ffee0519c143dd67e91572ea8
↪",
      "Topic": []
    },
    {
      "Agency": "agency",
      "IPAndPort": "127.0.0.1:62766",
      "Node": "node1",
      "NodeID":
↪"7718df20f0f7e27fdab97b3d69deebb6e289b07eb7799c7ba92fe2f43d2efb4c1250dd1f11fa5b5ce687c8283d6503
↪",
```

(continues on next page)

(续上页)

```
      "Topic": []
    },
    {
      "Agency": "agency",
      "IPAndPort": "127.0.0.1:30302",
      "Node": "node2",
      "NodeID":
↪"f306eb1066ceb9d46e3b77d2833a1bde2a9899cfc4d0433d64b01d03e79927aa60a40507c5739591b8122ee609cf56
↪",
      "Topic": []
    }
  ]
}
```

配置黑名单：node0拒绝node1的连接

将node1的NodeID写入node0的配置中

```
vim node0/config.ini
```

需要进行的配置如下，白名单为空（默认关闭）

```
[certificate_blacklist]
    ; crl.0 should be nodeid, nodeid's length is 128
    crl.
↪0=7718df20f0f7e27fdab97b3d69deebb6e289b07eb7799c7ba92fe2f43d2efb4c1250dd1f11fa5b5ce687c8283d650

[certificate_whitelist]
    ; cal.0 should be nodeid, nodeid's length is 128
    ; cal.0=
```

重启节点生效

```
$ cd node0
$ bash stop.sh && bash start.sh
```

查看节点连接

```
$ curl -X POST --data '{"jsonrpc":"2.0","method":"getPeers","params":[1],"id":1}'␣
↪http://127.0.0.1:8545 |jq
```

可看到只与两个节点建立的连接，未与node1建立连接

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": [
    {
      "Agency": "agency",
      "IPAndPort": "127.0.0.1:30303",
      "Node": "node3",
      "NodeID":
↪"38158ef34eb2d58ce1d31c8f3ef9f1fa829d0eb8ed1657f4b2a3ebd3265d44b243c69ffee0519c143dd67e91572ea8
↪",
      "Topic": []
    },
    {
      "Agency": "agency",
      "IPAndPort": "127.0.0.1:30302",
```

(continues on next page)

(续上页)

```
      "Node": "node2",
      "NodeID":
↪"f306eb1066ceb9d46e3b77d2833a1bde2a9899cfc4d0433d64b01d03e79927aa60a40507c5739591b8122ee609cf563
↪",
      "Topic": []
    }
  ]
}
```

配置白名单：node0拒绝与node1，node2之外的节点连接

将node1和node2的NodeID写入node0的配置中

```
$ vim node0/config.ini
```

需要进行的配置如下，黑名单置空，白名单配置上node1，node2

```
[certificate_blacklist]
    ; crl.0 should be nodeid, nodeid's length is 128
    ;crl.0=

[certificate_whitelist]
    ; cal.0 should be nodeid, nodeid's length is 128
    cal.
↪0=7718df20f0f7e27fdab97b3d69deebb6e289b07eb7799c7ba92fe2f43d2efb4c1250dd1f11fa5b5ce687c8283d650
    cal.
↪1=f306eb1066ceb9d46e3b77d2833a1bde2a9899cfc4d0433d64b01d03e79927aa60a40507c5739591b8122ee609cf5
```

重启节点生效

```
$ bash stop.sh && bash start.sh
```

查看节点连接

```
$ curl -X POST --data '{"jsonrpc":"2.0","method":"getPeers","params":[1],"id":1}'␣
↪http://127.0.0.1:8545 |jq
```

可看到只与两个节点建立的连接，未与node1建立连接

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": [
    {
      "Agency": "agency",
      "IPAndPort": "127.0.0.1:30302",
      "Node": "node2",
      "NodeID":
↪"f306eb1066ceb9d46e3b77d2833a1bde2a9899cfc4d0433d64b01d03e79927aa60a40507c5739591b8122ee609cf563
↪",
      "Topic": []
    },
    {
      "Agency": "agency",
      "IPAndPort": "127.0.0.1:30301",
      "Node": "node1",
      "NodeID":
↪"7718df20f0f7e27fdab97b3d69deebb6e289b07eb7799c7ba92fe2f43d2efb4c1250dd1f11fa5b5ce687c8283d6503
↪",
```

(continues on next page)

(续上页)

```
      "Topic": []
    }
  ]
}
```

黑名单与白名单混合配置：黑名单优先级高于白名单，白名单配置的基础上拒绝与node1建立连接

编辑node0的配置

```
$ vim node0/config.ini
```

需要进行的配置如下，黑名单配置上node1，白名单配置上node1，node2

```
[certificate_blacklist]
    ; crl.0 should be nodeid, nodeid's length is 128
    crl.
↪0=7718df20f0f7e27fdab97b3d69deebb6e289b07eb7799c7ba92fe2f43d2efb4c1250dd1f11fa5b5ce687c8283d650

[certificate_whitelist]
    ; cal.0 should be nodeid, nodeid's length is 128
    cal.
↪0=7718df20f0f7e27fdab97b3d69deebb6e289b07eb7799c7ba92fe2f43d2efb4c1250dd1f11fa5b5ce687c8283d650
    cal.
↪1=f306eb1066ceb9d46e3b77d2833a1bde2a9899cfc4d0433d64b01d03e79927aa60a40507c5739591b8122ee609cf5
```

重启节点生效

```
$ bash stop.sh && bash start.sh
```

查看节点连接

```
$ curl -X POST --data '{"jsonrpc":"2.0","method":"getPeers","params":[1],"id":1}'␣
↪http://127.0.0.1:8545 |jq
```

可看到虽然白名单上配置了node1，但由于node1在黑名单中也有配置，node0也不能与node1建立连接

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": [
    {
      "Agency": "agency",
      "IPAndPort": "127.0.0.1:30302",
      "Node": "node2",
      "NodeID":
↪"f306eb1066ceb9d46e3b77d2833a1bde2a9899cfc4d0433d64b01d03e79927aa60a40507c5739591b8122ee609cf56
↪",
      "Topic": []
    }
  ]
}
```

5.17 存储安全

联盟链的数据，只对联盟内部成员可见。落盘加密，保证了运行联盟链的数据，在硬盘上的安全性。一旦硬盘被带出联盟链自己的内网环境，数据将无法被解密。

落盘加密是对节点存储在硬盘上的内容进行加密，加密的内容包括：合约的数据、节点的私钥。

具体的落盘加密介绍，可参考：落盘加密的介绍

5.17.1 部署Key Manager

每个机构一个Key Manager，具体的部署步骤，可参考Key Manager README

重要： 若节点为国密版，Key Manager也需是国密版。

5.17.2 生成节点

用build_chain.sh脚本，用普通的操作方法，先生成节点。

```
curl -LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/`curl -s␣
↪https://api.github.com/repos/FISCO-BCOS/FISCO-BCOS/releases | grep "\"v2\.[0-9]\.
↪[0-9]\"" | sort -u | tail -n 1 | cut -d \" -f 4`/build_chain.sh && chmod u+x␣
↪build_chain.sh

bash build_chain.sh -l "127.0.0.1:4" -p 30300,20200,8545
```

重要： 节点生成后，不能启动，待dataKey配置后，再启动。节点在第一次运行前，必须配置好是否采用落盘加密。一旦节点开始运行，无法切换状态。

5.17.3 启动Key Manager

直接启动key-manager。若未部署key-manager，可参考Key Manager README

```
# 参数：端口，superkey
./key-manager 31443 123xyz
```

启动成功，打印日志

```
[1546501342949][TRACE][Load]key-manager started,port=31443
```

5.17.4 配置dataKey

重要： 配置dataKey的节点，必须是新生成，未启动过的节点。

执行脚本，定义dataKey，获取cipherDataKey

```
cd key-manager/scripts
bash gen_data_secure_key.sh 127.0.0.1 31443 123456

CiherDataKey generated: ed157f4588b86d61a2e1745efe71e6ea
Append these into config.ini to enable disk encryption:
[storage_security]
enable=true
key_manager_ip=127.0.0.1
key_manager_port=31443
cipher_data_key=ed157f4588b86d61a2e1745efe71e6ea
```

得到cipherDataKey，脚本自动打印出落盘加密需要的ini配置(如下)。此时得到节点的cipherDataKey：cipher_data_key=ed157f4588b86d61a2e1745efe71e6ea 将得到的落盘加密的ini配置，写入节点配置文件（config.ini）中。

```
vim nodes/127.0.0.1/node0/config.ini
```

修改[storage_security]中的字段如下。

```
[storage_security]
enable=true
key_manager_ip=127.0.0.1
key_manager_port=31443
cipher_data_key=ed157f4588b86d61a2e1745efe71e6ea
```

5.17.5 加密节点私钥

执行脚本，加密节点私钥

```
cd key-manager/scripts
# 参数: ip port 节点私钥文件 cipherDataKey
bash encrypt_node_key.sh 127.0.0.1 31443 ../../nodes/127.0.0.1/node0/conf/node.key␣
↪ed157f4588b86d61a2e1745efe71e6ea
```

执行后，节点私钥自动被加密，加密前的文件备份到了文件node.key.bak.xxxxxx中，**请将备份私钥妥善保管，并删除节点上生成的备份私钥**

```
[INFO] File backup to "nodes/127.0.0.1/node0/conf/node.key.bak.1546502474"
[INFO] "nodes/127.0.0.1/node0/conf/node.key" encrypted!
```

若查看node.key，可看到，已经被加密为密文

```
8b2eba71821a5eb15b0cbe710e96f23191419784f644389c58e823477cf33bd73a51b6f14af368d4d3ed647d9de681893
```

重要：所有需要加密的文件列举如下，若未加密，节点无法启动。

- 非国密版：conf/node.key
- 国密版：conf/gmnode.key和conf/origin_cert/node.key

5.17.6 节点运行

直接启动节点即可

```
cd nodes/127.0.0.1/node0/
./start.sh
```

5.17.7 正确性判断

（1）节点正常运行，正常共识，不断输出共识打包信息。

```
tail -f nodes/127.0.0.1/node0/log/* | grep +++
```

（2）key-manager在节点每次启动时，都会打印一条日志。例如，节点在一次启动时，Key Manager直接输出的日志如下。

```
[1546504272699][TRACE][Dec]Respond
{
   "dataKey" : "313233343536",
   "error" : 0,
   "info" : "success"
}
```

5.18 国密支持

为了充分支持国产密码学算法，金链盟基于国产密码学标准，在FISCO BCOS平台中集成了国密加解密、签名、验签、哈希算法、国密SSL通信协议，实现了对国家密码局认定的商用密码的完全支持。设计文档见国密版FISCO BCOS设计手册。

5.18.1 初次部署国密版FISCO BCOS

本节使用build_chain脚本在本地搭建一条4节点的FISCO BCOS链，以Ubuntu 16.04系统为例操作。本节使用预编译的静态fisco-bcos二进制文件，在CentOS 7和Ubuntu 16.04上经过测试。

```
# Ubuntu16安装依赖
$ sudo apt install -y openssl curl
# 准备环境
$ cd ~ && mkdir -p fisco && cd fisco
# 下载build_chain.sh脚本
$ curl -LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.5.0/build_
↪chain.sh && chmod u+x build_chain.sh
```

- 搭建4节点FISCO BCOS链

```
# 生成一条4节点的FISCO链 4个节点都属于group1 下面指令在fisco目录下执行
# -p指定起始端口，分别是p2p_port,channel_port,jsonrpc_port
# 根据下面的指令，需要保证机器的30300~30303，20200~20203，8545~8548端口没有被占用
# -g 国密编译选项，使用成功后会生成国密版的节点。默认从GitHub下载最新稳定版本可执行程序
$ ./build_chain.sh -l "127.0.0.1:4" -p 30300,20200,8545 -g
```

关于build_chain.sh脚本选项，请参考这里。命令正常执行会输出All completed。（如果没有输出，则参考nodes/build.log检查）。

```
[INFO] Downloading tassl binary ...
Generating CA key...
Generating Guomi CA key...
============================================================
Generating keys ...
Processing IP:127.0.0.1 Total:4 Agency:agency Groups:1
============================================================
Generating configurations...
Processing IP:127.0.0.1 Total:4 Agency:agency Groups:1
============================================================
[INFO] FISCO-BCOS Path   : bin/fisco-bcos
[INFO] Start Port        : 30300 20200 8545
[INFO] Server IP         : 127.0.0.1:4
[INFO] State Type        : storage
[INFO] RPC listen IP     : 127.0.0.1
[INFO] Output Dir        : /mnt/c/Users/asherli/Desktop/key-manager/build/nodes
[INFO] CA Key Path       : /mnt/c/Users/asherli/Desktop/key-manager/build/nodes/
↪gmcert/ca.key
[INFO] Guomi mode        : yes
============================================================
[INFO] All completed. Files in /mnt/c/Users/asherli/Desktop/key-manager/build/nodes
```

当国密联盟链部署完成之后，其余操作与安装的操作相同。

5.18.2 国密配置信息

国密版本FISCO BCOS节点之间采用SSL安全通道发送和接收消息，证书主要配置如下：

```
[network_security]

data_path：证书文件所在路径
key：节点私钥相对于data_path的路径
cert：证书gmnode.crt相对于data_path的路径
ca_cert：gmca证书路径

;certificate configuration
[network_security]
    ;directory the certificates located in
    data_path=conf/
    ;the node private key file
    key=gmnode.key
    ;the node certificate file
    cert=gmnode.crt
    ;the ca certificate file
    ca_cert=gmca.crt
```

FISCO-BCOS 2.5.0版本以后，节点与SDK之间既支持SSL连接进行通信，也可以以国密SSL连接进行通信，相关配置如下：

```
[chain]
    ; use SM crypto or not, should nerver be changed
    sm_crypto=true
    ; use SM SSL connection with SDK
    sm_crypto_channel=true
```

5.18.3 国密版SDK使用

详细操作参考SDK文档。

5.18.4 国密版控制台配置

详情操作参考配置国密版控制台。

5.18.5 国密控制台使用

国密版控制台功能与标准版控制台使用方式相同，见控制台操作手册。

5.18.6 国密落盘加密配置

国密版Key Manager

国密版的Key Manager需重新编译Key Manager，不同点在于cmake时带上-DBUILD_GM=ON选项。

```
# centos下
cmake3 .. -DBUILD_GM=ON
# ubuntu下
cmake .. -DBUILD_GM=ON
```

其它步骤与标准版Key Manager相同，请参考：key-manager repository。

国密版节点配置

FISCO BCOS国密版采用双证书模式，因此落盘加密需要加密的两套证书，分别为：conf/gmnode.key 和 conf/origin_cert/node.key。其它与标准版落盘加密操作相同。

```
cd key-manager/scripts
#加密 conf/gmnode.key 参数：ip port 节点私钥文件 cipherDataKey
bash encrypt_node_key.sh 127.0.0.1 31443 nodes/127.0.0.1/node0/conf/gmnode.key␣
↪ed157f4588b86d61a2e1745efe71e6ea
#加密 conf/origin_cert/node.key 参数：ip port 节点私钥文件 cipherDataKey
bash encrypt_node_key.sh 127.0.0.1 31443 nodes/127.0.0.1/node0/conf/origin_cert/
↪node.key ed157f4588b86d61a2e1745efe71e6ea
```

5.19 日志说明

FISCO BCOS的所有群组日志都输出到log目录下log_%YYYY%mm%dd%HH.%MM的文件中，且定制了日志格式，方便用户通过日志查看各群组状态。日志配置说明请参考日志配置说明

5.19.1 日志格式

每一条日志记录格式如下：

```
# 日志格式：
log_level|time|[g:group_id][module_name] content

# 日志示例：
info|2019-06-26 16:37:08.253147|[g:3][CONSENSUS][PBFT]^^^^^^^^Report,num=0,
↪sealerIdx=0,hash=a4e10062...,next=1,tx=0,nodeIdx=2
```

各字段含义如下：

- log_level: 日志级别，目前主要包括trace, debug, info, warning, error和fatal，其中在发生极其严重错误时会输出fatal
- time: 日志输出时间，精确到纳秒
- group_id: 输出日志记录的群组ID
- module_name：模块关键字，如同步模块关键字为SYNC，共识模块关键字为CONSENSUS
- content：日志记录内容

5.19.2 常见日志说明

共识打包日志

注解:

- 仅共识节点会周期性输出共识打包日志(节点目录下可通过命令 tail -f log/* | grep "${group_id}.*++" 查看指定群组共识打包日志)
- 打包日志可检查指定群组的共识节点是否异常，**异常的共识节点不会输出打包日志**

下面是共识打包日志的示例：

```
info|2019-06-26 18:00:02.551399|[g:2][CONSENSUS][SEALER]++++++++++++++++
↪Generating seal on,blkNum=1,tx=0,nodeIdx=3,hash=1f9c2b14...
```

日志中各字段的含义如下：

- blkNum: 打包区块的高度
- tx: 打包区块中包含的交易数
- nodeIdx: 当前共识节点的索引
- hash: 打包区块的哈希

共识异常日志

网络抖动、网络断连或配置出错(如同一个群组的创世块文件不一致)均有可能导致节点共识异常，PBFT共识节点会输出ViewChangeWarning日志，示例如下：

```
warning|2019-06-26 18:00:06.154102|[g:1][CONSENSUS][PBFT]ViewChangeWarning: not
↪caused by omit empty block ,v=5,toV=6,curNum=715,hash=ed6e856d...,nodeIdx=3,
↪myNode=e39000ea...
```

该日志各字段含义如下：

- v: 当前节点PBFT共识视图
- toV: 当前节点试图切换到的视图
- curNum: 节点最高块高
- hash: 节点最高块哈希
- nodeIdx: 当前共识节点索引
- myNode: 当前节点Node ID

区块落盘日志

区块共识成功或节点正在从其他节点同步区块，均会输出落盘日志。

注解： 向节点发交易，若交易被处理，非游离节点均会输出落盘日志(节点目录下可通过命令 `tail -f log/* | grep "${group_id}.*Report"` 查看节点出块情况)，若没有输出该日志，说明节点已处于异常状态，请优先检查网络连接是否正常、节点证书是否有效

下面是区块落盘日志：

```
info|2019-06-26 18:00:07.802027|[g:1][CONSENSUS][PBFT]^^^^^^^^Report,num=716,
↪sealerIdx=2,hash=dfd75e06...,next=717,tx=8,nodeIdx=3
```

日志中各字段说明如下：

- num: 落盘区块块高
- sealerIdx: 打包该区块的共识节点索引
- hash: 落盘区块哈希
- next: 下一个区块块高
- tx: 落盘区块中包含的交易数
- nodeIdx: 当前共识节点索引

网络连接日志

注解： 节点目录下可通过命令 `tail -f log/* | grep "connected count"` 检查网络状态，若日志输出的网络连接数目不符合预期，请通过 `netstat -anp | grep fisco-bcos` 命令检查节点连接

日志示例如下：

```
info|2019-06-26 18:00:01.343480|[P2P][Service] heartBeat,connected count=3
```

日志中各字段含义如下：

- `connected count`: 与当前节点建立P2P网络连接的节点数

5.19.3 日志模块关键字

FISCO BCOS日志中核心模块关键字如下：

5.20 Caliper压力测试指南

5.20.1 一、环境要求

1.1 硬件

- 需要外网权限

1.2 操作系统

- 版本要求：Ubuntu >= 16.04, CentOS >= 7, MacOS >= 10.14

1.3 基础软件

- python 2.7，make，g++，gcc，git

1.4 NodeJS

- 版本要求：

  NodeJS 8 (LTS), 9, 或 10 (LTS)，Caliper尚未在更高的NodeJS版本中进行过验证。

- 安装指南：

  建议使用nvm(Node Version Manager)安装，nvm的安装方式如下：

```
# 安装nvm
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.2/install.sh |␣
↪bash
# 加载nvm配置
source ~/.$(basename $SHELL)rc
# 安装Node.js 8
nvm install 8
# 使用Node.js 8
nvm use 8
```

1.5 Docker

- 版本要求：>= 18.06.01
- 安装指南：

  CentOS：

```
# 添加源
sudo yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/
↪centos/docker-ce.repo
# 更新缓存
sudo yum makecache fast
# 安装社区版Docker
sudo yum -y install docker-ce
# 将当前用户加入docker用户组（重要）
sudo groupadd docker
sudo gpasswd -a ${USER} docker
# 重启Docker服务
sudo service docker restart
newgrp - docker
# 验证Docker是否已经启动
sudo systemctl status docker
```

  Ubuntu

```
# 更新包索引
sudo apt-get update
# 安装基础依赖库
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common
# 添加Docker官方GPG key
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
# 添加docker仓库
sudo add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) \
stable"
# 更新包索引
sudo apt-get update
# 安装Docker
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

  MacOs下

  请安装Docker Desktop。

- 加入Docker用户组

  CentOS

```
sudo groupadd docker
sudo gpasswd -a ${USER} docker

# 重启Docker服务
sudo service docker restart
# 验证Docker是否已经启动
sudo systemctl status docker
```

  Ubuntu

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

1.6 Docker Compose

- 版本要求：>= 1.22.0
- 安装指南：

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.24.0/
↪docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

5.20.2 二、Caliper部署

2.1 部署

Caliper提供了方便易用的命令行界面工具`caliper-cli`，推荐在本地进行局部安装：

1. 建立一个工作目录

```
mkdir benchmarks && cd benchmarks
```

1. 对NPM项目进行初始化

```
npm init
```

这一步主要是为在工作目录下创建package.json文件以方便后续依赖项的安装，如果不需要填写项目信息的话可以直接执行`npm init -y`。

1. 安装`caliper-cli`

```
npm install --only=prod @hyperledger/caliper-cli@0.2.0
```

由于Caliper所有依赖项的安装较为耗时，因此使用`--only=prod`选项用于指定NPM只安装Caliper的核心组件，而不安装其他的依赖项（如各个区块链平台针对Caliper的适配器）。在部署完成后，可以通过`caliper-cli`显式绑定需要测试的区块链平台及相应的适配器。

1. 验证`caliper-cli`安装成功

```
npx caliper --version
```

若安装成功，则会打印相应的版本信息，如：

```
user@ubuntu:~/benchmarks$ npx caliper --version
v0.2.0
```

2.2 绑定

由于Caliper采用了轻量级的部署方式，因此需要显式的绑定步骤指定要测试的平台及适配器版本，`caliper-cli`会自动进行相应依赖项的安装。使用`npx caliper bind`命令进行绑定，命令所需的各项参数可以通过如下命令查看：

```
user@ubuntu:~/benchmarks$ npx caliper bind --help
Usage:
  caliper bind --caliper-bind-sut fabric --caliper-bind-sdk 1.4.1 --caliper-bind-
↪cwd ./ --caliper-bind-args="-g"
```

(continues on next page)

(续上页)

```
Options:
  --help                Show help  [boolean]
  -v, --version         Show version number  [boolean]
  --caliper-bind-sut    The name of the platform to bind to  [string]
  --caliper-bind-sdk    Version of the platform SDK to bind to  [string]
  --caliper-bind-cwd    The working directory for performing the SDK install␣
↪[string]
  --caliper-bind-args   Additional arguments to pass to "npm install". Use the "="␣
↪notation when setting this parameter  [string]
```

其中，

–caliper-bind-sut：用于指定需要测试的区块链平台，即受测系统（***S***ystem ***u***nder ***T***est）；**–caliper-bind-sdk**：用于指定适配器版本；**–caliper-bind-cwd**：用于绑定`caliper-cli`的工作目录，`caliper-cli`在加载配置文件等场合时均是使用相对于工作目录的相对路径；**caliper-bind-args**：用于指定`caliper-cli`在安装依赖项时传递给`npm`的参数，如用于全局安装的`-g`。

对于FISCO BCOS，可以采用如下方式进行绑定：

```
npx caliper bind --caliper-bind-sut fisco-bcos --caliper-bind-sdk latest
```

由于FISCO BCOS对于caliper 0.2.0版本的适配存在部分不兼容情况，需要手动按照https://github.com/FISCO-BCOS/FISCO-BCOS/issues/1248中的步骤修改代码后方可正常运行。

2.3 快速体验FISCO BCOS基准测试

为方便测试人员快速上手，FISCO BCOS已经为Caliper提供了一组预定义的测试样例，测试对象涵盖HelloWorld合约、Solidity版转账合约及预编译版转账合约。同时在测试样例中，Caliper测试脚本会使用docker在本地自动部署及运行4个互连的节点组成的链，因此测试人员无需手工搭链及编写测试用例便可直接运行这些测试样例。

1. 在工作目录下下载预定义测试用例：

```
git clone https://github.com/vita-dounai/caliper-benchmarks.git
```

1. 执行HelloWorld合约测试

```
npx caliper benchmark run --caliper-workspace caliper-benchmarks --caliper-
↪benchconfig benchmarks/samples/fisco-bcos/helloworld/config.yaml  --caliper-
↪networkconfig networks/fisco-bcos/4nodes1group/fisco-bcos.json
```

1. 执行Solidity版转账合约测试

```
npx caliper benchmark run --caliper-workspace caliper-benchmarks --caliper-
↪benchconfig benchmarks/samples/fisco-bcos/transfer/solidity/config.yaml  --
↪caliper-networkconfig networks/fisco-bcos/4nodes1group/fisco-bcos.json
```

1. 执行预编译版转账合约测试

```
npx caliper benchmark run --caliper-workspace caliper-benchmarks --caliper-
↪benchconfig benchmarks/samples/fisco-bcos/transfer/precompiled/config.yaml  --
↪caliper-networkconfig networks/fisco-bcos/4nodes1group/fisco-bcos.json
```

测试完成后，会在命令行界面中展示测试结果（TPS、延迟等）及资源消耗情况，同时会在`caliper-benchmarks`目录下生成一份包含上述内容的可视化HTML报告。

`caliper benchmark run`所需的各项参数可以通过如下命令查看：

```
user@ubuntu:~/benchmarks$ npx caliper benchmark run --help
caliper benchmark run --caliper-workspace ~/myCaliperProject --caliper-benchconfig␣
↪my-app-test-config.yaml --caliper-networkconfig my-sut-config.yaml

Options:
  --help                    Show help  [boolean]
  -v, --version             Show version number  [boolean]
  --caliper-benchconfig     Path to the benchmark workload file that describes the␣
↪test client(s), test rounds and monitor.  [string]
  --caliper-networkconfig  Path to the blockchain configuration file that contains␣
↪information required to interact with the SUT  [string]
  --caliper-workspace       Workspace directory that contains all configuration␣
↪information  [string]
```

其中，

–caliper-workspace：用于指定caliper-cli的工作目录，如果没有绑定工作目录，可以通过该选项动态指定工作目录； **–caliper-benchconfig**：用于指定测试配置文件，测试配置文件中包含测试的具体参数，如交易的发送方式、发送速率控制器类型、性能监视器类型等； **–caliper-networkconfig**：用于指定网络配置文件，网络配置文件中会指定Caliper与受测系统的连接方式及要部署测试的合约等。

三、自定义测试用例

本节将会以测试HelloWorld合约为例，介绍如何使用Caliper测试自定义的测试用例。

Caliper前后端分离的设计原则使得只要后端的区块链系统开放了相关网络端口，Caliper便可以对该系统进行测试。结合Docker提供的性能数据统计服务或本地的ps命令工具，Caliper能够在测试的同时收集节点所在机器上的各种性能数据，包括CPU、内存、网络及磁盘的使用等。尽管Caliper能工作在不使用Docker模式而是使用原生二进制ficos-bcos可执行程序搭建出的链上，但是那样Caliper将无法获知节点所在机器上的资源消耗。因此，在目前的Caliper版本下（v0.2.0），我们推荐使用Docker模式搭链。

3.1、配置Docker Daemon及部署FISCO BCOS网络

如果只想基于已经搭建好的链进行测试，可以跳过本小节。

3.1.1 配置Docker Daemon

为方便Caliper统一管理节点容器及监控性能数据，在运行节点的服务器上首先需要开启Docker Daemon服务。

开始之前，先停止docker进程：

```
sudo service docker stop
```

创建/etc/docker/daemon.json文件（如果已经存在则修改），加入以下内容：

```
{
  "hosts" : ["unix:///var/run/docker.sock", "tcp://0.0.0.0:2375"]
}
```

“unix:///var/run/docker.sock”：UNIX套接字，本地客户端将通过这个来连接Docker Daemon；**tcp://0.0.0.0:2375**，TCP套接字，表示允许任何远程客户端通过2375端口连接Docker Daemon.

使用sudo systemctl edit docker新建或修改/etc/systemd/system/docker.service.d/override.conf，其内容如下：

```
##Add this to the file for the docker daemon to use different ExecStart parameters␣
↪(more things can be added here)
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd
```

默认情况下使用systemd时，docker.service的设置为：ExecStart=/usr/bin/dockerd -H fd://，这将覆写daemon.json中的任何hosts。通过override.conf文件将ExecStart定义为：ExecStart=/usr/bin/dockerd，就能使daemon.json中设置的hosts生效。override.conf中的第一行ExecStart=必须要有，这一行将用于清除默认的ExecStart参数。

重新加载daemon并重启docker服务：

```
sudo systemctl daemon-reload
sudo systemctl restart docker.service
```

检查端口监听：

```
sudo netstat -anp | grep 2375
```

如果出现以下字样则表明配置成功：

```
tcp6       0      0 :::2375                 :::*                    LISTEN      ␣
↪79018/dockerd
```

此时能够在另一台机器上通过远程连接访问本机的Docker Daemon服务，例如：

```
# 假设开启Docker Daemon服务的机器IP地址为192.168.1.1
docker -H 192.168.1.1:2375 images
```

3.1.2 建链

使用开发部署工具 build_chain.sh脚本快速建链。本节以4个节点、全连接的形式搭链，但本节所述的测试方法能够推广任意数量节点及任意网络拓扑形式的链。

创建生成节点的配置文件（如一个名为ipconf的文件），文件内容如下：

```
192.168.1.1:1 agency1 1
192.168.1.2:1 agency1 1
192.168.1.3:1 agency1 1
192.168.1.4:1 agency1 1
```

生成链中节点的配置文件：

```
bash build_chain.sh -f ipconf -i -p 30914,20914,8914
```

将产生的节点配置文件夹分别拷贝至对应的服务器上：

```
scp -r 192.168.1.1/node0/ app@192.168.1.1:/data/test
scp -r 192.168.1.2/node0/ app@192.168.1.2:/data/test
scp -r 192.168.1.3/node0/ app@192.168.1.3:/data/test
scp -r 192.168.1.4/node0/ app@192.168.1.3:/data/test
```

3.2 配置FISCO BCOS适配器

在另外一台机器上部署Caliper，部署教程见第二节。

3.2.1 网络配置

新建一个名为4nodes1group的目录，本阶示例中的FISCO BCOS适配器的网络配置文件均会放置于此。新建一个名为fisco-bcos.json的配置文件，文件内容如下：

```
{
    "caliper": {
        "blockchain": "fisco-bcos",
        "command": {
            "start": "sh network/fisco-bcos/4nodes1group/start.sh",
            "end": "sh network/fisco-bcos/4nodes1group/end.sh"
        }
    },
    "fisco-bcos": {
        "config": {
            "privateKey":
↪"bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd",
            "account": "0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3"
        },
        "network": {
            "nodes": [
                {
                    "ip": "192.168.1.1",
                    "rpcPort": "8914",
                    "channelPort": "20914"
                },
                {
                    "ip": "192.168.1.2",
                    "rpcPort": "8914",
                    "channelPort": "20914"
                },
                {
                    "ip": "192.168.1.3",
                    "rpcPort": "8914",
                    "channelPort": "20914"
                },
           ],
            "authentication": {
                "key": "packages/caliper-samples/network/fisco-bcos/4nodes1group/
↪sdk/node.key",
                "cert": "packages/caliper-samples/network/fisco-bcos/4nodes1group/
↪sdk/node.crt",
                "ca": "packages/caliper-samples/network/fisco-bcos/4nodes1group/
↪sdk/ca.crt"
            },
            "groupID": 1,
            "timeout": 600000
        },
        "smartContracts": [
            {
                "id": "helloworld",
                "path": "src/contract/fisco-bcos/helloworld/HelloWorld.sol",
                "language": "solidity",
                "version": "v0"
            }
        ]
    },
    "info": {
        "Version": "2.0.0",
        "Size": "4 Nodes",
        "Distribution": "Remote Host"
    }
}
```

配置文件中每一项的具体含义如下：

- **caliper.command.start**

启动Caliper时会首先执行start配置中指定的命令，主要用于初始化SUT。本文示例中使用Docker模式启动，启动Caliper时首先执行当前目录下的`start.sh`文件，其具体内容是：

```
docker -H 192.168.1.1:2375 run -d --rm --name node0 -v /data/test/node0/:/data -p␣
↪8914:8914 -p 20914:20914 -p 30914:30914 -w=/data fiscoorg/fiscobcos:latest -c␣
↪config.ini 1> /dev/null
docker -H 192.168.1.2:2375 run -d --rm --name node1 -v /data/test/node0/:/data -p␣
↪8914:8914 -p 20914:20914 -p 30914:30914 -w=/data fiscoorg/fiscobcos:latest -c␣
↪config.ini 1> /dev/null
docker -H 192.168.1.3:2375 run -d --rm --name node2 -v /data/test/node0/:/data -p␣
↪8914:8914 -p 20914:20914 -p 30914:30914 -w=/data fiscoorg/fiscobcos:latest -c␣
↪config.ini 1> /dev/null
docker -H 192.168.1.4:2375 run -d --rm --name node3 -v /data/test/node0/:/data -p␣
↪8914:8914 -p 20914:20914 -p 30914:30914 -w=/data fiscoorg/fiscobcos:latest -c␣
↪config.ini 1> /dev/null
```

即启动远程的Docker容器。如果不需要在Caliper启动时执行命令，需要将该配置项置空。

- **caliper.command.end**

Caliper在退出流程的最后会执行end配置指定的命令，主要用于清理环境。本例中在测试结束时会执行当前目录下的`end.sh`文件，其具体内容是：

```
docker -H 192.168.1.1:2375 stop $(docker -H 192.168.1.1:2375 ps -a | grep node0 |␣
↪cut -d " " -f 1) 1> /dev/null && echo -e "\033[32mremote container node0␣
↪stopped\033[0m"
docker -H 192.168.1.2:2375 stop $(docker -H 192.168.1.2:2375 ps -a | grep node1 |␣
↪cut -d " " -f 1) 1> /dev/null && echo -e "\033[32mremote container node1␣
↪stopped\033[0m"
docker -H 192.168.1.3:2375 stop $(docker -H 192.168.1.3:2375 ps -a | grep node2 |␣
↪cut -d " " -f 1) 1> /dev/null && echo -e "\033[32mremote container node2␣
↪stopped\033[0m"
docker -H 192.168.1.4:2375 stop $(docker -H 192.168.1.3:2375 ps -a | grep node3 |␣
↪cut -d " " -f 1) 1> /dev/null && echo -e "\033[32mremote container node3␣
↪stopped\033[0m"
```

即停止并删除有所的远程容器。如果不需要在Caliper退出时执行命令，需要将该配置项置空。

- **network.nodes**

一个包含了所有要连接节点的列表，列表中每一项需要指明被连接节点的IP地址、RPC端口及Channel端口号，所有端口号需要和节点的配置文件保持一致。

- **network.authentication**

适配器向节点的Channel端口发起请求时需要使用CA根证书等文件，这些文件已在3.1.2节中调用`build_chain.sh`脚本时已经生成好，使用任一节点配置下的sdk文件夹中的相应文件即可，需要在该配置中写上所有文件的路径（使用相对路径时需要以`caliper-cli`工作目录为起始目录）。

- **network.smartContracts**

指定要测试的合约，Caliper会在启动时想后端区块链系统部署合约。目前FISCO BCOS适配器支持通过`language`字段指定两种类型的合约——Solidity合约和预编译合约，当测试合约为Solidity合约时，`language`字段需要指定为`solidity`，当测试合约为预编译合约时，`language`字段需要指定为`precompiled`。当测试合约为预编译合约时，需要在`address`字段中指定预编译合约的地址，否则需要在`path`字段中指定Solidity合约的路径。

3.2.2 测试配置

测试配置用于指定测试的具体运行方式。测试配置是一个YAML文件，HelloWorld合约的测试配置文件内容如下所示：

```
---
test:
  name: Hello World
  description: This is a helloworld benchmark of FISCO BCOS for caliper
  clients:
    type: local
    number: 1
  rounds:
  - label: get
    description: Test performance of getting name
    txNumber:
    - 10000
    rateControl:
    - type: fixed-rate
      opts:
        tps: 1000
    callback: benchmarks/samples/fisco-bcos/helloworld/get.js
  - label: set
    description: Test performance of setting name
    txNumber:
    - 10000
    rateControl:
    - type: fixed-rate
      opts:
        tps: 1000
    callback: benchmarks/samples/fisco-bcos/helloworld/set.js
monitor:
  type:
  - docker
  docker:
    name:
    - http://192.168.1.1:2375/node0
    - http://192.168.1.2:2375/node1
    - http://192.168.1.3:2375/node2
    - http://192.168.1.4:2375/node3
  interval: 0.1
```

测试文件中主要包括两部分：

- 测试内容配置

test项负责对测试内容进行配置。配置主要集中在round字段中指定如何对区块链系统进行测试。每一个测试可以包含多轮，每一轮可以向区块链发起不同的测试请求。具体要HelloWorld合约测试，测试中包含两轮，分别对合约的get接口和set接口进行测试。在每一轮测试中，可以通过txNumber或txDuration字段指定测试的交易发送数量或执行时间，并通过rateControl字段指定交易发送时的速率控制器，在本节的示例中，使用了QPS为1000的匀速控制器，更多速率控制器的介绍可以参考官方文档。

- 性能监视器配置

monitor项负责对测试所使用的性能监视器的进行配置。每项配置项的解释如下：

1. monitor.type，需要指定为docker，指对docker容器进行监控；
2. monitor.docker.name，一个包含所有要监视的节点的docker容器名称列表，名字必须以http://开头，其后跟随"{节点的IP}:{节点docker daemon端口}/{docker容器的名称}"；
3. monitor.interval，监视器的采样间隔，单位为秒。

如果是在**本地**搭好的链，则可以添加本地性能监视器，相应地监视器的配置更改如下：

```
monitor:
  type:
  - process
```

(continues on next page)

(续上页)

```
  process:
    - command: node0
      multiOutput: avg
    - command: node1
      multiOutput: avg
    - command: node2
      multiOutput: avg
    - command: node3
      multiOutput: avg
  interval: 0.1
```

其中每项配置项的解释如下：

1. monitor.type，需要指定为`process`，只对进程进行监控；
2. monitor.process，一个包含所有要监视的进称列表，其中每个进程的command属性为一个正则表达式，表示进程名称；每个进程还可以有一个arguments属性（未在上述示例中使用到），表示进程的参数。Caliper会先使用ps命令搜索commad + arguments,然后匹配以得到目标的进程的进程ID及系统资源的使用情况。每个进程的multiOutput属性用于指定结果的输出方式，目前支持平均值（avg）及总和（sum）两种方式；
3. monitor.interval，监视器的采样间隔，单位为秒。

需要注意的是，进程监控目前暂不支持监控进程对网络和磁盘的使用情况。

5.21 隐私保护

隐私保护是联盟链的一大技术挑战。为了保护链上数据、保障联盟成员隐私，并且保证监管的有效性，FISCO BCOS以预编译合约的形式集成了同态加密、群/环签名验证功能，提供了多种隐私保护手段。

文档一、二节分别对同态加密和群/环签名算法以及相关应用场景进行了简单介绍，第三、四节则详细介绍了FISCO BCOS隐私保护模块启用方法以及调用方式。

5.21.1 同态加密

算法简介

同态加密(Homomorphic Encryption)是公钥密码系统领域的明珠之一，已有四十余年的研究历史。其绝妙的密码特性，吸引密码学家前赴后继，在业界也受到了广泛的关注。

- 同态加密本质是一种公钥加密算法，即加密使用公钥pk，解密使用私钥sk；
- 同态加密支持密文计算，即由相同公钥加密生成的密文可以计算f( )操作，生成的新密文解密后恰好等于两个原始明文计算f( )的结果；
- 同态加密公式描述如下：

$$
\begin{aligned}
C_1 &= Encrypt(m_1, pk) \\
C_2 &= Encrypt(m_2, pk) \\
C_3 &= Hom_{f()}(C_1, C_2, pk) \\
Decrypt(C_3, sk) &= f(m_1, m_2)
\end{aligned}
$$

FISCO BCOS采用的是paillier加密算法，支持加法同态。paillier的公私钥兼容主流的RSA加密算法，接入门槛低。同时paillier作为一种轻量级的同态加密算法，计算开销小易被业务系统接受。因此经过功能性和可用性的权衡，最终选定了paillier算法。

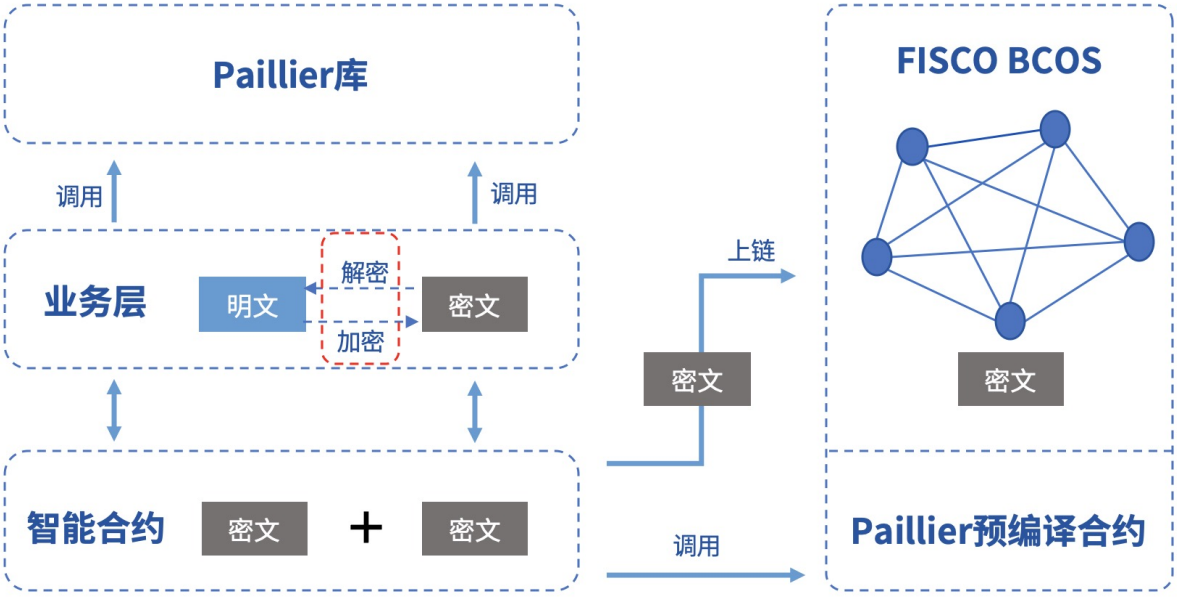
功能组件

FISCO BCOS同态加密模块提供的功能组件包括：

- paillier同态库，包括java库和c++同态接口。
- paillier预编译合约，供智能合约调用，提供密文同态运算接口。

使用方式

对于有隐私保护需求的业务，如果涉及简单密文计算，可借助本模块实现相关功能。凡是上链的数据可通过调用paillier库完成加密，链上的密文数据可通过调用paillier预编译合约实现密文的同态加运算，密文返还回业务层后，可通过调用paillier库完成解密，得到执行结果。具体流程如下图所示：



应用场景

在联盟链中，不同的业务场景需要配套不同的隐私保护策略。对于强隐私的业务，比如金融机构之间的对账，对资产数据进行加密是很有必要的。在FISCO BCOS中，用户可以调用同态加密库对数据进行加密，共识节点执行交易的时候调用同态加密预编译合约，得到密文计算的结果。

5.21.2 群/环签名

算法简介

群签名

群签名(Group Signature)是一种能保护签名者身份的具有相对匿名性的数字签名方案，用户可以代替自己所在的群对消息进行签名，而验证者可以验证该签名是否有效，但是并不知道签名属于哪一个群成员。同时，用户无法滥用这种匿名行为，因为群管理员可以通过群主私钥打开签名，暴露签名的归属信息。群签名的特性包括：

- 匿名性：群成员用群参数产生签名，其他人仅可验证签名的有效性，并通过签名知道签名者所属群组，却无法获取签名者身份信息；
- 不可伪造性：只有群成员才能生成有效可被验证的群签名；
- 不可链接性：给定两个签名，无法判断它们是否来自同一个签名者；
- 可追踪性：在监管介入的场景中，群主可通过签名获取签名者身份。

环签名

环签名(Ring Signature)是一种特殊的群签名方案，但具备完全匿名性，即不存在管理员这个角色，所有成员可主动加入环，且签名无法被打开。环签名的特性包括：

- 不可伪造性：环中其他成员不能伪造真实签名者签名；
- 完全匿名性：没有群主，只有环成员，其他人仅可验证环签名的有效性，但没有人可以获取签名者身份信息。

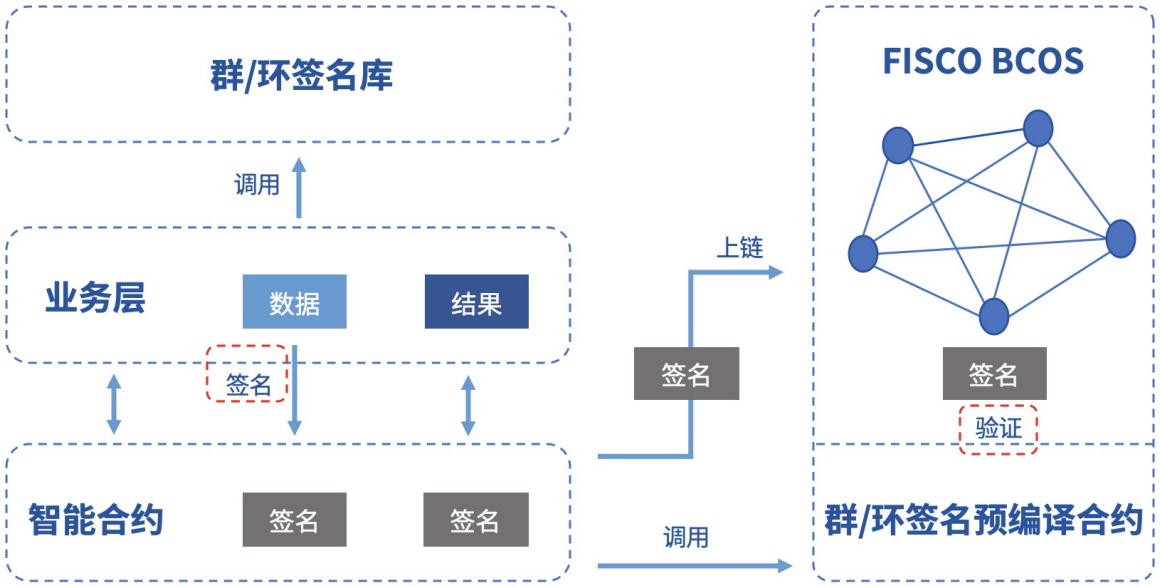
功能组件

FISCO BCOS群/环签名模块提供的功能组件包括：

- 群/环签名库，提供完整的群/环签名算法c++接口
- 群/环签名预编译合约，供智能合约调用，提供群/环签名验证接口。

使用方式

有签名者身份隐匿需求的业务可借助本模块实现相关功能。签名者通过调用群/环签名库完成对数据的签名，然后将签名上链，业务合约通过调用群/环签名预编译合约完成签名的验证，并将验证结果返还回业务层。如果是群签名，那么监管方还能打开指定签名数据，获得签名者身份。具体流程如下图所示：



应用场景

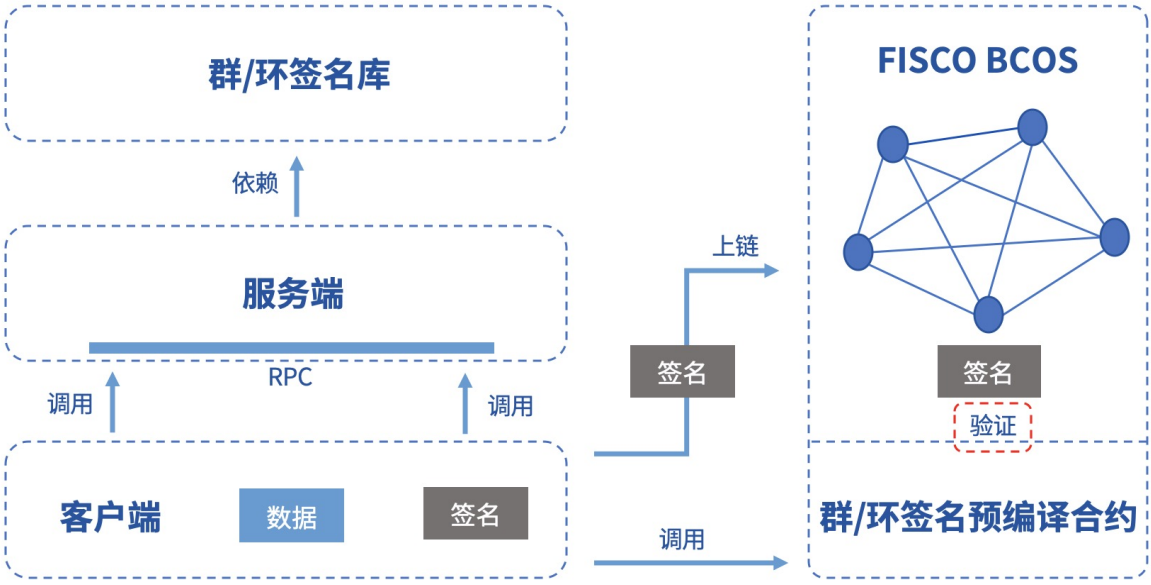
群/环签名由于其天然的匿名性，在需要对参与者身份进行隐匿的场景中有广泛的应用前景，例如匿名投票、匿名竞拍、匿名拍卖等等，甚至在区块链UTXO模型中可用于实现匿名转账。同时，由于群签名具备可追踪性，可以用于需要监管介入的场景，监管方作为群主或者委托群主揭露签名者身份。

开发示例

FISCO BCOS专门为用户提供了群/环签名开发示例，包括：

- 群/环签名服务端: 提供完整的群/环签名RPC服务。
- 群/环签名客户端: 调用RPC服务对数据进行签名，并提供签名上链以及链上验证等功能。

示例框架如下图所示，具体操作方法请参阅客户端指南。



5.21.3 启用方法

The FISCO BCOS privacy protection module is implemented through a precompiled contract and is enabled by default. For source code compilation FISCO BCOS, please refer to here.

搭建联盟链

假设当前位于`FISCO-BCOS/build`目录下，则使用下面的指令搭建本机4节点的链指令如下，更多选项参考这里。

```
bash ../tools/build_chain.sh -l "127.0.0.1:4" -e bin/fisco-bcos
```

5.21.4 预编译合约接口

隐私模块的代码和用户开发的预编译合约都位于`FISCO-BCOS/libprecompiled/extension`目录，因此隐私模块的调用方式和用户开发的预编译合约调用流程相同，不过有两点需要注意：

1.已为隐私模块的预编译合约分配了地址，无需另行注册。隐私模块实现的预编译合约列表以及地址分配如下：

2.需要通过`solidity`合约方式声明隐私模块预编译合约的接口，合约文件需保存在控制台合约目录`console/contracts/solidity`中，各个隐私功能的合约接口如下，可直接复制使用：

- 同态加密

```
// PaillierPrecompiled.sol
pragma solidity ^0.4.24;
contract PaillierPrecompiled{
    function paillierAdd(string cipher1, string cipher2) public constant␣
↪returns(string);
}
```

- 群签名

```
// GroupSigPrecompiled.sol
pragma solidity ^0.4.24;
contract GroupSigPrecompiled{
                function groupSigVerify(string signature, string message,␣
↪string gpkInfo, string paramInfo) public constant returns(bool);
}
```

- 环签名

```
// RingSigPrecompiled.sol
pragma solidity ^0.4.24;
contract RingSigPrecompiled{
     function ringSigVerify(string signature, string message, string␣
↪paramInfo) public constant returns(bool);
}
```

5.21.5 控制台调用

使用新编译出的二进制搭建节点后，部署控制台v1.0.2以上版本，将预编译合约接口声明文件拷贝到控制台合约目录。以调用同态加密为例，命令如下:

```
# 在console目录下启动控制台
bash start.sh

# 调用合约
call PaillierPrecompiled.sol 0x5003 paillierAdd
↪"0100E97E06A781DAAE6DBC9C094FC963D73B340D99FD934782A5D629E094D3B051FBBEA26F46BB681EB5314AE98A6A
↪"
↪"0100E97E06A781DAAE6DBC9C094FC963D73B340D99FD934782A5D629E094D3B051FBBEA26F46BB681EB5314AE98A6A
↪"

# 返回结果
0100E97E06A781DAAE6DBC9C094FC963D73B340D99FD934782A5D629E094D3B051FBBEA26F46BB681EB5314AE98A6A638
```

注： 控制台输入的密文可通过paillier库中的java库生成。

5.21.6 solidity合约调用

以调用同态加密为例，通过在solidity合约中创建预编译合约对象并调用其接口，在控制台`console/contracts/solidity`创建`CallPaillier.sol`文件，文件内容如下:

```
// CallPaillier.sol
pragma solidity ^0.4.24;
import "./PaillierPrecompiled.sol";

contract CallPaillier {
    PaillierPrecompiled paillier;
    function CallPaillier() {
        // 调用PaillierPrecompiled预编译合约
        paillier = PaillierPrecompiled(0x5003);
    }
    function add(string cipher1, string cipher2) public constant returns(string) {
        return paillier.paillierAdd(cipher1, cipher2);
    }
}
```

部署`CallPaillier`合约，然后调用`CallPaillier`合约的`add`接口，使用上面的密文作为输入，可以得到相同的结果。

5.22 国内镜像和CDN加速攻略

本节为访问GitHub较慢的用户提供国内镜像下载地址，以及CDN加速访问介绍。

5.22.1 FISCO BCOS源码与二进制程序

源码同步

FISCO BCOS当前所有仓库源码位于https://github.com/FISCO-BCOS/FISCO-BCOS，每个新的版本发布会将代码合入master分支。

为了方便国内用户，我们同样在gitee上提供了镜像仓库https://gitee.com/FISCO-BCOS/FISCO-BCOS，每次新版本发布后，镜像仓库会同步GitHub上官方仓库的更新，如果从GitHub下载失败，请尝试使用gitee镜像仓库。

二进制程序

FISCO BCOS每个新版本发布会在GitHub的tag中提供对应的二进制程序和部署工具，同时在官网https://www.fisco.com.cn也会提供同样的二进制程序，官网通过CDN加速下载。当前所提供的二进制程序包括：

1. fisco-bcos.tar.gz： 静态二进制程序，支持CentOS 7 和Ubuntu 16.04以上版本
2. fisco-bcos-gm.tar.gz： 国密版本的静态二进制程序，支持CentOS 7 和Ubuntu 16.04以上版本
3. build_chain.sh ： 对应版本的开发部署工具，依赖openssl和curl，支持CentOS 7/Ubuntu 16.04以上/macOS 10.15以上版本
4. fisco-bcos-macOS.tar.gz： 对应macOS系统的二进制程序

用户使用开发部署工具(build_chain)，工具先尝试从GitHub下载所需要的二进制程序，如果下载失败则尝试从官网下载。

用户运维部署工具(generator)的时候，工具默认从GitHub下载所需要的二进制程序，可以通过–cdn参数指定从官网下载。例如`./generator --download_fisco ./meta --cdn`

5.22.2 FISCO BCOS文档

FISCO BCOS文档使用readthedocs管理，全部开源于https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/，同样提供国内镜像http://www.fisco-bcos.org/developer/。

每个版本发布会为上个版本的文档打Tag，新版本的文档会合入主干分支，文档由于会持续改进，所以是下个版本发布才打上个版本的tag。readthedocs文档支持下载PDF格式，方便用户使用。

5.22.3 FISCO BCOS配套工具

控制台

FISCO BCOS控制台是一个交互式命令行工具，使用Java开发，代码位于https://github.com/FISCO-BCOS/console，国内镜像https://gitee.com/FISCO-BCOS/console。

控制台每个版本发布会提供编译好的包，用户下载后配置后即可使用，为了下载控制台用户需要获取download_console.sh脚本。此脚本会从GitHub下载最新版本console.tar.gz，如果下载失败则尝试从官网CDN下载。下面的指令从国内镜像获取download_console.sh脚本并执行。

```
curl -LO https://gitee.com/FISCO-BCOS/console/raw/master/tools/download_console.sh␣
↪&& bash download_console.sh
```

国密jar包官网CDN下载

控制台可以通过替换solcJ包，实现对solidity 0.4/0.5的国密和非国密版本支持。参考这里https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/console.html#jar，下面提供官网CDN下载地址。

- 0.4版本合约编译jar包

```
curl -LO https://www.fisco.com.cn/cdn/deps/tools/solcj/solcJ-all-0.4.25.jar
```

- 0.5版本合约编译jar包

```
curl -LO https://www.fisco.com.cn/cdn/deps/tools/solcj/solcJ-all-0.5.2.jar
```

- 国密0.4版本合约编译jar包

```
curl -LO https://www.fisco.com.cn/cdn/deps/tools/solcj/solcJ-all-0.4.25-gm.jar
```

- 国密0.5版本合约编译jar包

```
curl -LO https://www.fisco.com.cn/cdn/deps/tools/solcj/solcJ-all-0.5.2-gm.jar
```

TASSL

FISCO BCOS国密版本需要使用TASSL生成国密版本的证书，部署工具会自动从GitHub下载，解压后放置于~/.fisco/tassl，如果碰到下载失败，请尝试从https://gitee.com/FISCO-BCOS/LargeFiles/blob/master/tools/tassl.tar.gz下载并解压后，放置于~/.fisco/tassl

账户生成脚本

FISCO BCOS在国密模式下使用sm2曲线和对应签名算法，在非国密场景使用secp256k1曲线和ecdsa签名算法。为方便用户提供了生成脚本，脚本生成私钥并以账户地址命名，支持PEM和PKCS12两种格式。详情请参考这里https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/account.html

get_account.sh脚本依赖于openssl指令，用于生成secp256k1私钥，如果从GitHub下载失败，可以尝试镜像地址 https://gitee.com/FISCO-BCOS/console/raw/master/tools/get_account.sh

get_gm_account.sh脚本用于生成sm2私钥，依赖于TASSL。如果从GitHub下载失败，可以尝试镜像地址 https://gitee.com/FISCO-BCOS/console/raw/master/tools/get_gm_account.sh

5.22.4 举例：使用国内镜像建链

本节以搭建2.4.0国密版本为例，使用国内镜像建链，非国密版本的操作类似，参考https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/installation.html

下载开发部署工具

```
curl -LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.4.0/build_
↪chain.sh
```

如果下载失败请尝试curl -LO https://gitee.com/FISCO-BCOS/FISCO-BCOS/raw/master/tools/build_chain.sh

下载二进制程序

开发部署工具（build_chain）会自动下载二进制程序，下载失败自动切换官网CDN，不需要用户关注。用户也可以手动下载二进制程序或编译源码，通过开发部署工具的-e选项指定，此时工具不会再去下载。-e选项参考https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/build_chain.html#e-optional

搭建2.4.0国密FISCO BCOS链

搭建国密版本时，开发部署工具还依赖tassl，工具会自动下载，如果失败请用户参考TASSL手动下载方法，下载解压后放置于~/.fisco/tassl。执行下面的指令，输出All completed即表示执行成功。

```
bash build_chain.sh -l "127.0.0.1:4" -p 30300,20200,8545 -g -v 2.4.0
```

5.22.5 举例：使用国内源码镜像编译

本节以CentOS 7 为例，从gitee镜像下载源码并编译，其他操作系统编译流程类似，请参考https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/get_executable.html#id2

安装依赖

```
sudo yum install -y epel-release
sudo yum install -y openssl-devel openssl cmake3 gcc-c++ git
```

下载源码

```
git clone https://gitee.com/FISCO-BCOS/FISCO-BCOS.git
```

下载依赖包

FISCO BCOS在编译时会自动下载依赖包，每个依赖包有多个源。如果在编译阶段下载依赖包失败，请根据提示从下面的国内镜像手动下载，放置于FISCO-BCOS/deps/src目录下，再次make

https://gitee.com/FISCO-BCOS/LargeFiles/tree/master/libs

编译源码

```
cd FISCO-BCOS && mkdir build && cd build
cmake3 ..
make -j2
```

区块链部署

- 获取可执行程序
  - 下载二进制、docker镜像或手动编译
- 开发部署工具
  - 脚本选项、生成的节点目录结构
- 证书说明
  - 证书格式、证书对应角色、证书生成流程

- 配置文件与配置项
  - 节点所有的配置文件的详细说明
- 多群组部署
  - 多群组架构的配置指导
- 分布式存储
  - 分布式存储的配置指导

外部调用

- 控制台
  - 详细的控制台配置和使用说明
- 账户管理
  - 生成账户、用特定账户操作链
- SDK
  - 外部应用调用区块链上的智能合约
- 链上信使协议
  - 多个SDK间的消息相互推送

合约开发

- 智能合约开发
  - 普通智能合约开发、预编译合约开发合约开发
- 并行合约
  - 写并行合约，满足高并发场景

管理与安全

- 组员管理
  - 节点加入、退出群组
- 权限控制
  - 限制用户对区块链的操作
- CA黑白名单
  - 通过配置拒绝与不安全的节点建立连接
- 存储安全
  - 落盘加密，对节点存储的数据进行加密
- 隐私保护
  - 预编译合约支持同态加密、群/环签名验证

其它

- 国密支持
  - 国密版本的节点、SDK

- 日志说明
  - 节点日志格式和说明
- Caliper压力测试指南
  - Caliper压力测试指南

重要:

- 核心特性
  - 多群组部署
  - 并行合约
  - 分布式存储

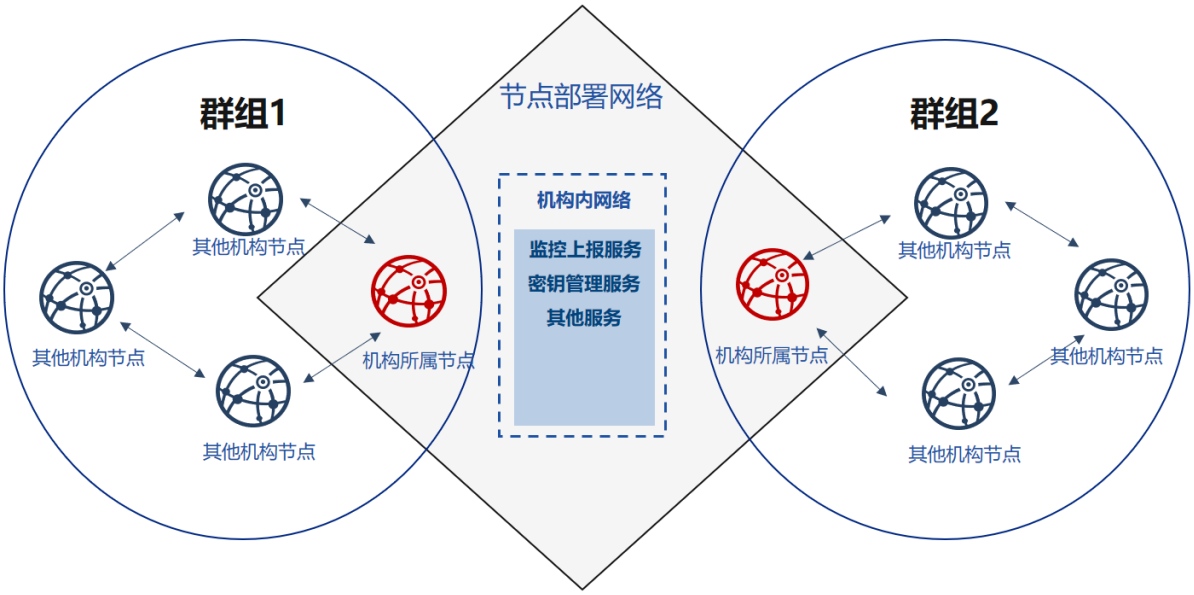
CHAPTER 6

运维部署工具

基本介绍

FISCO BCOS generator为企业用户提供了部署、管理和监控多机构多群组联盟链的便捷工具。

- 本工具降低了机构间生成与维护区块链的复杂度，提供了多种常用的部署方式。
- 本工具考虑了机构间节点安全性需求，所有机构间仅需要共享节点的证书，同时对应节点的私钥由各机构自己维护，不需要向机构外节点透露。
- 本工具考虑了机构间节点的对等性需求，多机构间可以通过交换数字证书对等安全地部署自己的节点。



设计背景

在联盟链中，多个对等机构是不完全信任的。联盟链的节点之间需要使用数字证书互相进行身份认证。

证书是机构对外身份的凭证，生成证书的过程中需要使用机构本身的公钥和私钥对。私钥即为机构在互联网上的身份信息，是私密的，不可对外告诉其他人的。节点在启动、运行过程中，需要使用私钥对数据包进行签名，从而完成身份认证过程。假设私钥泄露，则任何人都可以伪装成对应的机构，在不经过该机构授权行使该机构的权利。

重要: 即在联盟链部署、运行过程中，机构节点的私钥是不应该告诉任何人，应当只能由本机构生成和保管。

在FISCO BCOS的群组初始化过程中，需要多个节点协商生成群组的创世区块。创世区块在同一个群组中是唯一的，其中包含了初始节点身份信息的区块。这些身份信息需要通过交换数字证书的方式来构建。

现有的联盟链运维管理工具在初始化时都没有考虑联盟链间多个企业地位对等安全的诉求。联盟链在初始化时，需要协商创世节点中包含的节点信息。因此谁来生成这些信息就显得十分重要。现有做法为某一机构生成自己的节点信息，启动区块链，再加入其它机构的节点；或是由权威第三方机构直接生成所有机构内的节点信息，并将节点配置文件夹发送给各机构。

另一方面，FISCO BCOS 2.0引入了隐私性和可扩展性更强的多群组架构。在群组架构下，群组间数据、交易相互隔离，每个群组运行独立的共识算法，可满足区块链场景中的隐私保护需求。

在上述模式中，总有一个机构会优先加入到联盟链之中；并且在这种模式中，总有一个机构会获得所有节点的私钥。

如何保证企业间如何对等、安全、隐私地新建群组。新建群组之后如何保证节点可靠，有效的运行；群组账本的隐私性和安全性，以及企业建立群组、使用群组操作的隐私性都需要一个有效的方式来保证。

设计思路

FISCO BCOS generator从上述背景出发，根据灵活、安全、易用、对等的原则，从不同机构对等部署、新建群组的角度考虑，设计了解决上述问题的解决方案。

灵活:

- 无需安装即可使用
- 支持多种部署上报方式
- 支持多种架构改动

安全:

- 支持多种架构改动
- 节点私钥不出内网
- 机构间只需协商证书

易用:

- 支持多种组网模式
- 多种命令满足不同需求
- 监控审计脚本

对等:

- 机构地位对等
- 所有机构共同产生创世区块
- 机构对等管理所属群组

针对同一根证书的联盟链，本工具可以快速配置链内的多个群组，满足不同企业的不同业务需求。

不同机构间通过协商节点证书、IP、端口号等数据的模式，填写配置项，每个机构都可以在本地生成不含节点私钥的节点配置文件夹，节点的私钥可以不出内网，即使节点配置文件丢失，防止恶意攻击者伪装节点的同时，不会泄露链上任何信息。使用这种方式，在保证节点可用的同时，保护节点的安全性。

用户通过协商生成创世区块，生成节点配置文件夹后，启动节点，节点会根据用户配置信息进行多群组组网。

6.1 一键部署

one_click_generator.sh脚本为根据用户填写的节点配置，一键部署联盟链的脚本。脚本会根据用户指定文件夹下配置的node_deployment.ini，在文件夹下生成相应的节点。

本章主要以部署**3机构2群组6节点**的组网模式，为用户讲解单机构一键部署运维部署工具的使用方法。

本教程适用于单机构搭建所有节点的部署方式，运维部署工具多机构部署教程可以参考使用运维部署工具。

重要: 一键部署脚本使用时需要确保当前meta文件夹下不含节点证书信息，请清空meta文件夹。

6.1.1 下载安装

下载

```
cd ~/ && git clone https://github.com/FISCO-BCOS/generator.git
```

安装

此操作要求用户具有sudo权限。

```
cd ~/generator && bash ./scripts/install.sh
```

检查是否安装成功，若成功，输出 usage: generator xxx

```
./generator -h
```

获取节点二进制

拉取最新fisco-bcos二进制文件到meta中

```
./generator --download_fisco ./meta
```

检查二进制版本

若成功，输出 FISCO-BCOS Version : x.x.x-x

```
./meta/fisco-bcos -v
```

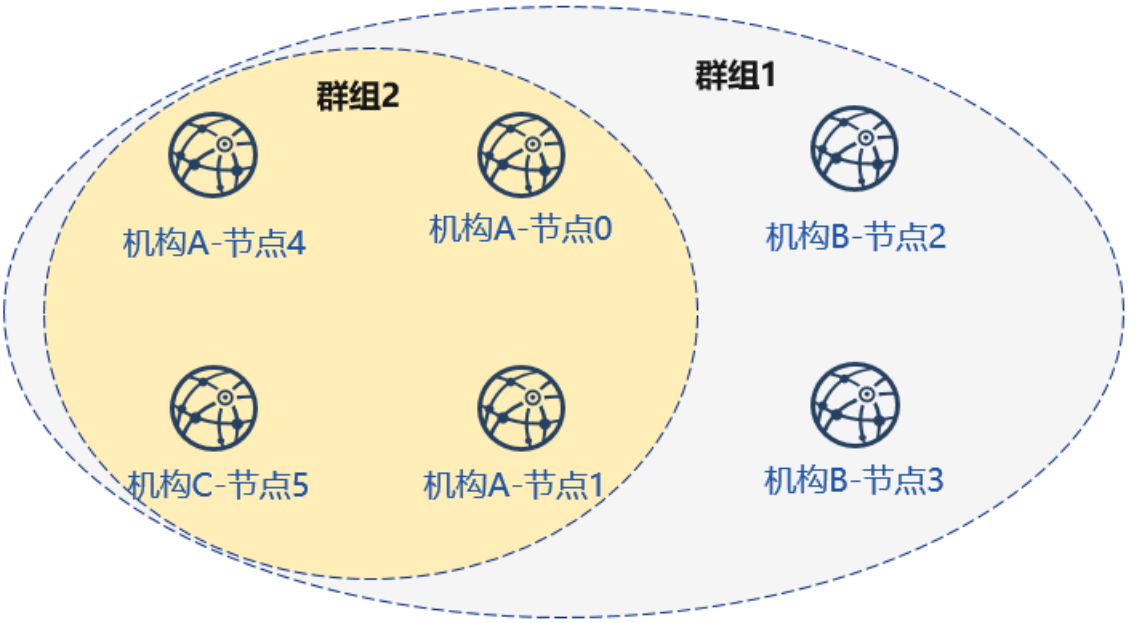
PS：源码编译节点二进制的用户，只需要用编译出来的二进制替换掉meta文件夹下的二进制即可。

6.1.2 背景介绍

本节以部署6节点3机构2群组的组网模式，演示如何使用运维部署工具一键部署功能，搭建区块链。

节点组网拓扑结构

一个如图所示的6节点3机构2群组的组网模式。机构B和机构C分别位于群组1和群组2中。机构A同属于群组1和群组2中。



机器环境

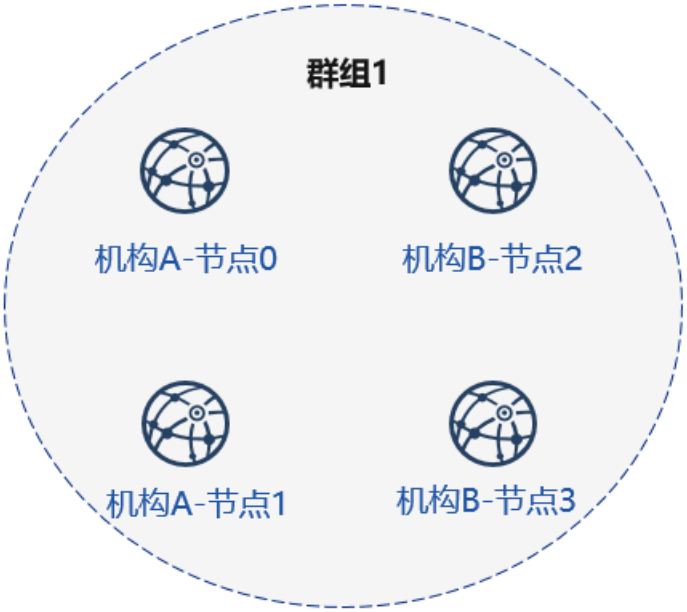
每个节点的IP，端口号为如下：

注解：

- 云主机的公网IP均为虚拟IP，若rpc_ip/p2p_ip/channel_ip填写外网IP，会绑定失败，须填写0.0.0.0
- RPC/P2P/Channel监听端口必须位于1024-65535范围内，且不能与机器上其他应用监听端口冲突
- 出于安全性和易用性考虑，FISCO BCOS v2.3.0版本最新节点config.ini配置将listen_ip拆分成jsonrpc_listen_ip和channel_listen_ip，但仍保留对listen_ip的解析功能，详细请参考 这里
- 为便于开发和体验，channel_listen_ip参考配置是 *0.0.0.0* ，出于安全考虑，请根据实际业务网络情况，修改为安全的监听地址，如：内网IP或特定的外网IP

6.1.3 部署网络

首先完成如图所示机构A、B搭建群组1的操作：



使用前用户需准备如图如tmp_one_click的文件夹，在文件夹下分别拥有不同机构的目录，每个机构目录下需要有对应的配置文件node_deployment.ini。使用前需要保证generator的meta文件夹没有进行过任何操作。

查看一键部署模板文件夹：

```
cd ~/generator
ls ./tmp_one_click
```

```
# 参数解释
# 如需多个机构，需要手动创建该文件夹
tmp_one_click # 用户指定进行一键部署操作的文件夹
├── agencyA # 机构A目录，命令执行后会在该目录下生成机构A的节点及相关文件
│   └── node_deployment.ini # 机构A节点配置文件，一键部署命令会根据该文件生成相应节点
└── agencyB # 机构B目录，命令执行后会在该目录下生成机构B的节点及相关文件
    └── node_deployment.ini # 机构B节点配置文件，一键部署命令会根据该文件生成相应节点
```

机构填写节点信息

教程中将配置文件放置与tmp_one_click文件夹下的agencyA, agencyB下

```
cat > ./tmp_one_click/agencyA/node_deployment.ini << EOF
[group]
group_id=1

[node0]
; Host IP for the communication among peers.
; Please use your ssh login IP.
p2p_ip=127.0.0.1
; listening IP for the communication between SDK clients.
; This IP is the same as p2p_ip for the physical host.
; But for virtual host e.g., VPS servers, it is usually different from p2p_ip.
; You can check accessible addresses of your network card.
; Please see https://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30300
```

(continues on next page)

(续上页)

```
channel_listen_port=20200
jsonrpc_listen_port=8545

[node1]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30301
channel_listen_port=20201
jsonrpc_listen_port=8546
EOF
```

```
cat > ./tmp_one_click/agencyB/node_deployment.ini << EOF
[group]
group_id=1

[node0]
; Host IP for the communication among peers.
; Please use your ssh login IP.
p2p_ip=127.0.0.1
; listening IP for the communication between SDK clients.
; This IP is the same as p2p_ip for the physical host.
; But for virtual host e.g., VPS servers, it is usually different from p2p_ip.
; You can check accessible addresses of your network card.
; Please see https://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30302
channel_listen_port=20202
jsonrpc_listen_port=8547

[node1]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30303
channel_listen_port=20203
jsonrpc_listen_port=8548
EOF
```

生成节点

```
bash ./one_click_generator.sh -b ./tmp_one_click
```

执行完毕后，./tmp_one_click文件夹结构如下：

查看执行后的一键部署模板文件夹：

```
ls ./tmp_one_click
```

```
├── agencyA # A机构文件夹
│   ├── agency_cert # A机构证书及私钥
│   ├── generator-agency # 自动代替A机构进行操作的generator文件夹
│   ├── node # A机构生成的节点，多机部署时推送至对应服务器即可
│   ├── node_deployment.ini # A机构的节点配置信息
│   └── sdk # A机构的sdk或控制台配置文件
├── agencyB
|   ├── agency_cert
```

(continues on next page)

(续上页)

```
|    ├── generator-agency
|    ├── node
|    ├── node_deployment.ini
|    └── sdk
|── ca.crt # 链证书
|── ca.key # 链私钥
|── group.1.genesis # 群组1创世区块
|── peers.txt # 节点的peers.txt信息
```

启动节点

调用脚本启动节点：

```
bash ./tmp_one_click/agencyA/node/start_all.sh
```

```
bash ./tmp_one_click/agencyB/node/start_all.sh
```

查看节点进程：

```
ps -ef | grep fisco
```

```
# 命令解释
# 可以看到如下进程
fisco  15347     1  0 17:22 pts/2    00:00:00 ~/generator/tmp_one_click/agencyA/
↪node/node_127.0.0.1_30300/fisco-bcos -c config.ini
fisco  15402     1  0 17:22 pts/2    00:00:00 ~/generator/tmp_one_click/agencyA/
↪node/node_127.0.0.1_30301/fisco-bcos -c config.ini
fisco  15442     1  0 17:22 pts/2    00:00:00 ~/generator/tmp_one_click/agencyB/
↪node/node_127.0.0.1_30302/fisco-bcos -c config.ini
fisco  15456     1  0 17:22 pts/2    00:00:00 ~/generator/tmp_one_click/agencyB/
↪node/node_127.0.0.1_30303/fisco-bcos -c config.ini
```

查看节点运行状态

查看节点log：

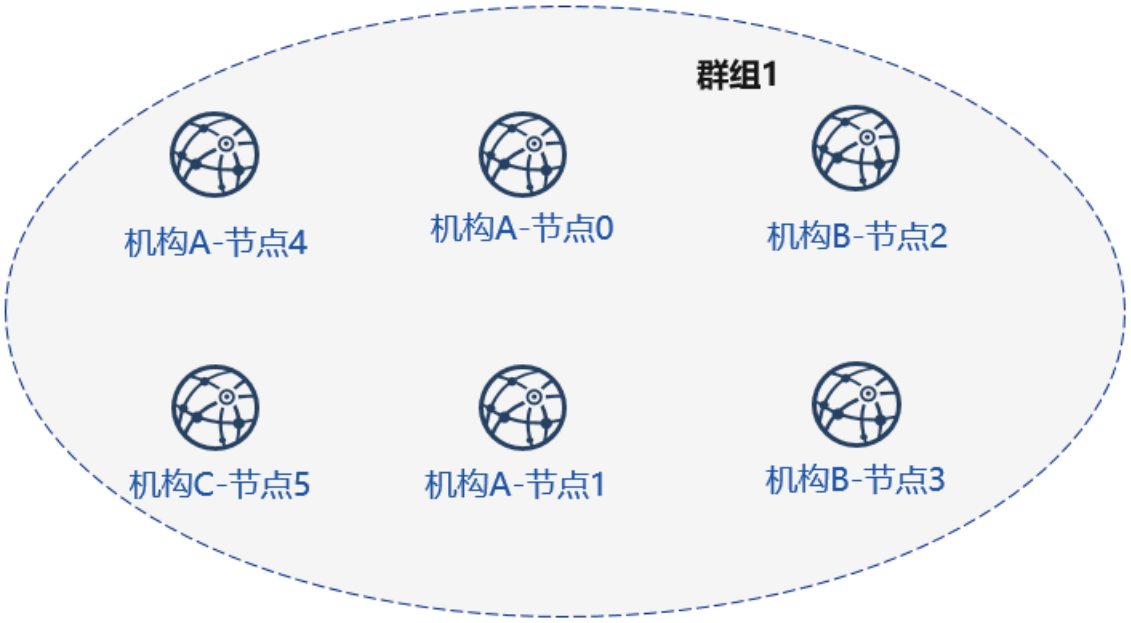
```
tail -f ~/generator/tmp_one_click/agency*/node/node*/log/log*  | grep +++
```

```
# 命令解释
# +++即为节点正常共识
info|2019-02-25 17:25:56.028692| [g:1][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=833bd983...
info|2019-02-25 17:25:59.058625| [g:1][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=343b1141...
info|2019-02-25 17:25:57.038284| [g:1][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=1,hash=ea85c27b...
```

6.1.4 新增节点 (扩容新节点)

重要： 一键部署脚本使用时需要确保当前meta文件夹下不含节点证书信息，请清空meta文件夹。

接下来，我们为机构A和机构C增加新节点，完成下图所示的组网：



初始化扩容配置

创建扩容文件夹，示例中tmp_one_click_expand可以为任意名称，请每次扩容使用新的文件夹

```
mkdir ~/generator/tmp_one_click_expand/
```

拷贝链证书及私钥至扩容文件夹

```
cp  ~/generator/tmp_one_click/ca.* ~/generator/tmp_one_click_expand/
```

拷贝群组1创世区块group.1.genesis至扩容文件夹

```
cp  ~/generator/tmp_one_click/group.1.genesis ~/generator/tmp_one_click_expand/
```

拷贝群组1节点P2P连接文件peers.txt至扩容文件夹

```
cp  ~/generator/tmp_one_click/peers.txt ~/generator/tmp_one_click_expand/
```

机构A配置节点信息

创建机构A扩容节点所在目录

```
mkdir ~/generator/tmp_one_click_expand/agencyA
```

此时机构A已经存在联盟链中，因此需拷贝机构A证书、私钥至对应文件夹

```
cp -r ~/generator/tmp_one_click/agencyA/agency_cert ~/generator/tmp_one_click_
↪expand/agencyA
```

机构A填写节点配置信息

```
cat > ./tmp_one_click_expand/agencyA/node_deployment.ini << EOF
[group]
group_id=1

[node0]
```

(continues on next page)

(续上页)

```
; Host IP for the communication among peers.
; Please use your ssh login IP.
p2p_ip=127.0.0.1
; listening IP for the communication between SDK clients.
; This IP is the same as p2p_ip for the physical host.
; But for virtual host e.g., VPS servers, it is usually different from p2p_ip.
; You can check accessible addresses of your network card.
; Please see https://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30304
channel_listen_port=20204
jsonrpc_listen_port=8549
EOF
```

机构C配置节点信息

创建机构C扩容节点所在目录

```
mkdir ~/generator/tmp_one_click_expand/agencyC
```

机构C填写节点配置信息

```
cat > ./tmp_one_click_expand/agencyC/node_deployment.ini << EOF
[group]
group_id=1

[node0]
; Host IP for the communication among peers.
; Please use your ssh login IP.
p2p_ip=127.0.0.1
; listening IP for the communication between SDK clients.
; This IP is the same as p2p_ip for the physical host.
; But for virtual host e.g., VPS servers, it is usually different from p2p_ip.
; You can check accessible addresses of your network card.
; Please see https://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30305
channel_listen_port=20205
jsonrpc_listen_port=8550
EOF
```

生成扩容节点

```
bash ./one_click_generator.sh -e ./tmp_one_click_expand
```

启动新节点

调用脚本启动节点：

```
bash ./tmp_one_click_expand/agencyA/node/start_all.sh
```

```
bash ./tmp_one_click_expand/agencyC/node/start_all.sh
```

查看节点进程：

```
ps -ef | grep fisco
```

```
# 命令解释
# 可以看到如下进程
fisco  15347     1  0 17:22 pts/2    00:00:00 ~/generator/tmp_one_click/agencyA/
↪node/node_127.0.0.1_30300/fisco-bcos -c config.ini
fisco  15402     1  0 17:22 pts/2    00:00:00 ~/generator/tmp_one_click/agencyA/
↪node/node_127.0.0.1_30301/fisco-bcos -c config.ini
fisco  15403     1  0 17:22 pts/2    00:00:00 ~/generator/tmp_one_click_expand/
↪agencyA/node/node_127.0.0.1_30304/fisco-bcos -c config.ini
fisco  15442     1  0 17:22 pts/2    00:00:00 ~/generator/tmp_one_click/agencyB/
↪node/node_127.0.0.1_30302/fisco-bcos -c config.ini
fisco  15456     1  0 17:22 pts/2    00:00:00 ~/generator/tmp_one_click/agencyB/
↪node/node_127.0.0.1_30303/fisco-bcos -c config.ini
fisco  15466     1  0 17:22 pts/2    00:00:00 ~/generator/tmp_one_click_expand/
↪agencyC/node/node_127.0.0.1_30305/fisco-bcos -c config.ini
```

重要： 为群组1扩容的新节点需要使用sdk或控制台加入到群组中。

使用控制台注册节点

由于控制台体积较大，一键部署中没有直接集成，用户可以使用以下命令获取控制台

注解： 企业部署工具会根据generator/meta文件夹下的机构证书及私钥生成sdk相应证书，如需手动生成可以参考操作手册中的generate_sdk_certificate命令

获取控制台，可能需要较长时间，国内用户可以使用--cdn命令：

以机构A使用控制台为例，此步需要切换到机构A对应的generator-agency文件夹

```
cd ~/generator/tmp_one_click/agencyA/generator-agency
```

```
./generator --download_console ./ --cdn
```

查看机构A节点4

机构A使用控制台加入机构A节点4为共识节点，其中参数第二项需要替换为加入节点的nodeid，nodeid在节点文件夹的conf的node.nodeid文件

查看机构A节点nodeid：

```
cat ~/generator/tmp_one_click_expand/agencyA/node/node_127.0.0.1_30304/conf/node.
↪nodeid
```

```
# 命令解释
# 可以看到类似于如下nodeid，控制台使用时需要传入该参数
ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c9715
```

使用控制台注册共识节点

启动控制台：

```
cd ~/generator/tmp_one_click/agencyA/generator-agency/console && bash ./start.sh 1
```

使用控制台addSealer命令将节点注册为共识节点，此步需要用到cat命令查看得到机构A节点的node.nodeid：

```
addSealer␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
```

```
# 命令解释
# 执行成功会提示success
$ [group:1]> addSealer␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
        "code":0,
        "msg":"success"
}
```

退出控制台：

```
exit
```

查看机构C节点5

机构A使用控制台加入机构C的节点5为观察节点，其中参数第二项需要替换为加入节点的nodeid，nodeid在节点文件夹的conf的node.nodeid文件

查看机构C节点nodeid：

```
cat ~/generator/tmp_one_click_expand/agencyC/node/node_127.0.0.1_30305/conf/node.
↪nodeid
```

```
# 命令解释
# 可以看到类似于如下nodeid，控制台使用时需要传入该参数
5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa9a
```

使用控制台注册观察节点

启动控制台：

```
cd ~/generator/tmp_one_click/agencyA/generator-agency/console && bash ./start.sh 1
```

使用控制台addObserver命令将节点注册为观察节点，此步需要用到cat命令查看得到机构C节点的node.nodeid：

```
addObserver␣
↪5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa
```

```
# 命令解释
# 执行成功会提示success
$ [group:1]> addObserver␣
↪5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa
{
        "code":0,
        "msg":"success"
}
```

退出控制台：

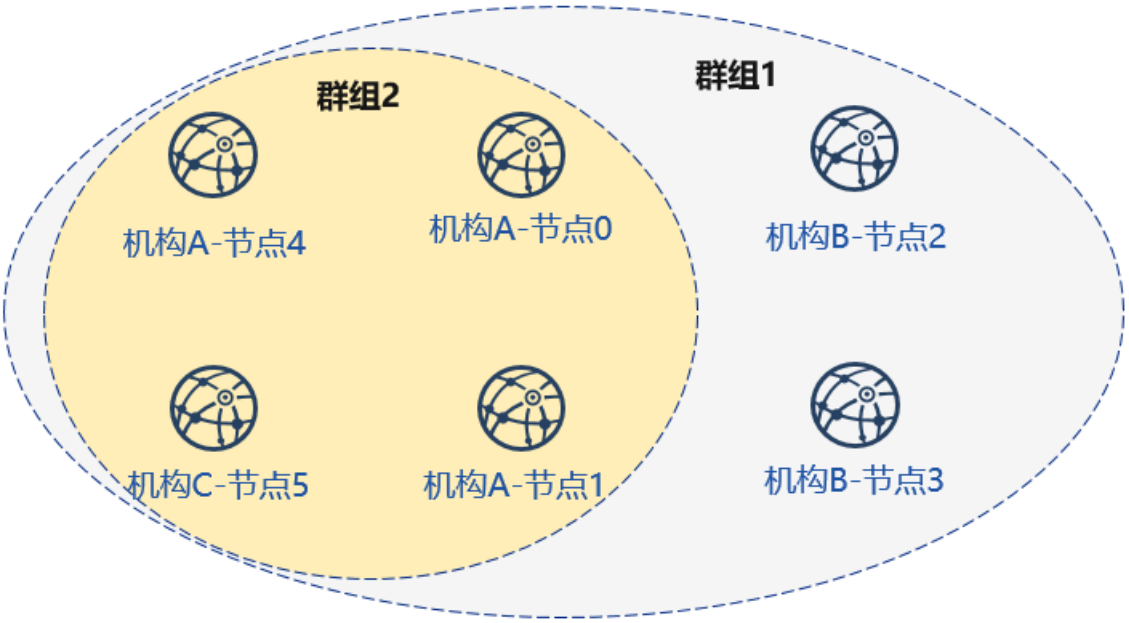
```
exit
```

至此，我们完成了新增节点至现有群组的操作。

6.1.5 新增群组 (扩容新群组)

新建群组的操作用户可以在执行one_click_generator.sh脚本的目录下，通过修改./conf/group_genesis.ini文件，并执行--create_group_genesis命令。

为如图4个节点生成群组2



配置群组2创世区块

重要： 此操作需要在和上述操作generator下执行。

```
cd ~/generator
```

配置群组创世区块文件，指定group_id为2。并在[node]下指定新群组中各个节点的IP和P2P端口，分别为机构A-节点0，机构A-节点1，机构A-节点4和机构C-节点5。

```
cat > ./conf/group_genesis.ini << EOF
[group]
group_id=2

[nodes]
node0=127.0.0.1:30300
node1=127.0.0.1:30301
node2=127.0.0.1:30304
node3=127.0.0.1:30305
EOF
```

获取对应节点证书

机构A-节点0（node0=127.0.0.1:30300）

```
cp ~/generator/tmp_one_click/agencyA/generator-agency/meta/cert_127.0.0.1_30300.
↪crt ~/generator/meta
```

机构A-节点1（node1=127.0.0.1:30301）

```
cp ~/generator/tmp_one_click/agencyA/generator-agency/meta/cert_127.0.0.1_30301.
↪crt ~/generator/meta
```

机构A-节点4（node2=127.0.0.1:30304）

```
cp ~/generator/tmp_one_click_expand/agencyA/generator-agency/meta/cert_127.0.0.1_
↪30304.crt ~/generator/meta
```

机构C-节点5（node3=127.0.0.1:30305）

```
cp ~/generator/tmp_one_click_expand/agencyC/generator-agency/meta/cert_127.0.0.1_
↪30305.crt ~/generator/meta
```

生成群组创世区块

```
./generator --create_group_genesis ./group2
```

将群组创世区块加入现有节点：

机构A-节点0（node0=127.0.0.1:30300）

```
./generator --add_group ./group2/group.2.genesis ./tmp_one_click/agencyA/node/node_
↪127.0.0.1_30300
```

机构A-节点1（node1=127.0.0.1:30301）

```
./generator --add_group ./group2/group.2.genesis ./tmp_one_click/agencyA/node/node_
↪127.0.0.1_30301
```

机构A-节点4（node2=127.0.0.1:30304）

```
 ./generator --add_group ./group2/group.2.genesis ./tmp_one_click_expand/agencyA/
↪node/node_127.0.0.1_30304
```

机构C-节点5（node3=127.0.0.1:30305）

```
./generator --add_group ./group2/group.2.genesis ./tmp_one_click_expand/agencyC/
↪node/node_127.0.0.1_30305
```

加载、启动新群组

节点在运行时，可直接用脚本load_new_groups.sh加载新群组配置，并调用startGroupRPC接口启动新群组：

机构A-节点0（node0=127.0.0.1:30300）

```
bash ./tmp_one_click/agencyA/node/node_127.0.0.1_30300/scripts/load_new_groups.sh
curl -X POST --data '{"jsonrpc":"2.0","method":"startGroup","params":[2],"id":1}'␣
↪http://127.0.0.1:8545
```

机构A-节点1（node1=127.0.0.1:30301）

```
bash ./tmp_one_click/agencyA/node/node_127.0.0.1_30301/scripts/load_new_groups.sh
curl -X POST --data '{"jsonrpc":"2.0","method":"startGroup","params":[2],"id":1}'␣
↪http://127.0.0.1:8546
```

机构A-节点4（node2=127.0.0.1:30304）

```
bash ./tmp_one_click_expand/agencyA/node/node_127.0.0.1_30304/scripts/load_new_
↪groups.sh
curl -X POST --data '{"jsonrpc":"2.0","method":"startGroup","params":[2],"id":1}'␣
↪http://127.0.0.1:8549
```

机构C-节点5（node3=127.0.0.1:30305）

```
bash ./tmp_one_click_expand/agencyC/node/node_127.0.0.1_30305/scripts/load_new_
↪groups.sh
curl -X POST --data '{"jsonrpc":"2.0","method":"startGroup","params":[2],"id":1}'␣
↪http://127.0.0.1:8550
```

查看节点

查看节点log内group1信息:

```
tail -f ~/generator/tmp_one_click/agency*/node/node*/log/log* | grep g:2 | grep +++
```

```
# 命令解释
# +++即为节点正常共识
info|2019-02-25 17:25:56.028692| [g:2][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=833bd983...
info|2019-02-25 17:25:59.058625| [g:2][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=343b1141...
info|2019-02-25 17:25:57.038284| [g:2][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=1,hash=ea85c27b...
```

至此 我们完成了所示构建教程中的所有操作。

注解: 使用完成后建议用以下命令对meta文件夹进行清理:

- rm ./meta/cert_*
- rm ./meta/group*

6.1.6 更多操作

更多操作，可以参考操作手册，或企业工具对等部署教程。

如果使用该教程遇到问题，请查看FAQ

6.2 使用运维部署工具

FISCO BCOS运维部署工具面向于真实的多机构生产环境。为了保证机构的密钥安全，运维部署工具提供了一种机构间相互合作部署联盟链方式。

本章以部署**6节点3机构2群组**的组网模式，演示运维部署工具的使用方法。更多参数选项说明请参考这里。

国密部署教程说明请参考使用运维部署工具部署国密区块链。

本章节为多机构对等部署的过程，适用于多机构部署，机构私钥不出内网的情况，由单机构一键生成所有机构节点配置文件的教程可以参考FISCO BCOS运维部署工具一键部署。

6.2.1 下载安装

下载

```
cd ~/ && git clone https://github.com/FISCO-BCOS/generator.git
```

安装

此操作要求用户具有sudo权限。

```
cd ~/generator && bash ./scripts/install.sh
```

检查是否安装成功，若成功，输出 usage: generator xxx

```
./generator -h
```

获取节点二进制

拉取最新fisco-bcos二进制文件到meta中

```
./generator --download_fisco ./meta
```

检查二进制版本

若成功，输出 FISCO-BCOS Version : x.x.x-x

```
./meta/fisco-bcos -v
```

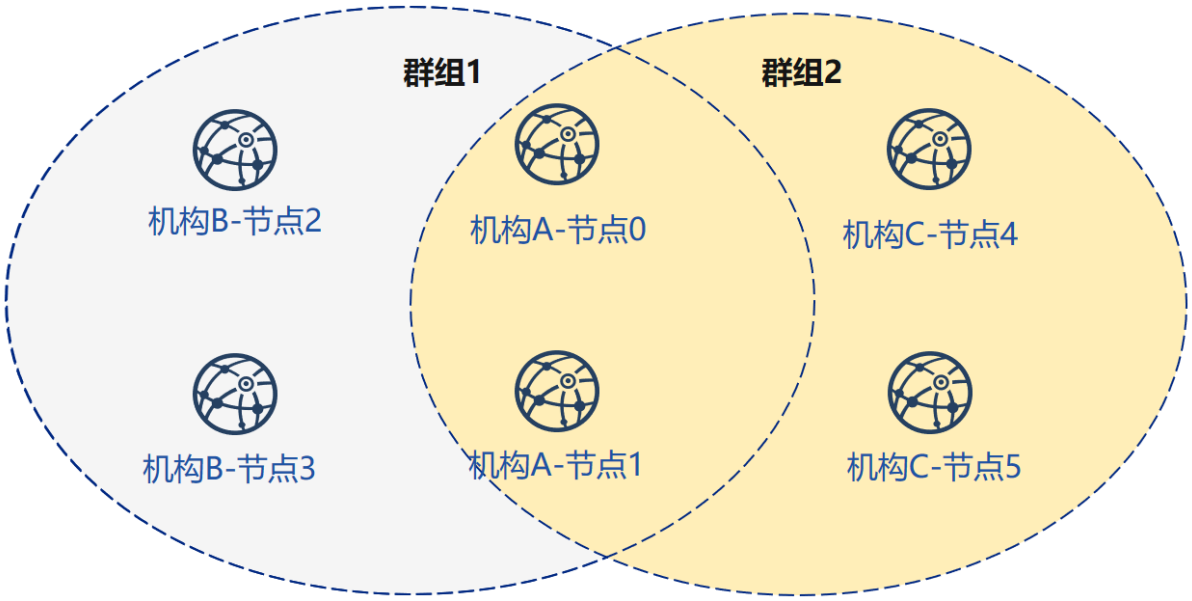
PS：源码编译节点二进制的用户，只需要用编译出来的二进制替换掉meta文件夹下的二进制即可。

6.2.2 典型示例

为了保证机构的密钥安全，运维部署工具提供了一种机构间相互合作的的搭链方式。本节以部署6节点3机构2群组的组网模式，演示企业间如何相互配合，搭建区块链。

节点组网拓扑结构

一个如图所示的6节点3机构2群组的组网模式。机构B和机构C分别位于群组1和群组2中。机构A同属于群组1和群组2中。



机器环境

每个节点的IP，端口号为如下：

注解:

- 云主机的公网IP均为虚拟IP，若rpc_ip/p2p_ip/channel_ip填写外网IP，会绑定失败，须填写0.0.0.0
- RPC/P2P/Channel监听端口必须位于1024-65535范围内，且不能与机器上其他应用监听端口冲突
- 出于安全性和易用性考虑，FISCO BCOS v2.3.0版本最新节点config.ini配置将listen_ip拆分成jsonrpc_listen_ip和channel_listen_ip，但仍保留对listen_ip的解析功能，详细请参考 这里
- 为便于开发和体验，channel_listen_ip参考配置是 *0.0.0.0* ，出于安全考虑，请根据实际业务网络情况，修改为安全的监听地址，如：内网IP或特定的外网IP

涉及机构

搭链操作涉及多个机构的合作，包括：

- 证书颁发机构
- 搭建节点的机构（简称“机构”）

关键流程

本流程简要的给出**证书颁发机构**，**节点机构间**如何相互配合搭建区块链。

一、初始化链证书

1. 证书颁发机构操作：
   - 生成链证书

二、生成群组1

1. 证书颁发机构操作：颁发机构证书
    - 生成机构证书
    - 发送证书
2. 机构间独立操作
    - 修改配置文件`node_deployment.ini`
    - 生成节点证书及节点P2P端口地址文件
3. 选取其中一个机构为群组生成创世块
    - 收集群组内所有节点证书
    - 修改配置文件`group_genesis.ini`
    - 为群组生成创世块文件
    - 分发创世块文件
4. 机构间独立操作：生成节点
    - 收集群组其他节点的P2P端口地址文件
    - 生成节点
    - 启动节点

三、初始化新机构

1. 证书颁发机构操作：颁发新机构证书
    - 生成机构证书
    - 发送证书

四、生成群组2

1. 新机构独立操作
    - 修改配置文件`node_deployment.ini`
    - 生成节点证书及节点P2P端口地址文件
2. 选取其中一个机构为群组生成创世块
    - 收集群组内所有节点证书
    - 修改配置文件`group_genesis.ini`
    - 为群组生成创世块文件
    - 分发创世块文件
3. 新机构独立操作：生成节点
    - 收集群组其他节点的P2P端口地址文件
    - 生成节点
    - 启动节点
4. 已有机构操作：配置新群组
    - 收集群组其他节点的P2P端口地址文件

- 配置新群组与新增节点的P2P端口地址
- 重启节点

五、现有节点加入群组1

1. 群组1原有机构操作:
   - 发送群组1创世区块至现有节点
   - 配置控制台
   - 获取加入节点nodeid
   - 使用控制台将节点加入群组1

6.2.3 联盟链初始化

为了操作简洁，本示例所有操作在同一台机器上进行，用不同的目录模拟不同的机构环境。用文件复制操作来模拟网络的发送。进行了教程中的下载安装后，请将generator复制到对应机构的generator目录中。

机构初始化

我们以**教程中下载的generator作为证书颁发机构**。

初始化机构A

```
cp -r ~/generator ~/generator-A
```

初始化机构B

```
cp -r ~/generator ~/generator-B
```

初始化链证书

在证书颁发机构上进行操作，一条联盟链拥有唯一的链证书ca.crt

用 `--generate_chain_certificate` 命令生成链证书

在证书生成机构目录下操作:

```
cd ~/generator
```

```
./generator --generate_chain_certificate ./dir_chain_ca
```

查看链证书及私钥:

```
ls ./dir_chain_ca
```

```
# 上述命令解释
# 从左至右分别为链证书、链私钥
ca.crt  ca.key
```

6.2.4 机构A、B构建群组1

初始化机构A

教程中为了简化操作直接生成了机构证书和私钥，实际应用时应该由机构本地生成私钥agency.key，再生成证书请求文件，向证书签发机构获取机构证书agency.crt。

在证书生成机构目录下操作:

```
cd ~/generator
```

生成机构A证书:

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyA
```

查看机构证书及私钥:

```
ls dir_agency_ca/agencyA/
```

```
# 上述命令解释
# 从左至右分别为机构证书、机构私钥、链证书
agency.crt  agency.key  ca.crt
```

发送链证书、机构证书、机构私钥至机构A，示例是通过文件拷贝的方式，从证书授权机构将机构证书发送给对应的机构，放到机构的工作目录的meta子目录下

```
cp ./dir_agency_ca/agencyA/* ~/generator-A/meta/
```

初始化机构B

在证书生成机构目录下操作:

```
cd ~/generator
```

生成机构B证书:

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyB
```

发送链证书、机构证书、机构私钥至机构B，示例是通过文件拷贝的方式，从证书授权机构将机构证书发送给对应的机构，放到机构的工作目录的meta子目录下

```
cp ./dir_agency_ca/agencyB/* ~/generator-B/meta/
```

重要: 一条联盟链中只能用到一个根证书ca.crt，多服务器部署时不要生成多个根证书和私钥。一个群组只能有一个群组创世区块group.x.genesis

---

机构A修改配置文件

node_deployment.ini为节点配置文件，运维部署工具会根据node_deployment.ini下的配置生成相关节点证书，及生成节点配置文件夹等。

机构A修改conf文件夹下的node_deployment.ini如下图所示:

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

```
cat > ./conf/node_deployment.ini << EOF
[group]
group_id=1

[node0]
; host ip for the communication among peers.
; Please use your ssh login ip.
p2p_ip=127.0.0.1
; listen ip for the communication between sdk clients.
; This ip is the same as p2p_ip for physical host.
; But for virtual host e.g. vps servers, it is usually different from p2p_ip.
; You can check accessible addresses of your network card.
; Please see https://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30300
channel_listen_port=20200
jsonrpc_listen_port=8545

[node1]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30301
channel_listen_port=20201
jsonrpc_listen_port=8546
EOF
```

机构B修改配置文件

机构B修改conf文件夹下的`node_deployment.ini`如下图所示:

在~/generator-B目录下执行下述命令

```
cd ~/generator-B
```

```
cat > ./conf/node_deployment.ini << EOF
[group]
group_id=1

[node0]
; host ip for the communication among peers.
; Please use your ssh login ip.
p2p_ip=127.0.0.1
; listen ip for the communication between sdk clients.
; This ip is the same as p2p_ip for physical host.
; But for virtual host e.g. vps servers, it is usually different from p2p_ip.
; You can check accessible addresses of your network card.
; Please see https://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30302
channel_listen_port=20202
jsonrpc_listen_port=8547

[node1]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
```

(continues on next page)

(续上页)

```
channel_ip=0.0.0.0
p2p_listen_port=30303
channel_listen_port=20203
jsonrpc_listen_port=8548
EOF
```

机构A生成并发送节点信息

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

机构A生成节点证书及P2P连接信息文件，此步需要用到上述配置的node_deployment.ini，及机构meta文件夹下的机构证书与私钥，机构A生成节点证书及P2P连接信息文件

```
./generator --generate_all_certificates ./agencyA_node_info
```

查看生成文件:

```
ls ./agencyA_node_info
```

```
# 上述命令解释
# 从左至右分别为需要交互给机构A的节点证书，节点P2P连接地址文件(根据node_deployment.ini生成的本机构节点信息)
cert_127.0.0.1_30300.crt cert_127.0.0.1_30301.crt peers.txt
```

机构生成节点时需要指定其他节点的节点P2P连接地址，因此，A机构需将节点P2P连接地址文件发送至机构B

```
cp ./agencyA_node_info/peers.txt ~/generator-B/meta/peersA.txt
```

机构B生成并发送节点信息

在~/generator-B目录下执行下述命令

```
cd ~/generator-B
```

机构B生成节点证书及P2P连接信息文件:

```
./generator --generate_all_certificates ./agencyB_node_info
```

生成创世区块的机构需要节点证书，示例中由A机构生成创世区块，因此B机构除了发送节点P2P连接地址文件外，还需发送节点证书至机构A

发送证书

```
cp ./agencyB_node_info/cert*.crt ~/generator-A/meta/
```

发送节点P2P连接地址文件

```
cp ./agencyB_node_info/peers.txt ~/generator-A/meta/peersB.txt
```

机构A生成群组1创世区块

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

机构A修改conf文件夹下的group_genesis.ini，配置项可参考手册。:

```
cat > ./conf/group_genesis.ini << EOF
[group]
group_id=1

[nodes]
node0=127.0.0.1:30300
node1=127.0.0.1:30301
node2=127.0.0.1:30302
node3=127.0.0.1:30303
EOF
```

命令执行之后会修改./conf/group_genesis.ini文件:

```
;命令解释
[group]
;群组id
group_id=1

[nodes]
;机构A节点p2p地址
node0=127.0.0.1:30300
;机构A节点p2p地址
node1=127.0.0.1:30301
;机构B节点p2p地址
node2=127.0.0.1:30302
;机构B节点p2p地址
node3=127.0.0.1:30303
```

教程中选择机构A生成群组创世区块，实际生产中可以通过联盟链委员会协商选择。

此步会根据机构A的meta文件夹下配置的节点证书，生成group_genesis.ini配置的群组创世区块，教程中需要机构A的meta下有名为cert_127.0.0.1_30300.crt，cert_127.0.0.1_30301.crt，cert_127.0.0.1_30302.crt，cert_127.0.0.1_30303.crt的节点证书，此步需要用到机构B的节点证书。

```
./generator --create_group_genesis ./group
```

分发群组1创世区块至机构B:

```
cp ./group/group.1.genesis ~/generator-B/meta
```

机构A生成所属节点

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

生成机构A所属节点，此命令会根据用户配置的node_deployment.ini文件生成相应的节点配置文件夹:

注意，此步指定的节点P2P连接信息peers.txt为群组内其他节点的链接信息，多个机构组网的情况下需要将其合并。

```
./generator --build_install_package ./meta/peersB.txt ./nodeA
```

查看生成节点配置文件夹:

```
ls ./nodeA
```

```
# 命令解释 此处采用tree风格显示
# 生成的文件夹nodeA信息如下所示，
├── monitor # monitor脚本
├── node_127.0.0.1_30300 # 127.0.0.1服务器 端口号30300的节点配置文件夹
├── node_127.0.0.1_30301
├── scripts # 节点的相关工具脚本
├── start_all.sh # 节点批量启动脚本
└── stop_all.sh # 节点批量停止脚本
```

机构A启动节点：

```
bash ./nodeA/start_all.sh
```

查看节点进程：

```
ps -ef | grep fisco
```

```
# 命令解释
# 可以看到如下进程
fisco  15347     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30300/fisco-bcos -c config.ini
fisco  15402     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30301/fisco-bcos -c config.ini
```

机构B生成所属节点

在~/generator-B目录下执行下述命令

```
cd ~/generator-B
```

生成机构B所属节点，此命令会根据用户配置的node_deployment.ini文件生成相应的节点配置文件夹：

```
./generator --build_install_package ./meta/peersA.txt ./nodeB
```

机构B启动节点：

```
bash ./nodeB/start_all.sh
```

注解： 节点启动只需要推送对应ip的node文件夹即可，如127.0.0.1的服务器，只需node_127.0.0.1_port对应的节点配置文件夹。多机部署时，只需要将生成的节点文件夹推送至对应服务器即可。

查看群组1节点运行状态

查看进程：

```
ps -ef | grep fisco
```

```
# 命令解释
# 可以看到如下所示的进程
fisco  15347     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30300/fisco-bcos -c config.ini
fisco  15402     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30301/fisco-bcos -c config.ini
```

(continues on next page)

(续上页)

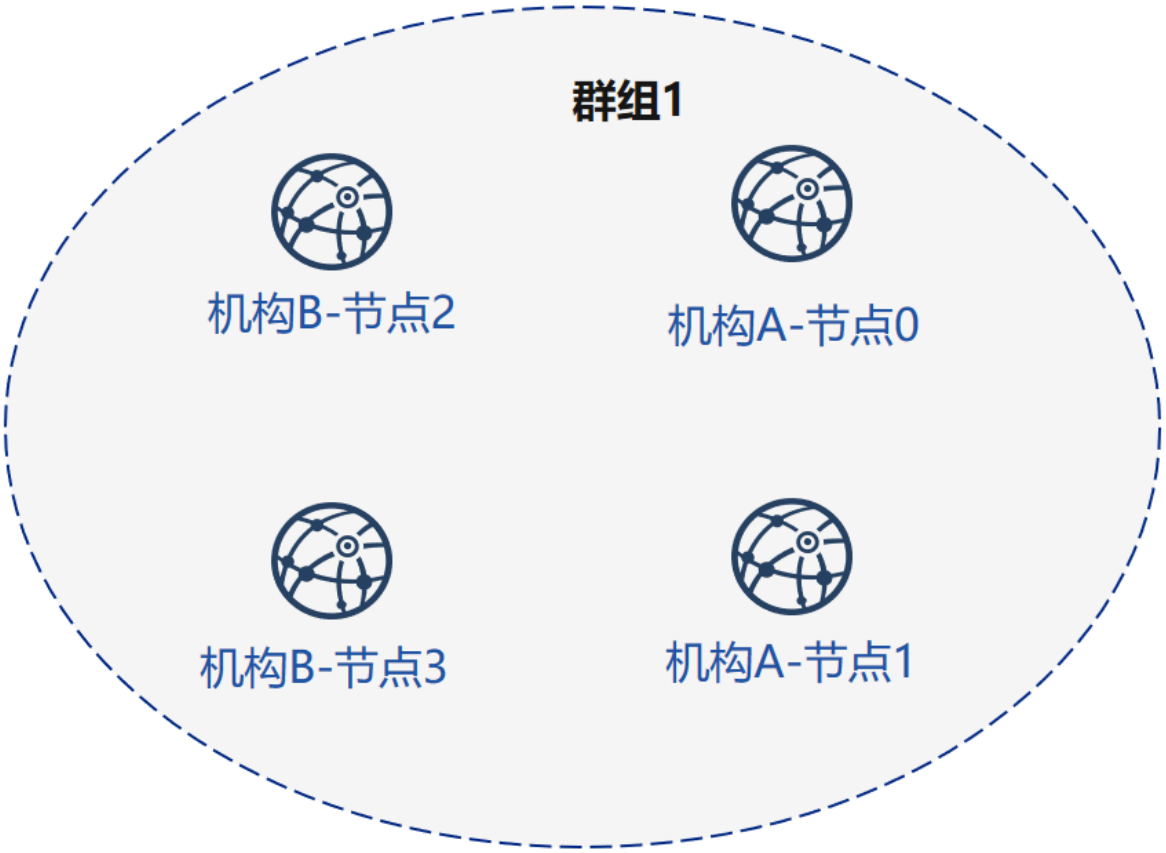
```
fisco  15457     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30302/fisco-bcos -c config.ini
fisco  15498     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30303/fisco-bcos -c config.ini
```

查看节点log:

```
tail -f ./node*/node*/log/log*  | grep +++
```

```
# 命令解释
# log中打印的+++即为节点正常共识
info|2019-02-25 17:25:56.028692| [g:1][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=833bd983...
info|2019-02-25 17:25:59.058625| [g:1][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=343b1141...
info|2019-02-25 17:25:57.038284| [g:1][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=1,hash=ea85c27b...
```

至此，我们完成了如图所示机构A、B搭建群组1的操作:



6.2.5 证书授权机构初始化机构C

在证书生成机构目录下操作:

```
cd ~/generator
```

初始化机构C，请注意，此时generator目录下有链证书及私钥，实际环境中机构C无法获取链证书及私钥。

```
cp -r ~/generator ~/generator-C
```

生成机构C证书：

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyC
```

查看机构证书及私钥:

```
ls dir_agency_ca/agencyC/
```

```
# 上述命令解释
# 从左至右分别为机构证书、机构私钥、链证书
agency.crt  agency.key  ca.crt
```

发送链证书、机构证书、机构私钥至机构C，示例是通过文件拷贝的方式，从证书授权机构将机构证书发送给对应的机构，放到机构的工作目录的meta子目录下

```
cp ./dir_agency_ca/agencyC/* ~/generator-C/meta/
```

6.2.6 机构A、C构建群组2

接下来，机构C需要与A进行新群组建立操作，示例中以C生成创世区块为例。

机构A发送节点信息

由于机构A已经生成过节点证书及peers文件，只需将之前生成的节点P2P连接信息以及节点证书发送至机构C，操作如下：

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

示例中由机构C生成群组创世区块，因此需要机构A的节点证书和节点P2P连接地址文件，将上述文件发送至机构C

发送证书

```
cp ./agencyA_node_info/cert*.crt ~/generator-C/meta/
```

发送节点P2P连接地址文件

```
cp ./agencyA_node_info/peers.txt ~/generator-C/meta/peersA.txt
```

机构C修改配置文件

机构C修改conf文件夹下的`node_deployment.ini`如下图所示:

在~/generator-C目录下执行下述命令

```
cd ~/generator-C
```

```
cat > ./conf/node_deployment.ini << EOF
[group]
group_id=2

[node0]
; host ip for the communication among peers.
```

(continues on next page)

(续上页)

```
; Please use your ssh login ip.
p2p_ip=127.0.0.1
; listen ip for the communication between sdk clients.
; This ip is the same as p2p_ip for physical host.
; But for virtual host e.g. vps servers, it is usually different from p2p_ip.
; You can check accessible addresses of your network card.
; Please see https://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30304
channel_listen_port=20204
jsonrpc_listen_port=8549

[node1]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30305
channel_listen_port=20205
jsonrpc_listen_port=8550
EOF
```

机构C生成并发送节点信息

在~/generator-C目录下执行下述命令

```
cd ~/generator-C
```

机构C生成节点证书及P2P连接信息文件:

```
./generator --generate_all_certificates ./agencyC_node_info
```

查看生成文件:

```
ls ./agencyC_node_info
```

```
# 上述命令解释
# 从左至右分别为需要交互给机构A的节点证书，节点P2P连接地址文件(根据node_deployment.ini生成的本机构节点信息)
cert_127.0.0.1_30304.crt cert_127.0.0.1_30305.crt peers.txt
```

机构生成节点时需要指定其他节点的节点P2P连接地址，因此，C机构需将节点P2P连接地址文件发送至机构A

```
cp ./agencyC_node_info/peers.txt ~/generator-A/meta/peersC.txt
```

机构C生成群组2创世区块

在~/generator-C目录下执行下述命令

```
cd ~/generator-C
```

机构C修改conf文件夹下的group_genesis.ini如下图所示:

```
cat > ./conf/group_genesis.ini << EOF
[group]
```

(continues on next page)

(续上页)

```
group_id=2

[nodes]
node0=127.0.0.1:30300
node1=127.0.0.1:30301
node2=127.0.0.1:30304
node3=127.0.0.1:30305
EOF
```

命令执行之后会修改./conf/group_genesis.ini文件：

```
;命令解释
[group]
group_id=2

[nodes]
node0=127.0.0.1:30300
;机构A节点p2p地址
node1=127.0.0.1:30301
;机构A节点p2p地址
node2=127.0.0.1:30304
;机构C节点p2p地址
node3=127.0.0.1:30305
;机构C节点p2p地址
```

教程中选择机构C生成群组创世区块，实际生产中可以通过联盟链委员会协商选择。

此步会根据机构C的meta文件夹下配置的节点证书，生成group_genesis.ini配置的群组创世区块。

```
./generator --create_group_genesis ./group
```

分发群组2创世区块至机构A：

```
cp ./group/group.2.genesis ~/generator-A/meta/
```

机构C生成所属节点

在~/generator-C目录下执行下述命令

```
cd ~/generator-C
```

```
./generator --build_install_package ./meta/peersA.txt ./nodeC
```

机构C启动节点：

```
bash ./nodeC/start_all.sh
```

```
ps -ef | grep fisco
```

```
# 命令解释
# 可以看到如下进程
fisco   15347     1   0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30300/fisco-bcos -c config.ini
fisco   15402     1   0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30301/fisco-bcos -c config.ini
fisco   15457     1   0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30302/fisco-bcos -c config.ini
fisco   15498     1   0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30303/fisco-bcos -c config.ini
```

(continues on next page)

(续上页)

```
fisco  15550     1  0 17:22 pts/2    00:00:00 ~/generator-C/nodeC/node_127.0.0.1_
↪30304/fisco-bcos -c config.ini
fisco  15589     1  0 17:22 pts/2    00:00:00 ~/generator-C/nodeC/node_127.0.0.1_
↪30305/fisco-bcos -c config.ini
```

机构A为现有节点初始化群组2

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

添加群组2配置文件至已有节点，此步将群组2创世区块`group.2.genesis`添加至./nodeA下的所有节点内：

```
./generator --add_group ./meta/group.2.genesis ./nodeA
```

添加机构C节点连接文件peers至已有节点，此步将`peersC.txt`的节点P2P连接地址添加至./nodeA下的所有节点内：

```
./generator --add_peers ./meta/peersC.txt ./nodeA
```

重启机构A节点：

```
bash ./nodeA/stop_all.sh
```

```
bash ./nodeA/start_all.sh
```

查看群组2节点运行状态

查看节点进程：

```
ps -ef | grep fisco
```

```
# 命令解释
# 可以看到如下进程
fisco  15347     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30300/fisco-bcos -c config.ini
fisco  15402     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30301/fisco-bcos -c config.ini
fisco  15457     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30302/fisco-bcos -c config.ini
fisco  15498     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30303/fisco-bcos -c config.ini
fisco  15550     1  0 17:22 pts/2    00:00:00 ~/generator-C/nodeC/node_127.0.0.1_
↪30304/fisco-bcos -c config.ini
fisco  15589     1  0 17:22 pts/2    00:00:00 ~/generator-C/nodeC/node_127.0.0.1_
↪30305/fisco-bcos -c config.ini
```

查看节点log：

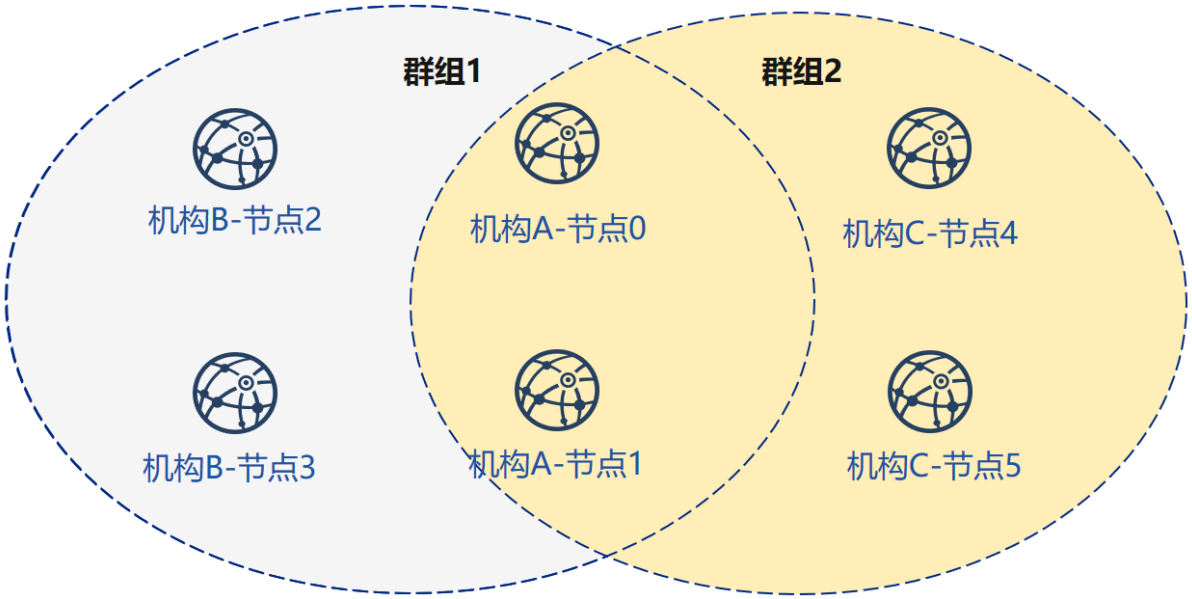
在~/generator-C目录下执行下述命令

```
cd ~/generator-C
```

```
tail -f ./node*/node*/log/log*  | grep +++
```

```
# 命令解释
# log中打印的+++即为节点正常共识
info|2019-02-25 17:25:56.028692| [g:2][p:264][CONSENSUS][SEALER]++++++++++++++++
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=833bd983...
info|2019-02-25 17:25:59.058625| [g:2][p:264][CONSENSUS][SEALER]++++++++++++++++
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=343b1141...
info|2019-02-25 17:25:57.038284| [g:2][p:264][CONSENSUS][SEALER]++++++++++++++++
↪Generating seal on,blkNum=1,tx=0,myIdx=1,hash=ea85c27b...
```

至此，我们完成了如图所示的机构A、C搭建群组2构建：



6.2.7 扩展教程–机构C节点加入群组1

将节点加入已有群组需要用户使用控制台发送指令，将节点加入群组，示例如下：

注解： 企业部署工具会根据generator/meta文件夹下的机构证书及私钥生成sdk相应证书，如需手动生成可以参考操作手册中的generate_sdk_certificate命令

此时群组1内有机构A、B的节点，机构C节点加入群组1需要经过群组内节点的准入，示例以机构A节点为例：

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

发送群组1创世区块至机构C

发送群组1配置文件至机构C节点：

```
./generator --add_group ./group/group.1.genesis  ~/generator-C/nodeC
```

当前FISCO BCOS暂不支持文件热更新，为机构C节点添加群组1创世区块后需重启节点。

重启机构C节点：

```
bash ~/generator-C/nodeC/stop_all.sh
```

```
bash ~/generator-C/nodeC/start_all.sh
```

配置控制台

机构A配置控制台或sdk，教程中以控制台为例：

注意：此命令会根据用户配置的node_deployment.ini中节点及群组完成了控制台的配置，用户可以直接启动控制台，启动前请确保已经安装java

国内用户推荐使用cdn下载，如果访问github较快，可以去掉--cdn选项：

```
./generator --download_console ./ --cdn
```

查看机构C节点4信息

机构A使用控制台加入机构C节点4为观察节点，其中参数第二项需要替换为加入节点的nodeid，nodeid在节点文件夹的conf的node.nodeid文件

查看机构C节点nodeid：

```
cat ~/generator-C/nodeC/node_127.0.0.1_30304/conf/node.nodeid
```

```
# 命令解释
# 可以看到类似于如下nodeid，控制台使用时需要传入该参数
ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c9715
```

使用控制台注册观察节点

启动控制台：

```
cd ~/generator-A/console && bash ./start.sh 1
```

使用控制台addObserver命令将节点注册为观察节点，此步需要用到cat命令查看得到机构C节点的node.nodeid：

```
addObserver␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
```

```
# 命令解释
# 执行成功会提示success
$ [group:1]> addObserver␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
        "code":0,
        "msg":"success"
}
```

退出控制台：

```
exit
```

查看机构C节点5信息

机构A使用控制台加入机构C的节点5为共识节点，其中参数第二项需要替换为加入节点的nodeid，nodeid在节点文件夹的conf的node.nodeid文件

查看机构C节点nodeid：

```
cat ~/generator-C/nodeC/node_127.0.0.1_30305/conf/node.nodeid
```

```
# 命令解释
# 可以看到类似于如下nodeid，控制台使用时需要传入该参数
5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa9ae
```

使用控制台注册共识节点

启动控制台：

```
cd ~/generator-A/console && bash ./start.sh 1
```

使用控制台addSealer命令将节点注册为共识节点，此步需要用到cat命令查看得到机构C节点的node.nodeid：

```
addSealer␣
↪5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa
```

```
# 命令解释
# 执行成功会提示success
$ [group:1]> addSealer␣
↪5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa
{
        "code":0,
        "msg":"success"
}
```

退出控制台：

```
exit
```

查看节点

在~/generator-C目录下执行下述命令

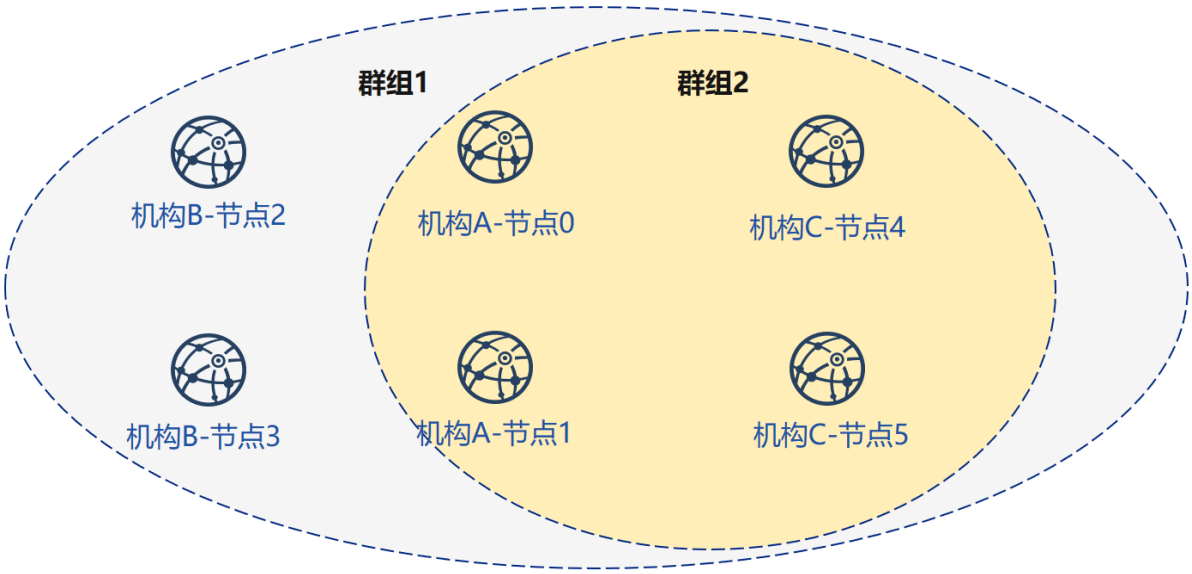
```
cd ~/generator-C
```

查看节点log内group1信息:

```
cat node*/node_127.0.0.1_3030*/log/log* | grep g:1  | grep Report
```

```
# 命令解释
# 观察节点只会同步交易数据，不会同步非交易状态的共识信息
# log中的^^^即为节点的交易信息，g:1为群组1打印的信息
info|2019-02-26 16:01:39.914367| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=0,sealerIdx=0,hash=9b76de5d...,next=1,tx=0,nodeIdx=65535
info|2019-02-26 16:01:40.121075| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=1,sealerIdx=3,hash=46b7f17c...,next=2,tx=1,nodeIdx=65535
info|2019-02-26 16:03:44.282927| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=2,sealerIdx=2,hash=fb982013...,next=3,tx=1,nodeIdx=65535
info|2019-02-26 16:01:39.914367| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=0,sealerIdx=0,hash=9b76de5d...,next=1,tx=0,nodeIdx=4
info|2019-02-26 16:01:40.121075| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=1,sealerIdx=3,hash=46b7f17c...,next=2,tx=1,nodeIdx=4
info|2019-02-26 16:03:44.282927| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=2,sealerIdx=2,hash=fb982013...,next=3,tx=1,nodeIdx=4
```

至此 我们完成了所示构建教程中的所有操作。



通过本节教程，我们在本机生成一个网络拓扑结构为3机构2群组6节点的多群组架构联盟链。

如果使用该教程遇到问题，请查看FAQ

6.3 使用运维部署工具部署国密区块链

此文档为使用运维部署工具部署的国密部署版本

本章以部署**6节点3机构2群组**的组网模式，演示运维部署工具的使用方法。更多参数选项说明请参考这里。

本章节为多机构对等部署的过程，适用于多机构部署，机构私钥不出内网的情况，由单机构一键生成所有机构节点配置文件的教程可以参考FISCO BCOS运维部署工具一键部署。

6.3.1 下载安装

下载

```
cd ~/ && git clone https://github.com/FISCO-BCOS/generator.git
```

安装

此操作要求用户具有sudo权限。

```
cd ~/generator && bash ./scripts/install.sh
```

检查是否安装成功，若成功，输出 usage: generator xxx

```
./generator -h
```

获取国密节点二进制

拉取最新fisco-bcos二进制文件到meta中

```
./generator --download_fisco ./meta -g
```

检查二进制版本

若成功，输出 FISCO-BCOS gm Version : x.x.x-x

```
./meta/fisco-bcos -v
```

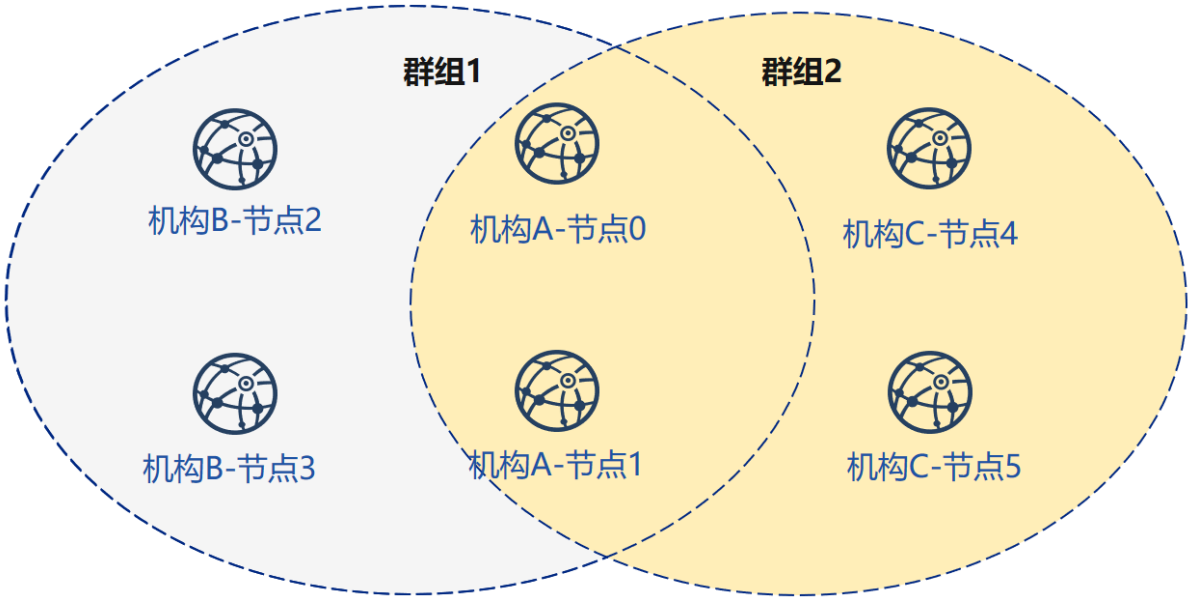
PS：源码编译节点二进制的用户，只需要用编译出来的二进制替换掉meta文件夹下的二进制即可。

6.3.2 典型示例

为了保证机构的密钥安全，运维部署工具提供了一种机构间相互合作的的搭链方式。本节以部署6节点3机构2群组的组网模式，演示企业间如何相互配合，搭建区块链。

节点组网拓扑结构

一个如图所示的6节点3机构2群组的组网模式。机构B和机构C分别位于群组1和群组2中。机构A同属于群组1和群组2中。



机器环境

每个节点的IP，端口号为如下：

注解:

- 云主机的公网IP均为虚拟IP，若rpc_ip/p2p_ip/channel_ip填写外网IP，会绑定失败，须填写0.0.0.0
- RPC/P2P/Channel监听端口必须位于1024-65535范围内，且不能与机器上其他应用监听端口冲突
- 出于安全性和易用性考虑，FISCO BCOS v2.3.0版本最新节点config.ini配置将listen_ip拆分成jsonrpc_listen_ip和channel_listen_ip，但仍保留对listen_ip的解析功能，详细请参考 这里
- 为便于开发和体验，channel_listen_ip参考配置是 *0.0.0.0* ，出于安全考虑，请根据实际业务网络情况，修改为安全的监听地址，如：内网IP或特定的外网IP

涉及机构

搭链操作涉及多个机构的合作，包括：

- 证书颁发机构

- 搭建节点的机构（简称“机构”）

关键流程

本流程简要的给出**证书颁发机构**，**节点机构间**如何相互配合搭建区块链。

一、初始化链证书

1. 证书颁发机构操作：
    - 生成链证书

二、生成群组1

1. 证书颁发机构操作：颁发机构证书
    - 生成机构证书
    - 发送证书
2. 机构间独立操作
    - 修改配置文件`node_deployment.ini`
    - 生成节点证书及节点P2P端口地址文件
3. 选取其中一个机构为群组生成创世块
    - 收集群组内所有节点证书
    - 修改配置文件`group_genesis.ini`
    - 为群组生成创世块文件
    - 分发创世块文件
4. 机构间独立操作：生成节点
    - 收集群组其他节点的P2P端口地址文件
    - 生成节点
    - 启动节点

三、初始化新机构

1. 证书颁发机构操作：颁发新机构证书
    - 生成机构证书
    - 发送证书

四、生成群组2

1. 新机构独立操作
    - 修改配置文件`node_deployment.ini`
    - 生成节点证书及节点P2P端口地址文件
2. 选取其中一个机构为群组生成创世块
    - 收集群组内所有节点证书

- 修改配置文件group_genesis.ini
- 为群组生成创世块文件
- 分发创世块文件

3. 新机构独立操作：生成节点
   - 收集群组其他节点的P2P端口地址文件
   - 生成节点
   - 启动节点
4. 已有机构操作：配置新群组
   - 收集群组其他节点的P2P端口地址文件
   - 配置新群组与新增节点的P2P端口地址
   - 重启节点

五、现有节点加入群组1

1. 群组1原有机构操作：
   - 发送群组1创世区块至现有节点
   - 配置控制台
   - 获取加入节点nodeid
   - 使用控制台将节点加入群组1

6.3.3 联盟链初始化

为了操作简洁，本示例所有操作在同一台机器上进行，用不同的目录模拟不同的机构环境。用文件复制操作来模拟网络的发送。进行了教程中的下载安装后，请将generator复制到对应机构的generator目录中。

机构初始化

我们以**教程中下载的generator作为证书颁发机构**。

初始化机构A

```
cp -r ~/generator ~/generator-A
```

初始化机构B

```
cp -r ~/generator ~/generator-B
```

初始化链证书

由于FISCO BCOS节点与SDK通信时采用非国密方式，因此节点需要生成非国密版本的证书

在证书颁发机构上进行操作，一条联盟链拥有唯一的链证书ca.crt和gmca.crt

用 --generate_chain_certificate 命令生成链证书

在证书生成机构目录下操作:

```
cd ~/generator
```

生成国密证书

```
./generator --generate_chain_certificate ./dir_chain_ca -g
```

生成普通证书

```
./generator --generate_chain_certificate ./dir_chain_ca_normal
```

查看链证书及私钥:

```
ls ./dir_chain_ca
```

```
# 上述命令解释
# 从左至右分别为链证书、链私钥
gmca.crt  gmca.key
```

```
ls ./dir_chain_ca_normal
```

```
# 上述命令解释
# 从左至右分别为链证书、链私钥
ca.crt  ca.key
```

6.3.4 机构A、B构建群组1

初始化机构A

教程中为了简化操作直接生成了机构证书和私钥，实际应用时应该由机构本地生成私钥agency.key和gmagency.key，再生成证书请求文件，向证书签发机构获取机构证书agency.crt和gmagency.crt。

在证书生成机构目录下操作:

```
cd ~/generator
```

生成机构A证书:

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyA -g
```

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca_normal␣
↪agencyA_normal
```

查看机构证书及私钥:

```
ls dir_agency_ca/agencyA/
```

```
# 上述命令解释
# 从左至右分别为机构证书、机构私钥、链证书
gmagency.crt  gmagency.key  gmca.crt
```

```
ls dir_agency_ca/agencyA_normal/
```

```
# 上述命令解释
# 从左至右分别为机构证书、机构私钥、链证书
agency.crt  agency.key  ca.crt
```

发送链证书、机构证书、机构私钥至机构A，示例是通过文件拷贝的方式，从证书授权机构将机构证书发送给对应的机构，放到机构的工作目录的meta子目录下

```
cp ./dir_agency_ca/agencyA/* ~/generator-A/meta/
```

```
cp ./dir_agency_ca/agencyA_normal/* ~/generator-A/meta/
```

初始化机构B

在证书生成机构目录下操作:

```
cd ~/generator
```

生成机构B证书:

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyB -g
```

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca_normal␣
↪agencyB_normal
```

发送链证书、机构证书、机构私钥至机构B，示例是通过文件拷贝的方式，从证书授权机构将机构证书发送给对应的机构，放到机构的工作目录的meta子目录下

```
cp ./dir_agency_ca/agencyB/* ~/generator-B/meta/
```

```
cp ./dir_agency_ca/agencyB_normal/* ~/generator-B/meta/
```

重要: 一条联盟链中只能用到一个根证书ca.crt，多服务器部署时不要生成多个根证书和私钥。一个群组只能有一个群组创世区块group.x.genesis

机构A修改配置文件

node_deployment.ini为节点配置文件，运维部署工具会根据node_deployment.ini下的配置生成相关节点证书，及生成节点配置文件夹等。

机构A修改conf文件夹下的node_deployment.ini如下图所示:

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

```
cat > ./conf/node_deployment.ini << EOF
[group]
group_id=1

[node0]
; host ip for the communication among peers.
; Please use your ssh login ip.
p2p_ip=127.0.0.1
; listen ip for the communication between sdk clients.
; This ip is the same as p2p_ip for physical host.
; But for virtual host e.g. vps servers, it is usually different from p2p_ip.
; You can check accessible addresses of your network card.
; Please see https://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30300
```

(continues on next page)

(续上页)

```
channel_listen_port=20200
jsonrpc_listen_port=8545

[node1]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30301
channel_listen_port=20201
jsonrpc_listen_port=8546
EOF
```

机构B修改配置文件

机构B修改conf文件夹下的node_deployment.ini如下图所示:

在~/generator-B目录下执行下述命令

```
cd ~/generator-B
```

```
cat > ./conf/node_deployment.ini << EOF
[group]
group_id=1

[node0]
; host ip for the communication among peers.
; Please use your ssh login ip.
p2p_ip=127.0.0.1
; listen ip for the communication between sdk clients.
; This ip is the same as p2p_ip for physical host.
; But for virtual host e.g. vps servers, it is usually different from p2p_ip.
; You can check accessible addresses of your network card.
; Please see https://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30302
channel_listen_port=20202
jsonrpc_listen_port=8547

[node1]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30303
channel_listen_port=20203
jsonrpc_listen_port=8548
EOF
```

机构A生成并发送节点信息

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

机构A生成节点证书及P2P连接信息文件，此步需要用到上述配置的node_deployment.ini，及机构meta文件夹下的机构证书与私钥，机构A生成节点证书及P2P连接信息文件

```
./generator --generate_all_certificates ./agencyA_node_info -g
```

查看生成文件:

```
ls ./agencyA_node_info
```

```
# 上述命令解释
# 从左至右分别为需要交互给机构A的节点证书，节点P2P连接地址文件(根据node_deployment.ini生成的本机构节点信息)
gmcert_127.0.0.1_30300.crt gmcert_127.0.0.1_30301.crt peers.txt
```

机构生成节点时需要指定其他节点的节点P2P连接地址，因此，A机构需将节点P2P连接地址文件发送至机构B

```
cp ./agencyA_node_info/peers.txt ~/generator-B/meta/peersA.txt
```

机构B生成并发送节点信息

在~/generator-B目录下执行下述命令

```
cd ~/generator-B
```

机构B生成节点证书及P2P连接信息文件:

```
./generator --generate_all_certificates ./agencyB_node_info -g
```

生成创世区块的机构需要节点证书，示例中由A机构生成创世区块，因此B机构除了发送节点P2P连接地址文件外，还需发送节点证书至机构A

发送证书

```
cp ./agencyB_node_info/gmcert*.crt ~/generator-A/meta/
```

发送节点P2P连接地址文件

```
cp ./agencyB_node_info/peers.txt ~/generator-A/meta/peersB.txt
```

机构A生成群组1创世区块

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

机构A修改conf文件夹下的group_genesis.ini，配置项可参考手册。:

```
cat > ./conf/group_genesis.ini << EOF
[group]
group_id=1

[nodes]
node0=127.0.0.1:30300
node1=127.0.0.1:30301
node2=127.0.0.1:30302
node3=127.0.0.1:30303
EOF
```

命令执行之后会修改./conf/group_genesis.ini文件:

```
;命令解释
[group]
;群组id
group_id=1

[nodes]
;机构A节点p2p地址
node0=127.0.0.1:30300
;机构A节点p2p地址
node1=127.0.0.1:30301
;机构B节点p2p地址
node2=127.0.0.1:30302
;机构B节点p2p地址
node3=127.0.0.1:30303
```

教程中选择机构A生成群组创世区块，实际生产中可以通过联盟链委员会协商选择。

此步会根据机构A的meta文件夹下配置的节点证书，生成group_genesis.ini配置的群组创世区块，教程中需要机构A的meta下有名为gmcert_127.0.0.1_30300.crt，gmcert_127.0.0.1_30301.crt，gmcert_127.0.0.1_30302.crt，gmcert_127.0.0.1_30303.crt的节点证书，此步需要用到机构B的节点证书。

```
./generator --create_group_genesis ./group -g
```

分发群组1创世区块至机构B：

```
cp ./group/group.1.genesis ~/generator-B/meta
```

机构A生成所属节点

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

生成机构A所属节点，此命令会根据用户配置的node_deployment.ini文件生成相应的节点配置文件夹:

注意，此步指定的节点P2P连接信息peers.txt为群组内其他节点的链接信息，多个机构组网的情况下需要将其合并。

```
./generator --build_install_package ./meta/peersB.txt ./nodeA -g
```

查看生成节点配置文件夹：

```
ls ./nodeA
```

```
# 命令解释 此处采用tree风格显示
# 生成的文件夹nodeA信息如下所示，
├── monitor # monitor脚本
├── node_127.0.0.1_30300 # 127.0.0.1服务器 端口号30300的节点配置文件夹
├── node_127.0.0.1_30301
├── scripts # 节点的相关工具脚本
├── start_all.sh # 节点批量启动脚本
└── stop_all.sh # 节点批量停止脚本
```

机构A启动节点：

```
bash ./nodeA/start_all.sh
```

查看节点进程:

```
ps -ef | grep fisco
```

```
# 命令解释
# 可以看到如下进程
fisco   15347     1   0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30300/fisco-bcos -c config.ini
fisco   15402     1   0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30301/fisco-bcos -c config.ini
```

机构B生成所属节点

在~/generator-B目录下执行下述命令

```
cd ~/generator-B
```

生成机构B所属节点，此命令会根据用户配置的node_deployment.ini文件生成相应的节点配置文件夹:

```
./generator --build_install_package ./meta/peersA.txt ./nodeB -g
```

机构B启动节点:

```
bash ./nodeB/start_all.sh
```

注解: 节点启动只需要推送对应ip的node文件夹即可，如127.0.0.1的服务器，只需node_127.0.0.1_port对应的节点配置文件夹。多机部署时，只需要将生成的节点文件夹推送至对应服务器即可。

查看群组1节点运行状态

查看进程:

```
ps -ef | grep fisco
```

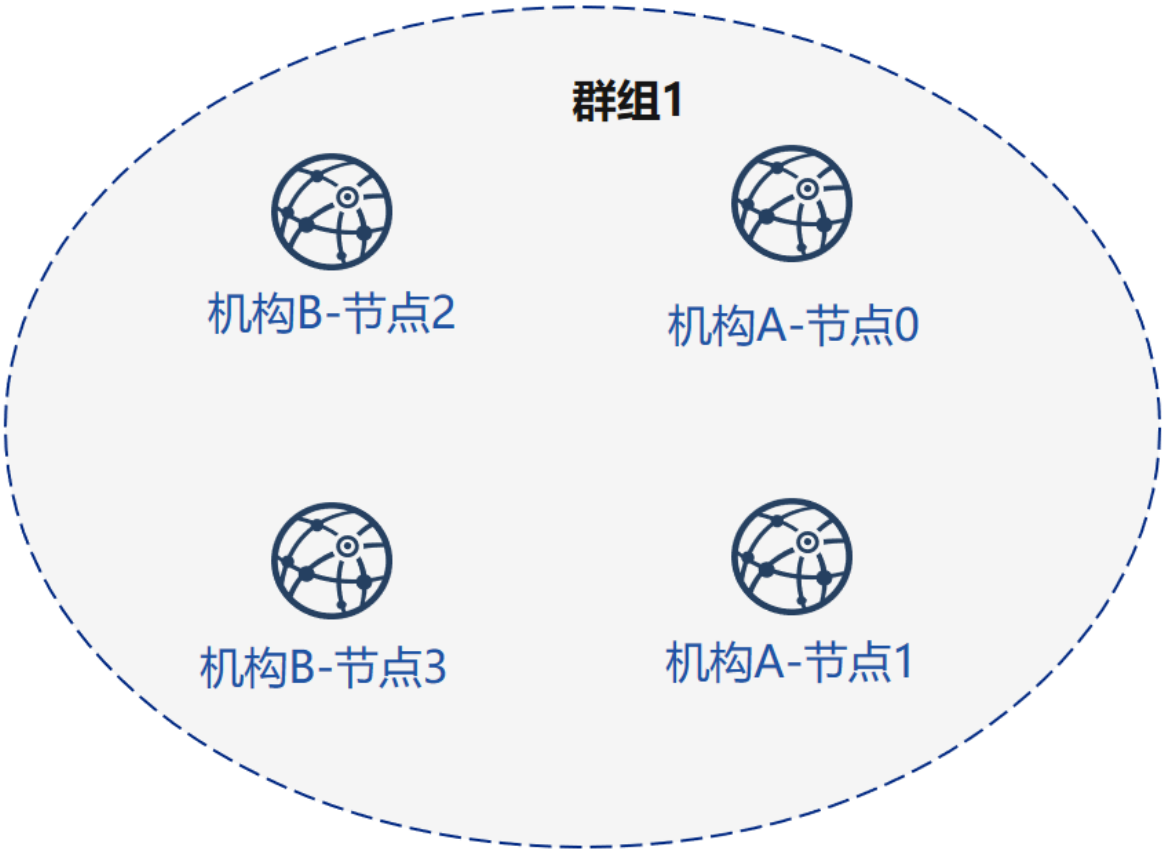
```
# 命令解释
# 可以看到如下所示的进程
fisco   15347     1   0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30300/fisco-bcos -c config.ini
fisco   15402     1   0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30301/fisco-bcos -c config.ini
fisco   15457     1   0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30302/fisco-bcos -c config.ini
fisco   15498     1   0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30303/fisco-bcos -c config.ini
```

查看节点log:

```
tail -f ./node*/node*/log/log*  | grep +++
```

```
# 命令解释
# log中打印的+++即为节点正常共识
info|2019-02-25 17:25:56.028692| [g:1][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=833bd983...
info|2019-02-25 17:25:59.058625| [g:1][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=343b1141...
info|2019-02-25 17:25:57.038284| [g:1][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=1,hash=ea85c27b...
```

至此，我们完成了如图所示机构A、B搭建群组1的操作：



6.3.5 证书授权机构初始化机构C

在证书生成机构目录下操作：

```
cd ~/generator
```

初始化机构C，请注意，此时generator目录下有链证书及私钥，实际环境中机构C无法获取链证书及私钥。

```
cp -r ~/generator ~/generator-C
```

生成机构C证书：

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyC -g
```

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca_normal␣
↪agencyC_normal
```

发送链证书、机构证书、机构私钥至机构C，示例是通过文件拷贝的方式，从证书授权机构将机构证书发送给对应的机构，放到机构的工作目录的meta子目录下

```
cp ./dir_agency_ca/agencyC/* ~/generator-C/meta/
```

```
cp ./dir_agency_ca/agencyC_normal/* ~/generator-C/meta/
```

6.3.6 机构A、C构建群组2

接下来，机构C需要与A进行新群组建立操作，示例中以C生成创世区块为例。

机构A发送节点信息

由于机构A已经生成过节点证书及peers文件，只需将之前生成的节点P2P连接信息以及节点证书发送至机构C，操作如下：

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

示例中由机构C生成群组创世区块，因此需要机构A的节点证书和节点P2P连接地址文件，将上述文件发送至机构C

发送证书

```
cp ./agencyA_node_info/gmcert*.crt ~/generator-C/meta/
```

发送节点P2P连接地址文件

```
cp ./agencyA_node_info/peers.txt ~/generator-C/meta/peersA.txt
```

机构C修改配置文件

机构C修改conf文件夹下的node_deployment.ini如下图所示:

在~/generator-C目录下执行下述命令

```
cd ~/generator-C
```

```
cat > ./conf/node_deployment.ini << EOF
[group]
group_id=2

[node0]
; host ip for the communication among peers.
; Please use your ssh login ip.
p2p_ip=127.0.0.1
; listen ip for the communication between sdk clients.
; This ip is the same as p2p_ip for physical host.
; But for virtual host e.g. vps servers, it is usually different from p2p_ip.
; You can check accessible addresses of your network card.
; Please see https://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30304
channel_listen_port=20204
jsonrpc_listen_port=8549

[node1]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30305
channel_listen_port=20205
jsonrpc_listen_port=8550
EOF
```

机构C生成并发送节点信息

在~/generator-C目录下执行下述命令

```
cd ~/generator-C
```

机构C生成节点证书及P2P连接信息文件:

```
./generator --generate_all_certificates ./agencyC_node_info -g
```

查看生成文件:

```
ls ./agencyC_node_info
```

```
# 上述命令解释
# 从左至右分别为需要交互给机构A的节点证书，节点P2P连接地址文件(根据node_deployment.ini生成的本机构节点信息)
gmcert_127.0.0.1_30304.crt gmcert_127.0.0.1_30305.crt peers.txt
```

机构生成节点时需要指定其他节点的节点P2P连接地址，因此，C机构需将节点P2P连接地址文件发送至机构A

```
cp ./agencyC_node_info/peers.txt ~/generator-A/meta/peersC.txt
```

机构C生成群组2创世区块

在~/generator-C目录下执行下述命令

```
cd ~/generator-C
```

机构C修改conf文件夹下的group_genesis.ini如下图所示:

```
cat > ./conf/group_genesis.ini << EOF
[group]
group_id=2

[nodes]
node0=127.0.0.1:30300
node1=127.0.0.1:30301
node2=127.0.0.1:30304
node3=127.0.0.1:30305
EOF
```

命令执行之后会修改./conf/group_genesis.ini文件:

```
;命令解释
[group]
group_id=2

[nodes]
node0=127.0.0.1:30300
;机构A节点p2p地址
node1=127.0.0.1:30301
;机构A节点p2p地址
node2=127.0.0.1:30304
;机构C节点p2p地址
node3=127.0.0.1:30305
;机构C节点p2p地址
```

教程中选择机构C生成群组创世区块，实际生产中可以通过联盟链委员会协商选择。

此步会根据机构C的meta文件夹下配置的节点证书，生成group_genesis.ini配置的群组创世区块。

```
./generator --create_group_genesis ./group -g
```

分发群组2创世区块至机构A:

```
cp ./group/group.2.genesis ~/generator-A/meta/
```

机构C生成所属节点

在~/generator-C目录下执行下述命令

```
cd ~/generator-C
```

```
./generator --build_install_package ./meta/peersA.txt ./nodeC -g
```

机构C启动节点:

```
bash ./nodeC/start_all.sh
```

```
ps -ef | grep fisco
```

```
# 命令解释
# 可以看到如下进程
fisco  15347     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30300/fisco-bcos -c config.ini
fisco  15402     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30301/fisco-bcos -c config.ini
fisco  15457     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30302/fisco-bcos -c config.ini
fisco  15498     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30303/fisco-bcos -c config.ini
fisco  15550     1  0 17:22 pts/2    00:00:00 ~/generator-C/nodeC/node_127.0.0.1_
↪30304/fisco-bcos -c config.ini
fisco  15589     1  0 17:22 pts/2    00:00:00 ~/generator-C/nodeC/node_127.0.0.1_
↪30305/fisco-bcos -c config.ini
```

机构A为现有节点初始化群组2

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

添加群组2配置文件至已有节点，此步将群组2创世区块`group.2.genesis`添加至./nodeA下的所有节点内:

```
./generator --add_group ./meta/group.2.genesis ./nodeA
```

添加机构C节点连接文件peers至已有节点，此步将`peersC.txt`的节点P2P连接地址添加至./nodeA下的所有节点内:

```
./generator --add_peers ./meta/peersC.txt ./nodeA
```

重启机构A节点:

```
bash ./nodeA/stop_all.sh
```

```
bash ./nodeA/start_all.sh
```

查看群组2节点运行状态

查看节点进程：

```
ps -ef | grep fisco
```

```
# 命令解释
# 可以看到如下进程
fisco  15347     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30300/fisco-bcos -c config.ini
fisco  15402     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30301/fisco-bcos -c config.ini
fisco  15457     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30302/fisco-bcos -c config.ini
fisco  15498     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30303/fisco-bcos -c config.ini
fisco  15550     1  0 17:22 pts/2    00:00:00 ~/generator-C/nodeC/node_127.0.0.1_
↪30304/fisco-bcos -c config.ini
fisco  15589     1  0 17:22 pts/2    00:00:00 ~/generator-C/nodeC/node_127.0.0.1_
↪30305/fisco-bcos -c config.ini
```

查看节点log：

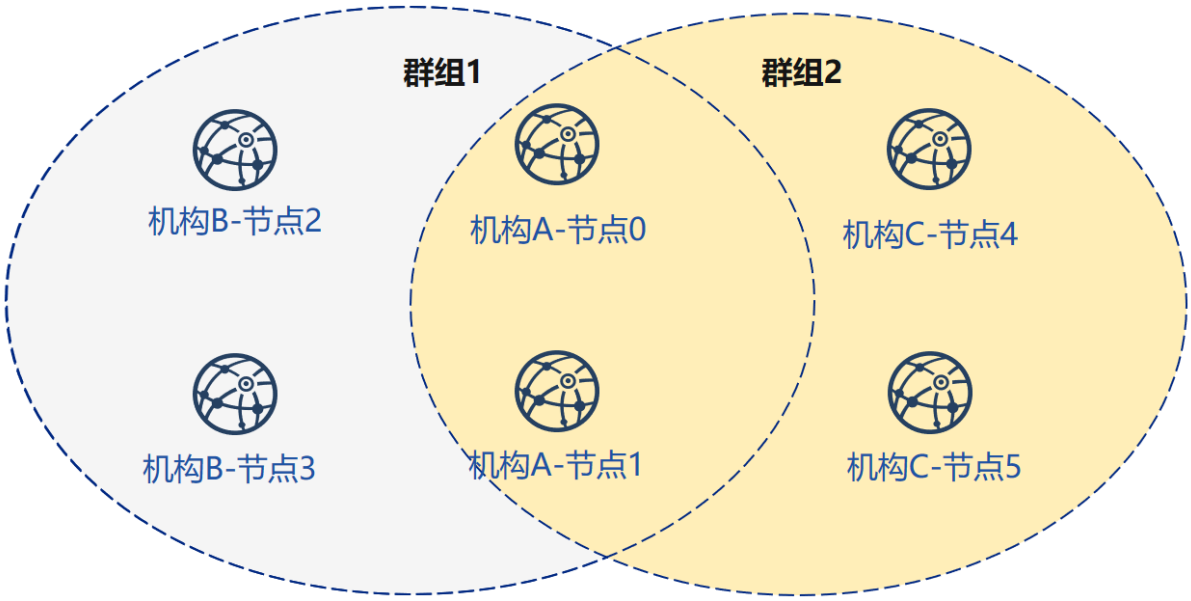
在~/generator-C目录下执行下述命令

```
cd ~/generator-C
```

```
tail -f ./node*/node*/log/log*  | grep +++
```

```
# 命令解释
# log中打印的+++即为节点正常共识
info|2019-02-25 17:25:56.028692| [g:2][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=833bd983...
info|2019-02-25 17:25:59.058625| [g:2][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=0,hash=343b1141...
info|2019-02-25 17:25:57.038284| [g:2][p:264][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,myIdx=1,hash=ea85c27b...
```

至此，我们完成了如图所示的机构A、C搭建群组2构建：



6.3.7 扩展教程–机构C节点加入群组1

将节点加入已有群组需要用户使用控制台发送指令，将节点加入群组，示例如下：

此时群组1内有机构A、B的节点，机构C节点加入群组1需要经过群组内节点的准入，示例以机构A节点为例：

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

发送群组1创世区块至机构C

发送群组1配置文件至机构C节点：

```
./generator --add_group ./group/group.1.genesis  ~/generator-C/nodeC
```

当前FISCO BCOS暂不支持文件热更新，为机构C节点添加群组1创世区块后需重启节点。

重启机构C节点：

```
bash ~/generator-C/nodeC/stop_all.sh
```

```
bash ~/generator-C/nodeC/start_all.sh
```

配置控制台

机构A配置控制台或sdk，教程中以控制台为例：

注解： 企业部署工具会根据generator/meta文件夹下的机构证书及私钥生成sdk相应证书，如需手动生成可以参考操作手册中的generate_sdk_certificate命令

国内用户推荐使用cdn下载，如果访问github较快，可以去掉--cdn选项：

```
./generator --download_console ./ --cdn
```

修改国密配置

```
vi ./console/conf/applicationContext.xml
```

进行如下修改

```
<bean id="encryptType" class="org.fisco.bcos.web3j.crypto.EncryptType">
    <!-- encryptType值设置为1，打开国密开关 -->
    <constructor-arg value="1"/> <!-- 0:standard 1:guomi -->
</bean>
```

替换国密jar包

```
cd console && curl -LO https://www.fisco.com.cn/cdn/deps/tools/solcj/solcJ-all-0.4.
↪25-gm.jar && bash replace_solc_jar.sh solcJ-all-0.4.25-gm.jar
```

注解： 建议用户下载完jar包后，手动检查jar包完整性。可以尝试手动解压观察是否完整

查看机构C节点4信息

机构A使用控制台加入机构C节点4为观察节点，其中参数第二项需要替换为加入节点的nodeid，nodeid在节点文件夹的conf的`gmnode.nodeid`文件

查看机构C节点nodeid：

```
cat ~/generator-C/nodeC/node_127.0.0.1_30304/conf/gmnode.nodeid
```

```
# 命令解释
# 可以看到类似于如下nodeid，控制台使用时需要传入该参数
ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c9715
```

使用控制台注册观察节点

启动控制台：

```
cd ~/generator-A/console && bash ./start.sh 1
```

使用控制台`addObserver`命令将节点注册为观察节点，此步需要用到`cat`命令查看得到机构C节点的`gmnode.nodeid`：

```
addObserver␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
```

```
# 命令解释
# 执行成功会提示success
$ [group:1]> addObserver␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
        "code":0,
        "msg":"success"
}
```

退出控制台：

```
exit
```

查看机构C节点5信息

机构A使用控制台加入机构C的节点5为共识节点，其中参数第二项需要替换为加入节点的nodeid，nodeid在节点文件夹的conf的node.nodeid文件

查看机构C节点nodeid：

```
cat ~/generator-C/nodeC/node_127.0.0.1_30305/conf/gmnode.nodeid
```

```
# 命令解释
# 可以看到类似于如下nodeid，控制台使用时需要传入该参数
5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa9ae
```

使用控制台注册共识节点

启动控制台：

```
cd ~/generator-A/console && bash ./start.sh 1
```

使用控制台addSealer命令将节点注册为共识节点，此步需要用到cat命令查看得到机构C节点的gmnode.nodeid:

```
addSealer␣
↪5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa
```

```
# 命令解释
# 执行成功会提示success
$ [group:1]> addSealer␣
↪5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa
{
        "code":0,
        "msg":"success"
}
```

退出控制台：

```
exit
```

查看节点

在~/generator-C目录下执行下述命令

```
cd ~/generator-C
```

查看节点log内group1信息:

```
cat node*/node_127.0.0.1_3030*/log/log* | grep g:1  | grep Report
```

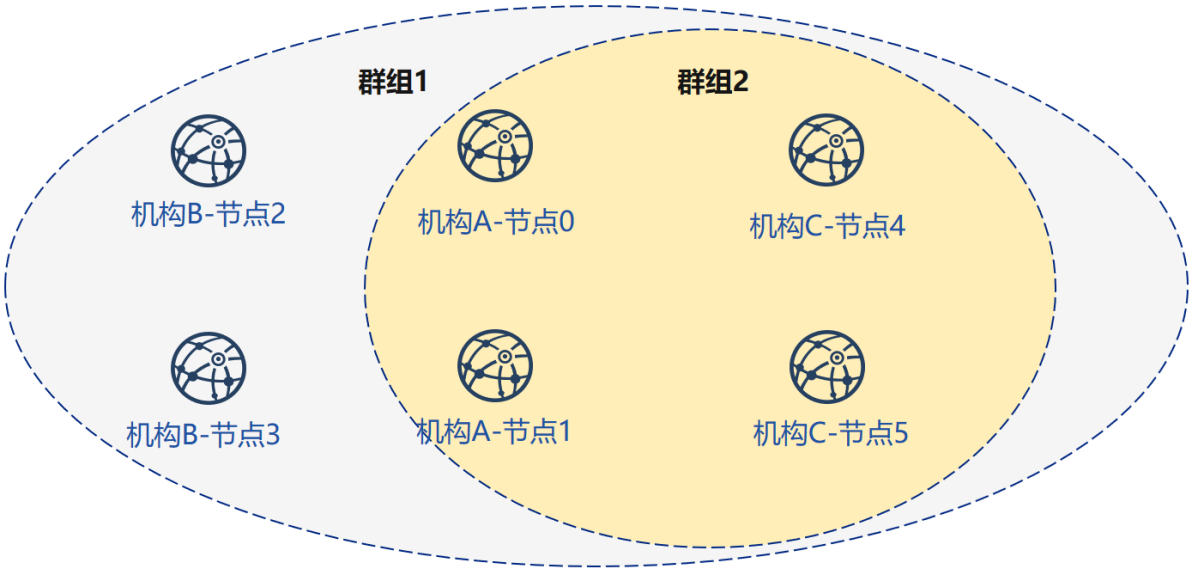
```
# 命令解释
# 观察节点只会同步交易数据，不会同步非交易状态的共识信息
# log中的^^^即为节点的交易信息，g:1为群组1打印的信息
info|2019-02-26 16:01:39.914367| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=0,sealerIdx=0,hash=9b76de5d...,next=1,tx=0,nodeIdx=65535
info|2019-02-26 16:01:40.121075| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=1,sealerIdx=3,hash=46b7f17c...,next=2,tx=1,nodeIdx=65535
info|2019-02-26 16:03:44.282927| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=2,sealerIdx=2,hash=fb982013...,next=3,tx=1,nodeIdx=65535
info|2019-02-26 16:01:39.914367| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=0,sealerIdx=0,hash=9b76de5d...,next=1,tx=0,nodeIdx=4
```

(continues on next page)

(续上页)

```
info|2019-02-26 16:01:40.121075| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=1,sealerIdx=3,hash=46b7f17c...,next=2,tx=1,nodeIdx=4
info|2019-02-26 16:03:44.282927| [g:1][p:65544][CONSENSUS][PBFT]^^^^^^^^Report,
↪num=2,sealerIdx=2,hash=fb982013...,next=3,tx=1,nodeIdx=4
```

至此 我们完成了所示构建教程中的所有操作。



通过本节教程，我们在本机生成一个网络拓扑结构为3机构2群组6节点的多群组架构联盟链。

如果使用该教程遇到问题，请查看FAQ

6.4 下载安装

6.4.1 环境依赖

FISCO BCOS generator依赖如下：

6.4.2 下载安装

下载

```
$ git clone https://github.com/FISCO-BCOS/generator.git
```

安装

```
$ cd generator
$ bash ./scripts/install.sh
```

检查是否安装成功

```
$ ./generator -h
# 若成功，输出 usage: generator xxx
```

6.4.3 拉取节点二进制

拉取最新fisco-bcos二进制文件到meta中

```
$ ./generator --download_fisco ./meta
```

检查二进制版本

```
$ ./meta/fisco-bcos -v
# 若成功，输出 FISCO-BCOS Version : x.x.x-x
```

PS：源码编译节点二进制的用户，只需要把编译出来的二进制放到meta文件夹下即可。

6.5 配置文件

FISCO BCOS generator的配置文件在./conf文件夹下，配置文件为：群组创世区块配置文件group_genesis.ini和生成节点配置文件node_deployment.ini。

用户通过对conf文件夹下文件的操作，配置生成节点配置文件夹的具体信息。

6.5.1 元数据文件夹meta

FISCO BCOS generator的meta文件夹为元数据文件夹，需要存放fisco bcos二进制文件、链证书ca.crt、本机构机构证书agency.crt、机构私钥节点证书、群组创世区块文件等。

证书的存放格式需要为cert_p2pip_port.crt的格式，如cert_127.0.0.1_30300.crt。

FISCO BCOS generator会根据用户在元数据文件夹下放置的相关证书、conf下的配置文件，生成用户下配置的节点配置文件夹。

6.5.2 group_genesis.ini

通过修改group_genesis.ini的配置，用户在指定目录及meta文件夹下生成新群组创世区块的相关配置，如group.1.genesis。

```
[group]
group_id=1

[nodes]
;群组创世区块的节点p2p地址
node0=127.0.0.1:30300
node1=127.0.0.1:30301
node2=127.0.0.1:30302
node3=127.0.0.1:30303
```

重要： 生成群组创世区块时需要节点的证书，如上述配置文件中需要4个节点的证书。分别为：cert_127.0.0.1_30301.crt，cert_127.0.0.1_30302.crt，cert_127.0.0.1_30303.crt和cert_127.0.0.1_30304.crt。

6.5.3 node_deployment.ini

通过修改node_deployment.ini的配置，用户可以使用–build_install_package命令在指定文件夹下生成节点不含私钥的节点配置文件夹。用户配置的每个section[node]即为用户需要生成的节点配置文件夹.section[peers]为需要连接的其他节点p2p信息。

配置文件示例如下：

```
[group]
group_id=1

# Owned nodes
[node0]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30300
channel_listen_port=20200
jsonrpc_listen_port=8545

[node1]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
channel_ip=0.0.0.0
p2p_listen_port=30301
channel_listen_port=20201
jsonrpc_listen_port=8546
```

读取节点配置的命令，如生成节点证书和节点配置文件夹等会读取该配置文件。

6.5.4 模板文件夹tpl

generator的模板文件夹如下图所示：

```
├── applicationContext.xml  # sdk配置文件模板
├── config.ini  # 节点配置文件模板
├── config.ini.gm  # 国密节点配置文件模板
├── group.i.genesis  # 群组创世区块模板
├── group.i.ini  # 群组区块配置模板
├── start.sh   # 节点启动脚本模板
├── start_all.sh  # 节点批量启动脚本模板
├── stop.sh  # 节点停止脚本模板
└── stop_all.sh  # 节点批量停止脚本模板
```

generator在进行如生成节点或群组配置的相关操作时，会根据模板文件夹下的配置文件生成相应的节点配置文件夹/群组配置，用户可以修改模板文件夹下的相关文件，再运行部署相关命令，即可生成自定义节点。

FISCO BCOS配置的相关解释可以参考FISCO BCOS配置文件

6.5.5 节点p2p连接文件peers.txt

节点p2p连接文件`peers.txt`为生成节点配置文件夹时指定的**其他机构**的节点连接信息，在使用`build_install_package`命令时，需要指定与本机构节点进行连接的节点p2p连接文件`peers.txt`，生成的本机构节点配置文件夹会根据该文件与其他节点进行通信。

采用`generate_all_certificates`命令的用户会根据在`conf`目录下填写的`node_deployment.ini`生成相应的`peers.txt`，采用其他方式生成证书的用户需要手动生成本机构节点的p2p连接文件并发送给对方，节点p2p连接文件的格式如下所示：

```
127.0.0.1:30300
127.0.0.1:30301
```

格式为 对应节点ip:p2p_listen_port

- 当需要与多机构节点通信时，需要将该文件合并

6.6 操作手册

FISCO BCOS generator 提供多种节点生成、扩容、群组划分、证书相关操作，简略介绍如下：

6.6.1 create_group_genesis (-c)

操作示例

```
$ cp node0/node.crt ./meta/cert_127.0.0.1_3030n.crt
...
$ vim ./conf/group_genesis.ini
$ ./generator --create_group_genesis ~/mydata
```

程序执行完成后，会在~/mydata文件夹下生成mgroup.ini中配置的`group.i.genesis`

用户生成的`group.i.genesis`即为群组的创世区块，即可完成新群组划分操作。

注解： FISCO BCOS 2.0中每个群组都会有一个群组创世区块。

6.6.2 build_install_package (-b)

操作示例

```
$ vim ./conf/node_deployment.ini
$ ./generator --build_install_package ./peers.txt ~/mydata
```

程序执行完成后，会在~/mydata文件夹下生成多个名为node_hostip_port的文件夹，推送到对应服务器后即可启动节点

6.6.3 generate_chain_certificate

```
$ ./generator --generate_chain_certificate ./dir_chain_ca
```

执行完成后用户可以在./dir_chain_ca文件夹下看到根证书`ca.crt` 和私钥`ca.key`。

6.6.4 generate_agency_certificate

```
$ ./generator --generate_agency_certificate ./dir_agency_ca ./chain_ca_dir The_
↪Agency_Name
```

执行完成后可以在./dir_agency_ca路径下生成名为The_Agency_Name的文件夹，包含相应的机构证书`agency.crt` 和私钥`agency.key`。

6.6.5 generate_node_certificate

```
$ ./generator --generate_node_certificate node_dir(SET) ./agency_dir  node_p2pip_
↪port
```

执行完成后可以在node_dir 路径下生成节点证书`node.crt` 和私钥`node.key`。

6.6.6 generate_sdk_certificate

```
$ ./generator --generate_sdk_certificate ./dir_sdk_ca ./dir_agency_ca
```

执行完成后可以在./dir_sdk_ca路径下生成名为SDK的文件夹，包含相应的SDK证书`node.crt` 和私钥`node.key`。

6.6.7 generate_all_certificates

```
$ ./generator --generate_all_certificates ./cert
```

注解： 上述命令会根据meta目录下存放的ca.crt、机构证书agency.crt和机构私钥agency.key生成相应的节点证书。

- 如果用户缺少上述三个文件，则无法生成节点证书，程序会抛出异常。

执行完成后会在./cert文件夹下生成节点的相关证书与私钥，并将节点证书放置于./meta下

6.6.8 merge_config (-m)

使用–merge_config命令可以合并两个config.ini中的p2p section

如 A目录下的config.ini文件的p2p section为

```
[p2p]
listen_ip = 127.0.0.1
listen_port = 30300
node.0 = 127.0.0.1:30300
node.1 = 127.0.0.1:30301
node.2 = 127.0.0.1:30302
node.3 = 127.0.0.1:30303
```

B目录下的config.ini文件的p2p section为

```
[p2p]
listen_ip = 127.0.0.1
listen_port = 30303
node.0 = 127.0.0.1:30300
node.1 = 127.0.0.1:30303
node.2 = 127.0.0.1:30300
node.3 = 127.0.0.1:30301
```

使用此命令后会成为：

```
[p2p]
listen_ip = 127.0.0.1
listen_port = 30304
node.0 = 127.0.0.1:30300
node.1 = 127.0.0.1:30301
node.2 = 127.0.0.1:30302
node.3 = 127.0.0.1:30303
node.4 = 127.0.0.1:30300
node.5 = 127.0.0.1:30301
```

使用示例

```
$ ./generator --merge_config ~/mydata/node_A/config.ini  ~/mydata/node_B/config.ini
```

使用成功后会将node_A和node_B的config.ini中p2p section合并与 ~/mydata/node_B/config.ini的文件中

6.6.9 deploy_private_key (-d)

使用–deploy_private_key可以将路径下名称相同的节点私钥导入到生成好的配置文件夹中。

使用示例:

```
$./generator --deploy_private_key ./cert ./data
```

如./cert下有名为node_127.0.0.1_30300，node_127.0.0.1_30301的文件夹，文件夹中有节点私钥文件node.key

./data下有名为node_127.0.0.1_30300，node_127.0.0.1_30301的配置文件夹

执行完成后可以将./cert下的对应的节点私钥导入./data的配置文件夹中

6.6.10 add_peers (-p)

使用–add_peers可以指定的peers文件导入到生成好的节点配置文件夹中。

使用示例:

```
$./generator --add_peers ./meta/peers.txt ./data
```

./data下有名为node_127.0.0.1_30300，node_127.0.0.1_30301的配置文件夹

执行完成后可以将peers文件中的连接信息导入./data下所有节点的配置文件`config.ini`中

6.6.11 add_group (-a)

使用–add_group可以指定的peers文件导入到生成好的节点配置文件夹中。

使用示例:

```
$./generator --add_group ./meta/group.2.genesis ./data
```

./data下有名为node_127.0.0.1_30300，node_127.0.0.1_30301的配置文件夹

执行完成后可以将群组2的连接信息导入./data下所有节点的`conf`文件夹中

6.6.12 download_fisco

使用–download_fisco可以指定的目录下下载`fisco-bcos`二进制文件，国内用户可以使用`--cdn`命令从cdn下载。

使用示例:

```
$./generator --download_fisco ./meta
```

或

```
$./generator --download_fisco ./meta --cdn
```

执行完成后会在./meta文件夹下下载`fisco-bcos`可执行二进制文件

6.6.13 download_console

使用–download_console可以指定的目录下下载并配置控制台，国内用户可以使用`--cdn`命令从cdn下载。。

使用示例:

```
$./generator --download_console ./meta
```

或

```
$./generator --download_console ./meta --cdn
```

执行完成后会在./meta文件夹下根据node_deployment.ini完成对控制台的配置

6.6.14 get_sdk_file

使用–get_sdk_file可以指定的目录下下获取控制台和sdk配置所需要的node.crt、node.key、ca.crt及applicationContext.xml。

使用示例:

```
$./generator --get_sdk_file ./sdk
```

执行完成后会在./sdk文件夹下根据node_deployment.ini生成上述配置文件

6.6.15 version (-v)

使用–version命令查看当前部署工具的版本号。

```
$ ./generator --version
```

6.6.16 help (-h)

用户可以使用-h或–help命令查看帮助菜单

使用示例:

```
$ ./generator -h
usage: generator [-h] [-v] [-b peer_path data_dir] [-c data_dir]
                 [--generate_chain_certificate chain_dir]
                 [--generate_agency_certificate agency_dir chain_dir agency_name]
                 [--generate_node_certificate node_dir agency_dir node_name]
                 [--generate_sdk_certificate sdk_dir agency_dir] [-g]
                 [--generate_all_certificates cert_dir] [-d cert_dir pkg_dir]
                 [-m config.ini config.ini] [-p peers config.ini]
                 [-a group genesis config.ini]
```

6.6.17 国密操作相关

FISCO BCOS generator的所有命令同时支持国密版fisco-bcos，使用时，国密证书、私钥均加以前缀gm。基本使用解释如下

国密开关 (-g)

国密开关-g打开时，生成证书、节点、群组创世区块的操作会相应生成国密版的上述文件。

生成证书操作

如generate_*_certificate操作时，配合-g命令会生成相应的国密证书。

操作示例：

```
$ ./generator --generate_all_certificates ./cert -g
```

注解： 上述命令会根据meta目录下存放的gmca.crt、机构证书gmagency.crt和机构私钥gmagency.key生成相应的节点证书。

- 如果用户缺少上述三个文件，则无法生成节点证书，程序会抛出异常。

生成国密群组创世区块

操作示例

```
$ cp node0/gmnode.crt ./meta/gmcert_127.0.0.1_3030n.crt
...
$ vim ./conf/group_genesis.ini
$ ./generator --create_group_genesis ~/mydata -g
```

程序执行完成后，会在~/mydata文件夹下生成mgroup.ini中配置的`group.i.genesis`

用户生成的`group.i.genesis`即为群组的创世区块，即可完成新群组划分操作。

生成国密节点配置文件夹

操作示例

```
$ vim ./conf/node_deployment.ini
$ ./generator --build_install_package ./peers.txt ~/mydata -g
```

程序执行完成后，会在~/mydata文件夹下生成多个名为node_hostip_port的文件夹，推送到对应服务器后即可启动节点

6.6.18 监控设计

FISCO BCOS generator 生成的节点配置文件夹中提供了内置的监控脚本，用户可以通过对其进行配置，将节点的告警信息发送至指定地址。FISCO BCOS generator会将monitor脚本放置于生成节点配置文件的指定目录下，假设用户指定生成的文件夹名为data，则monitor脚本会在data目录下的monitor文件夹下

使用方式如下：

```
$ cd ./data/monitor
```

用途如下：

1. 监控节点是否存活, 并且可以重新启动挂掉的节点.
2. 获取节点的块高和view信息, 判断节点共识是否正常.
3. 分析最近一分钟的节点日志打印, 收集日志关键错误打印信息, 准实时判断节点的状态.
4. 指定日志文件或者指定时间段, 分析节点的共识消息处理, 出块, 交易数量等信息, 判断节点的健康度.

配置告警服务

用户使用前，首先需要配置告警信息服务，这里以server酱的微信推送为例，可以参考配置server酱

绑定自己的github账号，以及微信后，可以使用本脚本向微信发送告警信息，使用本脚本的-s命令 可以向指定微信发送告警信息

如果用户希望使用其他服务，可以修改monitor.sh中的alarm() { # change http server }函数，个性化配置为自己需要的服务

help命令

使用help命令查看脚本使用方式

```
$ ./monitor.sh -h
Usage : bash monitor.sh
   -s : send alert to your address
   -m : monitor, statistics.  default : monitor .
   -f : log file to be analyzed.
   -o : dirpath
   -p : name of the monitored program , default is fisco-bcos
   -g : specified the group list to be analized
   -d : log analyze time range. default : 10(min), it should not bigger than max␣
↪value : 60(min).
   -r : setting alert receiver
   -h : help.
 example :
   bash  monitor.sh -s YourHttpAddr -o nodes -r your_name
   bash  monitor.sh -s YourHttpAddr -m statistics -o nodes -r your_name
   bash  monitor.sh -s YourHttpAddr -m statistics -f node0/log/log_2019021314.log -
↪g 1 2 -r your_name
```

命令解释如下：

- -s 指定告警配置地址，可以配置为告警上报服务的ip
- -m 设定监控模式，可以配置为statistics和monitor两种模式，默认为monitor模式。
- -f 分析节点log
- -o 指定节点路径
- -p 设定监控上报名称，默认为fisco-bcos
- -g 指定监控群组，默认分析所有群组
- -d log分析时间范围，默认10分钟内的log，最大不超过60分钟
- -r 指定上报接收者名称
- -h 帮助命令

使用示例

- 使用脚本监控指定路径下节点，发送给接收者Alice：

```
$ bash monitor.sh -s https://sc.ftqq.com/[SCKEY(登入后可见)].send -o alice/nodes -r␣
↪Alice
```

- 使用脚本统计指定路径下节点信息，发送给接收者Alice

```
$ bash monitor.sh -s https://sc.ftqq.com/[SCKEY(登入后可见)].send -m statistics -o␣
↪alice/nodes -r Alice
```

- 使用脚本统计指定路径下节点指定log指定群组1和群组2的信息，发送给接收者Alice

```
$ bash monitor.sh -s https://sc.ftqq.com/[SCKEY(登入后可见)].send -m statistics -f␣
↪node0/log/log_2019021314.log -g 1 2 -o alice/nodes -r Alice
```

6.6.19 handshake failed检测

FISCO BCOS generator 的scripts文件夹的check_certificates.sh脚本包含了节点log中提示handshake failed的异常检测。

获取脚本

如果用户需要检测由开发部署工具buildchain.sh生成的节点时，可以采用以下命令获取检测脚本：

```
curl -LO https://raw.githubusercontent.com/FISCO-BCOS/generator/master/scripts/
↪check_certificates.sh && chmod u+x check_certificates.sh
```

注解:

- 如果因为网络问题导致长时间无法下载，请尝试 *curl -LO https://gitee.com/FISCO-BCOS/generator/raw/master/scripts/check_certificates.sh && chmod u+x check_certificates.sh*

使用generator部署节点的用户可以从generator的根目录下，从scripts/check_certificates.sh获取脚本。

检测证书有效期

check_certificates.sh的-t命令会根据用户证书签发的有效期，以及当前的系统时间对证书进行检测。

使用示例：

```
$ ./check_certificates.sh -t ~/certificates.crt
```

参数第二项为任意符合x509格式的证书，验证成功时会提示check certificates time successful,验证失败会提示异常。

验证证书

check_certificates.sh的-v命令会根据用户指定的根证书从而验证节点证书。

```
$ ./check_certificates.sh -v ~/ca.crt ~/node.crt
```

验证成功时会提示use ~/ca.crt verify ~/node.crt successful,验证失败会提示异常。

6.6.20 节点配置错误检查

获取脚本

```
curl -LO https://raw.githubusercontent.com/FISCO-BCOS/FISCO-BCOS/master/tools/
↪check_node_config.sh && chmod u+x check_node_config.sh
```

注解:

- 如果因为网络问题导致长时间无法下载，请尝试 *curl -LO https://gitee.com/FISCO-BCOS/FISCO-BCOS/raw/master/tools/check_node_config.sh*

使用

使用下面的命令，脚本-p选项指定节点路径，脚本会根据路径下的config.ini分析配置是否有错误。

```
bash check_node_config.sh -p node_127.0.0.1_30300
```

CHAPTER 7

SDK

FISCO BCOS区块链向外部暴露了接口，外部业务程序能够通过FISCO BCOS提供的SDK来调用这些接口。开发者只需要根据自身业务程序的要求，选择相应语言的SDK，用SDK提供的API进行编程，即可实现对区块链的操作。

对接应用

目前，SDK接口可实现的功能包括（但不限于）：

- 合约操作
  - 合约编译、部署、查询
  - 交易发送、上链通知、参数解析、回执解析
- 链管理
  - 链状态查询、链参数设置
  - 组员管理
  - 权限设置
- 其它
  - SDK间的相互消息推送（AMOP）

内置控制台

为了方便开发者，部分SDK内置了控制台的功能。开发者可直接通过用命令行进行上述功能的操作。如编译合约、部署合约、发送交易、查询交易、链管理等等。

多种语言SDK

目前，FISCO BCOS提供的SDK包括：

- Java SDK （稳定、功能强大、无内置控制台）
- Python SDK （简单轻便、有内置控制台）
- Node-js SDK（简单轻便、有内置控制台）

7.1 Java SDK

Web3SDK可以支持访问节点、查询节点状态、修改系统设置和发送交易等功能。该版本（2.0）的技术文档只适用Web3SDK 2.0及以上版本(与FISCO BCOS 2.0及以上版本适配)，1.2.x版本的技术文档请查看Web3SDK 1.2.x版本技术文档。

2.0+版本主要特性包括：

- 提供调用FISCO BCOS 2.0 JSON-RPC的Java API
- 支持预编译（Precompiled）合约管理区块链
- 支持链上信使协议为联盟链提供安全高效的消息信道
- 支持使用国密算法发送交易
- 支持国密SSL与节点通信

7.1.1 环境要求

重要:

- Java版本

  JDK1.8 或者以上版本，推荐使用OracleJDK。

  注意：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。

- Java安装

  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Web3SDK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。

7.1.2 Java应用引入SDK

通过gradle或maven引入SDK到java应用

gradle:

```
compile ('org.fisco-bcos:web3sdk:2.5.0')
```

maven:

```
<dependency>
    <groupId>org.fisco-bcos</groupId>
    <artifactId>web3sdk</artifactId>
    <version>2.5.0</version>
</dependency>
```

由于引入了以太坊的solidity编译器相关jar包，需要在Java应用的gradle配置文件build.gradle中添加以太坊的远程仓库。

```
repositories {
        mavenCentral()
        maven { url "https://dl.bintray.com/ethereum/maven/" }
    }
```

注： 如果下载Web3SDK的依赖solcJ-all-0.4.25.jar速度过慢，可以参考这里进行下载。

7.1.3 配置SDK

证书配置

FISCO BCOS作为联盟链，SDK连接区块链节点时通过SSL进行双向认证。JavaSDK支持SSL与国密SSL两种认证方式。

SSL连接配置

国密区块链和非国密区块链环境下，节点与SDK之间均可以建立SSL的连接，将节点所在目录nodes/${ip}/sdk/目录下的ca.crt、sdk.crt和sdk.key文件拷贝到项目的资源目录。（低于2.1版本的FISCO BCOS节点目录下只有node.crt和node.key，需将其重命名为sdk.crt和sdk.key以兼容最新的SDK）

国密SSL连接配置

FISCO-BCOS 2.5及之后的版本，在国密区块链环境下支持节点与SDK建立国密SSL连接，将节点所在目录nodes/${ip}/sdk/gm/目录下的gmca.crt、gmensdk.crt、gmensdk.key、gmsdk.crt、gmsdk.key文件拷贝到项目的资源目录。

重要:

- 国密SSL连接只有在国密区块链环境下才可以使用。
- 是否选择国密SSL连接，SDK与区块链节点的配置要保持一致，节点配置参考 配置链属性

配置文件设置

Java应用的配置文件需要做相关配置。值得关注的是，FISCO BCOS 2.0+版本支持多群组功能，SDK需要配置群组的节点信息。将以Spring项目和Spring Boot项目为例，提供配置指引。

Spring项目配置

提供Spring项目中关于applicationContext.xml的配置下所示。

```
<?xml version="1.0" encoding="UTF-8" ?>

<beans xmlns="http://www.springframework.org/schema/beans"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://
↪www.springframework.org/schema/p"
           xmlns:tx="http://www.springframework.org/schema/tx" xmlns:aop="http://
↪www.springframework.org/schema/aop"
           xmlns:context="http://www.springframework.org/schema/context"
           xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
         http://www.springframework.org/schema/tx
```

(continues on next page)

(续上页)

```
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
         http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">


        <bean id="encryptType" class="org.fisco.bcos.web3j.crypto.EncryptType">
                <constructor-arg value="0"/> <!-- 0:standard 1:guomi -->
        </bean>

        <bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.
↪handler.GroupChannelConnectionsConfig">
                <!-- SSL certificate configuration -->
                <property name="caCert" value="ca.crt" />
                <property name="sslCert" value="sdk.crt" />
                <property name="sslKey" value="sdk.key" />
                <!-- GM SSL certificate configuration -->
                <property name="gmCaCert" value="gmca.crt" />
                <property name="gmEnSslCert" value="gmensdk.crt" />
                <property name="gmEnSslKey" value="gmensdk.key" />
                <property name="gmSslCert" value="gmsdk.crt" />
                <property name="gmSslKey" value="gmsdk.key" />
                <property name="allChannelConnections">
                        <list>  <!-- 每个群组需要配置一个bean，每个群组可以配置多个节点 -->
                                <bean id="group1"  class="org.fisco.bcos.channel.
↪handler.ChannelConnections">
                                        <property name="groupId" value="1" /> <!--␣
↪群组的groupID -->
                                        <property name="connectionsStr">
                                                <list>
                                                        <value>127.0.0.1:20200</
↪value>  <!-- IP:channel_port -->
                                                        <value>127.0.0.1:20201</
↪value>
                                                </list>
                                        </property>
                                </bean>
                                <bean id="group2"  class="org.fisco.bcos.channel.
↪handler.ChannelConnections">
                                        <property name="groupId" value="2" /> <!--␣
↪群组的groupID -->
                                        <property name="connectionsStr">
                                                <list>
                                                        <value>127.0.0.1:20202</
↪value>
                                                        <value>127.0.0.1:20203</
↪value>
                                                </list>
                                        </property>
                                </bean>
                        </list>
                </property>
        </bean>

        <bean id="channelService" class="org.fisco.bcos.channel.client.Service"␣
↪depends-on="groupChannelConnectionsConfig">
                <property name="groupId" value="1" /> <!-- 配置连接群组1 -->
                <property name="agencyName" value="fisco" /> <!-- 配置机构名 -->
                <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>
        </bean>
```

(continues on next page)

(续上页)

```
</beans>
```

applicationContext.xml配置项详细说明:

- encryptType: 国密开关(默认为0，关闭)
  - 0: 不使用国密
  - 1: 使用国密
    - 开启国密功能，需要连接的区块链节点是国密节点，搭建国密版FISCO BCOS区块链参考这里)
- groupChannelConnectionsConfig:
  - 配置待连接的群组，可以配置一个或多个群组，每个群组需要配置群组ID
  - 每个群组可以配置一个或多个节点，设置群组节点的配置文件**config.ini**中[rpc]部分的channel_listen_ip(若节点小于v2.3.0版本，查看配置项listen_ip)和channel_listen_port。
  - SSL配置项: SDK与节点SSL连接时使用
    - caCertSL连接根证书路径
    - sslCertSDK证书路径
    - sslKeySDK证书私钥路径
  - 国密SSL配置项: SDK与节点国密SSL连接时使用
    - gmCaCert国密SSL连接根证书路径
    - gmEnSslCert国密SSL连接加密证书路径
    - gmEnSslKey国密SSL连接加密证书私钥路径
    - gmSslCertSSL连接签名证书路径
    - gmSslKeySSL连接签名证书私钥路径
- channelService: 通过指定群组ID配置SDK实际连接的群组，指定的群组ID是groupChannelConnectionsConfig配置中的群组ID。SDK会与群组中配置的节点均建立连接，然后随机选择一个节点发送请求。

备注：刚下载项目时，有些插件可能没有安装，代码会报错。当你第一次在IDEA上使用lombok这个工具包时，请按以下步骤操作:

- 进入setting->Plugins->Marketplace->选择安装Lombok plugin
- 进入设置Setting-> Compiler -> Annotation Processors -> 勾选Enable annotation processing。

Spring Boot项目配置

提供Spring Boot项目中关于application.yml的配置如下所示。

```
encrypt-type: # 0: 普通， 1: 国密
  encrypt-type: 0

group-channel-connections-config:
  caCert: ca.crt
  sslCert: sdk.crt
  sslKey: sdk.key
  all-channel-connections:
    - group-id: 1 #group ID
      connections-str:
        # 若节点小于v2.3.0版本，查看配置项listen_ip:channel_listen_port
```

(continues on next page)

(续上页)

```
        - 127.0.0.1:20200 # node channel_listen_ip:channel_listen_port
        - 127.0.0.1:20201
    - group-id: 2
      connections-str:
        # 若节点小于v2.3.0版本，查看配置项listen_ip:channel_listen_port
        - 127.0.0.1:20202 # node channel_listen_ip:channel_listen_port
        - 127.0.0.1:20203
channel-service:
  group-id: 1 # sdk实际连接的群组
  agency-name: fisco # 机构名称
```

application.yml配置项与applicationContext.xml配置项相对应，详细介绍参考applicationContext.xml配置说明。

7.1.4 使用SDK

Spring项目开发指引

调用SDK的API(参考Web3SDK API列表设置或查询相关的区块链数据。

调用SDK Web3j的API

加载配置文件，SDK与区块链节点建立连接，获取web3j对象，根据Web3j对象调用相关API。示例代码如下：

```
    //读取配置文件，SDK与区块链节点建立连接
    ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
    Service service = context.getBean(Service.class);
    service.run();
    ChannelEthereumService channelEthereumService = new ChannelEthereumService();
    channelEthereumService.setChannelService(service);

    //获取Web3j对象
    Web3j web3j = Web3j.build(channelEthereumService, service.getGroupId());
    //通过Web3j对象调用API接口getBlockNumber
    BigInteger blockNumber = web3j.getBlockNumber().send().getBlockNumber();
    System.out.println(blockNumber);
```

注： SDK处理交易超时时间默认为60秒，即60秒内没有收到交易响应，判断为超时。该值可以通过ChannelEthereumService进行设置，示例如下：

```
// 设置交易超时时间为100000毫秒，即100秒
channelEthereumService.setTimeout(100000);
```

调用SDK Precompiled的API

加载配置文件，SDK与区块链节点建立连接。获取SDK Precompiled Service对象，调用相关的API。示例代码如下：

```
    //读取配置文件，SDK与区块链节点建立连接，获取Web3j对象
    ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
    Service service = context.getBean(Service.class);
    service.run();
    ChannelEthereumService channelEthereumService = new ChannelEthereumService();
```

(continues on next page)

(续上页)

```
    channelEthereumService.setChannelService(service);
    Web3j web3j = Web3j.build(channelEthereumService, service.getGroupId());
    String privateKey =
↪"b83261efa42895c38c6c2364ca878f43e77f3cddbc922bf57d0d48070f79feb6";
    //指定外部账户私钥，用于交易签名
    Credentials credentials = GenCredential.create(privateKey);
    //获取SystemConfigService对象
    SystemConfigService systemConfigService = new SystemConfigService(web3j,␣
↪credentials);
    //通过SystemConfigService对象调用API接口setValueByKey
    String result = systemConfigService.setValueByKey("tx_count_limit", "2000");
    //通过Web3j对象调用API接口getSystemConfigByKey
    String value = web3j.getSystemConfigByKey("tx_count_limit").send().
↪getSystemConfigByKey();
    System.out.println(value);
```

创建并使用指定外部账户

SDK发送交易需要一个外部账户，下面是随机创建一个外部账户的方法。

```
//创建普通外部账户
EncryptType.encryptType = 0;
//创建国密外部账户，向国密区块链节点发送交易需要使用国密外部账户
// EncryptType.encryptType = 1;
Credentials credentials = GenCredential.create();
//账户地址
String address = credentials.getAddress();
//账户私钥
String privateKey = credentials.getEcKeyPair().getPrivateKey().toString(16);
//账户公钥
String publicKey = credentials.getEcKeyPair().getPublicKey().toString(16);
```

使用指定的外部账户

```
//通过指定外部账户私钥使用指定的外部账户
Credentials credentials = GenCredential.create(privateKey);
```

加载账户私钥文件

如果通过账户生成脚本get_accounts.sh生成了PEM或PKCS12格式的账户私钥文件(账户生成脚本的用法参考账户管理文档)，则可以通过加载PEM或PKCS12账户私钥文件使用账户。加载私钥有两个类：P12Manager和PEMManager，其中，P12Manager用于加载PKCS12格式的私钥文件，PEMManager用于加载PEM格式的私钥文件。

- P12Manager用法举例： 在applicationContext.xml中配置PKCS12账户的私钥文件路径和密码

```
<bean id="p12" class="org.fisco.bcos.channel.client.P12Manager" init-method="load"␣
↪>
        <property name="password" value="123456" />
        <property name="p12File" value=
↪"classpath:0x0fc3c4bb89bd90299db4c62be0174c4966286c00.p12" />
</bean>
```

开发代码

```
//加载Bean
ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
```

(continues on next page)

(续上页)

```
P12Manager p12 = context.getBean(P12Manager.class);
//提供密码获取ECKeyPair，密码在生产p12账户文件时指定
ECKeyPair p12KeyPair = p12.getECKeyPair(p12.getPassword());

//以十六进制串输出私钥和公钥
System.out.println("p12 privateKey: " + p12KeyPair.getPrivateKey().toString(16));
System.out.println("p12 publicKey: " + p12KeyPair.getPublicKey().toString(16));

//生成Web3SDK使用的Credentials
Credentials credentials = GenCredential.create(p12KeyPair.getPrivateKey().
↪toString(16));
System.out.println("p12 Address: " + credentials.getAddress());
```

- PEMManager使用举例

在applicationContext.xml中配置PEM账户的私钥文件路径

```
<bean id="pem" class="org.fisco.bcos.channel.client.PEMManager" init-method="load"␣
↪>
        <property name="pemFile" value=
↪"classpath:0x0fc3c4bb89bd90299db4c62be0174c4966286c00.pem" />
</bean>
```

使用代码加载

```
//加载Bean
ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext-keystore-sample.xml");
PEMManager pem = context.getBean(PEMManager.class);
ECKeyPair pemKeyPair = pem.getECKeyPair();

//以十六进制串输出私钥和公钥
System.out.println("PEM privateKey: " + pemKeyPair.getPrivateKey().toString(16));
System.out.println("PEM publicKey: " + pemKeyPair.getPublicKey().toString(16));

//生成Web3SDK使用的Credentials
Credentials credentialsPEM = GenCredential.create(pemKeyPair.getPrivateKey().
↪toString(16));
System.out.println("PEM Address: " + credentialsPEM.getAddress());
```

通过SDK部署并调用合约

准备Java合约文件

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，具体使用方式参考这里。

部署并调用合约

SDK的核心功能是部署/加载合约，然后调用合约相关接口，实现相关业务功能。部署合约调用Java合约类的deploy方法，获取合约对象。通过合约对象可以调用getContractAddress方法获取部署合约的地址以及调用该合约的其他方法实现业务功能。如果合约已部署，则通过部署的合约地址可以调用load方法加载合约对象，然后调用该合约的相关方法。

```
    //读取配置文件，SDK与区块链节点建立连接，获取web3j对象
    ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
```

(continues on next page)

(续上页)

```
    Service service = context.getBean(Service.class);
    service.run();
    ChannelEthereumService channelEthereumService = new ChannelEthereumService();
    channelEthereumService.setChannelService(service);
    channelEthereumService.setTimeout(10000);
    Web3j web3j = Web3j.build(channelEthereumService, service.getGroupId());
    //准备部署和调用合约的参数
    BigInteger gasPrice = new BigInteger("300000000");
    BigInteger gasLimit = new BigInteger("300000000");
    String privateKey =
↪"b83261efa42895c38c6c2364ca878f43e77f3cddbc922bf57d0d48070f79feb6";
    //指定外部账户私钥，用于交易签名
    Credentials credentials = GenCredential.create(privateKey);
    //部署合约
    YourSmartContract contract = YourSmartContract.deploy(web3j, credentials, new␣
↪StaticGasProvider(gasPrice, gasLimit)).send();
    //根据合约地址加载合约
    //YourSmartContract contract = YourSmartContract.load(address, web3j,␣
↪credentials, new StaticGasProvider(gasPrice, gasLimit));
    //调用合约方法发送交易
    TransactionReceipt transactionReceipt = contract.someMethod(<param1>, ...).
↪send();
    //查询合约方法查询该合约的数据状态
    Type result = contract.someMethod(<param1>, ...).send();
```

Spring Boot项目开发指引

提供spring-boot-starter示例项目供参考。Spring Boot项目开发与Spring项目开发类似，其主要区别在于配置文件方式的差异。该示例项目提供相关的测试案例，具体描述参考示例项目的README文档。

SDK国密功能使用

- 前置条件：FISCO BCOS区块链采用国密算法，搭建国密版的FISCO BCOS区块链请参考国密使用手册。
- 启用国密功能：`applicationContext.xml/application.yml`配置文件中将`encryptType`属性设置为1。
- 加载私钥使用`GenCredential`类(适用于国密和非国密)，`Credential`类只适用于加载非国密私钥。

国密版SDK调用API的方式与普通版SDK调用API的方式相同，其差异在于国密版SDK需要生成国密版的Java合约文件。编译国密版的Java合约文件参考这里。

7.1.5 Web3SDK API

Web3SDK API主要分为Web3j API和Precompiled Service API。其中Web3j API可以查询区块链相关的状态，发送和查询交易信息；Precompiled Service API可以管理区块链相关配置以及实现特定功能。

Web3j API

Web3j API是由web3j对象调用的FISCO BCOS的RPC API，其API名称与RPC API相同，参考RPC API文档。

Precompiled Service API

预编译合约是FISCO BCOS底层通过C++实现的一种高效智能合约。SDK已提供预编译合约对应的Java接口，控制台通过调用这些Java接口实现了相关的操作命令，体验控制台，参考控制台手册。SDK提供Precompiled对应的Service类，分别是分布式控制权限相关的PermissionService，CNS相关的CnsService，系统属性配置相关的SystemConfigService和节点类型配置相关ConsensusService。相关错误码请参考：Precompiled Service API 错误码

PermissionService

SDK提供对分布式控制权限的支持，PermissionService可以配置权限信息，其API如下：

- **public String grantUserTableManager(String tableName, String address)**： 根据用户表名和外部账户地址设置权限信息。
- **public String revokeUserTableManager(String tableName, String address)**： 根据用户表名和外部账户地址去除权限信息。
- **public List<PermissionInfo> listUserTableManager(String tableName)**： 根据用户表名查询设置的权限记录列表(每条记录包含外部账户地址和生效块高)。
- **public String grantDeployAndCreateManager(String address)**： 增加外部账户地址的部署合约和创建用户表权限。
- **public String revokeDeployAndCreateManager(String address)**： 移除外部账户地址的部署合约和创建用户表权限。
- **public List<PermissionInfo> listDeployAndCreateManager()**： 查询拥有部署合约和创建用户表权限的权限记录列表。
- **public String grantPermissionManager(String address)**： 增加外部账户地址的管理权限的权限。
- **public String revokePermissionManager(String address)**： 移除外部账户地址的管理权限的权限。
- **public List<PermissionInfo> listPermissionManager()**： 查询拥有管理权限的权限记录列表。
- **public String grantNodeManager(String address)**： 增加外部账户地址的节点管理权限。
- **public String revokeNodeManager(String address)**： 移除外部账户地址的节点管理权限。
- **public List<PermissionInfo> listNodeManager()**： 查询拥有节点管理的权限记录列表。
- **public String grantCNSManager(String address)**： 增加外部账户地址的使用CNS权限。
- **public String revokeCNSManager(String address)**： 移除外部账户地址的使用CNS权限。
- **public List<PermissionInfo> listCNSManager()**： 查询拥有使用CNS的权限记录列表。
- **public String grantSysConfigManager(String address)**： 增加外部账户地址的系统参数管理权限。
- **public String revokeSysConfigManager(String address)**： 移除外部账户地址的系统参数管理权限。
- **public List<PermissionInfo> listSysConfigManager()**： 查询拥有系统参数管理的权限记录列表。

CnsService

SDK提供对CNS的支持。CnsService可以配置CNS信息，其API如下：

- **String registerCns(String name, String version, String address, String abi)**： 根据合约名、合约版本号、合约地址和合约abi注册CNS信息。
- **String getAddressByContractNameAndVersion(String contractNameAndVersion)**： 根据合约名和合约版本号(合约名和合约版本号用英文冒号连接)查询合约地址。若缺失合约版本号，默认使用合约最新版本。
- **List<CnsInfo> queryCnsByName(String name)**： 根据合约名查询CNS信息。

- **List<CnsInfo> queryCnsByNameAndVersion(String name, String version)**：根据合约名和合约版本号查询CNS信息。

SystemConfigService

SDK提供对系统配置的支持。SystemConfigService可以配置系统属性值（目前支持tx_count_limit和tx_gas_limit属性的设置），其API如下：

- **String setValueByKey(String key, String value)**：根据键设置对应的值（查询键对应的值，参考Web3j API中的getSystemConfigByKey接口）。

ConsensusService

SDK提供对节点类型配置的支持。ConsensusService可以设置节点类型，其API如下：

- **String addSealer(String nodeId)**：根据节点NodeID设置对应节点为共识节点。
- **String addObserver(String nodeId)**：根据节点NodeID设置对应节点为观察节点。
- **String removeNode(String nodeId)**：根据节点NodeID设置对应节点为游离节点。

CRUDService

SDK提供对CRUD(增删改查)操作的支持。CRUDService可以创建表，对表进行增删改查操作，其API如下：

- **int createTable(Table table)**：创建表，提供表对象。表对象需要设置其表名，主键字段名和其他字段名。其中，其他字段名是以英文逗号分隔拼接的字符串。返回创建表的状态值，返回为0则代表创建成功。
- **int insert(Table table, Entry entry)**：插入记录，提供表对象和Entry对象。表对象需要设置表名和主键值；Entry是map对象，提供插入的字段名和字段值。返回插入的记录数。
- **int update(Table table, Entry entry, Condition condition)**：更新记录，提供表对象，Entry对象和Condtion对象。表对象需要设置表名和主键值；Entry是map对象，提供更新的字段名和字段值；Condition对象是条件对象，可以设置更新的匹配条件。返回更新的记录数。
- **List<Map<String, String>> select(Table table, Condition condition)**：查询记录，提供表对象和Condtion对象。表对象需要设置表名和主键值；Condition对象是条件对象，可以设置查询的匹配条件。返回查询的记录。
- **int remove(Table table, Condition condition)**：移除记录，提供表对象和Condtion对象。表对象需要设置表名和主键值；Condition对象是条件对象，可以设置移除的匹配条件。返回移除的记录数。
- **Table desc(String tableName)**：根据表名查询表的信息，主要包含表的主键和其他属性字段。返回表类型，主要包含表的主键字段名和其他属性字段名。

7.1.6 交易解析

FISCO BCOS的交易是一段发往区块链系统的请求数据，用于部署合约，调用合约接口，维护合约的生命周期以及管理资产，进行价值交换等。当交易确认后会产生交易回执，交易回执和交易均保存在区块里，用于记录交易执行过程生成的信息，如结果码、日志、消耗的gas量等。用户可以使用交易哈希查询交易回执，判定交易是否完成。

交易回执包含三个关键字段，分别是input, output , logs:

交易解析功能帮助用户解析这三个字段为json数据和java对象。

接口说明

代码包路径`org.fisco.bcos.web3j.tx.txdecode`，使用`TransactionDecoderFactory`工厂类建立交易解析对象`TransactionDecoder`，有两种方式：

1. `TransactionDecoder buildTransactionDecoder(String abi, String bin);`

   abi：合约的ABI

   bin：合约bin，暂无使用，可以直接传入空字符串""

2. `TransactionDecoder buildTransactionDecoder(String contractName);`

   contractName：合约名称，在应用的根目录下创建`solidity`目录，将交易相关的合约放在`solidity`目录，通过指定合约名获取交易解析对象

交易解析对象`TransactionDecoder`接口列表：

1. `String decodeInputReturnJson(String input)`

   解析input，将结果封装为json字符串，json格式

   ```
   {"data":[{"name":"","type":"","data":} ... ],"function":"","methodID":""}
   ```

   function : 函数签名字符串

   methodID : 函数选择器

2. `InputAndOutputResult decodeInputReturnObject(String input)`

   解析input，返回Object对象，InputAndOutputResult结构:

   ```
   public class InputAndOutputResult {
      private String function; // 函数签名
      private String methodID; // methodID
      private List<ResultEntity> result; // 返回列表
    }

   public class ResultEntity {
      private String name;  // 字段名称, 解析output返回时，值为空字符串
      private String type;  // 字段类型
      private Object data;  // 字段值
    }
   ```

3. `String decodeOutputReturnJson(String input, String output)`

   解析output，将结果封装为json字符串，格式同`decodeInputReturnJson`

4. `InputAndOutputResult decodeOutputReturnObject(String input, String output)`

   解析output，返回java Object对象

5. `String decodeEventReturnJson(List<Log> logList)`

   解析event列表，将结果封装为json字符串，json格式

   ```
   {"event1签名":[[{"name":"","type":"","data":}...]...],"event2签名":[[{"name":"",
   ↪"type":"","data":}...]...]...}
   ```

6. `Map<String, List<List<ResultEntity>>> decodeEventReturnObject(List<Log> logList)`

   解析event列表，返回java Map对象，key为event签名字符串，`List<ResultEntity>`为交易中单个event参数列表，`List<List<ResultEntity>>`表示单个交易可以包含多个event

`TransactionDecoder`对input，output和event logs均分别提供返回json字符串和java对象的方法。json字符串方便客户端处理数据，java对象方便服务端处理数据。

示例

以TxDecodeSample合约为例说明接口的使用：

```
pragma solidity ^0.4.24;
contract TxDecodeSample
{
    event Event1(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _
↪s,bytes _bs);
    event Event2(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _
↪s,bytes  _bs);

    function echo(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string␣
↪_s,bytes _bs) public constant returns (uint256,int256,bool,address,bytes32,
↪string,bytes)
    {
      Event1(_u, _i, _b, _addr, _bs32, _s, _bs);
      return (_u, _i, _b, _addr, _bs32, _s, _bs);
    }

    function do_event(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32,␣
↪string _s,bytes _bs) public
    {
      Event1(_u, _i, _b, _addr, _bs32, _s, _bs);
      Event2(_u, _i, _b, _addr, _bs32, _s, _bs);
    }
}
```

使用buildTransactionDecoder 创建TxDecodeSample合约的解析对象：

```
// TxDecodeSample合约ABI
String abi = "[{\"constant\":false,\"inputs\":[{\"name\":\"_u\",\"type\":\"uint256\
↪"},{\"name\":\"_i\",\"type\":\"int256\"},{\"name\":\"_b\",\"type\":\"bool\"},{\
↪"name\":\"_addr\",\"type\":\"address\"},{\"name\":\"_bs32\",\"type\":\"bytes32\"}
↪,{\"name\":\"_s\",\"type\":\"string\"},{\"name\":\"_bs\",\"type\":\"bytes\"}],\
↪"name\":\"do_event\",\"outputs\":[],\"payable\":false,\"stateMutability\":\
↪"nonpayable\",\"type\":\"function\"},{\"anonymous\":false,\"inputs\":[{\"indexed\
↪":false,\"name\":\"_u\",\"type\":\"uint256\"},{\"indexed\":false,\"name\":\"_i\",
↪\"type\":\"int256\"},{\"indexed\":false,\"name\":\"_b\",\"type\":\"bool\"},{\
↪"indexed\":false,\"name\":\"_addr\",\"type\":\"address\"},{\"indexed\":false,\
↪"name\":\"_bs32\",\"type\":\"bytes32\"},{\"indexed\":false,\"name\":\"_s\",\
↪"type\":\"string\"},{\"indexed\":false,\"name\":\"_bs\",\"type\":\"bytes\"}],\
↪"name\":\"Event1\",\"type\":\"event\"},{\"anonymous\":false,\"inputs\":[{\
↪"indexed\":false,\"name\":\"_u\",\"type\":\"uint256\"},{\"indexed\":false,\"name\
↪":\"_i\",\"type\":\"int256\"},{\"indexed\":false,\"name\":\"_b\",\"type\":\"bool\
↪"},{\"indexed\":false,\"name\":\"_addr\",\"type\":\"address\"},{\"indexed\
↪":false,\"name\":\"_bs32\",\"type\":\"bytes32\"},{\"indexed\":false,\"name\":\"_
↪s\",\"type\":\"string\"},{\"indexed\":false,\"name\":\"_bs\",\"type\":\"bytes\"}
↪],\"name\":\"Event2\",\"type\":\"event\"},{\"constant\":true,\"inputs\":[{\"name\
↪":\"_u\",\"type\":\"uint256\"},{\"name\":\"_i\",\"type\":\"int256\"},{\"name\":\
↪"_b\",\"type\":\"bool\"},{\"name\":\"_addr\",\"type\":\"address\"},{\"name\":\"_
↪bs32\",\"type\":\"bytes32\"},{\"name\":\"_s\",\"type\":\"string\"},{\"name\":\"_
↪bs\",\"type\":\"bytes\"}],\"name\":\"echo\",\"outputs\":[{\"name\":\"\",\"type\
↪":\"uint256\"},{\"name\":\"\",\"type\":\"int256\"},{\"name\":\"\",\"type\":\
↪"bool\"},{\"name\":\"\",\"type\":\"address\"},{\"name\":\"\",\"type\":\"bytes32\
↪"},{\"name\":\"\",\"type\":\"string\"},{\"name\":\"\",\"type\":\"bytes\"}],\
↪"payable\":false,\"stateMutability\":\"view\",\"type\":\"function\"}]";
String bin = "";
TransactionDecoder txDecodeSampleDecoder = TransactionDecoderFactory.
↪buildTransactionDecoder(abi, bin);
```

解析input

调用function echo(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _s,bytes _bs) 接口，输入参数为[111111 -1111111 false 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a abcdefghiabcdefghiabcdefghiabhji FISCO-BCOS nice]

```
// function echo(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _
↪s,bytes _bs)
String input =
↪"0x406d373b0000000000000000000000000000000000000000000000000000000000000001b207fffffffffffffffffff
↪";
String jsonResult = txDecodeSampleDecoder.decodeInputReturnJson(input);
InputAndOutputResult objectResult = txDecodeSampleDecoder.
↪decodeInputReturnObject(input);
System.out.println("json => \n" + jsonResult);
System.out.println("object => \n" + objectResult);
```

输出：

```
json =>
{
  "function": "echo(uint256,int256,bool,address,bytes32,string,bytes)",
  "methodID": "0x406d373b",
  "result": [
    {
      "name": "_u",
      "type": "uint256",
      "data": 111111
    },
    {
      "name": "_i",
      "type": "int256",
      "data": -1111111
    },
    {
      "name": "_b",
      "type": "bool",
      "data": false
    },
    {
      "name": "_addr",
      "type": "address",
      "data": "0x692a70d2e424a56d2c6c27aa97d1a86395877b3a"
    },
    {
      "name": "_bs32",
      "type": "bytes32",
      "data": "abcdefghiabcdefghiabcdefghiabhji"
    },
    {
      "name": "_s",
      "type": "string",
      "data": "FISCO-BCOS"
    },
    {
      "name": "_bs",
      "type": "bytes",
      "data": "nice"
    }
  ]
}
```

(continues on next page)

(续上页)

```
object =>
InputAndOutputResult[
  function=echo(uint256,
  int256,
  bool,
  address,
  bytes32,
  string,
  bytes),
  methodID=0x406d373b,
  result=[
    ResultEntity[
      name=_u,
      type=uint256,
      data=111111
    ],
    ResultEntity[
      name=_i,
      type=int256,
      data=-1111111
    ],
    ResultEntity[
      name=_b,
      type=bool,
      data=false
    ],
    ResultEntity[
      name=_addr,
      type=address,
      data=0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
    ],
    ResultEntity[
      name=_bs32,
      type=bytes32,
      data=abcdefghiabcdefghiabcdefghiabhji
    ],
    ResultEntity[
      name=_s,
      type=string,
      data=FISCO-BCOS
    ],
    ResultEntity[
      name=_bs,
      type=bytes,
      data=nice
    ]
  ]
]
```

解析output

调用function echo(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _s,bytes _bs) 接口，输入参数为[111111 -1111111 false 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a abcdefghiabcdefghiabcdefghiabhji FISCO-BCOS nice]，echo接口直接将输入返回，因此返回与输入相同

```
//  function echo(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string␣
↪_s,bytes _bs)  public constant returns (uint256,int256,bool,address,bytes32,
↪string,bytes)
```

(continues on next page)

(续上页)

```
String input =␣
↪"0x406d373b000000000000000000000000000000000000000000000000000000000001b207ffffffffffffffffffff
↪

String output =
↪""0x000000000000000000000000000000000000000000000000000000000001b207ffffffffffffffffffffffffff
↪";

String jsonResult = txDecodeSampleDecoder.decodeOutputReturnJson(input, output);
InputAndOutputResult objectResult = txDecodeSampleDecoder.
↪decodeOutputReturnObject(input, output);
System.out.println("json => \n" + jsonResult);
System.out.println("object => \n" + objectResult);
```

结果:

```
json =>
{
  "function": "echo(uint256,int256,bool,address,bytes32,string,bytes)",
  "methodID": "0x406d373b",
  "result": [
    {
      "name": "",
      "type": "uint256",
      "data": 111111
    },
    {
      "name": "",
      "type": "int256",
      "data": -1111111
    },
    {
      "name": "",
      "type": "bool",
      "data": false
    },
    {
      "name": "",
      "type": "address",
      "data": "0x692a70d2e424a56d2c6c27aa97d1a86395877b3a"
    },
    {
      "name": "",
      "type": "bytes32",
      "data": "abcdefghiabcdefghiabcdefghiabhji"
    },
    {
      "name": "",
      "type": "string",
      "data": "FISCO-BCOS"
    },
    {
      "name": "",
      "type": "bytes",
      "data": "nice"
    }
  ]
}

object =>
InputAndOutputResult[
```

(continues on next page)

(续上页)

```
  function=echo(uint256,
  int256,
  bool,
  address,
  bytes32,
  string,
  bytes),
  methodID=0x406d373b,
  result=[
    ResultEntity[
      name=,
      type=uint256,
      data=111111
    ],
    ResultEntity[
      name=,
      type=int256,
      data=-1111111
    ],
    ResultEntity[
      name=,
      type=bool,
      data=false
    ],
    ResultEntity[
      name=,
      type=address,
      data=0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
    ],
    ResultEntity[
      name=,
      type=bytes32,
      data=abcdefghiabcdefghiabcdefghiabhji
    ],
    ResultEntity[
      name=,
      type=string,
      data=FISCO-BCOS
    ],
    ResultEntity[
      name=,
      type=bytes,
      data=nice
    ]
  ]
]
```

解析event logs

调用function do_event(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _s,bytes _bs) 接口，输入参数为[111111 -1111111 false 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a abcdefghiabcdefghiabcdefghiabhji FISCO-BCOS nice]，解析交易中的logs

```
// transactionReceipt为调用do_event接口的交易回执
String jsonResult = txDecodeSampleDecoder.decodeEventReturnJson(transactionReceipt.
↪getLogs());
String mapResult = txDecodeSampleDecoder.decodeEventReturnJson(transactionReceipt.
↪getLogs());
```

(continues on next page)

(续上页)

```
System.out.println("json => \n" + jsonResult);
System.out.println("map => \n" + mapResult);
```

结果:

```
json =>
{
  "Event1(uint256,int256,bool,address,bytes32,string,bytes)": [
    [
      {
        "name": "_u",
        "type": "uint256",
        "data": 111111,
        "indexed": false
      },
      {
        "name": "_i",
        "type": "int256",
        "data": -1111111,
        "indexed": false
      },
      {
        "name": "_b",
        "type": "bool",
        "data": false,
        "indexed": false
      },
      {
        "name": "_addr",
        "type": "address",
        "data": "0x692a70d2e424a56d2c6c27aa97d1a86395877b3a",
        "indexed": false
      },
      {
        "name": "_bs32",
        "type": "bytes32",
        "data": "abcdefghiabcdefghiabcdefghiabhji",
        "indexed": false
      },
      {
        "name": "_s",
        "type": "string",
        "data": "Fisco Bcos",
        "indexed": false
      },
      {
        "name": "_bs",
        "type": "bytes",
        "data": "sadfljkjkljkl",
        "indexed": false
      }
    ]
  ],
  "Event2(uint256,int256,bool,address,bytes32,string,bytes)": [
    [
      {
        "name": "_u",
        "type": "uint256",
        "data": 111111,
        "indexed": false
```

(continues on next page)

(续上页)

```
      },
      {
        "name": "_i",
        "type": "int256",
        "data": -1111111,
        "indexed": false
      },
      {
        "name": "_b",
        "type": "bool",
        "data": false,
        "indexed": false
      },
      {
        "name": "_addr",
        "type": "address",
        "data": "0x692a70d2e424a56d2c6c27aa97d1a86395877b3a",
        "indexed": false
      },
      {
        "name": "_bs32",
        "type": "bytes32",
        "data": "abcdefghiabcdefghiabcdefghiabhji",
        "indexed": false
      },
      {
        "name": "_s",
        "type": "string",
        "data": "FISCO-BCOS",
        "indexed": false
      },
      {
        "name": "_bs",
        "type": "bytes",
        "data": "nice",
        "indexed": false
      }
    ]
  ]
}

map =>
{
  Event1(uint256,
  int256,
  bool,
  address,
  bytes32,
  string,
  bytes)=[
    [
      ResultEntity[
        name=_u,
        type=uint256,
        data=111111
      ],
      ResultEntity[
        name=_i,
        type=int256,
        data=-1111111
      ],
```

(continues on next page)

(续上页)

```
      ResultEntity[
        name=_b,
        type=bool,
        data=false
      ],
      ResultEntity[
        name=_addr,
        type=address,
        data=0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
      ],
      ResultEntity[
        name=_bs32,
        type=bytes32,
        data=abcdefghiabcdefghiabcdefghiabhji
      ],
      ResultEntity[
        name=_s,
        type=string,
        data=FISCO-BCOS
      ],
      ResultEntity[
        name=_bs,
        type=bytes,
        data=nice
      ]
    ]
  ],
  Event2(uint256,
  int256,
  bool,
  address,
  bytes32,
  string,
  bytes)=[
    [
      ResultEntity[
        name=_u,
        type=uint256,
        data=111111
      ],
      ResultEntity[
        name=_i,
        type=int256,
        data=-1111111
      ],
      ResultEntity[
        name=_b,
        type=bool,
        data=false
      ],
      ResultEntity[
        name=_addr,
        type=address,
        data=0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
      ],
      ResultEntity[
        name=_bs32,
        type=bytes32,
        data=abcdefghiabcdefghiabcdefghiabhji
      ],
      ResultEntity[
```

(continues on next page)

(续上页)

```
        name=_s,
        type=string,
        data=FISCO-BCOS
      ],
      ResultEntity[
        name=_bs,
        type=bytes,
        data=nices
      ]
    ]
  ]
}
```

7.1.7 合约事件推送

功能简介

合约事件推送功能提供了合约事件的异步推送机制，客户端向节点发送注册请求，在请求中携带客户端关注的合约事件的参数，节点根据请求参数对请求区块范围的Event Log进行过滤，将结果分次推送给客户端。

交互协议

客户端与节点的交互基于Channel协议。交互分为三个阶段：注册请求，节点回复，Event Log数据推送。

注册请求

客户端向节点发送Event推送的注册请求:

```
// request sample:
{
  "fromBlock": "latest",
  "toBlock": "latest",
  "addresses": [
    "0xca5ed56862869c25da0bdf186e634aac6c6361ee"
  ],
  "topics": [
    "0x91c95f04198617c60eaf2180fbca88fc192db379657df0e412a9f7dd4ebbe95d"
  ],
  "groupID": "1",
  "filterID": "bb31e4ec086c48e18f21cb994e2e5967"
}
```

- filerID：字符串类型，每次请求唯一，标记一次注册任务
- groupID：字符串类型，群组ID
- fromBlock：整形字符串，初始区块。“latest” 当前块高
- toBlock：整形字符串，最终区块。“latest” 处理至当前块高时，继续等待新区块
- addresses：字符串或者字符串数组：字符串表示单个合约地址，数组为多个合约地址，数组可以为空
- topics：字符串类型或者数组类型：字符串表示单个topic，数组为多个topic，数组可以为空

节点回复

节点接受客户端注册请求时，会对请求参数进行校验，将是否成功接受该注册请求结果回复给客户端。

```
// response sample:
{
  "filterID": "bb31e4ec086c48e18f21cb994e2e5967",
  "result": 0
}
```

- filterID：字符串类型，每次请求唯一，标记一次注册任务
- result：整形，返回结果。0成功，其余为失败状态码

Event Log数据推送

节点验证客户端注册请求成功之后，根据客户端请求参数条件，向客户端推送Event的Log数据。

```
// event log push sample:
{
  "filterID": "bb31e4ec086c48e18f21cb994e2e5967",
  "result": 0,
  "logs": [

  ]
}
```

- filterID：字符串类型，每次请求唯一，标记一次注册任务
- result：整形 0：Event Log数据推送 1：推送完成。客户端一次注册请求对应节点的数据推送会有多次（请求区块范围比较大或者等待新的区块），result字段为1时说明节点推送已经结束
- logs：Log对象数组，result为0时有效

Java SDK教程

注册接口

Java SDK中org.fisco.bcos.channel.client.Service类提供合约事件的注册接口，用户可以调用registerEventLogFilter向节点发送注册请求，并设置回调函数。

```
  public void registerEventLogFilter(EventLogUserParams params,␣
↪EventLogPushCallback callback);
```

params注册参数

事件回调请求注册的参数：

```
public class EventLogUserParams {
    private String fromBlock;
    private String toBlock;
    private List<String> addresses;
    private List<Object> topics;
}
```

callback回调对象

```
public abstract class EventLogPushCallback {
    public void onPushEventLog(int status, List<LogResult> logs);
}
```

- status 回调返回状态:

```
    0        : 正常推送，此时logs为节点推送的Event日志
    1        : 推送完成，执行区间的区块都已经处理
    -41000   : 参数无效，客户端验证参数错误返回
    -41001   : 参数错误，节点验证参数错误返回
    -41002   : 群组不存在
    -41003   : 请求错误的区块区间
    -41004   : 节点推送数据格式错误
    -41005   : 请求发送超时
    -41006   : 其他错误
```

- logs表示回调的Event Log对象列表，status为0有效

```
  public class LogResult {
    private List<EventResultEntity> logParams;
    private Log log;
  }

  // Log对象
  public class Log {
    private String logIndex;
    private String transactionIndex;
    private String transactionHash;
    private String blockHash;
    private String blockNumber;
    private String address;
    private String data;
    private String type;
    private List<String> topics;
  }
```

Log log：Log对象

List<EventResultEntity> logParams：默认值null，可以在子类中解析Log的data字段，将结果保存入logParams [参考交易解析]

- 实现回调对象

Java SDK默认实现的回调类ServiceEventLogPushCallback，将status与logs在日志中打印，用户可以通过继承ServiceEventLogPushCallback类，重写onPushEventLog接口，实现自己的回调逻辑处理。

```
class MyEventLogPushCallBack extends ServiceEventLogPushCallback {
    @Override
    public void onPushEventLog(int status, List<LogResult> logs) {
        // ADD CODE
    }
}
```

注意：onPushEventLog接口多次回调的logs有重复的可能性，可以根据Log对象中的blockNumber，transactionIndex，logIndex进行去重

topic工具

org.fisco.bcos.channel.event.filter.TopicTools提供将各种类型参数转换为对应topic的工具，用户设置EventLogUserParams的topics参数可以使用。

```
class TopicTools {
    // int1/uint1~uint1/uint256
    public static String integerToTopic(BigInteger i)
    // bool
    public static String boolToTopic(boolean b)
    // address
    public static String addressToTopic(String s)
    // string
    public static String stringToTopic(String s)
    // bytes
    public static String bytesToTopic(byte[] b)
    // byte1~byte32
    public static String byteNToTopic(byte[] b)
}
```

Solidity To Java

为了简化使用，solidity合约生成对应的Java合约代码时，为每个Event生成两个重载的同名接口，接口命名规则: register + Event名称 + EventLogFilter。

这里以Asset合约的TransferEvent为例说明

```
contract Asset {
    event TransferEvent(int256 ret, string indexed from_account, string indexed to_
↪account, uint256 indexed amount)

    function transfer(string from_account, string to_account, uint256 amount)␣
↪public returns(int256) {
        // 结果
        int result = 0;

        // 其他逻辑，省略

        // TransferEvent 保存结果以及接口参数
        TransferEvent(result, from_account, to_account, amount);
    }
}
```

将Asset.sol生成对应Java合约文件[将solidity合约生成对应的Java调用文件]

```
class Asset {
    // 其他生成代码 省略

    public void registerTransferEventEventLogFilter(EventLogPushWithDecodeCallback␣
↪callback);
    public void registerTransferEventEventLogFilter(String fromBlock, String␣
↪toBlock, List<String> otherTopics, EventLogPushWithDecodeCallback callback);
}
```

registerTransferEventEventLogFilter

这两个接口对org.fisco.bcos.channel.client.Service.registerEventLogFilter进行了封装，调用等价于将registerEventLogFilter的params参数设置为：

```
    EventLogUserParams params = new EventLogUserParams();
    // fromBlock, 无参数设置为"latest"
    params.setFromBlock(fromBlock); // params.setFromBlock("latest");
    // toBlock, 无参数设置为"latest"
    params.setToBlock(toBlock); // params.setToBlock("latest");

    // addresses，设置为Java合约对象的地址
    // 当前java合约对象为：Asset asset
    ArrayList<String> addresses = new ArrayList<String>();
    addresses.add(asset.getContractedAddress());
    params.setAddresses(addresses);

    // topics, topic0设置为Event接口对应的topic
    ArrayList<Object> topics = new ArrayList<>();
    topics.add(TopicTools.stringToTopic("TransferEvent(int256,string,string,
↪uint256)"));
    // 其他topic设置， 没有则忽略
    topics.addAll(otherTopics);
```

可以看出，在关注指定地址特定合约的某个Event，使用生成的Java合约对象中的接口，更加简单方便。

EventLogPushWithDecodeCallback

EventLogPushWithDecodeCallback与ServiceEventLogPushCallback相同，是EventLogPushCallback的子类，区别在于：

- ServiceEventLogPushCallback回调接口onPushEventLog(int status, List<LogResult> logs) LogResult成员logParams为空，用户需要使用Log数据时需要解析数据
- EventLogPushWithDecodeCallback作为Asset对象的成员，可以根据其保存的ABI成员构造对应Event的解析工具，解析返回的Log数据，解析结果保存在logParams中。

示例

这里以Asset合约为例，给出合约事件推送的一些场景供用户参考。

- 场景1：将链上所有/最新的Event回调至客户端

```
        // 其他初始化逻辑，省略

        // 参数设置
        EventLogUserParams params = new EventLogUserParams();

        // 全部Event fromBlock设置为"1"
        params.setFromBlock("1");

        // 最新Event fromBlock设置为"latest"
        // params.setFromBlock("latest");

        // toBlock设置为"latest"，处理至最新区块继续等待新的区块
        params.setToBlock("latest");

        // addresses设置为空数组，匹配所有的合约地址
        params.setAddresses(new ArrayList<String>());

        // topics设置为空数组，匹配所有的Event
        params.setTopics(new ArrayList<Object>());
```

(continues on next page)

(续上页)

```
        // 回调，用户可以替换为自己实现的类的回调对象
        ServiceEventLogPushCallback callback = new ServiceEventLogPushCallback();
        service.registerEventLogFilter(params, callback);
```

- 场景2: 将Asset合约最新的TransferEvent事件回调至客户端

```
        // 其他初始化逻辑，省略

        // 设置参数
        EventLogUserParams params = new EventLogUserParams();

        // 从最新区块开始，fromBlock设置为"latest"
        params.setFromBlock("latest");
        // toBlock设置为"latest"，处理至最新区块继续等待新的区块
        params.setToBlock("latest");

        // addresses设置为空数组，匹配所有的合约地址
        params.setAddresses(new ArrayList<String>());

        // topic0，TransferEvent(int256,string,string,uint256)
        ArrayList<Object> topics = new ArrayList<>();
        topics.add(TopicTools.stringToTopic("TransferEvent(int256,string,string,
↪uint256)"));
        params.setTopics(topics);

        // 回调，用户可以替换为自己实现的类的回调对象
        ServiceEventLogPushCallback callback = new ServiceEventLogPushCallback();
        service.registerEventLogFilter(params, callback);
```

- 场景3: 将指定地址的Asset合约最新的TransferEvent事件回调至客户端

合约地址: `String addr = "0x06922a844c542df030a2a2be8f835892db99f324";`

方案1.

```
        // 其他初始化逻辑，省略

        String addr = "0x06922a844c542df030a2a2be8f835892db99f324";

        // 设置参数
        EventLogUserParams params = new EventLogUserParams();

        // 从最新区块开始，fromBlock设置为"latest"
        params.setFromBlock("latest");
        // toBlock设置为"latest"，处理至最新块并继续等待共识出块
        params.setToBlock("latest");

        // 合约地址
        ArrayList<String> addresses = new ArrayList<String>();
        addresses.add(addr);
        params.setAddresses(addresses);

        // topic0，匹配 TransferEvent(int256,string,string,uint256) 事件
        ArrayList<Object> topics = new ArrayList<>();
        topics.add(TopicTools.stringToTopic("TransferEvent(int256,string,uint256)
↪"));
        params.setTopics(topics);

        ServiceEventLogPushCallback callback = new ServiceEventLogPushCallback();
        service.registerEventLogFilter(params, callback);
```

方案2.

```
        // 其他初始化逻辑，省略

        String addr = "0x06922a844c542df030a2a2be8f835892db99f324";

        // 构造Asset合约对象
        Asset asset = Asset.load(addr, ... );

        EventLogPushWithDecodeCallback callback = new␣
↪EventLogPushWithDecodeCallback();
        asset.registerTransferEventEventLogFilter(callback);
```

- 场景4: 将指定地址的Asset合约所有TransferEvent事件回调至客户端

合约地址: `String addr = "0x06922a844c542df030a2a2be8f835892db99f324";`

方案1:

```
        // 其他初始化逻辑，省略

        // 设置参数
        EventLogUserParams params = new EventLogUserParams();

        // 从最初区块开始，fromBlock设置为"1"
        params.setFromBlock("1");
        // toBlock设置为"latest"，处理至最新块并继续等待共识出块
        params.setToBlock("latest");

        // 设置合约地址
        ArrayList<String> addresses = new ArrayList<String>();
        addresses.add(addr);
        params.setAddresses(addresses);

        // TransferEvent(int256,string,string,uint256) 转换为topic
        ArrayList<Object> topics = new ArrayList<>();
        topics.add(TopicTools.stringToTopic("TransferEvent(int256,string,string,
↪uint256)"));
        params.setTopics(topics);

        ServiceEventLogPushCallback callback = new ServiceEventLogPushCallback();
        service.registerEventLogFilter(params, callback);
```

方案2.

```
        // 其他初始化逻辑，省略

        Asset asset = Asset.load(addr, ... );

        // 设置区块范围
        String fromBlock = "1";
        String toBlock = "latest";

        // 参数topic为空
        ArrayList<Object> otherTopics = new ArrayList<>();

        EventLogPushWithDecodeCallback callback = new␣
↪EventLogPushWithDecodeCallback();

        asset.registerTransferEventEventLogFilter(fromBlock,toBlock,otherTopics,
↪callback);
```

- 场景5: 将Asset指定合约指定账户转账的所有事件回调至客户端

合约地址: `String addr = "0x06922a844c542df030a2a2be8f835892db99f324"`

转账账户: `String fromAccount = "account"`

方案1:

```
        // 其他初始化逻辑，省略

        String addr = "0x06922a844c542df030a2a2be8f835892db99f324";
        String fromAccount = "account";

        // 参数
        EventLogUserParams params = new EventLogUserParams();

        // 从最初区块开始，fromBlock设置为"1"
        params.setFromBlock("1");
        // toBlock设置为"latest"
        params.setToBlock("latest");

        // 设置合约地址
        ArrayList<String> addresses = new ArrayList<String>();
        addresses.add(addr);
        params.setAddresses(addresses);

        // 设置topic
        ArrayList<Object> topics = new ArrayList<>();
        // TransferEvent(int256,string,string,uint256) 转换为topic
        topics.add(TopicTools.stringToTopic("TransferEvent(int256,string,string,
↪uint256)"));
        // 转账账户 fromAccount转换为topic
        topics.add(TopicTools.stringToTopic(fromAccount));
        params.setTopics(topics);

        ServiceEventLogPushCallback callback = new ServiceEventLogPushCallback();
        service.registerEventLogFilter(params, callback);
```

方案2.

```
        // 其他初始化逻辑，省略

        String addr = "0x06922a844c542df030a2a2be8f835892db99f324";
        String fromAccount = "account";

        // 加载合约地址，生成Java合约对象
        Asset asset = Asset.load(addr, ... );

        // 回调函数
        EventLogPushWithDecodeCallback callback = new␣
↪EventLogPushWithDecodeCallback();

        // 设置区块范围
        String fromBlock = "1";
        String toBlock = "latest";
        // 参数topic
        ArrayList<Object> otherTopics = new ArrayList<>();
        // 转账账户 fromAccount转换为topic
        otherTopics.add(TopicTools.stringToTopic(fromAccount));

        asset.registerRegisterEventEventLogFilter(fromBlock,toBlock,otherTopics,
↪callback);
```

7.1.8 附录：JavaSDK启动异常场景

- Failed to connect to the node. Please check the node status and the console configuration.比较旧的SDK版本的提示，建议将JavaSDK版本升级至**2.2.2**或者以上(修改gradle.build或者maven配置文件中web3sdk的版本号)，可以获取更准确友好的提示，然后参考下面的错误提示解决问题。
- Failed to initialize the SSLContext: class path resource [ca.crt] cannot be opened because it does not exist. 无法加载到证书文件，证书文件没有正确拷贝至conf目录，可以参考控制台安装流程，拷贝证书文件至conf目录下。
- Failed to initialize the SSLContext: Input stream not contain valid certificates. 加载证书文件失败，CentOS系统使用OpenJDK的错误，参考CentOS环境安装JDK章节重新安装OracleJDK。
- Failed to connect to nodes: [connection timed out: /127.0.0.1:20200]连接超时，节点的网络不可达，请检查提示的IP是否配置错误，或者，当前JavaSDK运行环境与节点的环境网络确实不通，可以咨询运维人员解决网络不通的问题。
- Failed to connect to nodes: [Connection refused: /127.0.0.1:20200]拒绝连接，无法连接对端的端口，可以使用telnet命令检查端口是否连通，可能原因：
  1. 节点未启动，端口处于未监听状态，启动节点即可。
  2. 节点监听`127.0.0.1`的网段，监听`127.0.0.1`网络只能本机的客户端才可以连接，控制台位于不同服务器时无法连接节点，将节点配置文件`config.ini`中的`channel_listen_ip`修改为控制台连接节点使用的网段IP，或者将其修改为`0.0.0.0`。
  3. 错误的端口配置，配置的端口并不是节点监听的channel端口，修改连接端口为节点`config.ini`配置的`channel_listen_port`的值。**注意：控制台（或者JavaSDK）连接节点时使用Channel端口，并不是RPC端口，Channel端口在节点配置文件中通过channel_listen_ip字段配置，RPC端口通过jsonrpc_listen_port字段配置，注意区分，RPC默认从8545开始分配，Channel端口默认从20200开始分配。**
- Failed to connect to nodes: [ssl handshake failed:/127.0.0.1:20233] 与节点ssl握手失败，可能原因：
  1. 拷贝了错误的证书，检查拷贝的证书是否正确。
  2. 端口配置错误，连接其他服务正在监听的端口，检查连接端口是否为节点`channel_listen_port`端口。
  3. JDK版本问题，推荐使用1.8以及以上的OracleJDK，参考CentOS环境安装JDK章节安装OracleJDK。
- Failed to connect to [127.0.0.1:20233, 127.0.0.1:20234, 127.0.0.1:20235] ,groupId: 1 ,caCert: classpath:ca.crt ,sslKey: classpath:sdk.key ,sslCrt: classpath:sdk.crt ,java version: 1.8.0_231. 其他未知的错误，需要查看日志文件分析具体错误。

7.2 Node.js SDK

Node.js SDK 提供了访问 FISCO BCOS 节点的Node.js API，支持节点状态查询、部署和调用合约等功能，基于Node.js SDK可快速开发区块链应用，目前支持 FISCO BCOS 2.0+

注意

Node.js SDK目前仅处于个人开发者体验阶段，开发企业级应用请使用 Web3SDK

主要特性

- 提供调用FISCO BCOS JSON-RPC 的Node.js API
- 提供部署及调用Solidity合约（支持Solidity 0.4.x 及Solidity 0.5.x）的Node.js API
- 提供调用预编译（Precompiled）合约的Node.js API

- 使用 Channel协议 与FISCO BCOS节点通信，双向认证更安全
- 提供CLI（Command-Line Interface）工具供用户在命令行中方便快捷地与区块链交互

7.2.1 快速安装

环境要求

- Node.js开发环境
  - Node.js >= 8.10.0
  - npm >= 5.6.0

  如果您没有部署过Node.js环境，可以参考下列部署方式：
  - 如果您使用Linux或MacOS：

    推荐使用nvm快速部署，使用nvm同时也能够避免潜在的导致Node.js SDK部署失败的权限问题。以部署Node.js 8为例，部署步骤如下：

```
# 安装nvm
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.2/install.
↪sh | bash
# 加载nvm配置
source ~/.$(basename $SHELL)rc
# 安装Node.js 8
nvm install 8
# 使用Node.js 8
nvm use 8
```

  - 如果您使用Windows：

    请前往Node.js官网下载Windows下的安装包进行安装。
- 基本开发组件
  - Python 2（Windows、Linux及MacOS需要）
  - g++（Linux及MacOS需要）
  - make（Linux及MacOS需要）
  - Git（Windows、Linux及MacOS需要）
  - Git bash（仅Windows需要）
  - MSBuild构建环境（仅Windows需要）
- FISCO BCOS节点：请参考FISCO BCOS安装搭建

部署Node.js SDK

拉取源代码

```
git clone https://github.com/FISCO-BCOS/nodejs-sdk.git
```

使用npm安装依赖项

如果您的网络中使用了代理，请先为*npm*配置代理：

```
npm config set proxy <your proxy>
npm config set https-proxy <your proxy>
```

如果您所在的网络不便访问*npm*官方镜像，请使用其他镜像代替，如淘宝：

```
npm config set registry https://registry.npm.taobao.org
```

```
# 部署过程中请确保能够访问外网以能够安装第三方依赖包
cd nodejs-sdk
npm install
npm run repoclean
npm run bootstrap
```

Node.js CLI

Node.js SDK内嵌CLI工具，供用户在命令行中方便地与区块链进行交互。CLI工具在Node.js SDK提供的API的基础上开发而成，使用方式与结果输出对脚本友好，同时也是一个展示如何调用Node.js API进行二次开发的范例。

快速建链（可选）

若您的系统中已经搭建了*FISCO BCOS*链，请跳过本节。

```
# 获取开发部署工具
curl -LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/`curl -s␣
↪https://api.github.com/repos/FISCO-BCOS/FISCO-BCOS/releases | grep "\"v2\.[0-9]\.
↪[0-9]\"" | sort -u | tail -n 1 | cut -d \" -f 4`/build_chain.sh && chmod u+x␣
↪build_chain.sh
# 在本地建一个4节点的FISCO BCOS链
bash build_chain.sh -l "127.0.0.1:4" -p 30300,20200,8545 -i
# 启动FISCO BCOS链
bash nodes/127.0.0.1/start_all.sh
```

配置证书及Channel端口

- 配置证书

  修改配置文件，证书配置位于packages/cli/conf/config.json文件的authentication配置项中。你需要根据您实际使用的证书文件的路径修改该配置项的key、cert及ca配置，其中key为SDK私钥文件的路径，cert为SDK证书文件的路径，ca为链根证书文件的路径，这些文件可以由开发部署工具或运维部署工具自动生成，具体的生成方式及文件位置请参阅上述工具的说明文档。

- 配置Channel端口

  修改配置文件，节点IP及端口配置位于packages/cli/conf/config.json文件的nodes配置项中。您需要根据您要连接FISCO BCOS节点的实际配置修改该配置项的ip及port配置，其中ip为所连节点的IP地址，port为节点目录下的 config.ini 文件中的channel_listen_port配置项的值。如果您使用了快速搭链，可以跳过此步。

配置完成后，即可开始使用CLI工具，CLI工具位于packages/cli/cli.js，所有操作均需要在packages/cli/目录下执行，您需要先切换至该目录：

```
cd packages/cli
```

开启自动补全（仅针对bash及zsh用户，可选）

为方便用户使用CLI工具，CLI工具支持在bash或zsh中进行自动补全，此功能需要手动启用，执行命令：

```
rcfile=~/.$(basename $SHELL)rc && ./cli.js completion >> $rcfile && source $rcfile
```

便可启用自动补全。使用CLI工具时，按下Tab键（依据系统配置的不同，可能需要按两下）便可弹出候选命令或参数的列表并自动补全。

示例

以下给出几个使用示例：

查看CLI工具的帮助

```
./cli.js --help
```

查看CLI工具能够调用的命令及对应的功能

```
./cli.js list
```

以下示例中的输入、输出及参数仅供举例

查看所连的FISCO BCOS节点版本

```
./cli.js getClientVersion
```

输出如下：

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "Build Time": "20190705 21:19:13",
    "Build Type": "Linux/g++/RelWithDebInfo",
    "Chain Id": "1",
    "FISCO-BCOS Version": "2.0.0",
    "Git Branch": "master",
    "Git Commit Hash": "d8605a73e30148cfb9b63807fb85fa211d365014",
    "Supported Version": "2.0.0"
  }
}
```

获取当前的块高

```
./cli.js getBlockNumber
```

输出如下：

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": "0xfa"
}
```

部署SDK自带的HelloWorld合约

```
./cli.js deploy HelloWorld
```

输出如下：

```
{
  "contractAddress": "0x11b6d7495f2f04bdca45e9685ceadea4d4bd1832"
}
```

调用HelloWorld合约的set接口，请将合约地址改为实际地址

```
./cli.js call HelloWorld 0x11b6d7495f2f04bdca45e9685ceadea4d4bd1832 set vita
```

输出如下：

```
{
  "transactionHash":
↪"0xa71f136107389348d5a092a345aa6bc72770d98805a7dbab0dbf8fe569ff3f37",
  "status": "0x0"
}
```

调用HelloWorld合约的get接口，请将合约地址改为实际地址

```
./cli.js call HelloWorld 0xab09b29dd07e003776d22566ae5c078f2cb2279e get
```

输出如下：

```
{
  "status": "0x0",
  "output": {
    "0": "vita"
  }
}
```

CLI帮助

如果您想知道某一个命令该如何使用，可以使用如下的命令：

```
./cli.js <command> ?
```

其中command为一个命令名，使用?作为参数便可获取该命令的使用提示，如：

```
./cli.js call ?
```

会得到如下的输出：

```
cli.js call <contractName> <contractAddress> <function> [parameters...]

Call a contract by a function and parameters

位置:
  contractName     The name of a contract                          [字符串] [必需]
  contractAddress  20 Bytes - The address of a contract            [字符串] [必需]
  function         The function of a contract                      [字符串] [必需]
  parameters       The parameters(splited by a space) of a function
                                                          [数组] [默认值: []]

选项:
  --help     显示帮助信息                                                 [布尔]
  --version  显示版本号                                                   [布尔]
```

7.2.2 配置说明

Node.js SDK的配置文件为一个JSON文件，主要包括**通用配置**，**群组配置**，**通信配置**和**证书配置**。

通用配置

- `privateKey`: `object`，必需。外部账户的私钥，可以为一个256 bits的随机整数，也可以是一个pem或p12格式的私钥文件，后两者需要结合get_account.sh生成的私钥文件使用。`privateKey`包含两个必需字段，一个可选字段：
  - `type`: `string`，必需。用于指示私钥类型。`type`的值必需为下列三个值之一：
    - `ecrandom`：随机整数
    - `pem`：pem格式的文件

        * p12：p12格式的文件
    - value：string，必需。用于指示私钥具体的值：
        * 如果type为ecrandom，则value为一个长度为256 bits 的随机整数，其值介于1 ~ 0xFFFF FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4141之间。
        * 如果type为pem，则value为pem文件的路径，如果是相对路径，需要以配置文件所在的目录为相对路径起始位置。
        * 如果type为p12，则value为p12文件的路径，如果是相对路径，需要以配置文件所在的目录为相对路径起始位置。
    - password：string，可选。如果type为p12，则需要此字段以解密私钥，否则会忽略该字段。
- timeout：number。Node.js SDK所连节点可能会陷入停止响应的状态。为避免陷入无限等待，Node.js SDK的每一次API调用在timeout之后若仍没有得到结果，则强制返回一个错误对象。timeout的单位为毫秒。
- solc: string，可选。Node.js SDK已经自带0.4.26及0.5.10版本的Solidity编译器，如果您有特殊的编译器需求，可以设置本配置项为您的编译器的执行路径或全局命令

群组配置

- groupID: number。Node.js SDK访问的链的群组ID

通信配置

- nodes: list，必需。FISCO BCOS节点列表，Node.js SDK在访问节点时时会从该列表中随机挑选一个节点进行通信，要求节点数目必须 >= 1。在FISCO BCOS中，一笔交易上链并不代表网络中的所有节点都已同步到了最新的状态，如果Node.js SDK连接了多个节点，则可能会出现读取不到最新状态的情况，因此在对状态同步有较高要求的场合，请谨慎连接多个节点。每个节点包含两个字段：
    - ip: string，必需。FISCO BCOS节点的IP地址
    - port: string，必需，FISCO BCOS节点的Channel端口

证书配置

- authentication：object。必需，包含建立Channel通信时所需的认证信息，一般在建链过程中自动生成。authentication包含三个必需字段：
    - key: string，必需。私钥文件路径，如果是相对路径，需要以配置文件所在的目录为相对路径起始位置。
    - cert: string，必需。证书文件路径，如果是相对路径，需要以配置文件所在的目录为相对路径起始位置。
    - ca: string，必需。CA根证书文件路径，如果是相对路径，需要以配置文件所在的目录为相对路径起始位置。

7.2.3 Node.js API

Node.js SDK为区块链应用开发者提供了Node.js API接口，以服务的形式供外部调用。按照功能，Node.js API可以分为如下几类：

- **Web3jService**：提供访问FISCO BCOS 2.0+节点JSON-RPC接口支持；提供部署及调用合约的支持。

- **PrecompiledService**：

  Precompiled合约（预编译合约）是一种FISCO BCOS底层内嵌的、通过C++实现的高效智能合约，提供包括分布式权限控制、CNS、系统属性配置、节点类型配置等功能。PrecompiledService是调用这类功能的API的统称，分为：

  - **PermissionService**：提供对分布式权限控制的支持
  - **CNSService**：提供对CNS的支持
  - **SystemConfigService**：提供对系统配置的支持
  - **ConsensusService**：提供对节点类型配置的支持
  - **CRUDService**：提供对CRUD(增删改查)操作的支持，可以创建表或对表进行增删改查操作。

API调用约定

- 使用服务之前，首先需要初始化全局的`Configuration`对象，用以为各个服务提供必要的配置信息。`Configuration`对象位于`nodejs-sdk/packages/api/common/configuration.js`，其初始化参数为一个配置文件的路径或包含配置项的对象。配置文件的配置项说明见配置说明
- 如无特殊说明，Node.js SDK提供的API均为**异步**API。异步API的实际返回值是一个包裹了API返回值的Promise对象，开发者可以使用async/await语法或then...catch...finally方法操作该Promise对象以实现自己的程序逻辑
- 当API内部出现错误导致逻辑无法继续执行时（如合约地址不存在），均会直接抛出异常，所有异常均继承自Error类

Web3jService

位置：`nodejs-sdk/packages/api/web3j`

功能：访问FISCO BCOS 2.0+节点JSON-RPC；部署合约；调用合约

*调用接口：函数名(参数类型,...)，例如：func(uint256,uint256)，参数类型之间不能有空格

PrecompiledService

PermissionService

位置：`nodejs-sdk/packages/api/precompiled/permission`

功能：提供对分布式权限控制的支持

CNSService

位置：`nodejs-sdk/packages/api/precompiled/cns`

功能：提供对节点类型配置的支持

SystemConfigService

位置：`nodejs-sdk/packages/api/precompiled/systemConfig`

功能：提供对系统配置的支持

ConsensusService

位置：`nodejs-sdk/packages/api/precompiled/consensus`

功能：提供对节点类型配置的支持

CRUDService

位置：`nodejs-sdk/packages/api/precompiled/crud`

功能：提供对CRUD(增删改查)操作的支持

7.3 Python SDK

Python SDK 提供了访问 FISCO BCOS 节点的Python API，支持节点状态查询、部署和调用合约等功能，基于Python SDK可快速开发区块链应用，目前支持 FISCO BCOS 2.0+

注意

- **Python SDK当前为候选版本，可供开发测试使用，企业级应用可用 Web3SDK**
- Python SDK目前支持FISCO BCOS 2.0.0及其以上版本

主要特性

- 提供调用FISCO BCOS JSON-RPC 的Python API
- 支持使用 Channel协议 与FISCO BCOS节点通信，保证节点与SDK安全加密通信的同时，可接收节点推送的消息。
- 支持交易解析功能：包括交易输入、交易输出、Event Log等ABI数据的拼装和解析
- 支持合约编译，将 `sol` 合约编译成 `abi` 和 `bin` 文件
- 支持基于keystore的账户管理
- 支持合约历史查询

7.3.1 快速安装

环境要求

依赖软件

- **Ubuntu**: `sudo apt install -y zlib1g-dev libffi6 libffi-dev wget git`
- **CentOS**: `sudo yum install -y zlib-devel libffi-devel wget git`
- **MacOs**: `brew install wget git`

Python环境要求

- 支持版本：
  - **python 3.6.3**
  - **3.7.x**

部署Python SDK

环境要求

- Python环境：python 3.6.3, 3.7.x
- FISCO BCOS节点：请参考FISCO BCOS安装搭建

初始化环境(若python环境符合要求，可跳过)

Linux环境初始化

拉取源代码

```
git clone https://github.com/FISCO-BCOS/python-sdk
```

配置环境

注解:

- `bash init_env.sh -p` 主要功能是安装pyenv，并使用pyenv安装名称为 `python-sdk` 的python-3.7.3虚拟环境
- 若python环境符合要求，可以跳过此步
- 若脚本执行出错，请检查是否参考[依赖软件]说明安装了依赖
- 安装python-3.7.3可能耗时比较久
- 此步骤仅需初始化一遍，再次登录直接使用命令 `pyenv activate python-sdk` 激活 `python-sdk` 虚拟环境即可

```
# 判断python版本，并为不符合条件的python环境安装python 3.7.3的虚拟环境，命名为python-sdk
# 若python环境符合要求，可以跳过此步
# 若脚本执行出错，请检查是否参考[依赖软件]说明安装了依赖
# 提示：安装python-3.7.3可能耗时比较久
cd python-sdk && bash init_env.sh -p

# 激活python-sdk虚拟环境
source ~/.bashrc && pyenv activate python-sdk && pip install --upgrade pip
```

Windows环境初始化

在Windows运行Python SDK，需要按照以下步骤安装依赖软件并配置合约编译器：

安装依赖软件

注解:

- Microsoft Visual C++ 14.0 is required. Get it with "Microsoft Visual C++ Build Tools"解决方法: https://visualstudio.microsoft.com/downloads （注意选择vs 2005即14.0版） 或 https://pan.baidu.com/s/1ZmDUGZjZNgFJ8D14zBu9og 提取码: zrby
- solc编译器下载成功后，解压，将其中的 `solc.exe` 文件复制 `${python-sdk}\bin` 目录下，若python-sdk路径为 `D:\\open-source\\python-sdk` ，则 `solc.exe` 文件复制路径为 `D:\\open-source\\python-sdk\\bin\\solc.exe`

- 直接安装Python-3.7.x和git软件 python环境变量配置可参考这里

- Visual C++ 14.0库
- 下载Windows版本solc, 点击这里下载

拉取源代码

打开 git，在任意目录执行如下命令

```
git clone https://github.com/FISCO-BCOS/python-sdk
```

配置solc编译器

修改client_config.py.template，配置solc编译器路径，solc二进制下载请参考bcos_solc.py中的描述，并将client_config.py.template拷贝为client_config.py。

```
# 修改client_config.py.template:
# 配置solc编译器路径，若solc存放路径为D:\\open-source\\python-sdk\\bin\\solc.exe，则solc_
↪path配置如下:
solc_path = "D:\\open-source\\python-sdk\\bin\\solc.exe"

# 将client_config.py.template拷贝到client_config.py
```

安装Python SDK依赖

```
pip install -r requirements.txt
```

若因网络原因，安装依赖失败，可使用清华的pip源下载，安装命令如下：

```
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple -r requirements.txt
```

初始化配置(Windows环境可跳过)

```
# 该脚本执行操作如下:
# 1. 拷贝client_config.py.template->client_config.py
# 2. 下载solc编译器
bash init_env.sh -i
```

若没有执行以上初始化步骤，需要将contracts/目录下的sol代码手动编译成bin和abi文件并放置于contracts目录，才可以部署和调用相应合约。合约编译可以使用remix

配置Channel通信协议

Python SDK支持使用Channel协议与FISCO BCOS节点通信，通过SSL加密通信保障SDK与节点通信的机密性。

设SDK连接的节点部署在目录~/fisco/nodes/127.0.0.1目录下，则通过如下步骤使用Channel协议:

配置Channel信息

注解: 为便于开发和体验，channel_listen_ip参考配置是 *0.0.0.0* ，出于安全考虑，请根据实际业务网络情况，修改为安全的监听地址，如：内网IP或特定的外网IP

在节点目录下的 config.ini 文件中获取 channel_listen_port, 这里为20200

```
[rpc]
    channel_listen_ip=0.0.0.0
    jsonrpc_listen_ip=127.0.0.1
    channel_listen_port=20200
    jsonrpc_listen_port=8545
```

切换到python-sdk目录，修改 client_config.py 文件中channel_host为实际的IP，channel_port为上步获取的channel_listen_port：

```
channel_host = "127.0.0.1"
channel_port = 20200
```

配置证书

```
# 若节点与python-sdk位于不同机器，请将节点sdk目录下所有相关文件拷贝到bin目录
# 若节点与sdk位于相同机器，直接拷贝节点证书到SDK配置目录
cp ~/fisco/nodes/127.0.0.1/sdk/* bin/
```

配置证书路径

注解:

- client_config.py 的 channel_node_cert 和 channel_node_key 选项分别用于配置SDK证书和私钥
- release-2.1.0 版本开始，SDK证书和私钥更新为 sdk.crt 和 sdk.key ，配置证书路径前，请先检查上步拷贝的证书名和私钥名，并将 channel_node_cert 配置为SDK证书路径，将 channel_node_key 配置为SDK私钥路径

检查从节点拷贝的sdk证书路径，若sdk证书和私钥路径分别为bin/sdk.crt和bin/sdk.key，则client_config.py中相关配置项如下：

```
channel_node_cert = "bin/sdk.crt"   # 采用channel协议时，需要设置sdk证书,如采用rpc协议通信，这里可以留空
channel_node_key = "bin/sdk.key"    # 采用channel协议时，需要设置sdk私钥,如采用rpc协议通信，这里可以留空
```

若sdk证书和私钥路径分别为bin/node.crt和bin/node.key，则client_config.py中相关配置项如下:

```
channel_node_cert = "bin/node.crt"   # 采用channel协议时，需要设置sdk证书,如采用rpc协议通信，这里可以留空
channel_node_key = "bin/node.key"    # 采用channel协议时，需要设置sdk私钥,如采用rpc协议通信，这里可以留空
```

使用Channel协议访问节点

注解: windows环境下执行console.py请使用 .\console.py 或者 python console.py

---

```
# 获取FISCO BCOS节点版本号
./console.py getNodeVersion
```

开启命令行自动补全

Python SDK引入argcomplete支持命令行补全，运行如下命令开启此功能(**bashrc仅需设置一次，设置之后每次登陆自动生效**)

注解:

- 此步骤仅需设置一次，设置之后以后每次登陆自动生效
- 请在 **bash环境** 下执行此步骤
- 目前仅支持bash，不支持zsh

```
echo "eval \"\$(register-python-argcomplete ./console.py)\"" >> ~/.bashrc
source ~/.bashrc
```

7.3.2 配置说明

`client_config.py`是Python SDK的配置文件，主要包括**SDK算法类型配置**，**通用配置**，**账户配置**，**群组配置**，**通信配置**和**证书配置**。

注解:

- 确保连接端口开放：推荐使用 `telnet ip port` 确认客户端与节点网络是否连通

- 使用RPC通信协议，不需设置证书 - 日志配置参见 `client/clientlogger.py` ，默认在 `bin/logs` 目录下生成日志，默认级别为DEBUG

SDK算法类型配置

- **crypto_type**: SDK接口类型，目前支持国密接口(`GM`)和非国密接口(`ECDSA`)

通用配置

- **contract_info_file**: 保存已部署合约信息的文件，默认为`bin/contract.ini`
- **account_keyfile_path**: 存放keystore文件的目录，默认为`bin/accounts`
- **logdir**：默认日志输出目录，默认为`bin/logs`

账户配置

非国密账户的配置如下：

- **account_keyfile**: 存储非国密账号信息的keystore文件路径, 默认为`pyaccount.keystore`
- **account_password**: 非国密keystore文件的存储口令，默认为`123456`

国密账户的配置如下：

- **gm_account_keyfile**: 存储国密账号信息的加密文件路径, 默认为`gm_account.json`
- **gm_account_password**: 国密账户文件的存储口令，默认为`123456`

群组配置

群组配置主要包括链ID和群组ID：

- **fiscoChainId**：链ID，必须与通信节点的一致，默认为1
- **groupid**：群组ID，必须与通信的节点一致，获取节点群组ID请参考这里，默认为1

通信配置

- **client_protocol**：Python SDK与节点通信协议，包括rpc和channel选项，前者使用JSON-RPC接口访问节点，后者使用Channel访问节点，需要配置证书，默认为channel
- **remote_rpcurl**：采用**rpc**通信协议时，节点的rpc IP和端口，参考这里获取节点RPC信息，默认为http://127.0.0.1:8545，**如采用channel协议，可以留空**
- **channel_host**：采用channel协议时，节点的channel IP地址，参考这里获取节点Channel信息，默认为127.0.0.1，**如采用rpc协议通信，可以留空**
- **channel_port**：节点的channel 端口，默认为20200，**如采用rpc协议通信，可以留空**

证书配置

- **channel_ca**：链CA证书，**使用channel协议时设置**，默认为bin/ca.crt，
- **channel_node_cert**：节点证书，**使用channel协议时设置**，默认为bin/sdk.crt，**如采用rpc协议通信，可以留空**
- **channel_node_key**：Python SDK与节点通信私钥，采用channel协议时须设置，默认为bin/sdk.key，**如采用rpc协议通信，这里可以留空**

solc编译器配置

Python SDK支持使用配置的solc和solcjs编译器自动编译合约，同时配置solc和solcjs时，选择性能较高的solc编译器，编译选项如下：

- **solc_path**：非国密solc编译器路径
- **gm_solc_path**: 国密编译器路径
- **solcjs_path**：solcjs编译脚本路径，为./solc.js

配置项示例

配置项示例如下：

```
    """
    类成员常量和变量，便于用.调用和区分命名空间
    """
    # keyword used to represent the RPC Protocol
    PROTOCOL_RPC = "rpc"
    # keyword used to represent the Channel Protocol
    PROTOCOL_CHANNEL = "channel"

    # ---------crypto_type config--------------
    # crypto_type : 大小写不敏感: "GM" for 国密, "ECDSA" 或其他是椭圆曲线默认实现。
    crypto_type = "ECDSA"
    crypto_type = crypto_type.upper()
    set_crypto_type(crypto_type)   # 使密码算法模式全局生效，切勿删除此行

    # ----------------------------------------
    # configure below
    # ---------client communication config--------------
    client_protocol = "channel"  # or PROTOCOL_CHANNEL to use channel prototol
    # client_protocol = PROTOCOL_CHANNEL
    remote_rpcurl = "http://127.0.0.1:8545"  # 采用rpc通信时，节点的rpc端口,和要通信的节
点*必须*一致,如采用channel协议通信，这里可以留空
    channel_host = "127.0.0.1"  # 采用channel通信时，节点的channel ip地址,如采用rpc协议通
信，这里可以留空
```

(continues on next page)

(续上页)

```
    channel_port = 20200  # 节点的channel 端口,如采用rpc协议通信，这里可以留空
    channel_ca = "bin/ca.crt"  # 采用channel协议时，需要设置链证书,如采用rpc协议通信，这里
可以留空
    channel_node_cert = "bin/sdk.crt"  # 采用channel协议时，需要设置sdk证书,如采用rpc协议
通信，这里可以留空
    channel_node_key = "bin/sdk.key"  # 采用channel协议时，需要设置sdk私钥,如采用rpc协议
通信，这里可以留空
    fiscoChainId = 1  # 链ID，和要通信的节点*必须*一致
    groupid = 1  # 群组ID，和要通信的节点*必须*一致，如和其他群组通信，修改这一项，或者设
置bcosclient.py里对应的成员变量

    # ---------account &keyfile config--------------
    # 注意账号部分，国密和ECDSA采用不同的配置
    contract_info_file = "bin/contract.ini"  # 保存已部署合约信息的文件
    account_keyfile_path = "bin/accounts"  # 保存keystore文件的路径，在此路径下,
↪keystore文件以 [name].keystore命名
    account_keyfile = "pyaccount.keystore"
    account_password = "123456"  # 实际使用时建议改为复杂密码
    gm_account_keyfile = "gm_account.json"  # 国密账号的存储文件，可以加密存储,如果留空则不
加载gm_account_password = "123456"
    gm_account_password = "123456"
    # ---------console mode, support user input--------------
    background = True

    # ---------runtime related--------------
    # 非国密编译器路径
    solc_path = "./bin/solc/v0.4.25/solc"
    # 国密编译器路径
    gm_solc_path = "./bin/solc/v0.4.25/solc-gm"
    solcjs_path = "./solcjs"

    logdir = "bin/logs"  # 默认日志输出目录，该目录必须先建立
```

7.3.3 Python API

Python SDK为区块链应用开发者提供了Python API接口，主要包括：

- Python API：封装了访问FISCO BCOS 2.0+节点JSON-RPC的Python API
- 交易结构定义：定义了FISCO BCOS 2.0+的交易数据结构
- 交易输入输出解析：提供ABI、Event Log、交易输入和输出解析功能
- ChannelHandler：FISCO BCOS channel协议实现类，支持节点之间SSL加密通信

Python API：BcosClient

实现于`client/bcosclient.py`，封装了访问FISCO BCOS 2.0+节点JSON-RPC的Python API，主要接口包括：

Precompile Service

CNS

类名

```
client.precompile.cns.cns_service.CnsService
```

功能接口

- register_cns：注册合约名到(合约地址，合约版本)的映射到CNS系统表中
- query_cns_by_name：根据合约名查询CNS信息
- query_cns_by_nameAndVersion：根据合约名和合约名查询CNS信息

共识

类名

```
client.precompile.consensus.consensus_precompile.ConsensusPrecompile
```

功能接口

- addSealer：添加共识节点
- addObserver：添加观察者节点
- removeNode：将节点从群组中删除

权限控制

类名

```
client.precompile.permission.permission_service.PermissionService
```

功能接口

- grant: 将指定表的权限授权给用户
- revoke：收回指定用户对指定表的写权限
- list_permission: 显示对指定表有写权限的账户信息

CRUD

类名

```
client.precompile.crud.crud_service.Entry
```

功能接口

- create_table：创建用户表
- insert：向用户表插入记录
- update：更新用户表记录
- remove：删除用户表指定记录
- select：查询用户表指定记录
- desc: 查询用户表信息

系统配置

类名

```
client.precompile.config.config_precompile.ConfigPrecompile
```

功能接口

- setValueByKey: 设置系统配置项的值

交易结构定义：BcosTransaction

实现于`client/bcostransaction.py`，定义了FISCO BCOS 2.0+的交易数据结构：

交易输入输出解析：DatatypeParser

提供ABI、Event Log、交易输入和输出解析功能，实现于`client/datatype_parser.py`：

ChannelHandler

FISCO BCOS channel协议实现类，支持节点之间SSL加密通信，并可接收节点推送的消息，主要实现于`client/channelhandler.py`，channel协议编解码参考这里

合约历史查询

- **client/contratnote.py**： 采用ini配置文件格式保存合约的最新地址和历史地址，以便加载（如console命令里可以用(合约名 last)指代某个合约最新部署的地址）

日志模块

- **client/clientlogger.py**： logger定义，目前包括客户端日志和统计日志两种
- **client/stattool.py** 一个简单的统计数据收集和打印日志的工具类

7.3.4 控制台

Python SDK通过`console.py`实现了一个简单的控制台，支持合约操作、账户管理操作等。

注解:

- **Python SDK当前为候选版本，可供开发测试使用，企业级应用可用** Web3SDK
- 安装Java版本控制台可参考 这里
- windows环境下执行console.py请使用 `.\console.py` 或者 `python console.py`

常用命令

deploy

部署合约：

```
./console.py deploy [contract_name] [save]
```

参数包括：

- contract_name: 合约名，需要先放到`contracts`目录
- save：若设置了save参数，表明会将合约地址写入历史记录文件

```
$ ./console.py deploy HelloWorld save

INFO >> user input : ['deploy', 'HelloWorld', 'save']

backup [contracts/HelloWorld.abi] to [contracts/HelloWorld.abi.20190807102912]
backup [contracts/HelloWorld.bin] to [contracts/HelloWorld.bin.20190807102912]
INFO >> compile with solc compiler
deploy result  for [HelloWorld] is:
 {
    "blockHash":
↪"0x3912605dde5f7358fee40a85a8b97ba6493848eae7766a8c317beecafb2e279d",
    "blockNumber": "0x1",
    "contractAddress": "0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce",
    "from": "0x95198b93705e394a916579e048c8a32ddfb900f7",
    "gasUsed": "0x44ab3",
    "input": "0x6080604052...省略若干行...
↪c6f2c20576f726c6421000000000000000000000000000000",
    "logs": [],
    "logsBloom": "0x000...省略若干行...0000",
    "output": "0x",
    "status": "0x0",
    "to": "0x0000000000000000000000000000000000000000",
    "transactionHash":
↪"0xb291e9ca38b53c897340256b851764fa68a86f2a53cb14b2ecdcc332e850bb91",
    "transactionIndex": "0x0"
}
on block : 1,address: 0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce
address save to file:  bin/contract.ini
```

call

调用合约接口，并解析返回结果：

```
./console.py  call [contract_name] [contract_address] [function] [args]
```

参数包括：

- contract_name：合约名
- contract_address：调用的合约地址
- function：调用的合约接口
- args：调用参数

```
# 合约地址：0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce
# 调用接口：get
$./console.py  call HelloWorld 0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce get

INFO >> user input : ['call', 'HelloWorld',
↪'0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce', 'get']
INFO >> call HelloWorld , address: 0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce,␣
↪func: get, args:[]
INFO >> call result: ('Hello, World!',)
```

sendtx

发送交易调用指定合约的接口，交易结果会写入区块和状态：

```
./console.py sendtx [contract_name] [contract_address] [function] [args]
```

参数包括：

- contract_name：合约名
- contract_address：合约地址
- function：函数接口
- args：参数列表

```
# 合约名：HelloWorld
# 合约地址：0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce
# 调用接口：set
# 参数："Hello, FISCO"
$ ./console.py sendtx HelloWorld 0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce set
↪"Hello, FISCO"

INFO >> user input : ['sendtx', 'HelloWorld',
↪'0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce', 'set', 'Hello, FISCO']

INFO >> sendtx HelloWorld , address: 0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce,␣
↪func: set, args:['Hello, FISCO']

INFO >>  receipt logs :
INFO >> transaction hash : ␣
↪0xc20cbc6b0f28ad8fe1c560c8ce28c0e7eb7719a4a618a81604ac87ac46cc60f0
tx input data detail:
 {'name': 'set', 'args': ('Hello, FISCO',), 'signature': 'set(string)'}
receipt output : ()
```

newaccount

创建新账户，并将结果以加密的形式把保存与bin/accounts/${accoutname}.keystore文件中，**如同目录下已经有同名帐户文件，旧文件会复制一个备份**：

```
./console.py newaccount [account_name] [account_password]
```

参数包括：

- account_name：账户名
- account_password：加密keystore文件的口令

注解：

- 采用创建帐号的命令创建帐号后，若需作为默认帐号使用，注意修改client_config.py的 account_keyfile 和 account_password 配置项
- 账户名不可超过240个字符
- 若 account_password 中包含特殊字符，请在 account_password 周围加上单引号，否则无法解析

```
$ ./console.py newaccount test_account "123456"

>> user input : ['newaccount', 'test_account', '123456']

starting : test_account 123456
new address :    0x247e7AE892a94c9e089D61A7DB08af23CEDBec16
```

(continues on next page)

(续上页)

```
new privkey :     0xe2cf070a7c1da05577841b54b4f8ca7d9f7eb52e688bb7e61a2c6ada8a4c5c77
new pubkey :      ␣
↪0x71317d52a7f8b5bb3fa882b9936d7d31a04e6a122e6fdf790d39aeee8ed2883d3c0b90f644cab0b30153d700d93da
encrypt use time : 1.453 s
save to file : [bin/accounts/test_account.keystore]
>>-------------------------------------------------------
>> read [bin/accounts/test_account.keystore] again after new account,address &␣
↪keys in file:
decrypt use time : 1.447 s
address:          0x247e7AE892a94c9e089D61A7DB08af23CEDBec16
privkey:          0xe2cf070a7c1da05577841b54b4f8ca7d9f7eb52e688bb7e61a2c6ada8a4c5c77
pubkey :          ␣
↪0x71317d52a7f8b5bb3fa882b9936d7d31a04e6a122e6fdf790d39aeee8ed2883d3c0b90f644cab0b30153d700d93da

account store in file: [bin/accounts/test_account.keystore]

**** please remember your password !!! *****
```

showaccount

根据账户名和账户keystore文件口令，输出账户公私钥信息:

```
./console.py showaccount [account_name] [account_password]
```

参数包括:

- name：账户名称
- password: 账户keystore文件口令

```
$ ./console.py showaccount test_account "123456"

>> user input : ['showaccount', 'test_account', '123456']

show account : test_account, keyfile:bin/accounts/test_account.keystore ,password␣
↪123456
decrypt use time : 1.467 s
address:          0x247e7AE892a94c9e089D61A7DB08af23CEDBec16
privkey:          0xe2cf070a7c1da05577841b54b4f8ca7d9f7eb52e688bb7e61a2c6ada8a4c5c77
pubkey :          ␣
↪0x71317d52a7f8b5bb3fa882b9936d7d31a04e6a122e6fdf790d39aeee8ed2883d3c0b90f644cab0b30153d700d93da

account store in file: [bin/accounts/test_account.keystore]

**** please remember your password !!! *****
```

usage

输出控制台使用方法:

```
 $ ./console.py usage

INFO >> user input : ['usage']

FISCO BCOS 2.0 @python-SDK Usage:
newaccount [name] [password] [save]
        创建一个新帐户，参数为帐户名(如alice,bob)和密码
        结果加密保存在配置文件指定的帐户目录 *如同目录下已经有同名帐户文件，旧文件会复制一个备份
```

(continues on next page)

(续上页)

```
        如输入了"save"参数在最后，则不做询问直接备份和写入
        create a new account ,save to :[bin/accounts] (default) ,
        the path in client_config.py:[account_keyfile_path]
        if account file has exist ,then old file will save to a backup
        if "save" arg follows,then backup file and write new without ask
        the account len should be limitted to 240

    ... 省略若干行...
    [getTransactionByBlockHashAndIndex] [blockHash] [transactionIndex]
    [getTransactionByBlockNumberAndIndex] [blockNumber] [transactionIndex]
    [getSystemConfigByKey] [tx_count_limit/tx_gas_limit]
```

list

输出Python SDK支持的所有接口：

```
$ ./console.py list

INFO >> user input : ['list']

 >> RPC commands
    [getNodeVersion]
    [getBlockNumber]
    ... 省略若干行...
    [getTransactionByBlockHashAndIndex] [blockHash] [transactionIndex]
    [getTransactionByBlockNumberAndIndex] [blockNumber] [transactionIndex]
    [getSystemConfigByKey] [tx_count_limit/tx_gas_limit]
```

CNS

Python SDK控制台提供了CNS命令，主要包括注册CNS、查询CNS信息，CNS设计使用方法请参考这里。

registerCNS

将(合约地址, 合约版本)到合约名的映射注册到CNS系统表中：

```
./console.py registerCNS [contract_name] [contract_address] [contract_version]
```

参数包括:

- contract_name: 合约名
- contract_address: 合约地址
- contract_version: 合约版本

```
# 将合约地址0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce和合约版本v_1.0映射到合约
名HelloWorld
$ ./console.py registerCNS HelloWorld 0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce v_
↪1.0

INFO >> user input : ['registerCNS', 'HelloWorld',
↪'0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce', 'v_1.0']

INFO >> CNS version (strip space): v_1.0
INFO >> registerCNS
```

(continues on next page)

(续上页)

```
    >> status: 0x0
    >> transactionHash:␣
↪0x14720764a67c669811c02e9d9b4c7faa5ea94328e1e33fb7e35e885a27843a4e
    >> gasUsed: 0x6a98
    >> registerCNS succ, output: 1
```

queryCNSByName

根据合约名查询CNS信息:

```
./console.py queryCNSByName [contract_name]
```

参数包括:

- contract_name: 合约名

```
查询HelloWorld合约名对应的CNS信息
$ ./console.py queryCNSByName HelloWorld

INFO >> user input : ['queryCNSByName', 'HelloWorld']

    >> ('[{"abi":"\\"\\"","address":"0x2d1c577e41809453C50e7E5C3F57D06f3CDD90Ce",
↪"name":"HelloWorld","version":"v_1.0"}]\n',)
CNS ITEM 0 >>
        ContractName: HelloWorld
        ContractVersion: v_1.0
        ContractAddress: 0x2d1c577e41809453C50e7E5C3F57D06f3CDD90Ce
```

queryCNSByNameAndVersion

根据合约名和合约版本查询CNS信息:

```
./console.py queryCNSByNameAndVersion [contract_name] [contract_version]
```

参数包括:

- contract_name: 合约名
- contract_version: 合约版本

```
# 查询合约名为HelloWorld，版本为v_1.0的CNS信息
$ ./console.py queryCNSByNameAndVersion HelloWorld v_1.0

INFO >> user input : ['queryCNSByNameAndVersion', 'HelloWorld', 'v_1.0']

INFO >> CNS version (strip space): v_1.0
    >> ('[{"abi":"\\"\\"","address":"0x2d1c577e41809453C50e7E5C3F57D06f3CDD90Ce",
↪"name":"HelloWorld","version":"v_1.0"}]\n',)
CNS ITEM 0 >>
        ContractName: HelloWorld
        ContractVersion: v_1.0
        ContractAddress: 0x2d1c577e41809453C50e7E5C3F57D06f3CDD90Ce
```

节点管理

Python SDK提供了节点管理命令，包括添加共识节点、添加观察者节点、将节点从群组中删除，节点管理的详细资料可参考这里。

removeNode

将指定该节点从群组中删除:

```
./console.py removeNode [nodeId]
```

参数包括:

- nodeId：被删除节点的nodeID

```
# 设节点位于~/fisco/nodes目录，查询node1的nodeID
$ cat ~/fisco/nodes/127.0.0.1/node1/conf/node.nodeid
12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb24ff
# 将节点1从群组中删除node1
./console.py removeNode␣
↪12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb24

INFO >> user input : ['removeNode',
↪'12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb2
↪']

INFO >> removeNode
     >> status: 0x0
     >> transactionHash:␣
↪0x68cde78d76f490b35431905d2336d9811966a370da8b4041db092feb09981f28
     >> gasUsed: 0x7698
     >> removeNode succ, output: 1
```

addSealer

将指定节点加入共识节点列表:

```
./console.py addSealer [nodeId]
```

参数包括:

- nodeId: 加入的共识节点nodeID，获取节点nodeID可参考这里

```
# 设节点位于~/fisco/nodes目录，查询node1的nodeID
$ cat ~/fisco/nodes/127.0.0.1/node1/conf/node.nodeid
12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb24ff

# 将节点node1加入为共识节点
$./console.py addSealer␣
↪12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb24

INFO >> user input : ['addSealer',
↪'12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb2
↪']

INFO >> addSealer
     >> status: 0x0
     >> transactionHash:␣
↪0xfddfa618419880e37f82c8cd385994fcb1ee1d4c5b4b506ae0d67f223c8b723d
     >> gasUsed: 0x7698
     >> addSealer succ, output: 1
```

addObserver

将指定节点加入为观察者节点:

```
./console.py addObserver [nodeId]
```

参数包括：

- nodeId: 加入的观察者节点nodeID，获取节点nodeID可参考这里

```
# 设节点位于~/fisco/nodes目录，查询node1的nodeID
$ cat ~/fisco/nodes/127.0.0.1/node1/conf/node.nodeid
12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb24ff

# 将节点node1加入为观察节点
$ ./console.py addObserver␣
↪12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb24

INFO >> user input : ['addObserver',
↪'12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb2
↪']

INFO >> addObserver
     >> status: 0x0
     >> transactionHash:␣
↪0xb126900787205a5f913e6643058359a07ace1cc550190a5a9478ae4f49cfc1eb
     >> gasUsed: 0x7658
     >> addObserver succ, output: 1
```

系统配置

Python SDK提供了系统配置修改命令，FISCO BCOS目前支持的系统配置参考这里。

```
./console.py setSystemConfigByKey [key(tx_count_limit/tx_gas_limit)] [value]
```

参数包括：

- key：配置关键字，目前主要包括tx_count_limit和tx_gas_limit
- value: 配置关键字的值

```
# 将区块内最大交易数目调整为500
$ ./console.py setSystemConfigByKey tx_count_limit 500

INFO >> user input : ['setSystemConfigByKey', 'tx_count_limit', '500']

INFO >> setSystemConfigByKey
     >> status: 0x0
     >> transactionHash:␣
↪0xded8abc0858f8a7be5961ae38958928c98f75ee78dbe8197a47c382cb2549de1
     >> gasUsed: 0x5b58
     >> setSystemConfigByKey succ, output: 1

# 将交易gas限制调整为400000000
$ ./console.py setSystemConfigByKey tx_gas_limit 400000000

INFO >> user input : ['setSystemConfigByKey', 'tx_gas_limit', '400000000']

INFO >> setSystemConfigByKey
     >> status: 0x0
     >> transactionHash:␣
↪0x4b78868ec183c432e07971f578f5ab8222a9effda39dfa8e87643410cb2cea05
     >> gasUsed: 0x5c58
     >> setSystemConfigByKey succ, output: 1
```

权限管理

Python SDK提供了权限管理功能，包括授权、撤销权限和列出权限列表等，权限控制的详细内容可参考这里。

grantPermissionManager

将控制权限的功能授权给指定账户：

```
./console.py grantPermissionManager [account_adddress]
```

参数包括：

- account_adddress：被授予权限的账户地址，账户可通过newaccount命令生成

```
# 获取默认账户地址
./console.py showaccount pyaccount "123456"
INFO >> user input : ['showaccount', 'pyaccount', '123456']
show account : pyaccount, keyfile:bin/accounts/pyaccount.keystore ,password 123456
decrypt use time : 1.450 s
address:         0x95198B93705e394a916579e048c8A32DdFB900f7
privkey:         0x48140af2cf0879631d558833aa48b7bb4b37091dbfe902a573886538041b69c0
pubkey :
↪0x142d340c0f4df64bf56bbc0a3931e5228c7836add09cf8ff3cefeb3d7e610deb458ec871a9da86bae1ffc029f5aba
account store in file: [bin/accounts/pyaccount.keystore]
**** please remember your password !!! *****

# 为账户0x95198B93705e394a916579e048c8A32DdFB900f7添加权限管理权限
$ ./console.py grantPermissionManager 0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['grantPermissionManager',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> grantPermissionManager
     >> status: 0x0
     >> transactionHash:
↪0xdac11796dcfb663842a13333976626d844527605edb5bf9daadcfa28236bb5c8
     >> gasUsed: 0x6698
     >> grantPermissionManager succ, output: 1
```

listPermissionManager

列出有权限管理功能的账户信息：

```
# 列出所有权限管理账户信息
$ ./console.py listPermissionManager
INFO >> user input : ['listPermissionManager']
----->> ITEM 0
     = address: 0x95198B93705e394a916579e048c8A32DdFB900f7
     = enable_num: 9
```

grantUserTableManager

将给定用户表权限授予指定用户：

```
./console.py grantUserTableManager [tableName] [account_adddress]
```

注解： 给用户授权用户表权限前，请确保用户表存在，可用 createTable 命令创建用户表

参数包括：

- tableName： 用户表名
- account_adddress：被授权用户账户地址

```
# 创建用户表t_test
$./console.py createTable t_test "key" "value1, value2, value3"
INFO >> user input : ['createTable', 't_test', 'key', 'value1, value2, value3']
INFO >> createTable
     >> status: 0x0
     >> transactionHash:␣
↪0xfbc10c0d9e4652f59655903e5ba772bb7f127e8e9de12be250d487f0ff9c5268
     >> gasUsed: 0x6098
     >> createTable succ, output: 0

# 为账户0x95198B93705e394a916579e048c8A32DdFB900f7对用户表t_test的管理功能
$ ./console.py grantUserTableManager t_test␣
↪0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['grantUserTableManager', 't_test',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> table t_test
     >> key_field: key
     >> value_field: value1,value2,value3
INFO >> grantUserTableManager
     >> status: 0x0
     >> transactionHash:␣
↪0x2b9640f02db7afa839b5bdf158cca33a96a9718dc2e80f2c7b8af6100f6f8e92
     >> gasUsed: 0x6398
     >> grantUserTableManager succ, output: 1
```

listUserTableManager

列出对指定用户表有管理权限的账户信息：

```
./console.py listUserTableManager [tableName]
```

参数包括：

- tableName: 用户表

```
# 查看用户表t_test的管理信息
$./console.py listUserTableManager t_test
INFO >> user input : ['listUserTableManager', 't_test']
----->> ITEM 0
     = address: 0x95198B93705e394a916579e048c8A32DdFB900f7
     = enable_num: 11
```

grantNodeManager

将节点管理权限授予指定账户：

```
./console.py grantNodeManager [account_adddress]
```

参数包括：

- account_adddress：被授权用户账户地址

```
# 为账户0x95198B93705e394a916579e048c8A32DdFB900f7添加节点管理功能
$ ./console.py grantNodeManager 0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['grantNodeManager',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> grantNodeManager
     >> status: 0x0
     >> transactionHash:␣
↪0x3a8839bfdfefcd3fff2678f91f231d44d8d442e40fc7f3af726daec624ba80c8
     >> gasUsed: 0x65d8
     >> grantNodeManager succ, output: 1
```

listNodeManager

列出有节点管理功能的账户信息：

```
$ ./console.py listNodeManager
INFO >> user input : ['listNodeManager']
----->> ITEM 0
     = address: 0x95198B93705e394a916579e048c8A32DdFB900f7
     = enable_num: 12
```

grantCNSManager

将CNS管理权限授予指定账户：

```
./console.py grantCNSManager [account_adddress]
```

参数包括：

- account_adddress：被授权用户账户地址

```
# 为账户0x95198B93705e394a916579e048c8A32DdFB900f7添加CNS管理权限
$ ./console.py grantCNSManager 0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['grantCNSManager',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> grantCNSManager
     >> status: 0x0
     >> transactionHash:␣
↪0x4a112be9f582fb1ae98ae9d6a84706930f4ab3523b45722cc4bf08341397dd1e
     >> gasUsed: 0x6458
     >> grantCNSManager succ, output: 1
```

listCNSManager

列出有CNS管理权限的账户信息

```
$ ./console.py listCNSManager

INFO >> user input : ['listCNSManager']

----->> ITEM 0
     = address: 0x95198B93705e394a916579e048c8A32DdFB900f7
     = enable_num: 13
```

grantSysConfigManager

将系统配置修改权限授予指定账户：

```
./console.py grantSysConfigManager [account_adddress]
```

参数包括：

- account_adddress：被授权用户账户地址

```
# 为账户0x95198B93705e394a916579e048c8A32DdFB900f7添加系统配置权限
$ ./console.py grantSysConfigManager 0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['grantSysConfigManager',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> grantSysConfigManager
     >> status: 0x0
     >> transactionHash:␣
↪0xf6ec040686496256a8c01233d1339ee147551f6a2dfcbd7bd6d7647f240f1411
     >> gasUsed: 0x6518
     >> grantSysConfigManager succ, output: 1
```

listSysConfigManager

列出有系统配置修改权限的账户信息：

```
$ ./console.py listSysConfigManager
INFO >> user input : ['listSysConfigManager']
----->> ITEM 0
     = address: 0x95198B93705e394a916579e048c8A32DdFB900f7
     = enable_num: 14
```

grantDeployAndCreateManager

将部署和创建表的权限授予指定账户：

```
./console.py grantDeployAndCreateManager [account_adddress]
```

参数包括：

- account_adddress：被授权用户账户地址

```
# 为账户0x95198B93705e394a916579e048c8A32DdFB900f7添加创建表和部署合约权限
$./console.py grantDeployAndCreateManager␣
↪0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['grantDeployAndCreateManager',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> grantDeployAndCreateManager
     >> status: 0x0
     >> transactionHash:␣
↪0xf60452a12d5346fa641bca6bee662c261fa0c67ef90aca3944cdb29a5803c625
     >> gasUsed: 0x6518
     >> grantDeployAndCreateManager succ, output: 1
```

listDeployAndCreateManager

列出有创建合约和用户表的账户信息：

```
$ ./console.py listDeployAndCreateManager
INFO >> user input : ['listDeployAndCreateManager']
------>> ITEM 0
      = address: 0x95198B93705e394a916579e048c8A32DdFB900f7
      = enable_num: 15
```

revokeUserTableManager

撤销指定用户对指定用户表的写入权限：

```
./console.py revokeUserTableManager [tableName] [account_adddress]
```

参数包括：

- tableName：禁止指定用户写入的表名
- account_adddress：被撤销权限的账户地址

```
# 撤销账户0x95198B93705e394a916579e048c8A32DdFB900f7对用户表t_test的控制权限
$ ./console.py revokeUserTableManager t_test␣
↪0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['revokeUserTableManager', 't_test',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> revokeUserTableManager
      >> status: 0x0
      >> transactionHash:␣
↪0xc7ffbd0f79bfe06f43c603afde5997f9127a9fe499338362e64c653a593ded36
      >> gasUsed: 0x6398
      >> revokeUserTableManager succ, output: 1
```

revokeDeployAndCreateManager

撤销指定账户创建表、部署合约的权限：

```
./console.py revokeDeployAndCreateManager [account_adddress]
```

参数包括：

- account_adddress：被撤销权限的账户地址

```
# 撤销账户0x95198B93705e394a916579e048c8A32DdFB900f7部署和创建表权限
$ ./console.py revokeDeployAndCreateManager␣
↪0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['revokeDeployAndCreateManager',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> revokeDeployAndCreateManager
      >> status: 0x0
      >> transactionHash:␣
↪0xeac82f3464093f0659eb6412c39599d51b64082401ac43df9d7670cf17882f78
      >> gasUsed: 0x6518
      >> revokeDeployAndCreateManager succ, output: 1
```

revokeNodeManager

撤销指定账户的节点管理权限：

```
./console.py revokeNodeManager [account_adddress]
```

参数包括：

- account_adddress：被撤销权限的账户地址

```
# 撤销账户0x95198B93705e394a916579e048c8A32DdFB900f7节点管理权限
$ ./console.py revokeNodeManager 0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['revokeNodeManager',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> revokeNodeManager
     >> status: 0x0
     >> transactionHash:␣
↪0xc9f3799dc81a146f562fe10b493d14920676a8e49a6de94e7b4b998844198342
     >> gasUsed: 0x65d8
     >> revokeNodeManager succ, output: 1
```

revokeCNSManager

撤销指定账户CNS管理权限：

```
./console.py revokeCNSManager [account_adddress]
```

参数包括：

- account_adddress：被撤销权限的账户地址

```
# 撤销账户0x95198B93705e394a916579e048c8A32DdFB900f7 CNS管理权限
$ ./console.py revokeCNSManager 0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['revokeCNSManager',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> revokeCNSManager
     >> status: 0x0
     >> transactionHash:␣
↪0xa5aa6d115875156512af8c9974e353336e00bc3b9c2f2c2e21749d728e45abb4
     >> gasUsed: 0x6458
     >> revokeCNSManager succ, output: 1
```

revokeSysConfigManager

撤销指定账户修改系统配置权限：

```
./console.py revokeSysConfigManager [account_adddress]
```

参数包括：

- account_adddress：被撤销权限的账户地址

```
# 撤销账户0x95198B93705e394a916579e048c8A32DdFB900f7系统表管理权限
$ ./console.py revokeSysConfigManager 0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['revokeSysConfigManager',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> revokeSysConfigManager
     >> status: 0x0
     >> transactionHash:␣
↪0xfaffc25a4b111cfdaddca323d8b125c553c5e8f83b85fae1de21a6bc3bef792a
     >> gasUsed: 0x6518
     >> revokeSysConfigManager succ, output: 1
```

revokePermissionManager

撤销指定账户管理权限的权限：

```
./console.py revokePermissionManager [account_adddress]
```

参数包括：

- account_adddress：被撤销权限的账户地址

```
# 撤销账户0x95198B93705e394a916579e048c8A32DdFB900f7权限管理权限
$ ./console.py revokePermissionManager 0x95198B93705e394a916579e048c8A32DdFB900f7
INFO >> user input : ['revokePermissionManager',
↪'0x95198B93705e394a916579e048c8A32DdFB900f7']
INFO >> revokePermissionManager
     >> status: 0x0
     >> transactionHash:␣
↪0xa9398d4de7a3e86238a48bdbf5e053c61bc57ccd1aa57ebaa3c070bc47ea0f98
     >> gasUsed: 0x6698
     >> revokePermissionManager succ, output: 1
```

RPC

可以通过Python SDK查询节点信息，目前Python SDK支持的查询命令如下：

getNodeVersion

获取节点版本信息：

```
$ ./console.py getNodeVersion

INFO >> user input : ['getNodeVersion']

INFO >> getNodeVersion
     >> {
    "Build Time":"20200619 06:32:10",
    "Build Type":"Linux/clang/Release",
    "Chain Id":"1",
    "FISCO-BCOS Version":"2.5.0",
    "Git Branch":"HEAD",
    "Git Commit Hash":"72c6d770e5cf0f4197162d0e26005ec03d30fcfe",
    "Supported Version":"2.5.0"
}
```

getBlockNumber

获取节点最新块高：

```
$ ./console.py getBlockNumber
INFO >> user input : ['getBlockNumber']
INFO >> getBlockNumber
     >> 21
```

getPbftView

获取节点共识视图：

```
$ ./console.py getPbftView

INFO >> user input : ['getPbftView']

INFO >> getPbftView
     >> 0x34e
```

getSealerList

获取当前共识节点列表:

```
$ ./console.py getSealerList

INFO >> user input : ['getSealerList']

INFO >> getSealerList
     >>␣
↪3ad90ae5a10b8d88c9936492a37f564884e82b176e91f5e2e2f75a347be87212aac148ee7fa2060be8a790eaa3d44a2
     >>␣
↪6bd07f2f8180ac9d56b40ff9977ba528a4f65e83d4ca95a0537e12f6638f78848e0765cbee0cb2b5f581d7eb5027d18
     >>␣
↪b8783cfe3c073a532e9cbc47978d45a187da179d2fef4a85990c3b286d69d1afcd061de1b8cba07a59819d94f021db1
```

getObserverList

获取观察者节点列表:

```
$ ./console.py getObserverList
INFO >> user input : ['getObserverList']
INFO >> getObserverList
     >>␣
↪12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb24
```

getConsensusStatus

获取节点共识状态信息:

```
$ ./console.py getConsensusStatus

INFO >> user input : ['getConsensusStatus']

INFO >> getConsensusStatus
     >> {
    "accountType": 1,
    "allowFutureBlocks": true,
    "cfgErr": false,
    "connectedNodes": 3,
    "consensusedBlockNumber": 22,
    "currentView": 904,
    "groupId": 1,
    "highestblockHash":
↪"0x2aa73c33c054eb168dd1cb5d62cd211c780731c3fe40333be0f32069568d0083",
    "highestblockNumber": 21,
    "leaderFailed": false,
    "max_faulty_leader": 0,
    "nodeId":
↪"b8783cfe3c073a532e9cbc47978d45a187da179d2fef4a85990c3b286d69d1afcd061de1b8cba07a59819d94f021db
↪",
```

(continues on next page)

(续上页)

```
    "nodeNum": 3,
    "node_index": 2,
    "omitEmptyBlock": true,
    "protocolId": 65544,
    ... 省略若干行 ...
}
```

getSyncStatus

获取节点同步状态信息：

```
$ ./console.py getSyncStatus

INFO >> user input : ['getSyncStatus']

INFO >> getSyncStatus
     >> {
    "blockNumber": 21,
    "genesisHash":
↪"0xff1404962c6c063a98cc9e6a20b408e6a612052dc4267836bb1dc378acc6ce04",
    "isSyncing": false,
    "knownHighestNumber": 21,
    "knownLatestHash":
↪"2aa73c33c054eb168dd1cb5d62cd211c780731c3fe40333be0f32069568d0083",
    "latestHash":
↪"0x2aa73c33c054eb168dd1cb5d62cd211c780731c3fe40333be0f32069568d0083",
    "nodeId":
↪"b8783cfe3c073a532e9cbc47978d45a187da179d2fef4a85990c3b286d69d1afcd061de1b8cba07a59819d94f021db
↪",
    "peers": [
        {
            "blockNumber": 21,
            "genesisHash":
↪"0xff1404962c6c063a98cc9e6a20b408e6a612052dc4267836bb1dc378acc6ce04",
            "latestHash":
↪"0x2aa73c33c054eb168dd1cb5d62cd211c780731c3fe40333be0f32069568d0083",
            "nodeId":
↪"12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb2
↪"
        },
        {
            "blockNumber": 21,
            "genesisHash":
↪"0xff1404962c6c063a98cc9e6a20b408e6a612052dc4267836bb1dc378acc6ce04",
            "latestHash":
↪"0x2aa73c33c054eb168dd1cb5d62cd211c780731c3fe40333be0f32069568d0083",
            "nodeId":
↪"3ad90ae5a10b8d88c9936492a37f564884e82b176e91f5e2e2f75a347be87212aac148ee7fa2060be8a790eaa3d44a
↪"
        },
        {
            "blockNumber": 21,
            "genesisHash":
↪"0xff1404962c6c063a98cc9e6a20b408e6a612052dc4267836bb1dc378acc6ce04",
            "latestHash":
↪"0x2aa73c33c054eb168dd1cb5d62cd211c780731c3fe40333be0f32069568d0083",
            "nodeId":
↪"6bd07f2f8180ac9d56b40ff9977ba528a4f65e83d4ca95a0537e12f6638f78848e0765cbee0cb2b5f581d7eb5027d1
↪"
```

(continues on next page)

(续上页)

```
        }
    ],
    "protocolId": 65545,
    "txPoolSize": "0"
}
```

getPeers

获取节点连接列表：

```
$ ./console.py getPeers
INFO >> user input : ['getPeers']
INFO >> getPeers
     >> {
    "Agency": "agency",
    "IPAndPort": "127.0.0.1:30301",
    "Node": "node1",
    "NodeID":
↪"12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb24
↪",
    "Topic": []
}
     >> {
    "Agency": "agency",
    "IPAndPort": "127.0.0.1:30302",
    "Node": "node2",
    "NodeID":
↪"6bd07f2f8180ac9d56b40ff9977ba528a4f65e83d4ca95a0537e12f6638f78848e0765cbee0cb2b5f581d7eb5027d18
↪",
    "Topic": []
}
     >> {
    "Agency": "agency",
    "IPAndPort": "127.0.0.1:30303",
    "Node": "node3",
    "NodeID":
↪"3ad90ae5a10b8d88c9936492a37f564884e82b176e91f5e2e2f75a347be87212aac148ee7fa2060be8a790eaa3d44a2
↪",
    "Topic": []
}
```

getGroupPeers

获取群组内节点连接信息：

```
$ ./console.py getGroupPeers
INFO >> user input : ['getGroupPeers']
INFO >> getGroupPeers
     >>␣
↪3ad90ae5a10b8d88c9936492a37f564884e82b176e91f5e2e2f75a347be87212aac148ee7fa2060be8a790eaa3d44a2
     >>␣
↪6bd07f2f8180ac9d56b40ff9977ba528a4f65e83d4ca95a0537e12f6638f78848e0765cbee0cb2b5f581d7eb5027d18
     >>␣
↪b8783cfe3c073a532e9cbc47978d45a187da179d2fef4a85990c3b286d69d1afcd061de1b8cba07a59819d94f021db1
     >>␣
↪12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb24
```

getNodeIDList

获取区块链所有组网节点列表：

```
$ ./console.py getNodeIDList
INFO >> user input : ['getNodeIDList']
INFO >> getNodeIDList
     >>␣
↪b8783cfe3c073a532e9cbc47978d45a187da179d2fef4a85990c3b286d69d1afcd061de1b8cba07a59819d94f021db1
     >>␣
↪12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb24
     >>␣
↪6bd07f2f8180ac9d56b40ff9977ba528a4f65e83d4ca95a0537e12f6638f78848e0765cbee0cb2b5f581d7eb5027d18
     >>␣
↪3ad90ae5a10b8d88c9936492a37f564884e82b176e91f5e2e2f75a347be87212aac148ee7fa2060be8a790eaa3d44a2
```

getGroupList

获取群组列表：

```
$ ./console.py getGroupList
INFO >> user input : ['getGroupList']
INFO >> getGroupList
     >> 1
```

getPendingTransactions

获取交易池内还未上链的交易信息：

```
$ ./console.py getPendingTransactions
INFO >> user input : ['getPendingTransactions']
INFO >> getPendingTransactions
     >> Empty Set
```

getPendingTxSize

获取交易池内还未上链的交易数目：

```
$ ./console.py getPendingTxSize
INFO >> user input : ['getPendingTxSize']
INFO >> getPendingTxSize
     >> 0x0
```

getTotalTransactionCount

获取已经上链的交易数目：

```
$ ./console.py getTotalTransactionCount
INFO >> user input : ['getTotalTransactionCount']
INFO >> getTotalTransactionCount
     >> {
    "blockNumber": "0x16",
    "failedTxSum": "0x0",
    "txSum": "0x16"
     }
```

getBlockByNumber

根据块高查询区块：

```
$ ./console.py getBlockByNumber [block_number] [True/False]
```

参数包括：

- block_number：区块高度
- True/False: 可选，True表明返回的区块信息内包含具体的交易信息；False表明返回的区块内仅包含交易哈希

```
$ ./console.py getBlockByNumber 0

INFO >> user input : ['getBlockByNumber', '0']

INFO >> getBlockByNumber
     >> {
    "dbHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData": [
        "0x312d623837383336366533633037336135333265396362633437393738 64
        ... 省略若干行...
        7652d313030302d333030303030303030"
    ],
    "gasLimit": "0x0",
    "gasUsed": "0x0",
    "hash": "0xff1404962c6c063a98cc9e6a20b408e6a612052dc4267836bb1dc378acc6ce04",
    "logsBloom": "0x00000000... 省略若干行...00000000000000000000",
    "number": "0x0",
    "parentHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
    "receiptsRoot":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
    "sealer": "0x0",
    "sealerList": [],
    "stateRoot":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
    "timestamp": "0x16c61113388",
    "transactions": [],
    "transactionsRoot":
↪"0x0000000000000000000000000000000000000000000000000000000000000000"
}
```

getBlockHashByNumber

根据块高查询区块哈希：

```
$ ./console.py getBlockHashByNumber 0
INFO >> user input : ['getBlockHashByNumber', '0']
INFO >> getBlockHashByNumber
     >> 0xff1404962c6c063a98cc9e6a20b408e6a612052dc4267836bb1dc378acc6ce04
```

getBlockByHash

根据区块哈希获取区块信息：

```
$ ./console.py getBlockByHash [block_hash] [True/False]
```

参数包括:

- block_hash：区块哈希
- True/False: 可选，True表明返回的区块内包含交易具体信息；False表明返回的区块仅包含交易哈希

```
$ ./console.py getBlockByHash␣
↪0xff1404962c6c063a98cc9e6a20b408e6a612052dc4267836bb1dc378acc6ce04
INFO >> user input : ['getBlockByHash',
↪'0xff1404962c6c063a98cc9e6a20b408e6a612052dc4267836bb1dc378acc6ce04']
INFO >> getBlockByHash
     >> {
    "extraData": [
        "0x312d623...省略若干...3030303030"
    ],
    "gasLimit": "0x0",
    "gasUsed": "0x0",
    "hash": "0xff1404962c6c063a98cc9e6a20b408e6a612052dc4267836bb1dc378acc6ce04",
    "logsBloom": "0x0000...省略若干...000000",
    "number": "0x0",
    "parentHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
    "sealer": "0x0",
    "sealerList": [],
    "stateRoot":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
    "timestamp": "0x16c61113388",
    "transactions": [],
    "transactionsRoot":
↪"0x0000000000000000000000000000000000000000000000000000000000000000"
}
```

getCode

获取指定合约的二进制编码:

```
$ ./console.py getCode 0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce
INFO >> user input : ['getCode', '0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce']
INFO >> getCode
     >> 0x60806040526...省略若干...a40029
```

getTransactionByHash

根据交易哈希获取交易信息:

```
./console.py getTransactionByHash [hash] [contract_name]
```

参数包括:

- hash: 交易哈希
- contract_name：可选，该交易相关的合约名，若输入该参数，会解析返回交易的具体内容

```
$ ./console.py getTransactionByHash␣
↪0xb291e9ca38b53c897340256b851764fa68a86f2a53cb14b2ecdcc332e850bb91
INFO >> user input : ['getTransactionByHash',
↪'0xb291e9ca38b53c897340256b851764fa68a86f2a53cb14b2ecdcc332e850bb91']
INFO >> getTransactionByHash
     >> {
    "blockHash":
↪"0x3912605dde5f7358fee40a85a8b97ba6493848eae7766a8c317beecafb2e279d",
```

(continues on next page)

(续上页)

```
    "blockNumber": "0x1",
    "from": "0x95198b93705e394a916579e048c8a32ddfb900f7",
    "gas": "0x1c9c380",
    "gasPrice": "0x1c9c380",
    "hash": "0xb291e9ca38b53c897340256b851764fa68a86f2a53cb14b2ecdcc332e850bb91",
    "input": "0x60806...省略若干...ddd81c4a40029",
    "nonce": "0x2b2350c8",
    "to": "0x0000000000000000000000000000000000000000",
    "transactionIndex": "0x0",
    "value": "0x0"
}
```

getTransactionReceipt

根据交易哈希获取交易回执信息：

```
./console.py getTransactionReceipt [hash] [contract_name]
```

参数包括：

- hash：交易哈希
- contract_name：可选，该交易相关的合约名，若输入该参数，会解析交易和回执的具体内容

```
$ ./console.py getTransactionReceipt␣
↪0xb291e9ca38b53c897340256b851764fa68a86f2a53cb14b2ecdcc332e850bb91
INFO >> user input : ['getTransactionReceipt',
↪'0xb291e9ca38b53c897340256b851764fa68a86f2a53cb14b2ecdcc332e850bb91']
INFO >> getTransactionReceipt
     >> {
    "blockHash":
↪"0x3912605dde5f7358fee40a85a8b97ba6493848eae7766a8c317beecafb2e279d",
    "blockNumber": "0x1",
    "contractAddress": "0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce",
    "from": "0x95198b93705e394a916579e048c8a32ddfb900f7",
    "gasUsed": "0x44ab3",
    "input": "0x608060405234...省略若干...d9acf16e2fc2d570d491ddd81c4a40029",
    "logs": [],
    "logsBloom": "0x00000...省略若干...00000000000",
    "output": "0x",
    "status": "0x0",
    "to": "0x0000000000000000000000000000000000000000",
    "transactionHash":
↪"0xb291e9ca38b53c897340256b851764fa68a86f2a53cb14b2ecdcc332e850bb91",
    "transactionIndex": "0x0"
}
```

getTransactionByBlockHashAndIndex

根据区块哈希和交易索引查询交易信息：

```
./console.py getTransactionByBlockHashAndIndex [blockHash] [transactionIndex]␣
↪[contract_name]
```

参数包括：

- blockHash: 交易所在的区块哈希
- transactionIndex：交易索引

- contract_name：可选，该交易相关的合约名，若输入该参数，会解析返回交易的具体内容

```
$  ./console.py getTransactionByBlockHashAndIndex␣
↪0x3912605dde5f7358fee40a85a8b97ba6493848eae7766a8c317beecafb2e279d 0

INFO >> user input : ['getTransactionByBlockHashAndIndex',
↪'0x3912605dde5f7358fee40a85a8b97ba6493848eae7766a8c317beecafb2e279d', '0']

INFO >> getTransactionByBlockHashAndIndex
     >> {
    "blockHash":
↪"0x3912605dde5f7358fee40a85a8b97ba6493848eae7766a8c317beecafb2e279d",
    "blockNumber": "0x1",
    "from": "0x95198b93705e394a916579e048c8a32ddfb900f7",
    "gas": "0x1c9c380",
    "gasPrice": "0x1c9c380",
    "hash": "0xb291e9ca38b53c897340256b851764fa68a86f2a53cb14b2ecdcc332e850bb91",
    "input": "0x6080...省略若干...4a40029",
    "nonce": "0x2b2350c8",
    "to": "0x0000000000000000000000000000000000000000",
    "transactionIndex": "0x0",
    "value": "0x0"
}
```

getTransactionByBlockNumberAndIndex

根据块高和交易索引查询交易信息：

```
$ ./console.py getTransactionByBlockNumberAndIndex [blockNumber]␣
↪[transactionIndex] [contract_name]
```

参数包括：

- blockNumber：交易所在的区块块高
- transactionIndex：交易索引
- contract_name：可选，该交易相关的合约名，若输入该参数，会解析返回交易的具体内容

```
$ ./console.py getTransactionByBlockNumberAndIndex 1 0
INFO >> user input : ['getTransactionByBlockNumberAndIndex', '1', '0']
INFO >> getTransactionByBlockNumberAndIndex
    >> {
   "blockHash": "0x3912605dde5f7358fee40a85a8b97ba6493848eae7766a8c317beecafb2e279d
↪",
   "blockNumber": "0x1",
   "from": "0x95198b93705e394a916579e048c8a32ddfb900f7",
   "gas": "0x1c9c380",
   "gasPrice": "0x1c9c380",
   "hash": "0xb291e9ca38b53c897340256b851764fa68a86f2a53cb14b2ecdcc332e850bb91",
   "input": "0x608060...省略若干...a40029",
   "nonce": "0x2b2350c8",
   "to": "0x0000000000000000000000000000000000000000",
   "transactionIndex": "0x0",
   "value": "0x0"
}
```

getSystemConfigByKey

获取系统配置信息：

```
# 获取区块可打包最大交易数目
$ ./console.py getSystemConfigByKey tx_count_limit
INFO >> user input : ['getSystemConfigByKey', 'tx_count_limit']
INFO >> getSystemConfigByKey tx_count_limit
     >> 500
# 获取系统gas限制
$ ./console.py getSystemConfigByKey  tx_gas_limit
INFO >> user input : ['getSystemConfigByKey', 'tx_gas_limit']
INFO >> getSystemConfigByKey tx_gas_limit
     >> 400000000
```

7.3.5 开发样例

Python SDK的源码中提供了完整的Demo供开发者学习

- 调用节点API
- 部署合约、发送交易、处理回执、查询合约数据

调用节点API

正确的配置了SDK连接的节点信息后。在代码中实例化client结构，并调用client的接口即可。返回json，可以根据对fisco bcos rpc接口json格式的理解，进行字段获取和转码。

完整Demo: demo_get.py

```
# 实例化client
client = BcosClient()

# 调用查节点版本接口
res = client.getNodeVersion()
print("getClientVersion",res)

# 调用查节点块高接口
try:
    res = client.getBlockNumber()
    print("getBlockNumber",res)
except BcosError as e:
    print("bcos client error,",e.info())
```

操作合约

正确的配置了SDK连接的节点信息后。可进行部署合约、发送交易、处理回执、查询合约数据的操作。按照举例，调用deploy，sendRawTransactionGetReceipt，call，parse_event_logs等函数。

完整Demo: demo_transaction.py

```
# 实例化client
client = BcosClient()

# 从文件加载abi定义
abi_file  ="contracts/SimpleInfo.abi"
data_parser = DatatypeParser()
data_parser.load_abi_file(abi_file)
contract_abi = data_parser.contract_abi

# 部署合约
```

(continues on next page)

(续上页)

```
print("\n>>Deploy:----------------------------------------------------------------
↪----")
with open("contracts/SimpleInfo.bin", 'r') as load_f:
    contract_bin = load_f.read()
    load_f.close()
result = client.deploy(contract_bin)
print("deploy",result)
print("new address : ",result["contractAddress"])
contract_name =  os.path.splitext(os.path.basename(abi_file))[0]
memo = "tx:"+result["transactionHash"]
#把部署结果存入文件备查
from client.contractnote import ContractNote
ContractNote.save_address(contract_name, result["contractAddress"], int(result[
↪"blockNumber"], 16), memo)


#发送交易，调用一个改写数据的接口
print("\n>>sendRawTransaction:----------------------------------------------------
↪-----")
to_address = result['contractAddress'] #use new deploy address
args = ['simplename', 2024, to_checksum_address(
↪'0x7029c502b4F824d19Bd7921E9cb74Ef92392FB1c')]

receipt = client.sendRawTransactionGetReceipt(to_address,contract_abi,"set",args)
print("receipt:",receipt)

#解析receipt里的log
print("\n>>parse receipt and transaction:-----------------------------------------
↪----------------")
txhash = receipt['transactionHash']
print("transaction hash: " ,txhash)
logresult = data_parser.parse_event_logs(receipt["logs"])
i = 0
for log in logresult:
    if 'eventname' in log:
        i = i + 1
        print("{}): log name: {} , data: {}".format(i,log['eventname'],log[
↪'eventdata']))
#获取对应的交易数据，解析出调用方法名和参数

txresponse = client.getTransactionByHash(txhash)
inputresult = data_parser.parse_transaction_input(txresponse['input'])
print("transaction input parse:",txhash)
print(inputresult)

#解析该交易在receipt里输出的output,即交易调用的方法的return值
outputresult  = data_parser.parse_receipt_output(inputresult['name'], receipt[
↪'output'])
print("receipt output :",outputresult)


#调用一下call，获取数据
print("\n>>Call:------------------------------------------------------------------
↪-----")
res = client.call(to_address,contract_abi,"getbalance")
print("call getbalance result:",res)
res = client.call(to_address,contract_abi,"getbalance1",[100])
print("call getbalance1 result:",res)
res = client.call(to_address,contract_abi,"getname")
print("call getname:",res)
res = client.call(to_address,contract_abi,"getall")
```

(continues on next page)

(续上页)

```
print("call getall result:",res)
print("demo_tx,total req {}".format(client.request_counter))
client.finish()
```

CHAPTER 8

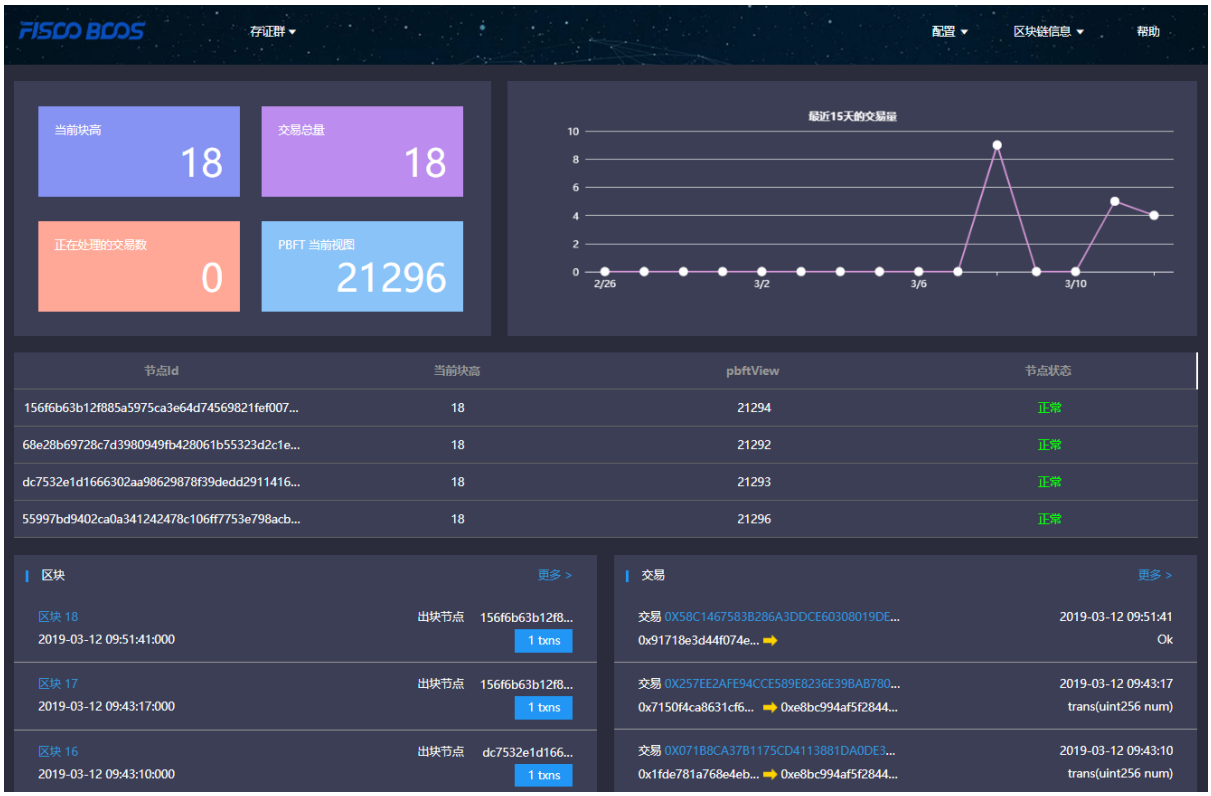
区块链浏览器

8.1 一、描述

8.1.1 1.1、基本描述

全新适配FISCO BCOS 2.0+版本，如果使用FISCO BCOS 1.2或1.3版本请用v1.2.1版本。

区块链浏览器将区块链中的数据可视化，并进行实时展示。方便用户以Web页面的方式，获取当前区块链中的信息。本浏览器版本适配FISCO BCOS 2.0+，关于2.0+版本的特性可以参考此链接。在使用本浏览器之前需要先理解2.0+版本的群组特性，详情可以参考此链接。

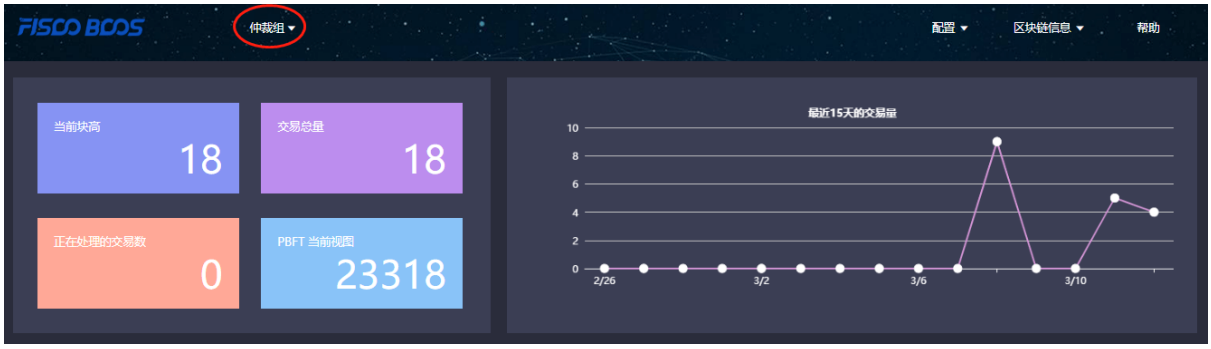


1.2、主要功能模块

本小节概要介绍浏览器的各个模块，方便大家对浏览器有一个整体的认识。区块链浏览器主要的功能模块有：群组切换模块，配置模块，区块链信息展示模块。

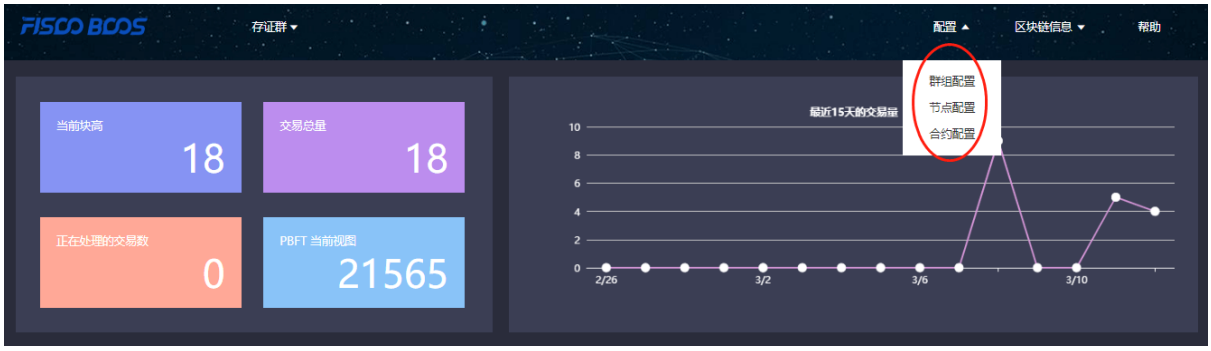
1.2.1、群组切换模块

群组切换主要用于在多群组场景中切换到不同群组，进行区块链信息浏览。



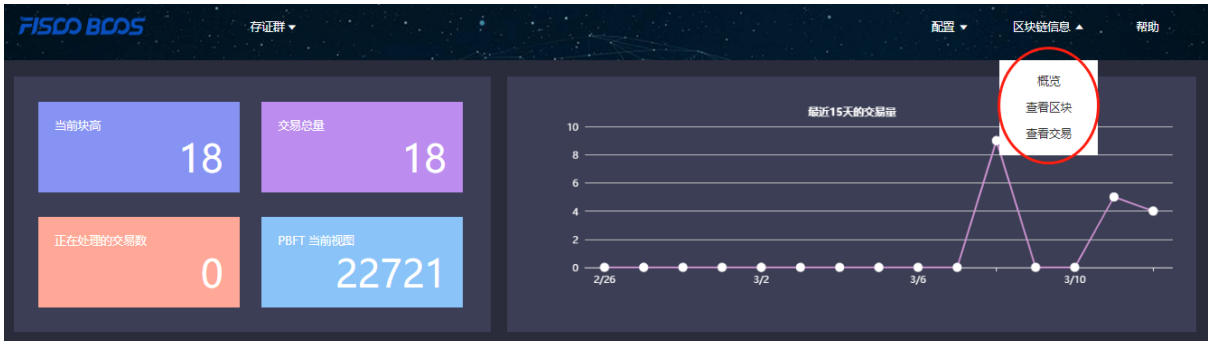
1.2.2、配置模块

主要包括群组配置，节点配置，合约配置。



1.2.3、区块链信息展示模块

区块链浏览器主要展示了链上群组的具体信息，这些信息包括：概览信息，区块信息，交易信息。



8.2 二、使用前提

8.2.1 2.1、群组搭建

区块链浏览器展示的数据是从区块链上同步下来的。为了同步数据需要初始化配置（添加群组信息和节点信息），故在同步数据展示前需要用户先搭建好区块链群组。FISCO BCOS 2.0+提供了多种便捷的群组搭建方式。

1. 如果是开发者进行开发调试，建议使用开发部署工具 build_chain。
2. 如果是开发企业级应用，建议使用企业部署工具运维部署工具 FISCO BCOS generator。

两者的主要区别在于build_chain为了使体验更好，搭建速度更快，辅助生成了群组内各个节点的私钥；但企业部署工具出于安全的考虑不辅助生成私钥，需要用户自己生成并设置。

8.3 三、区块链浏览器搭建

区块链浏览器分为两个部分：后台服务fisco-bcos-browser、前端web页面fisco-bcos-browser-front。

当前版本我们提供了两种搭建方式：一键搭建和手动搭建。

8.3.1 3.1.1、一键搭建

适合前后端同机部署，快速体验的情况使用。具体搭建流程参见安装文档。

8.3.2 3.1.2、手动搭建

后台服务搭建

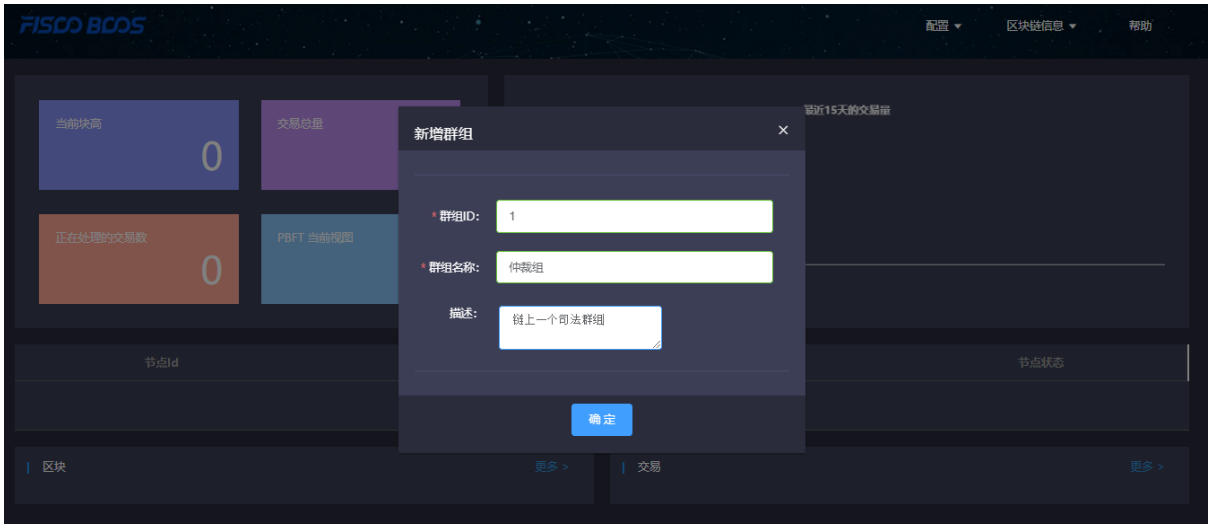
区块链浏览器后台服务使用Spring Boot的JAVA后台服务，具体搭建流程参见安装文档。

前端web页面服务搭建

区块链浏览器前端web页面使用框架`vue-cli`，具体搭建流程参见安装文档。

8.4 四、初始化环境

8.4.1 4.1、添加群组



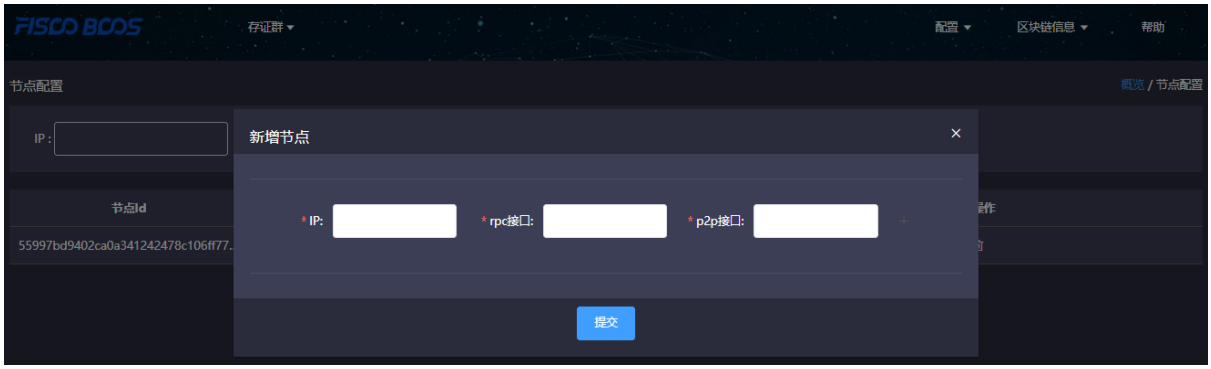
服务搭建成功后，可使用网页浏览器访问nginx配置的前端IP和前端端口，进入到浏览器页面。未初始化群组的浏览器页面会引导大家到新建群组配置页面，新建群组需要配置群组ID，群组名称，描述。

群组ID需要和区块链群组ID保持一致。 群组ID有多种查看方式，1、rpc接口获取。2、控制台命令。

群组名称是为群组ID取的一个有意义，便于理解的名字。

描述字段是对名称的进一步说明。

8.4.2 4.2、添加节点



添加群组所在的节点信息，用于区块链浏览器连接拉取相关展示信息。节点的rpc端口信息和p2p端口信息可以从节点的 **config.ini**配置文件中获取。

为了使用方便，新添加的群组会自动同步添加其他群组已经配置的共用节点信息。

8.4.3 4.3、添加合约

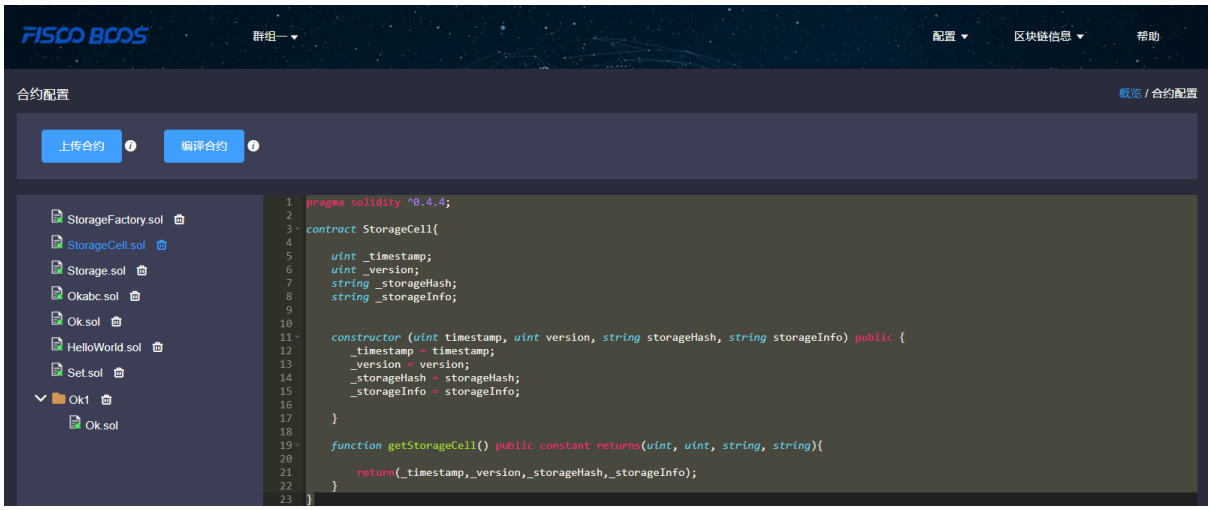
本浏览器版本提供合约解析的功能。此功能需要用户把本群组使用的所有合约进行导入。本版本支持用zip包上传一级目录，用于解决同名合约的问题。

导入步骤:

4.3.1 上传合约

1. 合约上传支持sol文件上传和将sol文件打包成zip包上传。
2. zip包最多支持一级目录，如果没有目录默认上传到根目录。zip包中只能有sol文件。

4.3.2 编译合约



8.5 五、功能介绍

8.5.1 5.1、概览

5.1.1 概览信息

主要包括当前群组的块高，交易总量，正在处理的交易数，PBFT视图。

5.1.2 最近15天的交易量

用折线图的形式展示了当前群组15内的交易情况。

5.1.3 节点概览

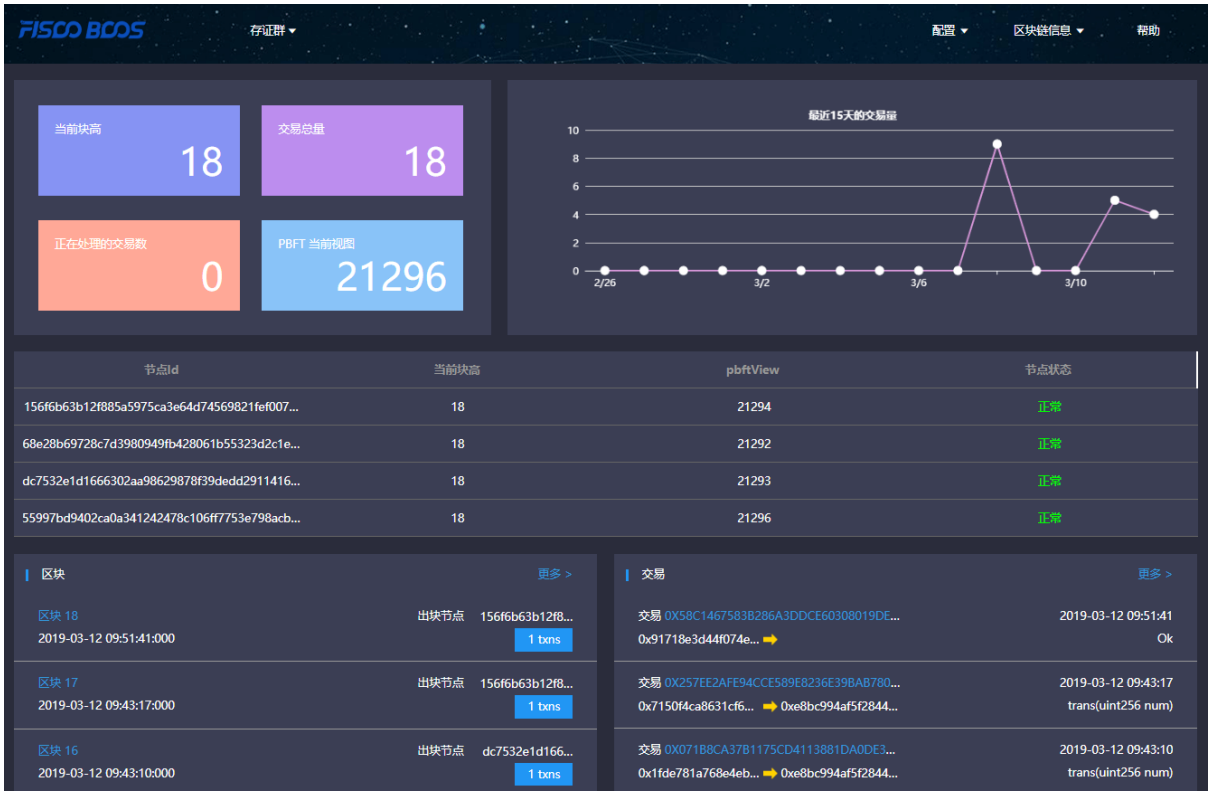
节点概览展示了当前群组内各个节点的ID，当前快高，pbftView，和节点状态。

5.1.4 区块概览

区块概览展示了最近4个区块的信息，包括每个区块的块高，出块者，块产生的时间及块上的交易总量。

5.1.5 交易概览

交易概览展示了最近四个交易，包括交易hash，交易时间，交易的发送者、交易的接收者，如果是正确导入了交易相关的合约还能展出交易调用的接口信息。



8.5.2 5.2、区块信息浏览

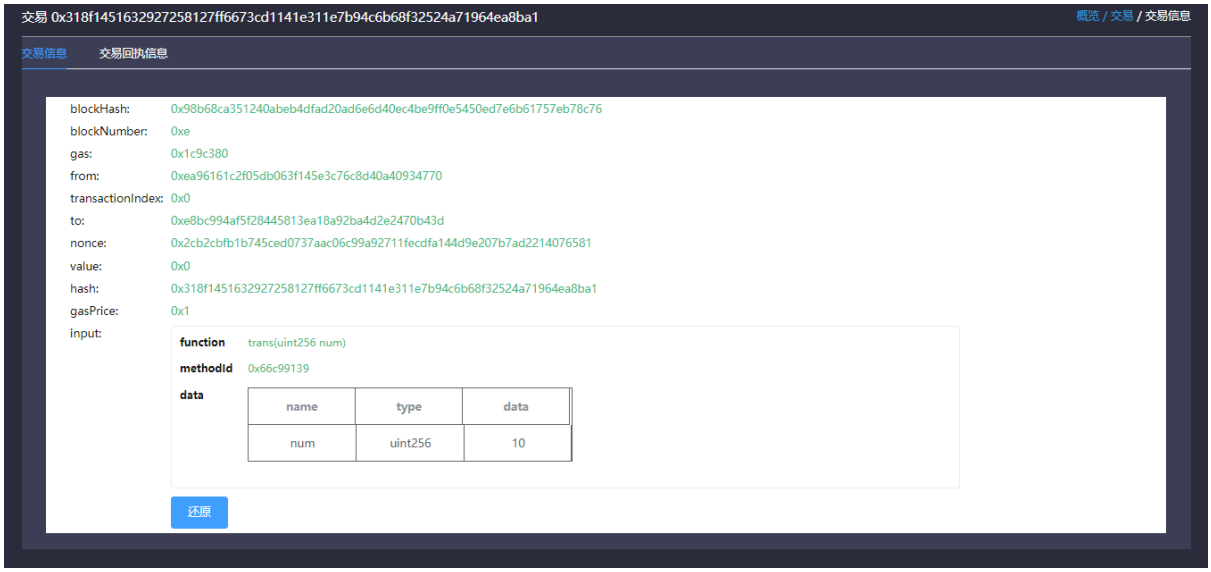
区块信息浏览主要包括区块列表页面和区块详情页面。

8.5.3 5.3、交易浏览

交易信息浏览主要包括交易列表页面和交易详情页面。

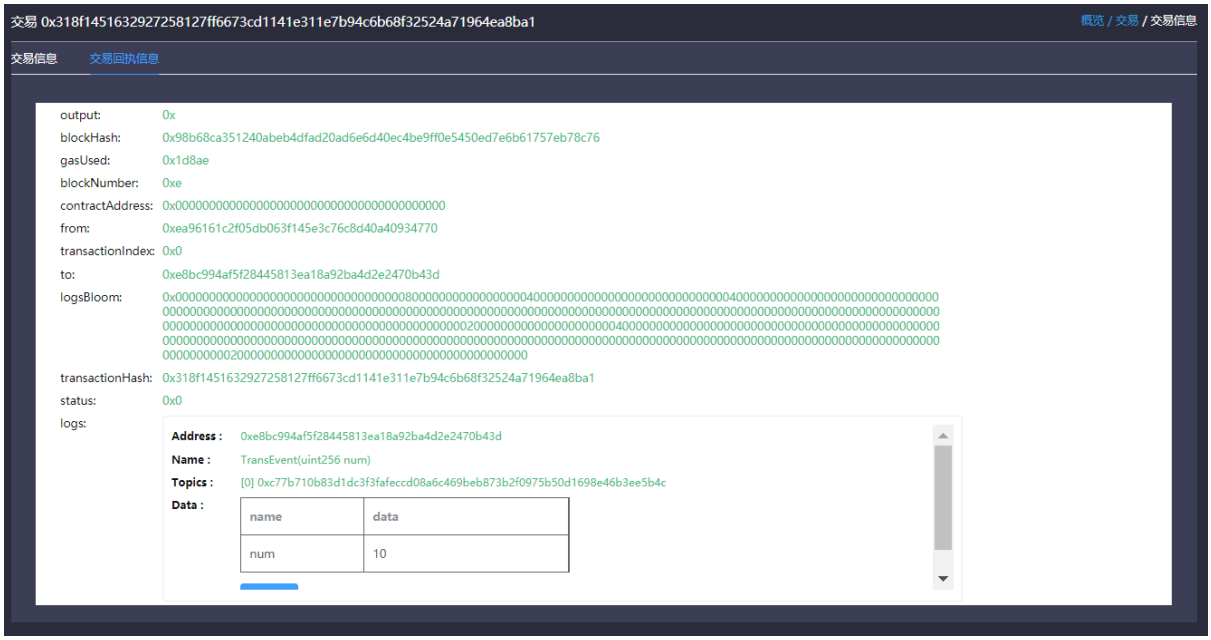
5.3.1、交易解析

合约成功上传并编译后，区块链浏览器能够解析出此合约相关交易的方法名和参数。浏览器的解析建立在合约的准确导入的基础上，故提醒用户在使用java和js等语言调用合约时，请**注意保存合约的正确版本。**



5.3.2、事件解析

合约成功上传并编译后，区块链浏览器能够解析出此合约相关交易回执中的事件方法名和参数。



CHAPTER 9

系统设计

本章介绍FISCO BCOS平台的设计思路，包括每个模块的结构以及实现，面向FISCO BCOS平台开发者。

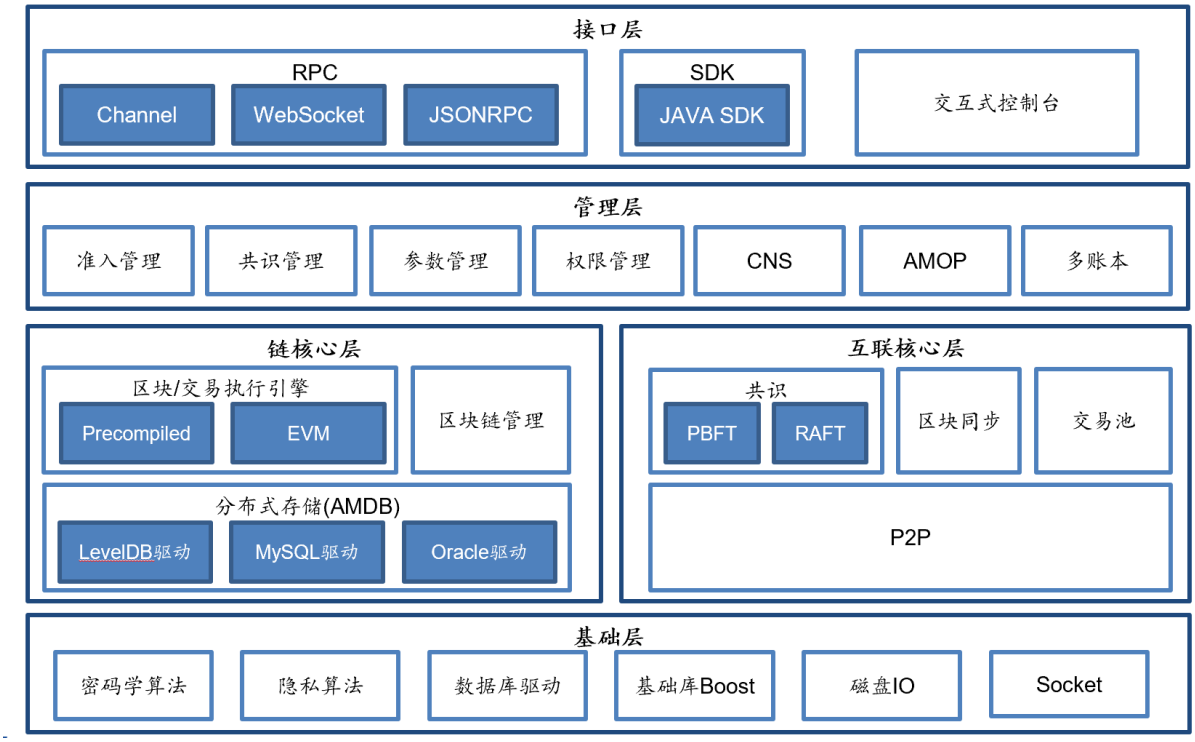
9.1 整体架构

整体架构上，FISCO BCOS划分成基础层、核心层、管理层和接口层：

- **基础层**:提供区块链的基础数据结构和算法库
- **核心层**: 实现了区块链的核心逻辑，核心层分为两大部分：
  1. 链核心层: 实现区块链的链式数据结构、交易执行引擎和存储驱动
  2. 互联核心层: 实现区块链的基础P2P网络通信、共识机制和区块同步机制
- **管理层**: 实现区块链的管理功能，包括参数配置、账本管理和AMOP
- **接口层**: 面向区块链用户，提供多种协议的RPC接口、SDK和交互式控制台

FISCO BCOS基于多群组架构实现了强扩展性的群组多账本，基于清晰的模块设计，构建了稳定、健壮的区块系统。

本章重点介绍FISCO BCOS的群组架构和系统运行时的交易流(包括交易提交、打包、执行和上链)。



9.1.1 群组架构

考虑到真实的业务场景需求，FISCO BCOS引入多群组架构，支持区块链节点启动多个群组，群组间交易处理、数据存储、区块共识相互隔离，保障区块链系统隐私性的同时，降低了系统的运维复杂度。

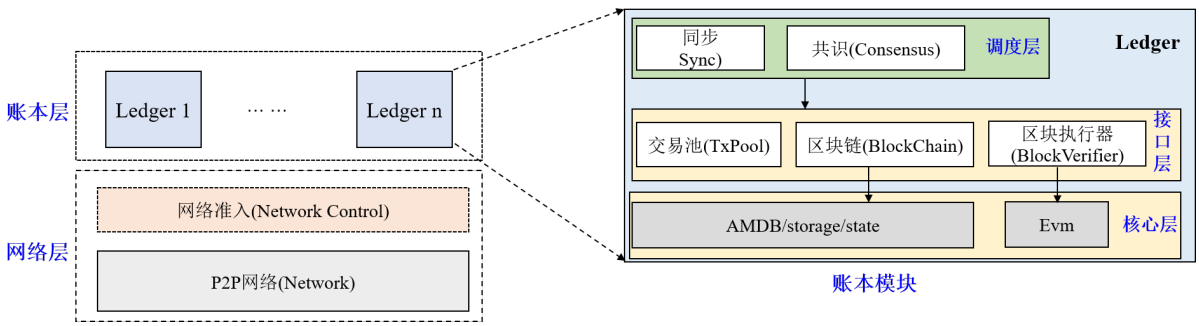
注解： 举个例子:

机构A、B、C所有节点构成一个区块链网络，运行业务1；一段时间后，机构A、B启动业务2，且不希望该业务相关数据、交易处理被机构C感知，有何解?

- **1.3系列FISCO BCOS系统**：机构A和机构B重新搭一条链运行业务2；运维管理员需要运维两条链，维护两套端口
- **FISCO BCOS 2.0+**：机构A和机构B新建一个群组运行业务2；运维管理员仅需维护一条链

显然在达到相同隐私保护需求基础上，FISCO BCOS 2.0+具有更好的扩展性、可运维性和灵活性。

多群组架构中，群组间共享网络，通过网络准入和账本白名单实现各账本间网络消息隔离。

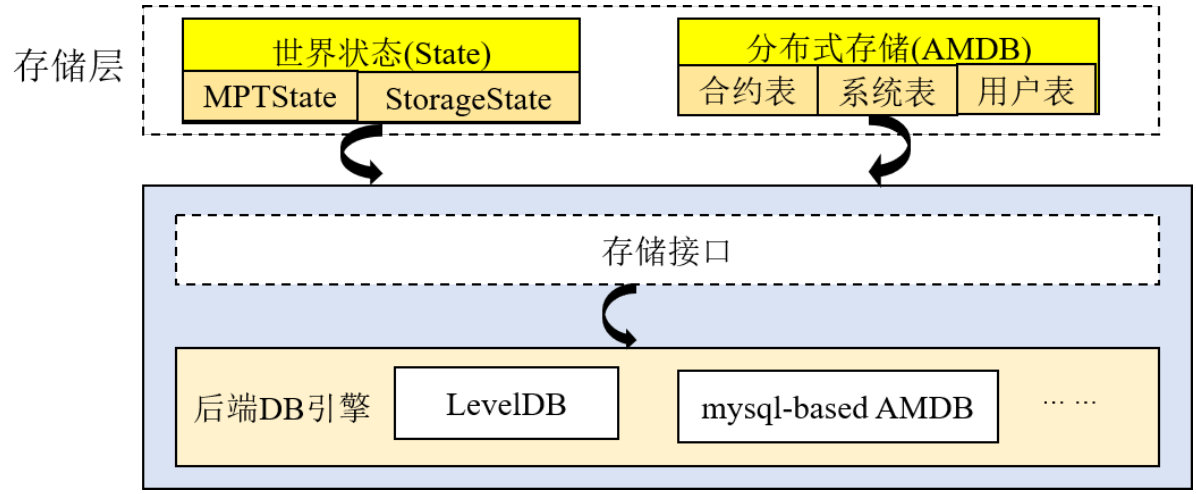


群组间数据隔离，每个群组独立运行各自的共识算法，不同群组可使用不同的共识算法。每个账本模块自底向上主要包括核心层、接口层和调度层三层，这三层相互协作，FISCO BCOS可保证单个群组独立健壮地运行。

核心层

核心层负责将群组的区块数据、区块信息、系统表以及区块执行结果写入底层数据库。

存储分为世界状态(State)和分布式存储(AMDB)两部分，世界状态包括MPTState和StorageState，负责存储交易执行的状态信息，StorageState性能高于MPTState，但不存储区块历史信息；AMDB则向外暴露简单的查询(select)、提交(commit)和更新(update)接口，负责操作合约表、系统表和用户表，具有可插拔特性，后端可支持多种数据库类型，目前支持RocksDB数据库和MySQLstorage。



接口层

接口层包括交易池(TxPool)、区块链(BlockChain)和区块执行器(BlockVerifier)三个模块。

- **交易池(TxPool)**: 与网络层以及调度层交互，负责缓存客户端或者其他节点广播的交易，调度层(主要是同步和共识模块)从交易池中取出交易进行广播或者区块打包；
- **区块链(BlockChain)**: 与核心层和调度层交互，是调度层访问底层存储的唯一入口，调度层(同步、共识模块)可通过区块链接口查询块高、获取指定区块、提交区块；
- **区块执行器(BlockVerifier)**: 与调度层交互，负责执行从调度层传入的区块，并将区块执行结果返回给调度层。

调度层

调度层包括共识模块(Consensus)和同步模块(Sync)。

- **共识模块**：包括Sealer线程和Engine线程，分别负责打包交易、执行共识流程。Sealer线程从交易池(TxPool)取交易，并打包成新区块；Engine线程执行共识流程，共识过程会执行区块，共识成功后，将区块以及区块执行结果提交到区块链(BlockChain)，区块链统一将这些信息写入底层存储，并触发交易池删除上链区块中包含的所有交易、将交易执行结果以回调的形式通知客户端，目前FISCO BCOS主要支持PBFT和Raft共识算法；
- **同步模块**：负责广播交易和获取最新区块， 考虑到共识过程中，leader负责打包区块，而leader随时有可能切换，因此必须保证客户端的交易尽可能发送到每个区块链节点，节点收到新交易后，同步模块将这些新交易广播给所有其他节点；考虑到区块链网络中机器性能不一致或者新节点加入都会导致部分节点区块高度落后于其他节点，同步模块提供了区块同步功能，该模块向其他节点发送自己节点的最新块高，其他节点发现块高落后于其他节点后，会主动下载最新区块。

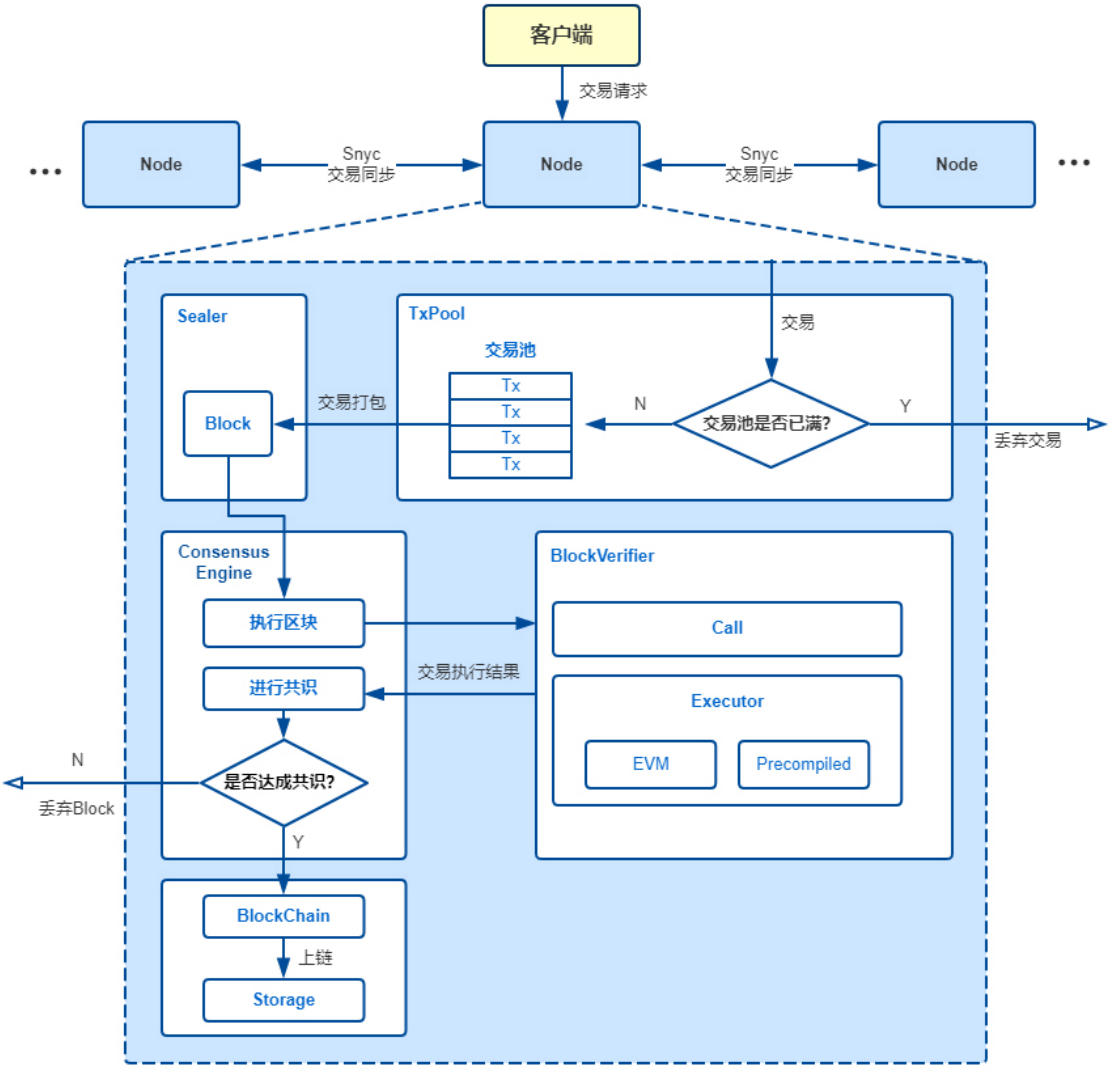
9.1.2 交易流

1 总体方案

用户通过SDK或curl命令向节点发起RPC请求以发起交易，节点收到交易后将交易附加到交易池中，打包器不断从交易池中取出交易并通过一定条件触发将取出交易打包为区块。生成区块后，由共识引擎进行验证及共识，验证区块无误且节点间达成共识后，将区块上链。当节点通过同步模块从其他节点处下载缺失的区块时，会同样对区块进行执行及验证。

2 整体架构

整体架构如下图所示：



Node：区块节点

TxPool：交易池，节点自身维护的、用于暂存收到的交易的内存区域

Sealer：打包器

Consensus Engine：共识引擎

BlockVerifier：区块验证器，用于验证一个区块的正确性

Executor：执行引擎，执行单个交易

BlockChain：区块链管理模块，是唯一有写权限的模块，提交区块接口需要同时传入区块数据和执行上下文数据，区块链管理将两种数据整合成一个事务提交到底层存储

Storage：底层存储

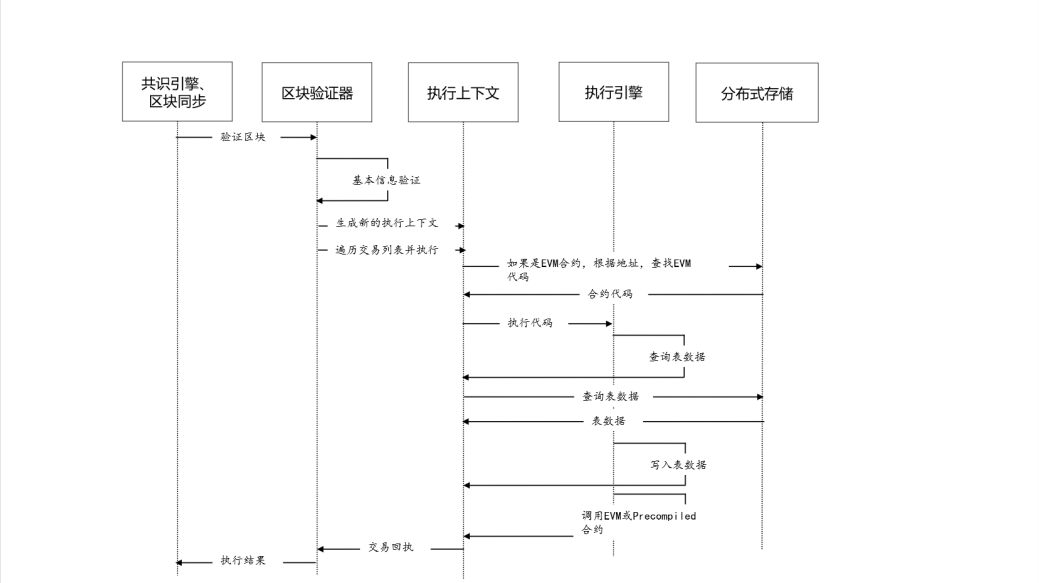
主要关系如下：

1. 用户通过操作SDK或直接编写curl命令向所连接的节点发起交易。
2. 节点收到交易后，若当前交易池未满则将交易附加至TxPool中并向自己所连的节点广播该交易；否则丢弃交易并输出告警。
3. Sealer会不断从交易池中取出交易，并立即将收集到的交易打包为区块并发送至共识引擎。
4. 共识引擎调用BlockVerifier对区块进行验证并在网络中进行共识，BlockVerifier调用Executor执行区块中的每笔交易。当区块验证无误且网络中节点达成一致后，共识引擎将区块发送至BlockChain。
5. BlockChain收到区块，对区块信息（如块高等）进行检查，并将区块数据与表数据写入底层存储中，完成区块上链。

3 方案流程

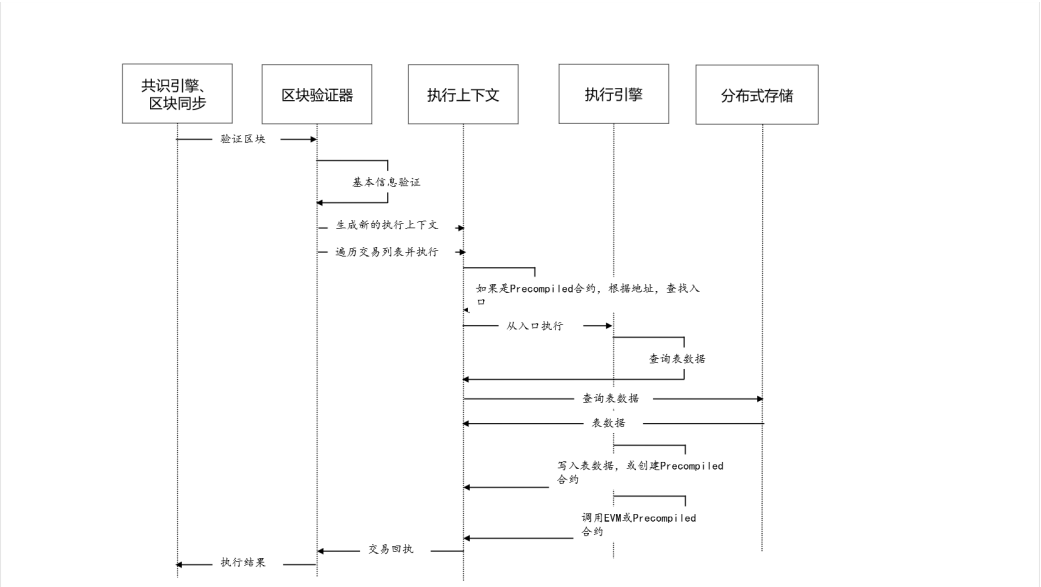
3.1 合约执行流程

执行引擎基于执行上下文（Executive Context）执行单个交易，其中执行上下文由区块验证器创建用于缓存暂存区块执行过程中执行引擎产生的所有数据，执行引擎同时支持EVM合约与预编译合约，其中EVM合约可以通过交易创建合约、合约创建合约两种方式来创建，其执行流程如下：



EVM合约创建后，保存到执行上下文的_sys_contracts_表中，EVM合约的地址在区块链全局状态内自增，从0x1000001开始（可定制），EVM合约执行过程中，Storage变量保存到执行上下文的c_(合约地址)表中。

预编译合约分永久和临时两种：(1) 永久预编译合约，整合在底层或插件中，合约地址固定；(2) 临时预编译合约，EVM合约或预编译合约执行时动态创建，合约地址在执行上下文内自增，从0x1000开始，至0x1000000截止，临时预编译合约仅在执行上下文内有效预编译合约没有Storage变量，只能操作表，其执行流程如下：



9.2 同步

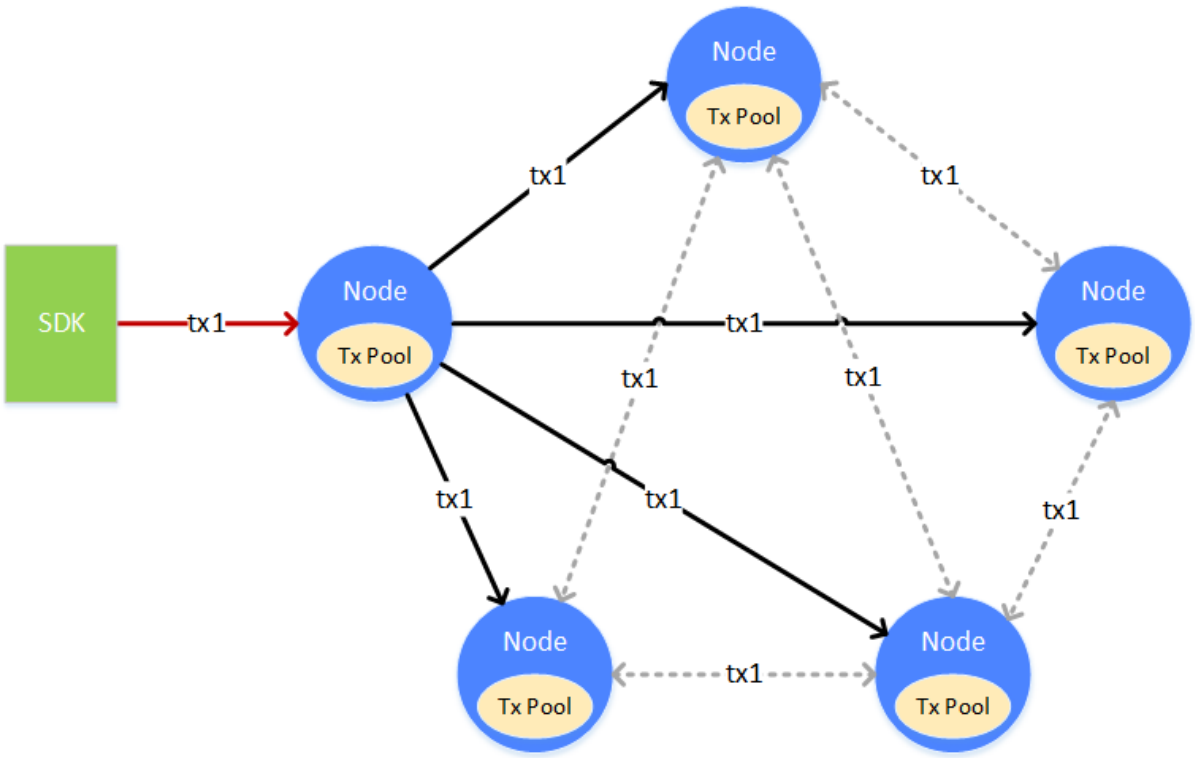
本节介绍节点间的同步机制以及同步优化策略。

9.2.1 同步基础流程

同步，是区块链节点非常重要的功能。它是共识的辅助，给共识提供必需的运行条件。同步分为交易的同步和状态的同步。交易的同步，确保了每笔交易能正确的到达每个节点上。状态的同步，能确保区块落后的节点能正确的回到最新的状态。只有持有最新区块状态的节点，才能参与到共识中去。

交易同步

交易同步，是让区块链的上的交易尽可能的到达所有的节点。为共识中将交易打包成区块提供基础。



一笔交易（tx1），从客户端上发往某个节点，节点在接收到交易后，会将交易放入自身的交易池（Tx Pool）中供共识去打包。与此同时，节点会将交易广播给其它的节点，其它节点收到交易后，也会将交易放到自身的交易池中。交易在发送的过程中，会有丢失的情况，为了能让交易尽可能的到达所有的节点，收到广播过来交易的节点，会根据一定的策略，选择其它的节点，再进行一次广播。

交易广播策略

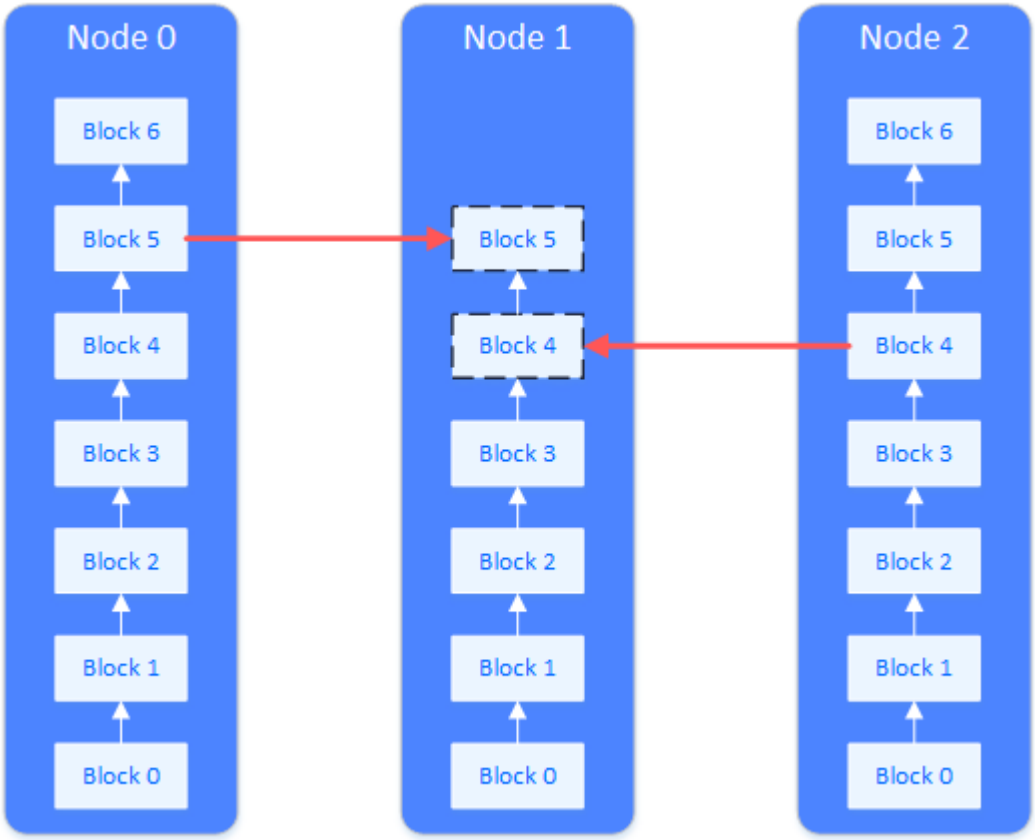
如果每个节点都没有限制的转发/广播收到的交易，带宽将被占满，出现交易广播雪崩的问题。为了避免交易广播的雪崩，FISCO BCOS根据经验，选择了较为精巧的交易广播策略。在尽可能保证交易可达性的前提下，尽量的减少重复的交易广播。

- 对于SDK来的交易，广播给所有的节点
- 对于其它节点广播来的交易，随机选择25%的节点再次广播
- 一条交易在一个节点上，只广播一次，当收到了重复的交易，不会进行二次广播

通过上述的策略，能够尽量的让交易到达所有的节点，但也会在极小的概率下出现某交易无法到达某节点的情况。此情况是允许的。交易尽可能到达更多的节点，是为了让此交易尽快的被打包、共识、确认，尽量的让交易能够更快的得到执行的结果。当交易未到达某个节点时，只会使得交易的执行时间变长，不会影响交易的正确性。

状态同步

状态同步，是让区块链节点的状态保持在最新。区块链的状态的新旧，是指区块链节点当前持有数据的新旧，即节点持有的当前区块块高的高低。若一个节点的块高是区块链的最高块高，则此节点就拥有区块链的最新状态。只有拥有最新状态的节点，才能参与到共识中去，进行下一个新区块的共识。

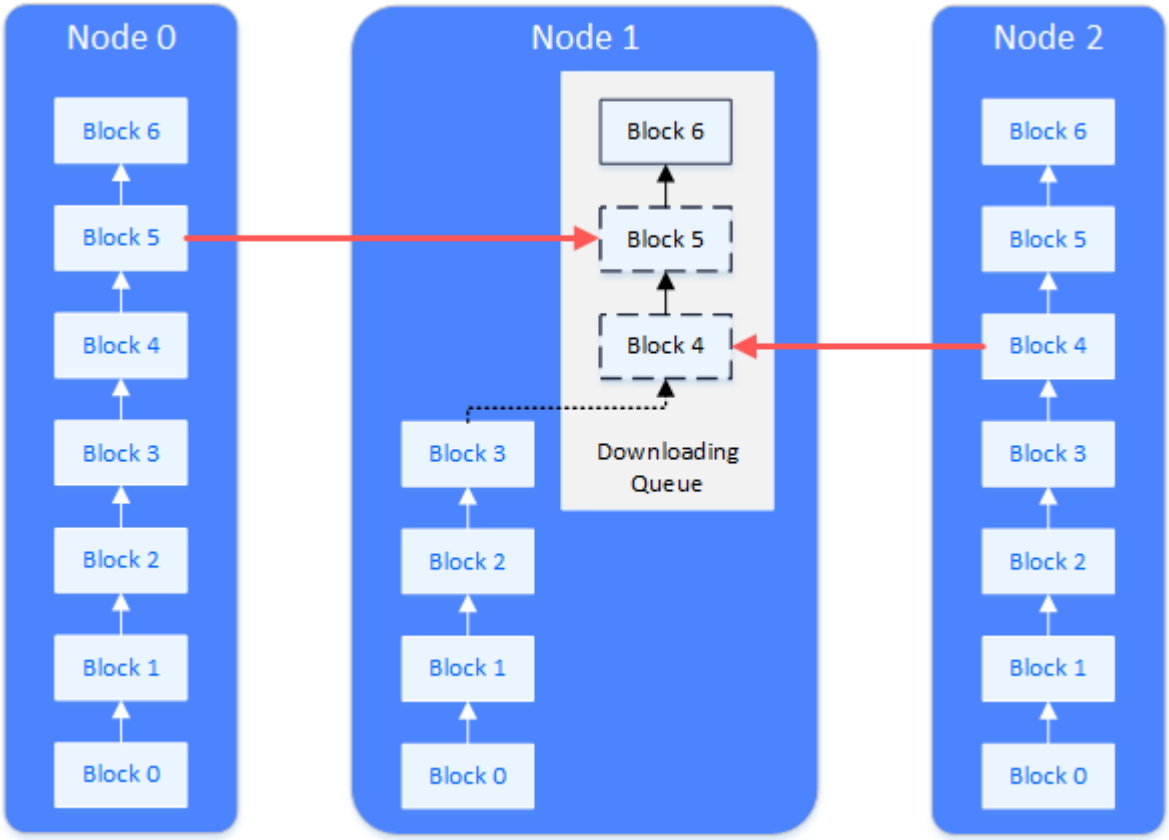


在一个全新的节点加入到区块链上，或一个已经断网的节点恢复了网络时，此节点的区块落后于其它节点，状态不是最新的。此时就需要进行状态同步。如图，需要状态同步的节点（Node 1），会主动向其它节点请求下载区块。整个下载的过程会将下载的负载分散到多个节点上。

状态同步与下载队列

区块链节点在运行时，会定时向其它节点广播自身的最高块高。节点收到其它节点广播过来的块高后，会和自身的块高进行比较，若自身的块高落后于此块高，就会启动区块下载流程。

区块的下载通过请求的方式完成。进入下载流程的节点，会随机的挑选满足要求的节点，发送需要下载的区块区间。收到下载请求的节点，会根据请求的内容，回复相应的区块。



收到回复区块的节点，在本地维护一个下载队列，用来对下载下来的区块进行缓冲和排序。下载队列是一个以块高为顺序的优先队列。下载下来的区块，会不断的插入到下载队列中，当队列中的区块能连接上节点当前本地的区块链，则将区块从下载队列中取出，真正的连接到当前本地的区块链上。

同步场景举例

交易同步

一笔交易被广播到所有节点的过程：

1. 一笔交易通过channel或RPC发送到某节点上
2. 收到交易的节点全量广播此交易给其它节点
3. 其它节点收到交易后，为了保险起见，选择25%的节点再广播一次
4. 节点收到广播过的交易，不会再次广播

状态同步

节点出块时的广播逻辑

1. 某个节点出块
2. 此节点将自己最新的状态（最新块高，最高块哈希，创世块哈希）广播给所有的节点
3. 其它的节点收到peer的状态后，更新在本地管理的peer数据

组内成员的同步

组内成员在某时刻意外关闭，但其它成员在出块，当此组员再次启动时，发现区块落后于其它组员：

1. 组员再次启动

2. 收到其它组员发来的状态包
3. 比较发现自己的最高块高落后于其它组员，启动下载流程
4. 将相差的区块按区间划分成多个下载请求包，发送给多个组员，负载均衡
5. 等待其它节点回复区块包
6. 其它节点接受响应，从自己的区块链上查询出区块，回复给启动的节点
7. 节点收到区块，放入下载队列
8. 节点从下载队列中将区块拿出，写到区块链上
9. 若下载未结束，则继续请求，若下载结束，则切换自身状态，开启交易同步，开启共识

新组员的同步

非组员作为一个新组员加入到某个组中，且此节点第一次启动，从原来的组员中同步区块：

1. 非组员未被注册到组中，但非组员先启动
2. 此时发现自己不在组中，不进行状态广播，也不进行交易广播，只等待其它组员发来状态消息
3. 此时组员中并没有此新组员，不会向新组员广播状态
4. 管理员将新组员加入到组中
5. 组员向新组员广播自身状态
6. 新组员收到组员状态，比较自身块高（为0），启动下载流程
7. 之后的下载流程，与组内成员区块同步流程相同

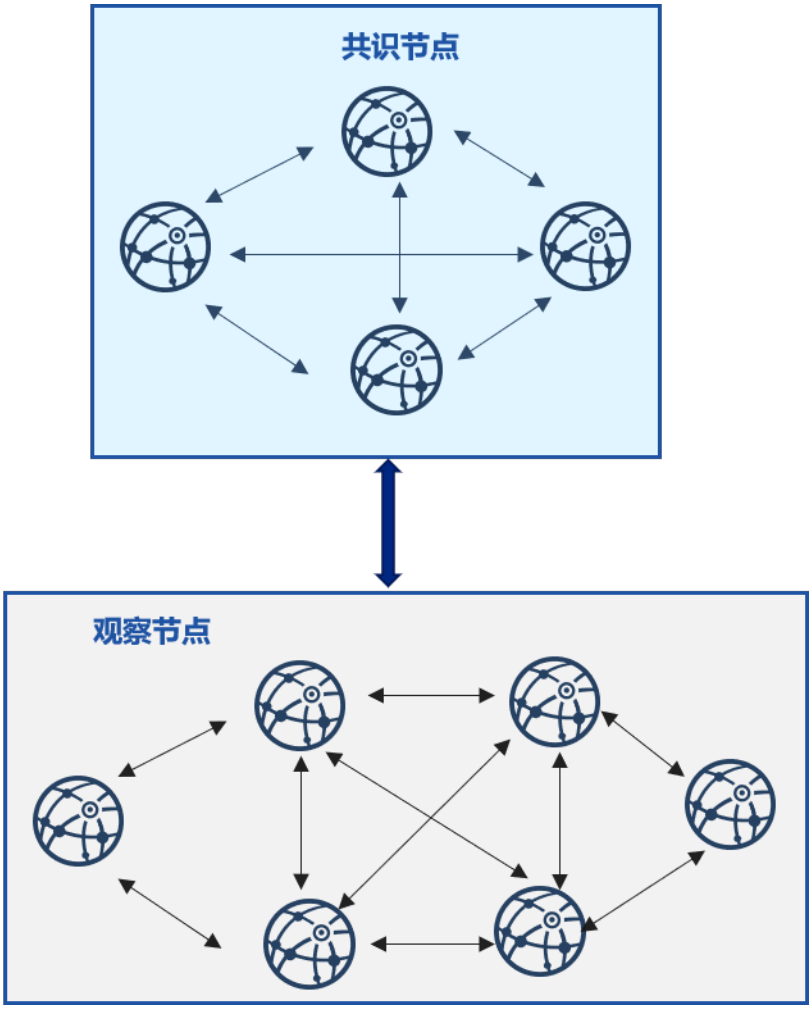
9.2.2 区块同步优化

为了增强区块链系统在网络带宽受限情况下的可扩展性，FISCO BCOS v2.2.0对区块同步进行了优化：

- 为了降低单个节点的出带宽，消除网络带宽对网络规模的限制，支持更大网络规模，采用树状拓扑进行区块同步
- 采用gossip协议来保障树状拓扑区块同步的健壮性，定期同步区块状态，使得在部分节点网络断连的情况下，所有节点均能同步到最新区块状态

背景

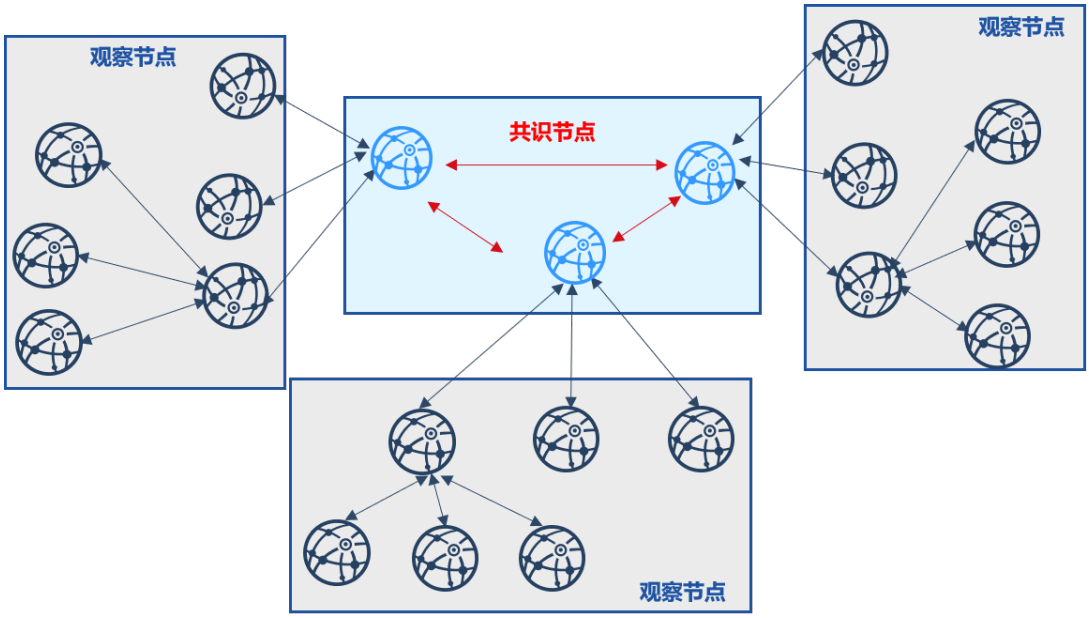
考虑到目前使用BFT类共识算法的区块链网络复杂度较高、不具有无限可扩展性，因此大部分业务架构仅有部分节点作为共识节点，其他节点均作为观察节点(不参与共识，但拥有区块链全量数据)，如下图所示。



在这种架构中，大部分观察节点均随机从拥有最新区块的共识节点同步区块，在包含n个共识节点、m个观察节点的区块链系统中，设每个区块大小为block_size，理想情况下(即负载均衡)，每共识一个区块，每个共识节点需要向m/n个观察节点发送区块，共识节点出带宽大约是(m/n)*block_size，设网络带宽是bandwidth，则每个共识节点最多可向(bandwidth/block_size)个节点同步区块，即区块链网络规模最大是(n*bandwidth/block_size)，在公网带宽bandwidth较小，区块较大的情况下，能容纳的节点数有限，因此随机的区块同步策略不具有可扩展性。

区块状态树状广播

为降低多个观察节点向单个共识节点同步区块时，共识节点的网络出带宽对网络规模的影响，FISCO BCOS v2.2.0实现了区块状态树状广播策略。下图是由3个共识节点、18个观察节点构成的区块链系统沿三叉树进行区块同步的示意图：



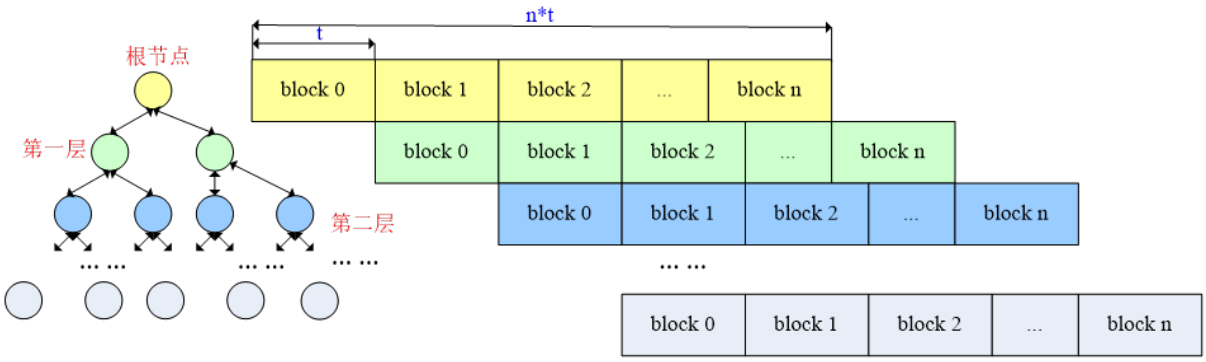
该策略将观察节点分摊给每个共识节点，并以共识节点为顶点构造一颗三叉树，共识节点出块后，优先向其子观察节点发送最新区块状态，子观察节点同步最新区块后，优先向自己的子节点发送最新区块状态，以此类推。采用了区块状态树状广播策略后，每个节点仅将最新区块状态发送给子节点，设区块大小为block_size，树的宽度为w，则用于区块同步的网络带宽均为(block_size * w)，与区块链系统的节点总数无关，具有可扩展性。上图所示的共识节点采用区块状态树状广播后，出带宽降低了2倍。

区块状态树状广播工作流程如下：

- 共识节点共识提交新区块block i后，若其与子节点连通，则向其子节点同步最新区块状态，包括高度和区块哈希，记为{i, block_hash(i)}；否则递归判断是否与孙子节点连通，若连通，则向孙子节点同步最新区块状态；
- 子节点收到共识节点的区块状态后，判断接收到的区块状态{i, block_hash(i)}比自身区块状态新，则向共识节点发送区块请求，共识节点收到请求后，向该节点发送对应的区块；
- 子节点收到共识节点的区块后，验证成功后将其落盘，继续向自己的子节点发送自身的区块状态，同样，若该节点与自己的子节点断连，会递归判断是否与孙子节点连通，并向连通的孙子节点发送最新区块状态；
- 收到新区块状态的子节点，重复步骤(2)，进行区块同步。

当然，使用区块状态树状广播策略时，由于区块并非由拥有最新区块的根节点直接下发到所有观察者节点，作为叶子节点的观察者节点同步区块的时延会相对长一些。

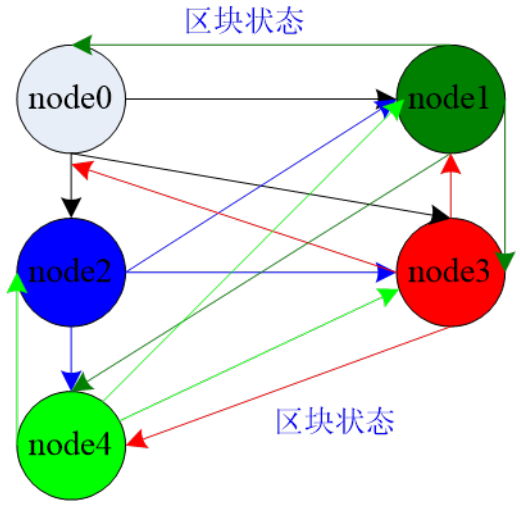
下图展示了各层节点提交n个区块的时延，设树的深度为d，每个区块提交时延为t，则根节点(共识节点)提交n个区块的时延为n*t，第一层节点(观察者节点)同步并提交区块的时延为n*t + t，第二层节点(观察者节点)同步并提交区块的时延为n*t + 2*t，叶子节点同步并提交区块的时延为n*t + d * t，与共识节点的时延差为d*t，n远大于d时，这个时延几乎可以忽略，因此该策略对观察者节点TPS的影响非常小。



定期同步区块状态

考虑到若树状拓扑中部分节点断连，可能会导致区块无法到达部分节点，区块状态树状广播优化策略还采用了gossip协议定期同步区块状态。

即：随机挑选若干个节点，同步最新区块状态信息。由于最终区块状态信息会收敛所有区块链节点，树状拓扑中断连节点也能从其邻居节点同步最新区块，保证了树状区块状态广播的健壮性。



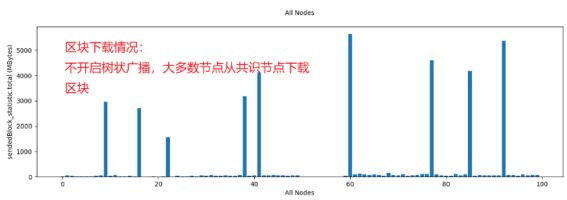
上图展示了各个节点如何使用gossip协议定期同步区块状态：

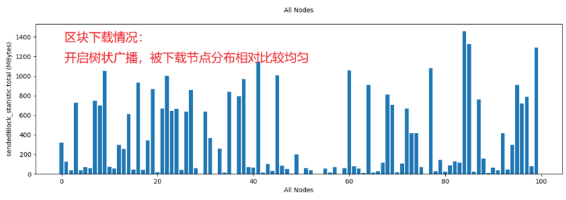
- 各个区块链节点每2s随机选择三个邻居节点广播当前区块状态，包括{区块高度，区块哈希}
- 节点收到这些区块状态包后，更新本地缓存的各个节点区块状态到最新
- 若某节点区块高度高于本节点区块高度，该节点会向拥有更高区块的节点同步区块

由于区块链节点之间定期同步区块状态，即使树状拓扑中部分节点断连，也可以保证每个节点同步到尽可能多的节点区块状态，并从拥有最高区块的节点下载最新区块，保障了树状区块状态广播可扩展性的同时，增强了整个系统的健壮性。

带宽对比

下图是采用了区块状态树状广播、定期同步区块状态策略后，区块同步优化效果：

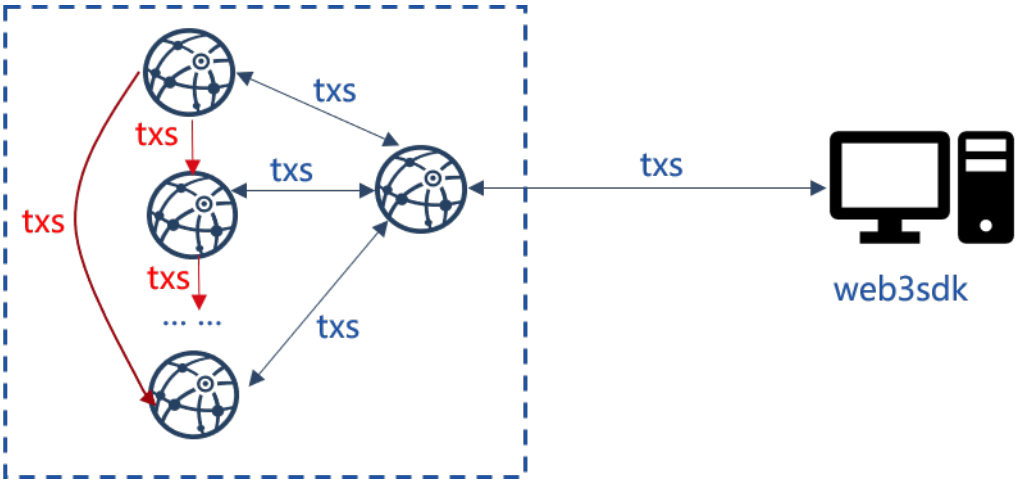




整个区块链网络中包含10个共识节点，90个观察者节点，树的度设置为2。优化前，观察者节点主要从10个共识节点下载区块，共识节点的出流量可达到5000MB；优化后，部分下载流量分摊到了观察者节点，节点由区块下载带来的流量开销降低到了1400MB，降低了3倍多，基本接近最优(最优的情况是优化前峰值出带宽是优化后峰值出带宽的4.5倍，由于gossip协议导致的区块随机拉取情况的存在，无法达到最优，只能接近最优)。

9.2.3 交易同步优化

区块链系统中，为了保障客户端发送的交易能到达所有节点，SDK直连的区块链节点需要将收到的交易广播给其他节点，如下图所示：



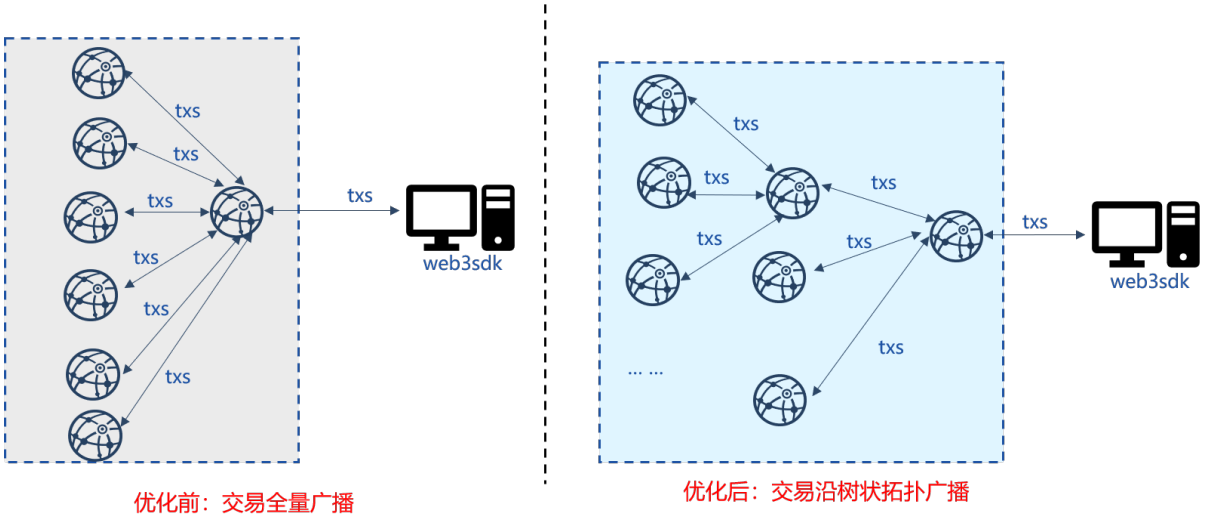
显然，SDK直连节点的出带宽与区块链节点总数成正比，随着区块链系统节点数的增加，该节点必然成为整个系统的瓶颈。

此外，为了保障节点网络断连的情况下，交易也能尽量到达所有节点，还引入了交易转发逻辑，节点收到其他节点广播过来的交易后，会随机选取25%的邻居节点转发收到的交易，在网络全连的情况下，这种交易转发策略无疑会带来巨大的带宽浪费，且节点数目越多，因交易转发带来的数据包冗余越多。

为降低SDK直连节点的出带宽、降低交易转发引起的大量冗余消息包，提升区块链系统的可扩展性，FISCO BCOS v2.2.0提出了**交易广播优化策略**和**交易转发优化策略**。

交易广播优化策略

为了降低SDK直连节点交易广播带来的网络压力，FISCO BCOS v2.2.0中，SDK直连节点收到交易后，沿着树状拓扑广播交易(树的宽度默认为3)。下图展示了优化前后7节点区块链系统交易广播拓扑：

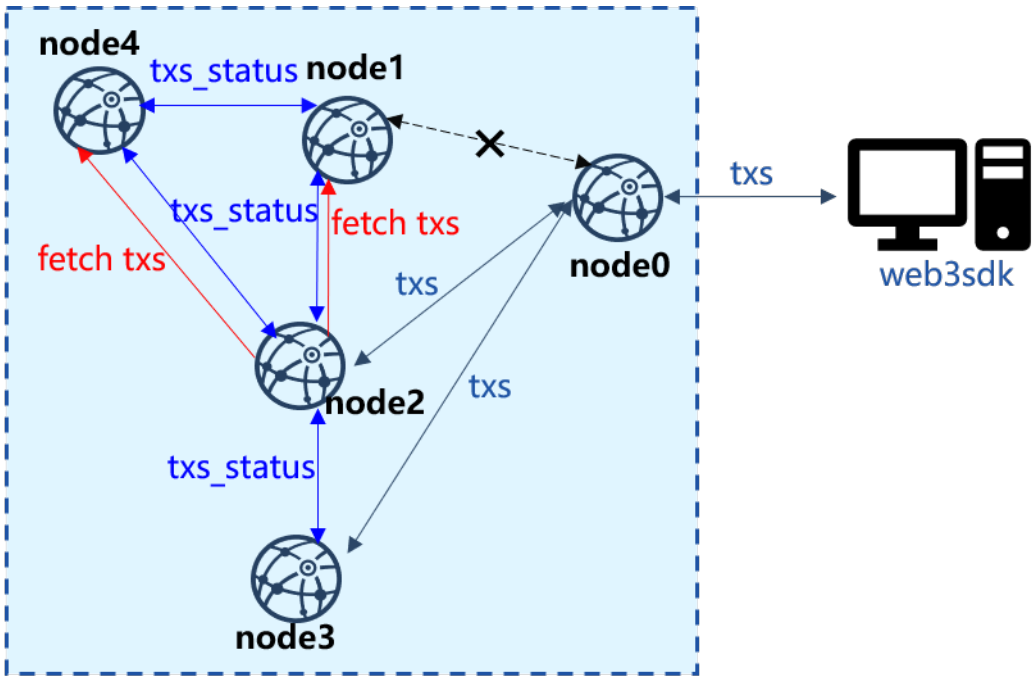


- 优化前：节点收到SDK的交易后，全量广播给其他节点；
- 优化后：节点收到SDK的交易后，将其发送给子节点，子节点收到交易后，继续将其发送给自身的子节点。

采用交易树状广播后，上图所示的7节点区块链系统，SDK直连节点的带宽降低为原先的一半，且由于SDK直连节点以及其他节点广播交易的出带宽仅与树状拓扑的宽度有关，因此优化后的交易同步具有可扩展性。

交易转发优化策略

交易转发对于交易同步尤为重要，可以包含部分节点网络断连情况下，SDK发出的交易能尽量到达所有节点。但正如前面提到的，已有的交易转发策略会带来大量的带宽冗余，因此在交易树状广播的基础上，FISCO BCOS v2.2.0提出了交易转发优化策略，如下图所示，优化后的交易转发策略不直接转发交易，仅转发交易状态，节点根据其他节点的交易状态，获取缺失的交易，然后直接向对应节点请求交易。



上图中，SDK直连node0，但是node0与node1断连，此时node0仅能将交易广播给node2和node3。node2和node3收到交易后，将最新交易的列表打包成状态包发送给其他节点，node1和node4收到状态包后，与本地交易池内的交易列表做对比，获取缺失的交易列表，并批量向拥有这些交易的node2或node3请求这些交易。

交易转发具体流程如下：

- 节点收到新交易txs后，获取所有新交易的哈希，记为txs_hash_list，并将其打包成状态包，随机发送给25%的节点；
- 节点node_x收到某节点node_i交易状态包后，从中解出交易哈希列表txs_hash_list，并将其与本地交易池中的交易列表做对比，获取缺失的交易列表，记为missed_txs_hash_list，将其打包成交易请求，向node_i发出交易请求；
- node_i接收到交易请求后，从交易池中取出missed_txs_hash_list对应的所有交易，回复给node_x。

由于在全连的网络拓扑中，所有节点交易状态基本一致，因此节点间交易请求较少，相较于直接转发交易，大大降低了转发冗余交易引起的带宽浪费。

9.3 共识算法

区块链系统通过共识算法保障系统一致性。理论上，共识是对某个提案(proposal)达成一致意见的过程，分布式系统中提案的含义十分宽泛，包括事件发生顺序、谁是leader等。区块链系统中，共识是各个共识节点对交易执行结果达成一致的过程。

共识算法分类

根据是否容忍 拜占庭错误 ，共识算法可分为容错(Crash Fault Tolerance, CFT)类算法和拜占庭容错(Byzantine Fault Tolerance, BFT)类算法：

- **CFT类算法**：普通容错类算法，当系统出现网络、磁盘故障，服务器宕机等普通故障时，仍能针对某个提议达成共识，经典的算法包括Paxos、Raft等，这类算法性能较好、处理速度较快、可以容忍不超过一半的故障节点；
- **BFT类算法**：拜占庭容错类算法，除了容忍系统共识过程中出现的普通故障外，还可容忍部分节点故意欺骗(如伪造交易执行结果)等拜占庭错误，经典算法包括PBFT等，这类算法性能较差，能容忍不超过三分之一的故障节点。

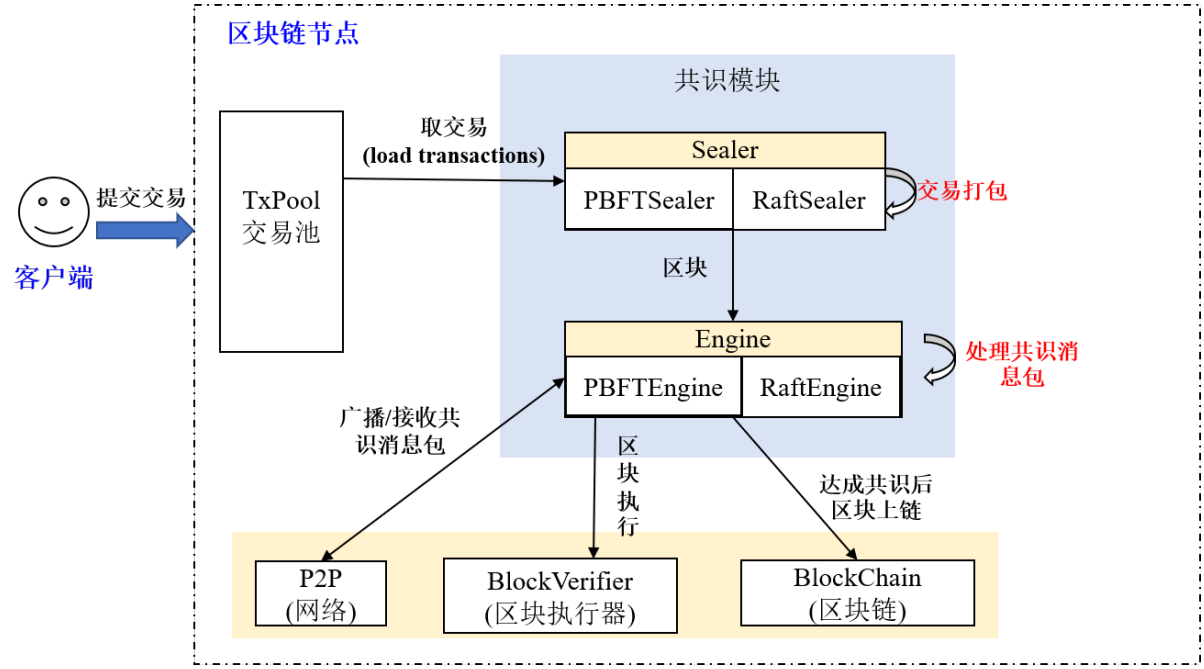
FISCO BCOS共识算法

FISCO BCOS基于多群组架构实现了插件化的共识算法，不同群组可运行不同的共识算法，组与组之间的共识过程互不影响，FISCO BCOS目前支持PBFT(Practical Byzantine Fault Tolerance)和Raft(Replication and Fault Tolerant)两种共识算法：

- **PBFT共识算法**: BFT类算法，可容忍不超过三分之一的故障节点和作恶节点，可达到最终一致性；
- **Raft共识算法**: CFT类算法, 可容忍一半故障节点，不能防止节点作恶，可达到一致性。

9.3.1 框架

FISCO BCOS实现了一套可扩展的共识框架，可插件化扩展不同的共识算法，目前支持 **PBFT(Practical Byzantine Fault Tolerance)** 和 **Raft(Replication and Fault Tolerant)** 共识算法，共识模块框架如下图：



Sealer线程

交易打包线程，负责从交易池取交易，并基于节点最高块打包交易，产生新区块，产生的新区块交给Engine线程处理，PBFT和Raft的交易打包线程分别为PBFTSealer和RaftSealer。

Engine线程

共识线程，负责从本地或通过网络接收新区块，并根据接收的共识消息包完成共识流程，最终将达成共识的新区块写入区块链(BlockChain)，区块上链后，从交易池中删除已经上链的交易，PBFT和Raft的共识线程分别为PBFTEngine和RaftEngine。

9.3.2 PBFT基础流程

PBFT(Practical Byzantine Fault Tolerance)共识算法可以在少数节点作恶(如伪造消息)场景中达成共识，它采用签名、签名验证、哈希等密码学算法确保消息传递过程中的防篡改性、防伪造性、不可抵赖性，并优化了前人工作，将拜占庭容错算法复杂度从指数级降低到多项式级别，在一个由(3*f+1)个节点构成的系统中，只要有不少于(2*f+1)个非恶意节点正常工作，该系统就能达成一致性，如：7个节点的系统中允许2个节点出现拜占庭错误。

FISCO BCOS区块链系统实现了PBFT共识算法。

1. 重要概念

节点类型、节点ID、节点索引和视图是PBFT共识算法的关键概念。区块链系统基本概念请参考关键概念。

1.1 节点类型

- **Leader/Primary**: 共识节点，负责将交易打包成区块和区块共识，每轮共识过程中有且仅有一个leader，为了防止leader伪造区块，每轮PBFT共识后，均会切换leader;
- **Replica**: 副本节点，负责区块共识，每轮共识过程中有多个Replica节点，每个Replica节点的处理过程类似;
- **Observer**: 观察者节点，负责从共识节点或副本节点获取最新区块，执行并验证区块执行结果后，将产生的区块上链。

其中Leader和Replica统称为共识节点。

1.2 节点ID && 节点索引

为了防止节点作恶，PBFT共识过程中每个共识节点均对其发送的消息进行签名，对收到的消息包进行验签名，因此每个节点均维护一份公私钥对，私钥用于对发送的消息进行签名，公钥作为节点ID，用于标识和验签。

> **节点ID** : 共识节点签名公钥和共识节点唯一标识, 一般是64字节二进制串，其他节点使用消息包发送者的节点ID对消息包进行验签

考虑到节点ID很长，在共识消息中包含该字段会耗费部分网络带宽，FISCO BCOS引入了节点索引，每个共识节点维护一份公共的共识节点列表，节点索引记录了每个共识节点ID在这个列表中的位置，发送网络消息包时，只需要带上节点索引，其他节点即可以从公共的共识节点列表中索引出节点的ID，进而对消息进行验签:

> **节点索引** : 每个共识节点ID在这个公共节点ID列表中的位置

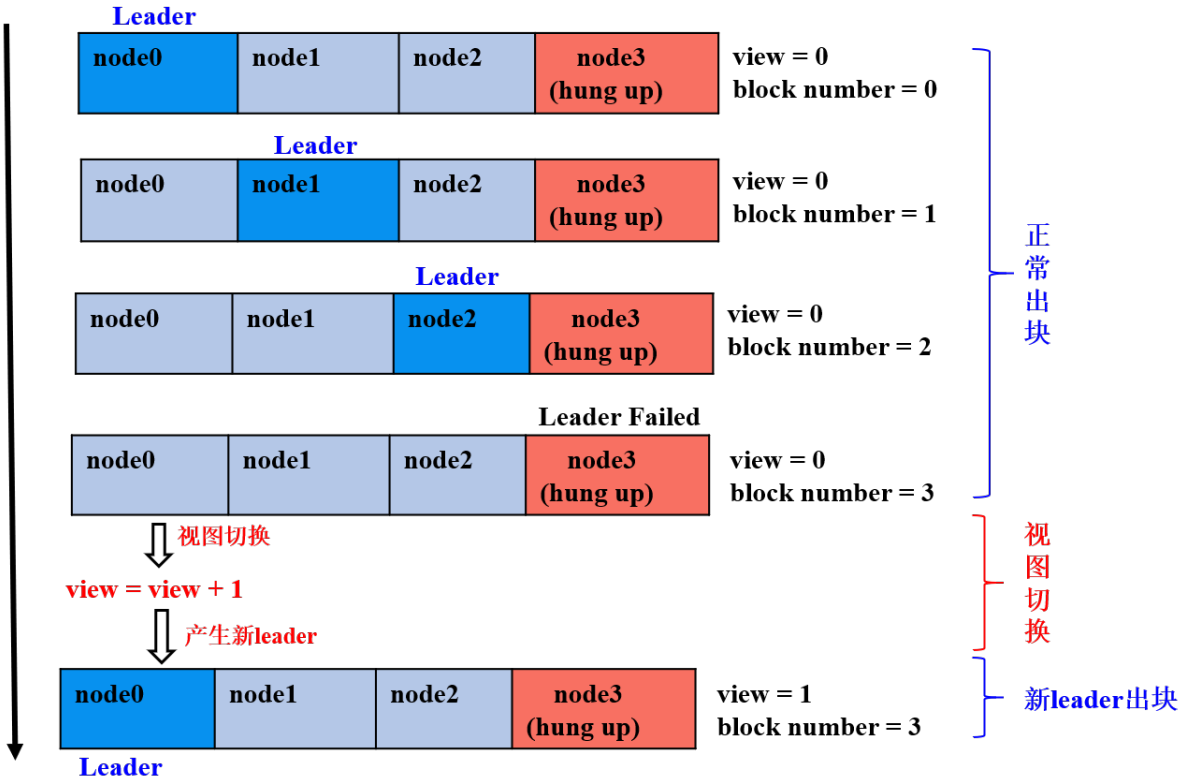
1.3 视图(view)

PBFT共识算法使用视图view记录每个节点的共识状态，相同视图节点维护相同的Leader和Replicas节点列表。当Leader出现故障，会发生视图切换，若视图切换成功(至少2*f+1个节点达到相同视图)，则根据新的视图选出新leader，新leader开始出块，否则继续进行视图切换，直至全网大部分节点(大于等于2*f+1)达到一致视图。

FISCO BCOS系统中，leader索引的计算公式如下：

```
leader_idx = (view + block_number) % node_num
```

下图简单展示了4(3*f+1, f=1)节点FISCO BCOS系统中，第三个节点(node3)为拜占庭节点情况下，视图切换过程：



- 前三轮共识： node0、node1、node2为leader，且非恶意节点数目等于2*f+1，节点正常出块共识；
- 第四轮共识：node3为leader，但node3为拜占庭节点，node0-node2在给定时间内未收到node3打包的区块，触发视图切换，试图切换到view_new=view+1的新视图，并相互间广播viewchange包，节点收集满在视图view_new上的(2*f+1)个viewchange包后，将自己的view切换为view_new，并计算出新leader；
- 为第五轮共识：node0为leader，继续打包出块。

1.4 共识消息

PBFT模块主要包括**PrepareReq**、**SignReq**、**CommitReq**和**ViewChangeReq**四种共识消息：

- **PrepareReqPacket**: 包含区块的请求包，由leader产生并向所有Replica节点广播，Replica节点收到Prepare包后，验证PrepareReq签名、执行区块并缓存区块执行结果，达到防止拜占庭节点作恶、保证区块执行结果的最终确定性的目的；
- **SignReqPacket**: 带有区块执行结果的签名请求，由收到Prepare包并执行完区块的共识节点产生，SignReq请求带有执行后区块的hash以及该hash的签名，分别记为SignReq.block_hash和SignReq.sig，节点将SignReq广播到所有其他共识节点后，其他节点对SignReq(即区块执行结果)进行共识；
- **CommitReqPacket**: 用于确认区块执行结果的提交请求，由收集满(2*f+1)个block_hash相同且来自不同节点SignReq请求的节点产生，CommitReq被广播给所有其他共识节点，其他节点收集满(2*f+1)个block_hash相同、来自不同节点的CommitReq后，将本地节点缓存的最新区块上链；
- **ViewChangeReqPacket**: 视图切换请求，当leader无法提供正常服务(如网络连接不正常、服务器宕机等)时，其他共识节点会主动触发视图切换，ViewChangeReq中带有该节点即将切换到的

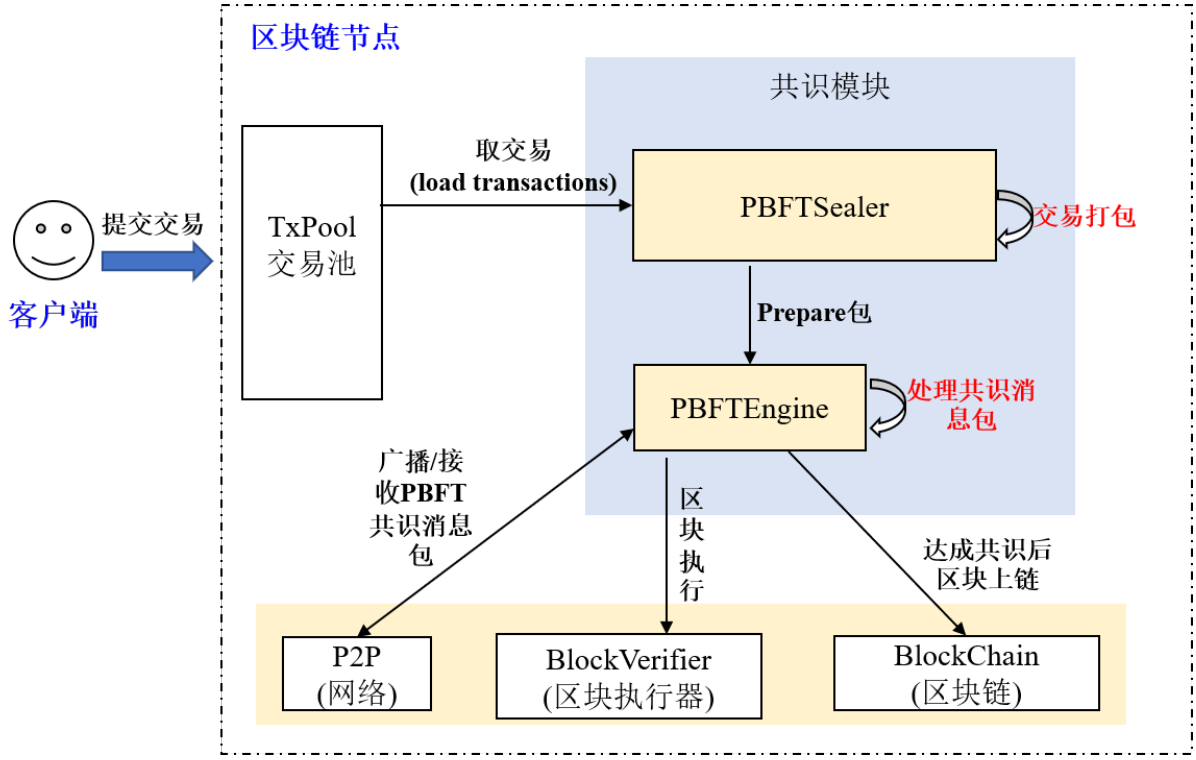
视图(记为toView，为当前视图加一)，某节点收集满(2*f+1)个视图等于toView、来自不同节点的ViewChangeReq后，会将当前视图切换为toView。

这四类消息包包含的字段大致相同，所有消息包共有的字段如下：

PrepareReqPacket类型消息包包含了正在处理的区块信息:

2. 系统框架

系统框架如下图:



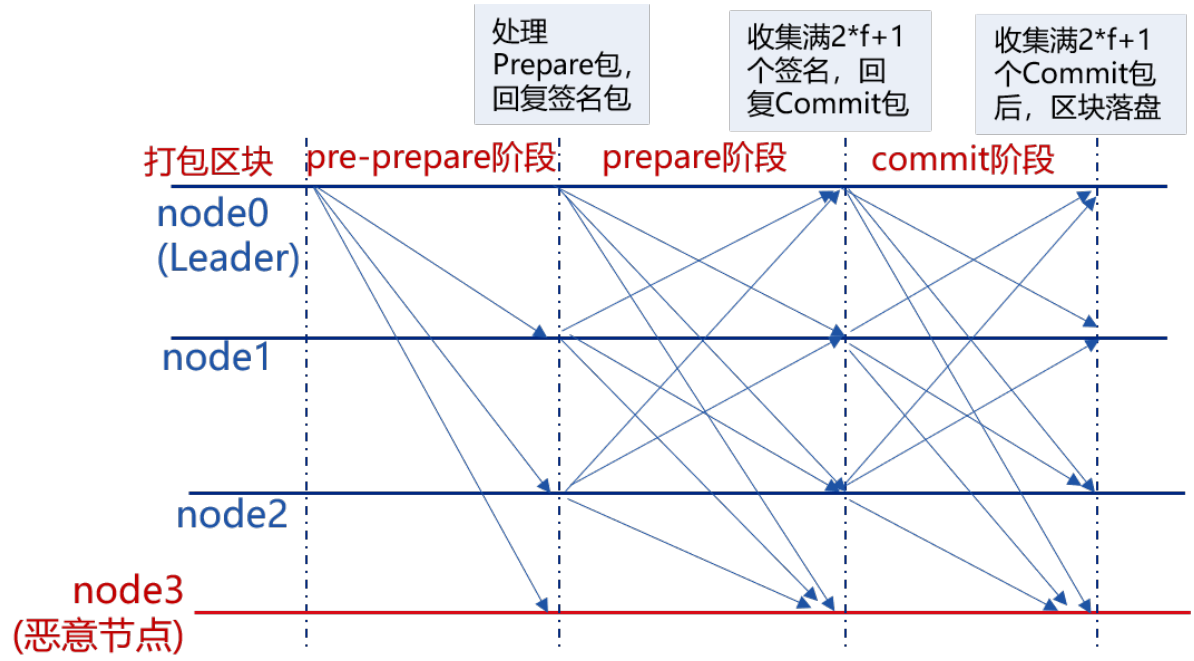
PBFT共识主要包括两个线程:

- PBFTSealer: PBFT打包线程，负责从交易池取交易，并将打包好的区块封装成PBFT Prepare包，交给PBFTEngine处理;
- PBFTEngine: PBFT共识线程，从PBFTSealer或者P2P网络接收PBFT共识消息包，区块验证器(Blockverifier)负责开始执行区块，完成共识流程，将达成共识的区块写入区块链，区块上链后，从交易池中删除已经上链的交易。

3. 核心流程

PBFT共识主要包括Pre-prepare、Prepare和Commit三个阶段:

- **Pre-prepare**: 负责执行区块，产生签名包，并将签名包广播给所有共识节点;
- **Prepare**: 负责收集签名包，某节点收集满2*f+1的签名包后，表明自身达到可以提交区块的状态，开始广播Commit包;
- **Commit**: 负责收集Commit包，某节点收集满2*f+1的Commit包后，直接将本地缓存的最新区块提交到数据库。

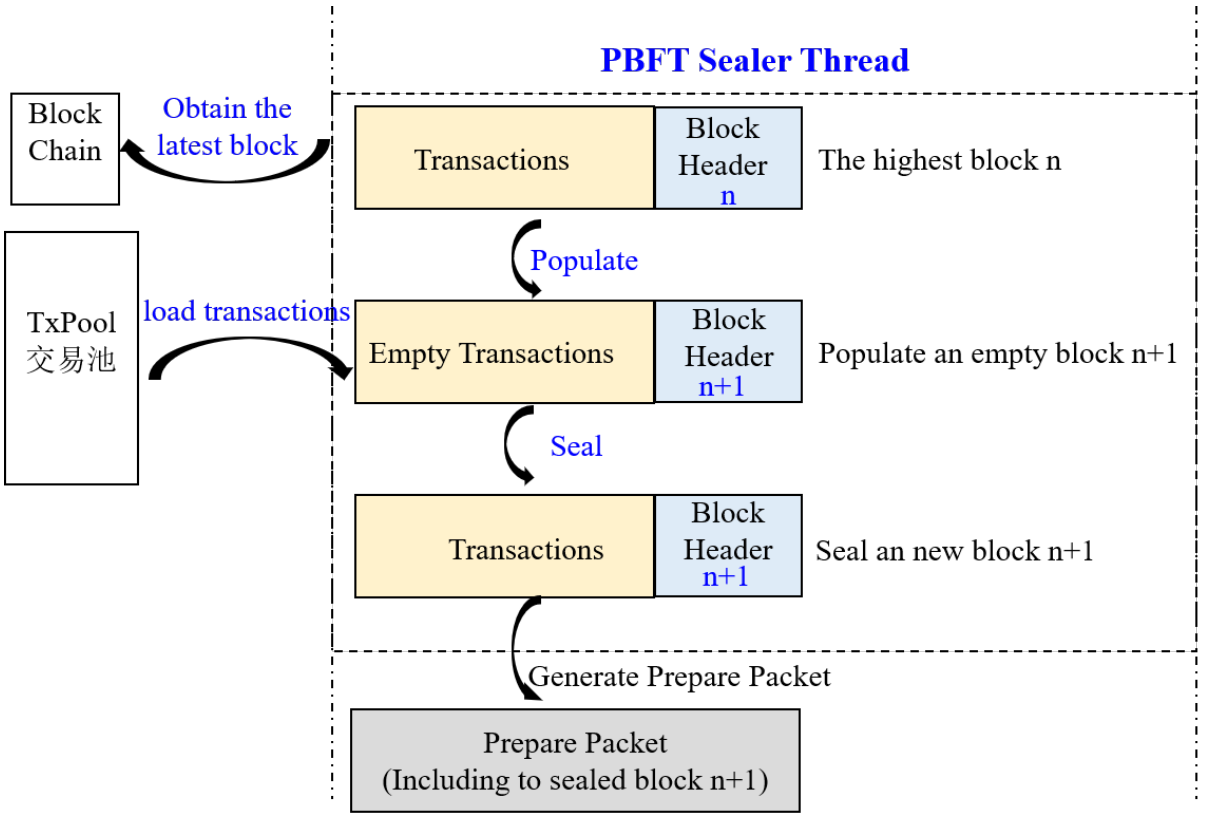


下图详细介绍了PBFT各个阶段的具体流程：

3.1 leader打包区块

PBFT共识算法中，共识节点轮流出块，每一轮共识仅有一个leader打包区块，leader索引通过公式`(block_number + current_view) % consensus_node_num`计算得出。

节点计算当前leader索引与自己索引相同后，就开始打包区块。区块打包主要由PBFTSealer线程完成，Sealer线程的主要工作如下图所示：



- **产生新的空块**: 通过区块链(BlockChain)获取当前最高块，并基于最高块产生新空块(将新区块父哈希置为最高块哈希，时间戳置为当前时间，交易清空);
- **从交易池打包交易**: 产生新空块后，从交易池中获取交易，并将获取的交易插入到产生的新区块中;
- **组装新区块**: Sealer线程打包到交易后，将新区块的打包者(Sealer字段)置为自己索引，并根据打包的交易计算出所有交易的transactionRoot;
- **产生Prepare包**: 将组装的新区块编码到Prepare包内，通过PBFTEngine线程广播给组内所有共识节点，其他共识节点收到Prepare包后，开始进行三阶段共识。

3.2 pre-prepare阶段

共识节点收到Prepare包后，进入pre-prepare阶段，此阶段的主要工作流程包括:

- **Prepare包合法性判断**：主要判断是否是重复的Prepare包、Prepare请求中包含的区块父哈希是否是当前节点最高块哈希(防止分叉)、Prepare请求中包含区块的块高是否等于最高块高加一;
- **缓存合法的Prepare包**: 若Prepare请求合法，则将其缓存到本地，用于过滤重复的Prepare请求;
- **空块判断**：若Prepare请求包含的区块中交易数目是0，则触发空块视图切换，将当前视图加一，并向所有其他节点广播视图切换请求;
- **执行区块并缓存区块执行结果**: 若Prepare请求包含的区块中交易数目大于0，则调用BlockVerifier区块执行器执行区块，并缓存执行后的区块;
- **产生并广播签名包**：基于执行后的区块哈希，产生并广播签名包，表明本节点已经完成区块执行和验证。

3.3 Prepare阶段

共识节点收到签名包后，进入Prepare阶段，此阶段的主要工作流程包括:

- **签名包合法性判断**：主要判断签名包的哈希与pre-prepare阶段缓存的执行后的区块哈希相同，若不相同，则继续判断该请求是否属于未来块签名请求(产生未来块的原因是本节点处理性能低于其他节点，还在进行上一轮共识，判断未来块的条件是：签名包的height字段大于本地最高块高加一)，若请求也非未来块，则是非法的签名请求，节点直接拒绝该签名请求;
- **缓存合法的签名包**：节点会缓存合法的签名包;
- **判断pre-prepare阶段缓存的区块对应的签名包缓存是否达到2*f+1，若收集满签名包，广播Commit包**：若pre-prepare阶段缓存的区块哈希对应的签名包数目超过`2*f+1`，则说明大多数节点均执行了该区块，并且执行结果一致，说明本节点已经达到可以提交区块的状态，开始广播Commit包;
- **若收集满签名包，备份pre-prepare阶段缓存的Prepare包落盘**：为了防止Commit阶段区块未提交到数据库之前超过`2*f+1`个节点宕机，这些节点启动后重新出块，导致区块链分叉(剩余的节点最新区块与这些节点最高区块不同)，还需要备份pre-prepare阶段缓存的Prepare包到数据库，节点重启后，优先处理备份的Prepare包。

3.4 Commit阶段

共识节点收到Commit包后，进入Commit阶段，此阶段工作流程包括:

- **Commit包合法性判断**：主要判断Commit包的哈希与pre-prepare阶段缓存的执行后的区块哈希相同，若不相同，则继续判断该请求是否属于未来块Commit请求(产生未来块的原因是本节点处理性能低于其他节点，还在进行上一轮共识，判断未来块的条件是：Commit的height字段大于本地最高块高加一)，若请求也非未来块，则是非法的Commit请求，节点直接拒绝该请求;
- **缓存合法的Commit包**：节点缓存合法的Commit包;

- **判断pre-prepare阶段缓存的区块对应的Commit包缓存是否达到2*f+1，若收集满Commit包，则将新区块落盘**：若pre-prepare阶段缓存的区块哈希对应的Commit请求数目超过2*f+1，则说明大多数节点达到了可提交该区块状态，且执行结果一致，则调用BlockChain模块将pre-prepare阶段缓存的区块写入数据库；

3.5 视图切换处理流程

当PBFT三阶段共识超时或节点收到空块时，PBFTEngine会试图切换到更高的视图(将要切换到的视图toView加一)，并触发ViewChange处理流程；节点收到ViewChange包时，也会触发ViewChange处理流程：

- **判断ViewChange包是否有效**: 有效的ViewChange请求中带有的块高值必须不小于当前节点最高块高，视图必须大于当前节点视图；
- **缓存有效ViewChange包**： 防止相同的ViewChange请求被处理多次，也作为判断节点是否可以切换视图的统计依据；
- **收集ViewChange包**： 若收到的ViewChange包中附带的view等于本节点的即将切换到的视图toView且本节点收集满2*f+1来自不同节点view等于toView的ViewChange包，则说明超过三分之二的节点要切换到toView视图，切换当前视图到toView。

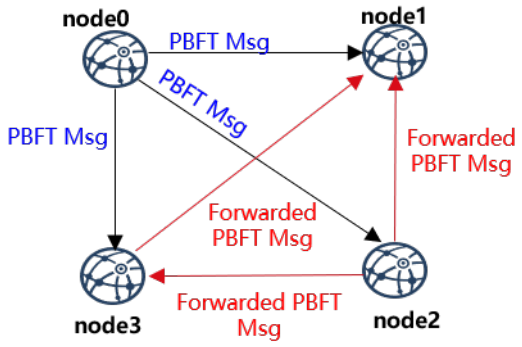
9.3.3 PBFT网络优化

FISCO BCOS v2.2.0优化了PBFT消息转发机制和Prepare包的结构，尽量减少网络中冗余的数据包，提升网络效率。

PBFT消息转发优化

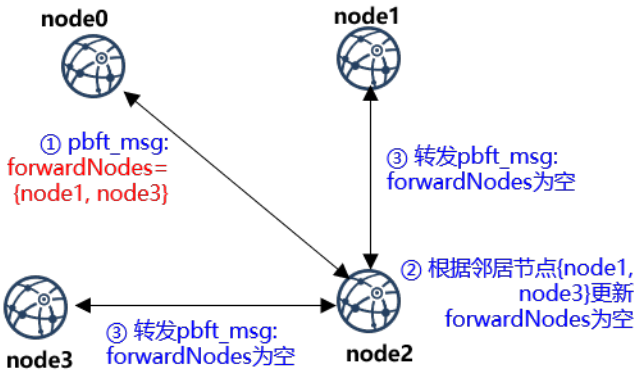
为了保证节点断连情况下共识消息包能到达所有节点，FISCO BCOS PBFT共识模块采用了消息转发机制，优化前的消息转发机制如下：

对于全连四节点区块链系统，系统TTL设置为2时，每个共识消息包均会被转发三次，且节点规模越大、TTL值越大冗余的共识消息包越多。且Leader广播的Prepare包内含有整个区块，多次转发同样的Prepare包会带来巨大的网络开销。



为了在网络全连的情况下，避免冗余的共识消息包；在网络断连情况下，共识消息包能尽量到达每个共识节点，FISCO BCOS v2.2.0对PBFT消息转发机制进行了优化，优化后的PBFT消息转发流程如下：

下图展示了四节点区块链系统在节点断连情况下，PBFT消息包转发流程：



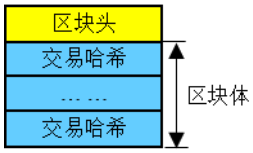
- node0向{node1, node2, node3}发送PBFT消息，发现{node1, node3}不在连接列表内，则将PBFT消息msg的forwardNodes字段设置为{node1, node3}，并将其转发给node2；
- node2收到node1的PBFT消息后，判断forwardNodes字段不为空，则遍历邻居节点列表{node1, node3}，并将邻居节点从forwardNodes中移除；
- node2向node1和node3转发更新后的PBFT消息msg；
- node1和node3收到msg后，判断forwardNodes字段为空，认为该消息已经到达了所有节点，不继续转发PBFT消息。

优化后的PBFT消息转发策略，源节点在PBFT消息包中加入了forwardNodes字段记录断连节点信息，其他节点收到PBFT消息包后，将消息转发给forwardNodes记录的可达节点，保障PBFT消息包尽量能到达所有节点的同时，减少了网络中冗余的PBFT消息，提升网络效率。

Prepare包结构优化

PBFT共识算法中，Leader向所有节点广播Prepare包，Prepare包内包含Leader节点从交易池打包的整个区块，由于同步模块会将交易同步到所有共识节点，因此Prepare包内区块的交易有很大概率在其他共识节点的交易池命中。基于这点，FISCO BCOS 2.2.0优化了Prepare包结构，Prepare消息包内的区块仅包含交易哈希，其他节点收到Prepare包后，优先从本地交易池内获取命中交易，缺失的交易向Leader请求。

优化后的Prepare消息包内的区块结构如下：



Prepare包处理流程如下：

优化Prepare结构后，充分利用了交易池缓存的交易，进一步降低了Prepare消息包的大小，节省了网络流量。

9.3.4 Raft

1 名词解释

1.1 Raft

Raft（Replication and Fault Tolerant）是一个允许网络分区（Partition Tolerant）的一致性协议，它保证了在一个由N个节点构成的系统中有(N+1)/2（向上取整）个节点正常工作的情况下的系统的一致性，比如在一个5个节点的系统中允许2个节点出现非拜占庭错误，如节点宕机、网络分区、消息延时。Raft相比于Paxos更容易理解，且被证明可以提供与Paxos相同的容错性以及性能，其详细介绍可见官网及动态演示。

1.2 节点类型

在Raft算法中，每个网络节点只能如下三种身份之一：**Leader**、**Follower**以及**Candidate**，其中：

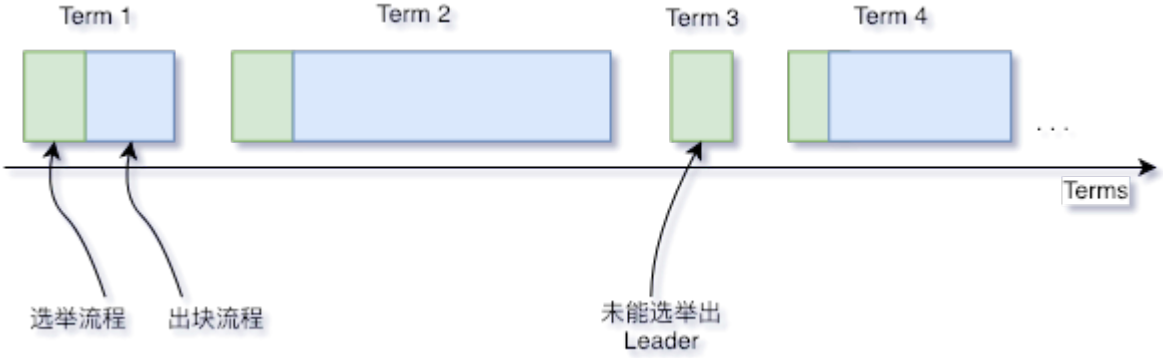
- **Leader**：主要负责与外界交互，由Follower节点选举而来，在每一次共识过程中有且仅有一个Leader节点，由Leader全权负责从交易池中取出交易、打包交易组成区块并将区块上链；
- **Follower**：以Leader节点为准进行同步，并在Leader节点失效时举行选举以选出新的Leader节点；
- **Candidate**：Follower节点在竞选Leader时拥有的临时身份。

1.3 节点ID & 节点索引

在Raft算法中，每个网络节点都会有一个固定且全局的唯一的用于表明节点身份的ID（一般是一个64字节表示数字），这称为节点ID；同时每个共识节点还会维护一份公共的共识节点列表，这个列表记录了每个共识节点的ID，而自己在这个列表中的位置被称为节点索引。

1.4 任期

Raft算法将时间划分为不定长度的任期Terms，Terms为连续的数字。每个Term以选举开始，如果选举成功，则由当前leader负责出块，如果选举失败，并没有选举出新的单一Leader，则会开启新的Term，重新开始选举。



1.5 消息

在Raft算法中，每个网络节点间通过发送消息进行通讯，当前Raft模块包括四种消息：**VoteReq**、**VoteResp**、**Heartbeat**、**HeartbeatResp**，其中：

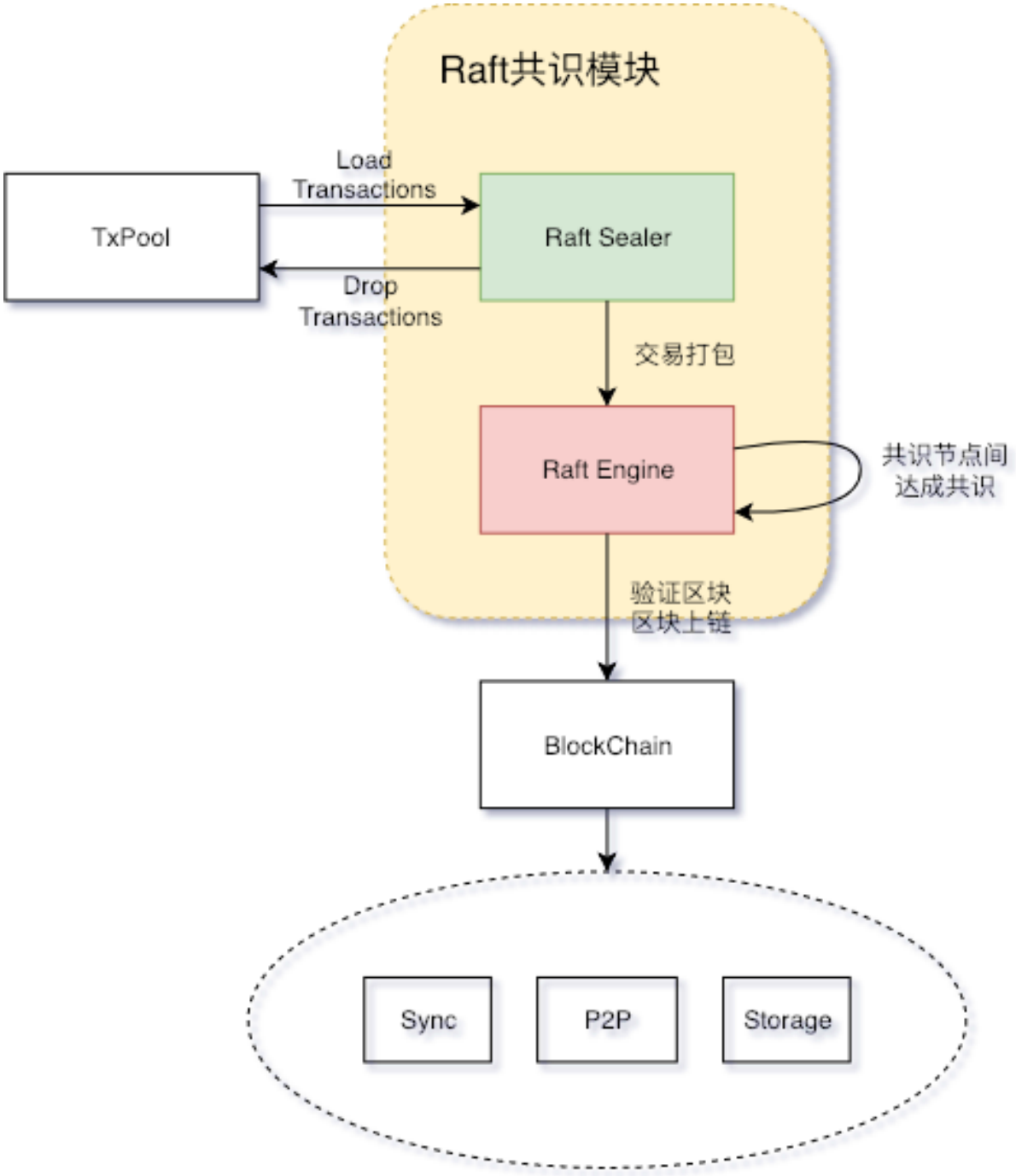
- **VoteReq**：投票请求，由Candidate节点主动发出，用于向网络中其他节点请求投票以竞选Leader；
- **VoteResp**：投票响应，在节点收到投票请求后，用于对投票请求进行响应，响应内容为同意或拒绝该投票请求；
- **Heartbeat**：心跳，由Leader节点主动周期发出，其作用有两个：(1) 用于维护Leader节点身份，只要Leader能够一直正常发送心跳且被其他节点响应，Leader身份就不会发生变化；(2) 区块数据复制，当Leader节点成功打包一个区块后，会将区块数据编码至心跳中以将区块进行广播，其他节点在收到该心跳后会解码出区块数据并将区块放入自己的缓冲区中；
- **HeartbeatResp**：心跳响应，在节点收到心跳后后，用于对心跳进行效应，特别的，当收到一个包含区块数据的心跳时，该心跳的响应中会带上该区块的哈希；

所有消息共有的字段如下表所示：

每种消息类型特有的字段如下表所示：

2 系统框架

系统框架如下图所示：

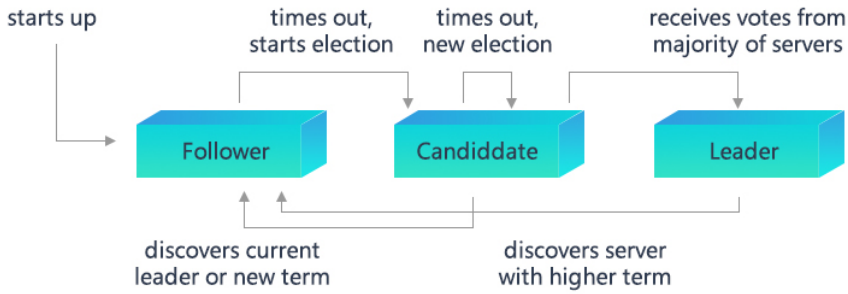


- Raft Sealer：负责从交易池取出交易并打包成区块，并发送至Raft Engine进行共识。区块上链后，Raft Sealer负责从交易池中删除已上链交易；
- Raft Engine：负责在共识节点进行共识，将达成共识的区块上链。

3 核心流程

3.1 节点状态转换

节点类型之间转换关系如下图所示，每种状态转换形式将在接下来的各个小节进行阐述：



3.1.1 选举

Raft共识模块中使用心跳机制来触发Leader选举。当节点启动时，节点自动成为Follower且将Term置0。只要Follower从Leader或者Candidate收到有效的Heartbeat或RequestVote消息，其就会保持在Follower状态，如果Follower在一段时间内（这段时间称为 ***Election Timeout***）没收到上述消息，则它会假设系统当前的Leader已经失活，然后增加自己的Term并转换为Candidiate，开启新一轮的Leader选举流程，流程如下:

1. Follower增加当前的Term，转换为Candidate；
2. Candidate将票投给自己，并广播RequestVote到其他节点请求投票；
3. Candidate节点保持在Candidate状态，直到下面三种情况中的一种发生：(1)该节点赢得选举；(2) 在等待选举期间，Candidate收到了其他节点的Heartbeat；(3) 经过*Election Timeout*后，没有Leader被选出。Raft算法采用随机定时器的方法来避免节点选票出现平均瓜分的情况以保证大多数时候只会有一个节点超时进入Candidate状态并获得大部分节点的投票成为Leader。

3.1.2 投票

节点在收到VoteReq消息后，会根据消息的内容选择不同的响应策略:

1. ***VoteReq*的*Term*小于或等于自己的*Term***
   - 如果节点是Leader，则拒绝该投票请求，Candidate收到此响应后会放弃选举转变为Follower，并增加投票超时；
   - 如果节点不是Leader:
     - 如果VoteReq的Term小于自己的Term，则拒绝该投票请求，如果Candidate收到超过半数的该种响应则表明其已经过时，此时Candidate会放弃选举转变为Follower，并增加投票超时；
     - 如果VoteReq的Term等于自己的Term，则拒绝该投票请求，对于该投票请求不作任何处理。对于每个节点而言，只能按照先到先得的原则投票给一个Candidate，从而保证每轮选举中至多只有一个Candidate被选为Leader。
2. ***VoteReq*的*lastLeaderTerm*小于自己的*lastLeaderTerm***

   每个节点中会有一个lastLeaderTerm字段表示该节点见过的最后一个Leader的Term，lastLeaderTerm仅能由Heartbeat进行更新。如果VoteReq中的lastLeaderTerm小于自己的lastLeaderTerm，表明Leader访问这个Candidate存在问题，如果此时Candidate处于网络孤岛的环境中，会不断向外提起投票请求，因此需要打断它的投票请求，所以此时节点会拒绝该投票请求。

3. ***VoteReq*的*lastBlockNumber*小于自己的*lastBlockNumber***

   每个节点中会有一个lastBlockNumber字段表示节点见到过的最新块的块高。在出块过程中，节点间会进行区块复制（详见3.2节），在区块复制的过程中，可能有部分节点收到了较新的区块数据而部分没有，从而导致不同节点的lastBlockNumber不一致。为了使系统能够达成一致，需要要求节点必须把票投给拥有较新数据的节点，因此在这种情况下节点会拒绝该投票请求。

4. **节点是第一次投票**

   为了避免出现Follower因为网络抖动导致重新发起选举，规定如果节点是第一次投票，直接拒绝该投票请求，同时会将自己的firstVote字段置为该Candidate的节点索引。

5. ***1~4*步骤中都没有拒绝投票请求**

   同意该投票请求。

3.1.3 心跳超时

在Leader成为网络孤岛时，Leader可以发出心跳、Follower可以收到心跳但是Leader收不到心跳回应，这种情况下Leader此时已经出现网络异常，但是由于一直可以向外发送心跳包会导致Follower无法切换状态进行选取，系统陷入停滞。为了避免第二种情况发生，模块中设置了心跳超时机制，Leader每次收到心跳回应时会进行相应记录，一旦一段时间后记录没有更新则Leader放弃Leader身份并转换为Follower节点。

3.2 区块复制

Raft协议强依赖Leader节点的可用性来确保集群数据的一致性，因为数据只能从Leader节点向Follower节点转移。当Raft Sealer向集群Leader提交区块数据后，Leader将该数据置为未提交（uncommitted）状态，接着Leader 节点会通过在Heartbeat中附加数据的形式并发向所有Follower节点复制数据并等待接收响应，在确保网络中超过半数节点已接收到数据后，再将区块数据写入底层存储中，此时区块数据状态已经进入已提交（committed）状态。此后Leader节点再通过Sync模块向其他Follower节点广播该区块数据，区块复制及提交的流程图如下图所示：

其中RaftSealer验证是否当前是否能打包交易的验证条件包括：(1) 是否为Leader；(2) 是否存在尚未完成同步的peer； (3) uncommitBlock字段是否为空，只有三个条件均符合才允许打包。

9.3.5 rPBFT

区块链共识困境

POW类算法

POW算法因如下特点，不适用于交易吞吐量大、交易时延要求低的联盟链场景：

- 性能低：10分钟出一个区块，交易确认时延一个小时，耗电多
- 无最终一致性保证
- 吞吐量低

基于分布式一致性原理的共识算法

基于分布式一致性原理的共识算法，如BFT类和CFT类共识算法具有秒级交易确认时延、最终一致性、吞吐量高、不耗电等优势，尤其是BFT类共识算法还可应对节点作恶的场景，在性能、安全性等方面均可达到联盟链需求。

但这类算法复杂度均与节点规模有关，可支撑的网络规模有限，极大限制了联盟链节点规模。

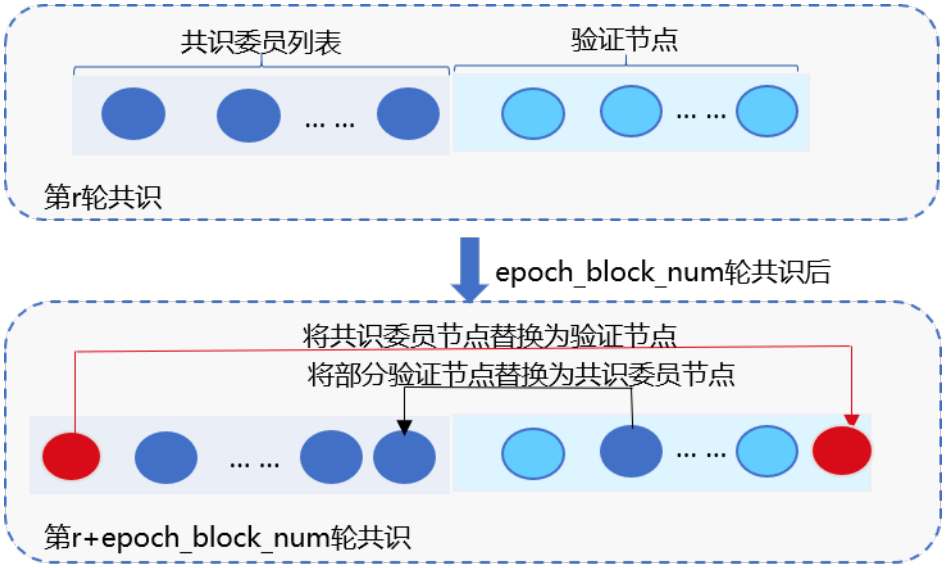
综上所述，FISCO BCOS v2.3.0提出了rPBFT共识算法，旨在保留BFT类共识算法高性能、高吞吐量、高一致性、安全性的同时，尽量减少节点规模对共识算法的影响。

rPBFT共识算法

节点类型

- 共识委员：执行PBFT共识流程的节点，有轮流出块权限
- 验证节点：不执行共识流程，验证共识节点是否合法、区块验证，经过若干轮共识后，会切换为共识节点

核心思想



rPBFT算法每轮共识流程仅选取若干个共识节点出块，并根据区块高度周期性地替换共识节点，保障系统安全，主要包括2个系统参数：

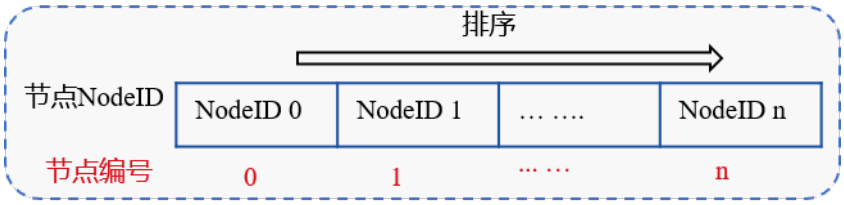
- `epoch_sealer_num`：每轮共识过程中参与共识的节点数目，可通过控制台发交易方式动态配置该参数
- `epoch_block_num`：共识节点替换周期，为防止选取的共识节点联合作恶，rPBFT每出epoch_block_num个区块，会替换一个共识节点，可通过控制台发交易的方式动态配置该参数

这两个配置项记录在系统配置表中，配置表主要包括配置关键字、配置对应的值、生效块高三个字段，其中生效块高记录了配置最新值最新生效块高，例：在100块发交易将`epoch_sealer_num`和`epoch_block_num`分别设置为4和10000，此时系统配置表如下：

算法流程

确定各共识节点编号IDX

对所有共识节点的NodeID进行排序，如下图，节点排序后的NodeID索引即为该共识节点编号：



链初始化

链初始化时，rPBFT需要选取epoch_sealer_num个共识节点到共识委员中参与共识，目前初步实现是选取索引为0到epoch_sealer_num-1的节点参与前epoch_block_num个区块共识。

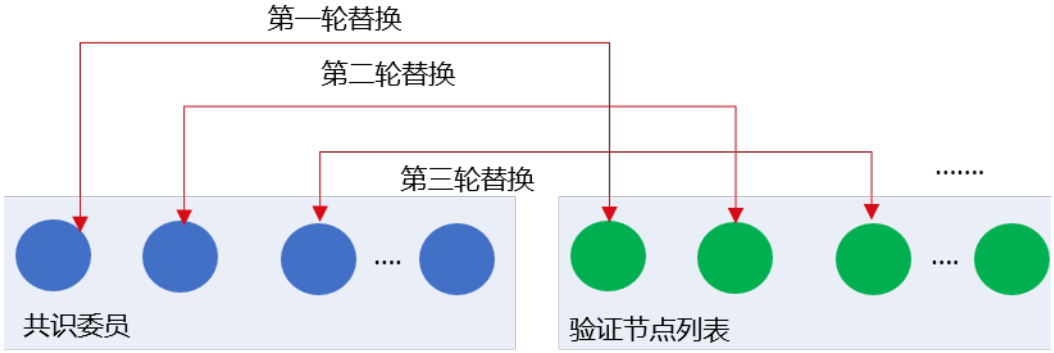
共识委员节点运行PBFT共识算法

选取的epoch_sealer_num个共识委员节点运行PBFT共识算法，验证节点同步并验证这些共识委员节点共识产生的区块，验证节点的验证步骤包括：

- 校验区块签名列表：每个区块必须至少包含三分之二共识委员的签名
- 校验区块执行结果：本地区块执行结果须与共识委员在区块头记录的执行结果一致

动态替换共识委员列表

为保障系统安全性，rPBFT算法每出epoch_block_num个区块后，会从共识委员列表中剔除一个节点作为验证节点，并加入一个验证节点到共识委员列表中，如下图所示：



rPBFT算法目前实现中，轮流将共识委员列表节点替换为验证节点，设当前有序的共识委员会节点列表为CommitteeSealersList，共识节点总数为N，则共识epoch_block_num个区块后，会将CommitteeSealersList[0]剔除共识委员列表，并加入索引为(CommitteeSealersList[0].IDX + epoch_sealer_num) % N的验证节点到共识委员列表中。第i轮替换周期，将CommitteeSealersList[i % epoch_sealer_num]剔除共识委员列表，加入索引为(CommitteeSealersList[i%epoch_sealer_num].IDX + epoch_sealer_num) % N的验证节点到共识委员列表中。

节点重启

节点重启后，rPBFT算法需要快速确定共识委员列表，由于epoch_block_num可通过控制台动态更新，需要结合epoch_block_num最新配置生效块高获取共识委员列表，主要步骤如下：

计算共识周期rotatingRound

设当前块高为blockNum，epoch_block_num生效块高为enableNum，则共识周期为：
rotatingRound = (blockNumber - enableNum) % epoch_block_num

确定共识委员起始节点索引: N为共识节点总数，索引从`(rotatingRound * epoch_block_num) % N`到`(rotatingRound * epoch_block_num + epoch_sealer_num) % N`之间的节点均属于共识委员节点

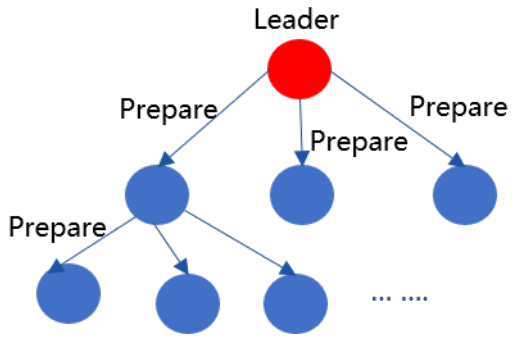
rPBFT算法分析

- 网络复杂度：O(epoch_sealer_num * epoch_sealer_num)，与节点规模无关，可扩展性强于PBFT共识算法
- 性能：可秒级确认，且由于算法复杂度与节点数无关，性能衰减远小于PBFT
- 一致性、可用性要求：需要至少三分之二的共识委员节点正常工作，系统才可正常共识
- 安全性：未来将引入VRF算法，随机、私密地替换共识委员，增强共识算法安全性

rPBFT网络优化

Prepare包广播优化

为进一步提升Prepare包在带宽有限场景下广播效率，FISCO BCOS v2.3.0在rPBFT的基础上实现了Prepare包树状广播，如下图所示：



- 根据共识节点索引，构成完全n叉树(默认是3)
- Leader产生Prepare包后，沿着树状拓扑将Prepare包转发给其所有下属子节点

优势:

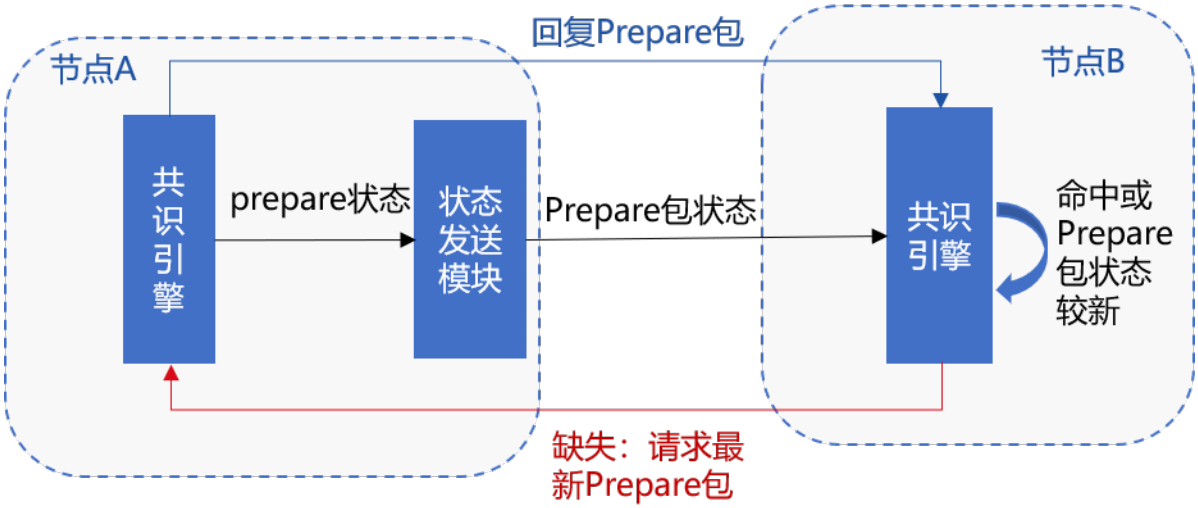
- 传播速度比gossip快，无冗余消息包
- 分而治之，每个节点出带宽为O(1)，可扩展性强

劣势: 中间节点是单点，需要额外的容错策略

基于状态包的容错方案

注解: 基于状态包的容错策略仅在开启Prepare包树状广播时生效

为保证节点断连情况下，开启树状广播时，Prepare包能到达每个节点，rPBFT引入了基于状态包的容错机制，如下图所示：



主要流程包括：

(1) 节点A收到Prepare后，随机选取33%节点广播Prepare包状态，记为prepareStatus，包括{blockNumber, blockHash, view, idx}

(2) 节点B收到节点A随机广播过来的prepareStatus后，判断节点A的Prepare包状态是否比节点B当前Prepare包localPrepare状态新，主要判断包括：

- prepareStatus.blockNumber是否大于当前块高
- prepareStatus.blockNumber是否大于localPrepare.blockNumber
- prepareStatus.blockNumber等于localPrepare.blockNumber情况下，prepareStatus.view是否大于localPrepare.view

以上任意一个条件成立，都说明节点A的Prepare包状态比节点B的状态新

(3) 若节点B的状态落后于节点A，且节点B与其父节点断连，则节点B向节点A发出prepareRequest请求，请求相应的Prepare包

(4) 若节点B的状态落后于节点A，但节点B与其父节点相连，若节点B最多等待100ms(可配)后，状态仍然落后于节点A，则节点B向节点A发出prepareRequest请求，请求相应的Prepare包

(5) 节点B收到节点A的prepareRequest请求后，向其回复相应的Prepare消息包

(6) 节点A收到节点B的Prepare消息包后，执行handlePrepare流程处理收到的Prepare包。

流量负载均衡策略

注解： 流量负载均衡策略仅在开启Prepare包树状广播时生效

rPBFT开启Prepare包结构优化后，其他共识节点交易缺失后，向leader请求交易，导致leader出带宽成为瓶颈，FISCO BCOS v2.3.0结合Prepare包状态，设计并实现了负载均衡策略，该策略时序图如下：

Leader的子节点sealerA的主要处理流程如下：

(1) leader产生新区块后，将仅包含交易哈希列表的Prepare包发送给三个子节点

(2) 子节点sealerA收到Prepare包后，将其沿树状拓扑转发给三个子节点

(3) 子节点sealerA开始处理Prepare包：

- 从交易池中获取命中的交易，填充到Prepare包内的区块中
- 向父节点Leader请求缺失的交易

(4) sealerA收到Leader的回包后，将回包内的交易填充到Prepare包内，并随机选取33%的节点广播Prepare包的状态，主要包括{blockNumber, blockHash, view, idx}，其他节点收到该状态包后，将sealerA最新状态包更新到缓存中

sealerA的子节点sealerB的主要处理流程如下

(1) sealerB收到SealerA转发过来的Prepare包后，同样继续将该Prepare包转发给sealerB的子节点

(2) sealerB开始处理Prepare包，首先从交易池中获取命中的交易，填充到Prepare包的区块中，并选取节点获取缺失的交易：

- 若sealerB缓存来自节点sealerA的prepareStatus.blockHash等于Prepare.blockHash，则直接向父节点sealerA请求缺失交易
- 若sealerB缓存的sealerA状态包哈希不等于Prepare.blockHash，但存在来自其他节点C的prepareStatus.blockHash等于prepare.blockHash，则向C请求缺失交易
- 若sealerB缓存的任何节点prepareStatus的哈希均不但等于prepare.blockHash，最多等待100ms(可配)后，向Leader请求缺失的交易

(3) sealerB收到被请求节点回复的交易后，填充Prepare包内区块，并随机选取33%(可配)节点广播Prepare包状态

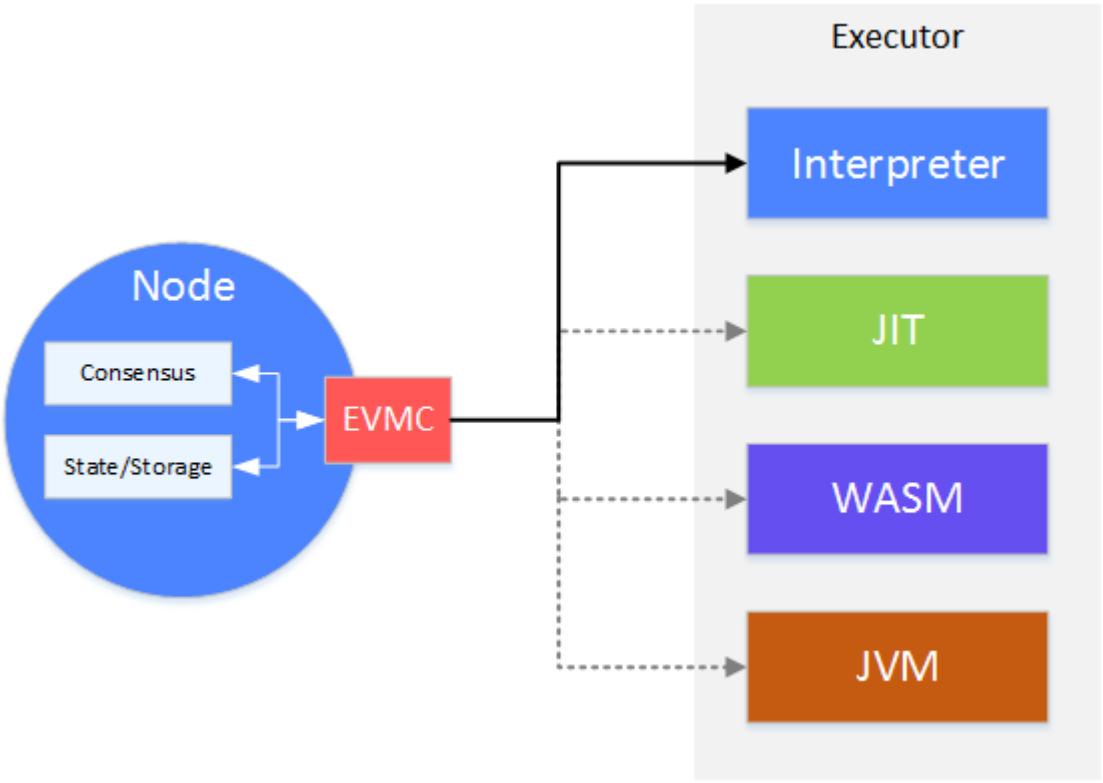
(4) 其他节点收到sealerB的状态包后，将其sealerB的最新状态包更新到缓存中

9.4 虚拟机与合约

交易的执行是区块链节点上的一个重要的功能。交易的执行，是把交易中的智能合约二进制代码取出来，用执行器（Executor）执行。共识模块（Consensus）把交易从交易池(TxPool)中取出，打包成区块，并调用执行器去执行区块中的交易。在交易的执行过程中，会对区块链的状态（State）进行修改，形成新区块的状态储存下来（Storage）。执行器在这个过程中，类似于一个黑盒，输入是智能合约代码，输出是状态的改变。

随着技术的发展，人们开始关注执行器的性能和易用性。一方面，人们希望智能合约在区块链上能有更快的执行速度，满足大规模交易的需求。另一方面，人们希望能用更熟悉更好用的语言进行开发。进而出现了一些替代传统的执行器（EVM）的方案，如：JIT、WASM_甚至JVM。然而，传统的EVM是耦合在节点代码中的。首先要做的，是将执行器的接口抽象出来，兼容各种虚拟机的实现。因此，EVMC被设计出来。

EVMC (Ethereum Client-VM Connector API)，是以太坊抽象出来的执行器的接口，旨在能够对接各种类型的执行器。FISCO BCOS目前采用了以太坊的智能合约语言Solidity，因此也沿用了以太坊对执行器接口的抽象。



在节点上，共识模块会调用EVMC，将打包好的交易交由执行器执行。执行器执行时，对状态进行的读写，会通过EVMC的回调反过来操作节点上的状态数据。

经过EVMC一层的抽象，FISCO BCOS能够对接今后出现的更高效、易用性更强的执行器。目前，FISCO BCOS采用的是传统的EVM根据EVMC抽象出来的执行器—Interpreter。因此能够支持基于Solidity语言的智能合约。目前其他类型的执行器发展尚未成熟，后续将持续跟进。

9.4.1 EVM 以太坊虚拟机

在区块链上，用户通过运行部署在区块链上的合约，完成需要共识的操作。以太坊虚拟机，是智能合约代码的执行器。

当智能合约被编译成二进制文件后，被部署到区块链上。用户通过调用智能合约的接口，来触发智能合约的执行操作。EVM执行智能合约的代码，修改当前区块链上的数据（状态）。被修改的数据，会被共识，确保一致性。

EVMC – Ethereum Client-VM Connector API

新版本的以太坊将EVM从节点代码中剥离出来，形成一个独立的模块。EVM与节点的交互，抽象出EVMC接口标准。通过EVMC，节点可以对接多种虚拟机，而不仅限于传统的基于solidity的虚拟机。

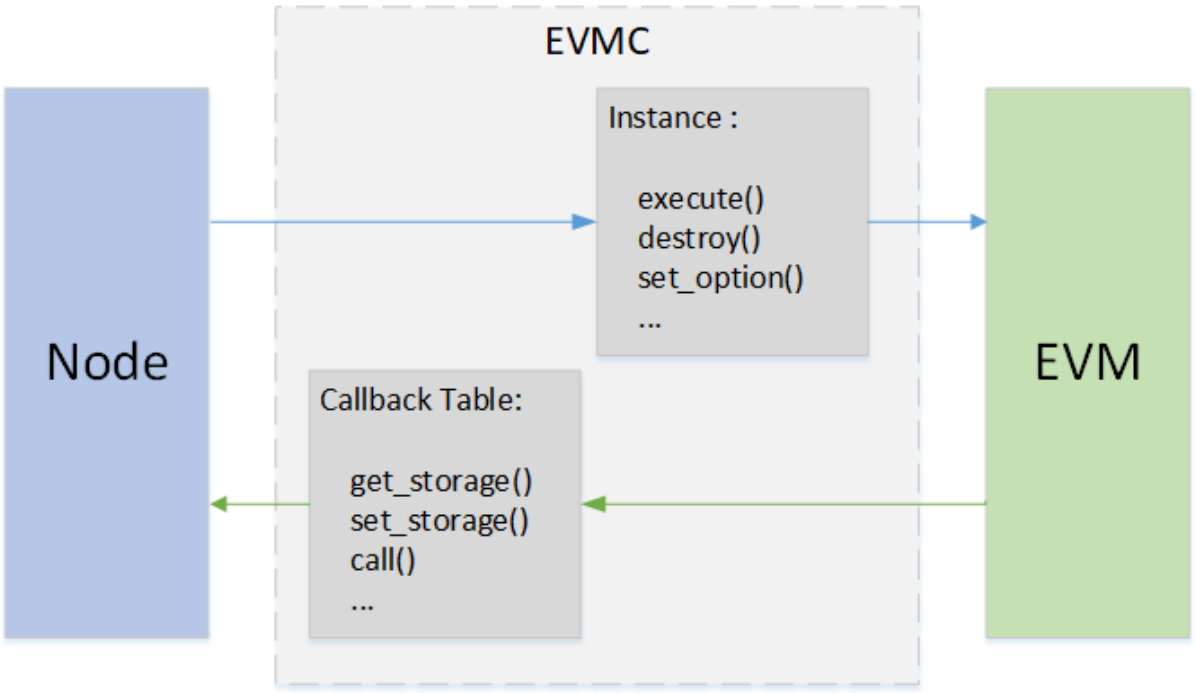
传统的solidity虚拟机，在以太坊中称为interpreter，下文主要解释interpreter的实现。

EVMC 接口

EVMC主要定义了两种调用的接口：

- Instance接口：节点调用EVM的接口
- Callback接口：EVM回调节点的接口

EVM本身不保存状态数据，节点通过instance接口操作EVM，EVM反过来，调Callback接口，对节点的状态进行操作。



Instance 接口

定义了节点对虚拟机的操作，包括创建，销毁，设置等。

接口定义在evmc_instance（evmc.h）中

- abi_version
- name
- version
- destroy
- execute
- set_tracer
- set_option

Callback接口

定义了EVM对节点的操作，主要是对state读写、区块信息的读写等。

接口定义在evmc_context_fn_table（evmc.h）中。

- evmc_account_exists_fn account_exists
- evmc_get_storage_fn get_storage
- evmc_set_storage_fn set_storage
- evmc_get_balance_fn get_balance
- evmc_get_code_size_fn get_code_size
- evmc_get_code_hash_fn get_code_hash
- evmc_copy_code_fn copy_code
- evmc_selfdestruct_fn selfdestruct
- evmc_call_fn call
- evmc_get_tx_context_fn get_tx_context
- evmc_get_block_hash_fn get_block_hash

- evmc_emit_log_fn emit_log

EVM 执行

EVM 指令

solidity是合约的执行语言，solidity被solc编译后，变成类似于汇编的EVM指令。Interpreter定义了一套完整的指令集。solidity被编译后，生成二进制文件，二进制文件就是EVM指令的集合，交易以二进制的形式发往节点，节点收到后，通过EVMC调用EVM执行这些指令。在EVM中，用代码模拟实现了这些指令的逻辑。

Solidity是基于堆栈的语言，EVM在执行二进制时，也是以堆栈的方式进行调用。

算术指令举例

一条ADD指令，在EVM中的代码实现如下。SP是堆栈的指针，从栈顶第一和第二个位置（SP[0]、SP[1]）拿出数据，进行加和后，写入结果堆栈SPP的顶端SPP[0]。

```
CASE(ADD)
{
    ON_OP();
    updateIOGas();

    // pops two items and pushes their sum mod 2^256.
    m_SPP[0] = m_SP[0] + m_SP[1];
}
```

跳转指令举例

JUMP指令，实现了二进制代码间的跳转。首先从堆栈顶端SP[0]取出待跳转的地址，验证一下是否越界，放到程序计数器PC中，下一个指令，将从PC指向的位置开始执行。

```
CASE(JUMP)
{
    ON_OP();
    updateIOGas();
    m_PC = verifyJumpDest(m_SP[0]);
}
```

状态读指令举例

SLOAD可以查询状态数据。大致过程是，从堆栈顶端SP[0]取出要访问的key，把key作为参数，然后调evmc的callback函数get_storage()，查询相应的key对应的value。之后将读到的value写到结果堆栈SPP的顶端SPP[0]。

```
CASE(SLOAD)
{
    m_runGas = m_rev >= EVMC_TANGERINE_WHISTLE ? 200 : 50;
    ON_OP();
    updateIOGas();

    evmc_uint256be key = toEvmC(m_SP[0]);
    evmc_uint256be value;
    m_context->fn_table->get_storage(&value, m_context, &m_message->destination, &
↪key);
    m_SPP[0] = fromEvmC(value);
}
```

状态写指令举例

SSTORE指令可以将数据写到节点的状态中，大致过程是，从栈顶第一和第二个位置（SP[0]、SP[1]）拿出key和value，把key和value作为参数，调用evmc的callback函数set_storage()，写入节点的状态。

```
CASE(SSTORE)
{
    ON_OP();
    if (m_message->flags & EVMC_STATIC)
        throwDisallowedStateChange();

    static_assert(
        VMSchedule::sstoreResetGas <= VMSchedule::sstoreSetGas, "Wrong SSTORE gas␣
↪costs");
    m_runGas = VMSchedule::sstoreResetGas;  // Charge the modification cost up␣
↪front.
    updateIOGas();

    evmc_uint256be key = toEvmC(m_SP[0]);
    evmc_uint256be value = toEvmC(m_SP[1]);
    auto status =
        m_context->fn_table->set_storage(m_context, &m_message->destination, &key,␣
↪&value);

    if (status == EVMC_STORAGE_ADDED)
    {
        // Charge additional amount for added storage item.
        m_runGas = VMSchedule::sstoreSetGas - VMSchedule::sstoreResetGas;
        updateIOGas();
    }
}
```

合约调用指令举例

CALL指令能够根据地址调用另外一个合约。首先，EVM判断是CALL指令，调用caseCall()，在caseCall()中，用caseCallSetup()从堆栈中拿出数据，封装成msg，作为参数，调用evmc的callback函数call。Eth在被回调call()后，启动一个新的EVM，处理调用，之后将新的EVM的执行结果，通过call()‘‘‘的参数返回给当前的EVM，当前的EVM将结果写入结果堆栈SSP中，调用结束。合约创建的逻辑与此逻辑类似。

```
CASE(CALL)
CASE(CALLCODE)
{
    ON_OP();
    if (m_OP == Instruction::DELEGATECALL && m_rev < EVMC_HOMESTEAD)
        throwBadInstruction();
    if (m_OP == Instruction::STATICCALL && m_rev < EVMC_BYZANTIUM)
        throwBadInstruction();
    if (m_OP == Instruction::CALL && m_message->flags & EVMC_STATIC && m_SP[2] !=␣
↪0)
        throwDisallowedStateChange();
    m_bounce = &VM::caseCall;
}
BREAK

void VM::caseCall()
{
    m_bounce = &VM::interpretCases;

    evmc_message msg = {};

    // Clear the return data buffer. This will not free the memory.
    m_returnData.clear();

    bytesRef output;
    if (caseCallSetup(msg, output))
    {
```

(continues on next page)

(续上页)

```
        evmc_result result;
        m_context->fn_table->call(&result, m_context, &msg);

        m_returnData.assign(result.output_data, result.output_data + result.output_
↪size);
        bytesConstRef{&m_returnData}.copyTo(output);

        m_SPP[0] = result.status_code == EVMC_SUCCESS ? 1 : 0;
        m_io_gas += result.gas_left;

        if (result.release)
            result.release(&result);
    }
    else
    {
        m_SPP[0] = 0;
        m_io_gas += msg.gas;
    }
    ++m_PC;
}
```

总结

EVM是一个状态执行的机器，输入是solidity编译后的二进制指令和节点的状态数据，输出是节点状态的改变。以太坊通过EVMC实现了多种虚拟机的兼容。但截至目前，并未出现除开interpreter之外的，真正生产可用的虚拟机。也许要做到同一份代码在不同的虚拟机上跑出相同的结果，是一件很难的事情。BCOS将持续跟进此部分的发展。

9.4.2 Precompiled

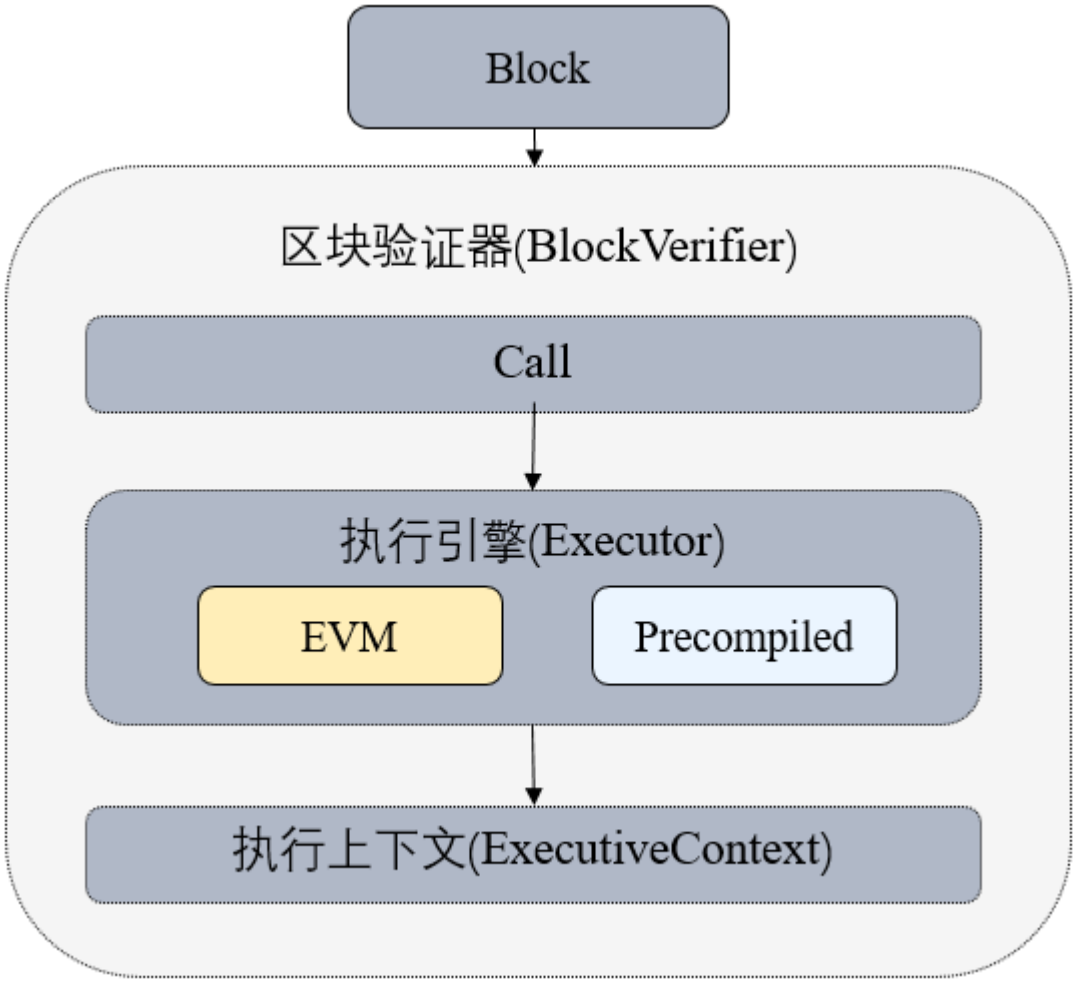
预编译合约提供一种使用C++编写合约的方法，合约逻辑与数据分离，相比于solidity合约具有更好的性能，可以通过修改底层代码实现合约升级。

预编译合约与Solidity合约对比

模块架构

Precompiled的架构如下图所示：

- 区块验证器在执行交易的时候会根据被调用合约的地址来判断类型。地址1-4表示以太坊预编译合约，地址0x1000-0x10000是C++预编译合约，其他地址是EVM合约。



关键流程

- 执行预编译合约时首先需要根据合约地址获取到预编译合约的对象。
- 每个预编译合约对象都会实现`call`接口，预编译合约的具体逻辑在该接口中实现。
- `call`根据交易的abi编码，获取到`Function Selector`和参数，然后执行对应的逻辑。

接口定义

每个预编译合约都必须实现自己的`call`接口，接口接受三个参数，分别是`ExecutiveContext`执行上下文、`bytesConstRef`参数的abi编码和外部账户地址，其中外部账户地址用于判断是否具有写权限。`Precompiled源码`。

9.4.3 Gas

EVM虚拟机有一整套Gas机制来衡量每笔交易上链消耗的CPU、内存和存储资源。FISCO BCOS 2.0引入了Precompiled合约，支持内置的C++合约，为了提升Precompiled合约的安全性，FISCO BCOS v2.4.0在Precompiled合约中引入了Gas机制。

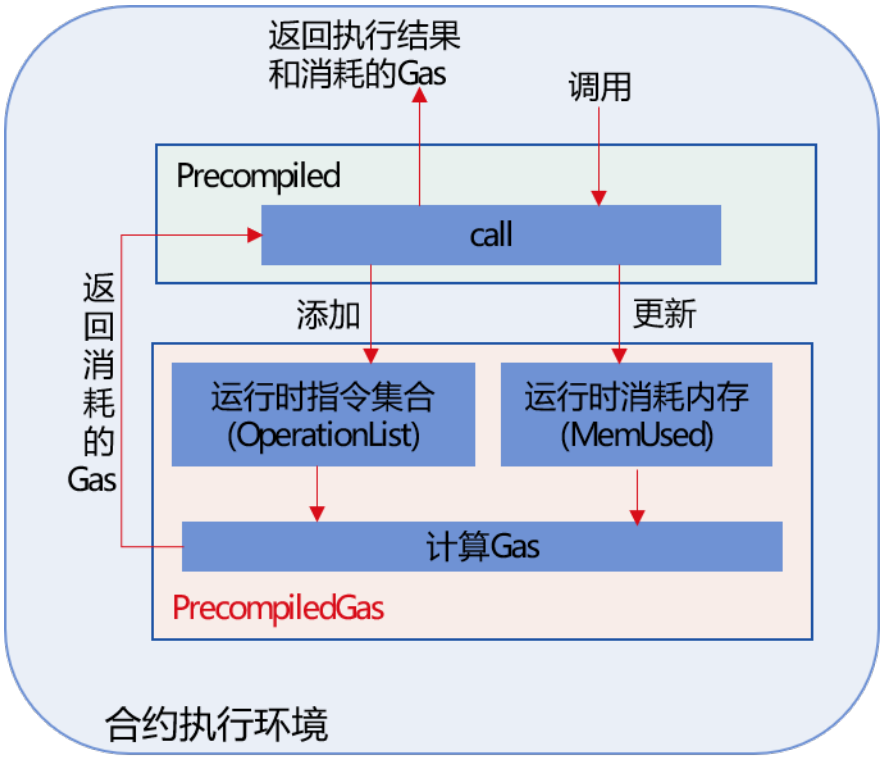
此外，EVM原始的Gas机制中，交易的主要Gas消耗来源于存储，考虑到联盟链场景更关注CPU和内存消耗，FISCO BCOS v2.4.0调整了存储Gas，引入`Free Storage` Gas衡量模式，提升CPU和内存在交易Gas消耗中的占比。

Precompiled合约支持Gas计算

注解： Precompiled合约支持Gas计算的特性从v2.4.0开始支持，当 `supported_version` 小于v2.4.0，或者旧链直接替换二进制升级时，不支持该特性

模块架构

FISCO BCOS v2.4.0新增了`PrecompiledGas`模块进行Gas计算，Gas开销主要包括CPU、内存和存储三个维度，模块图如下：



`PrecompiledGas`主要记录了每个交易执行Precompiled合约过程中调用的基础操作、占用内存消耗的Gas，交易调用Precompiled合约时Gas计算机制如下：

- 虚拟机执行交易调用`Precompiled`合约的`call`接口时，每调用一个基础操作，会将其对应的`OPCode`添加到`PrecompiledGas`的**运行时指令集合**中
- 虚拟机执行交易调用`Precompiled`合约的`call`接口时，基础操作占用的内存变化时，会更新`PrecompiledGas`的运行时消耗的内存
- `Precompiled`合约执行完毕后，可调用接口，根据运行`Precompiled`合约过程中执行的指令集合、消耗的内存，计算出该`Precompiled`合约Gas消耗。

Precompiled合约Gas衡量标准

FISCO BCOS Precompiled合约Gas衡量标准参考了EVM，主要包括CPU、内存和存储三个维度。下面详细介绍Precompiled合约具体的Gas计算方法。

Precompiled合约内存Gas计算

Precompiled合约内存消耗主要来自于输入、输出以及运行时产生的额外内存消耗。某笔交易消耗的总内存为txMemUsed时，其对应的内存Gas计算公式如下。即：每32字节增加memoryGasUnit个Gas，memoryGasUnit的值为3.

```
    MemoryGas(txMemUsed) = memoryGasUnit * txMemUsed / 32 + (txMemUsed *␣
↪txMemUsed)/512
```

Precompiled合约CPU、存储Gas计算

为了计算Precompiled合约基础操作消耗的Gas，FISCO BCOS v2.4.0将Precompiled合约映射到具体的操作码，并定义了每个基础操作对应的Gas。

Precompiled合约基础操作对应的操作码

PrecompiledGas模块将Precompiled合约基础操作映射到了操作码如下：

Precompiled合约基础操作衡量标准

PrecompiledGas定义了Precompiled合约每个基础操作对应的Gas消耗，具体如下：

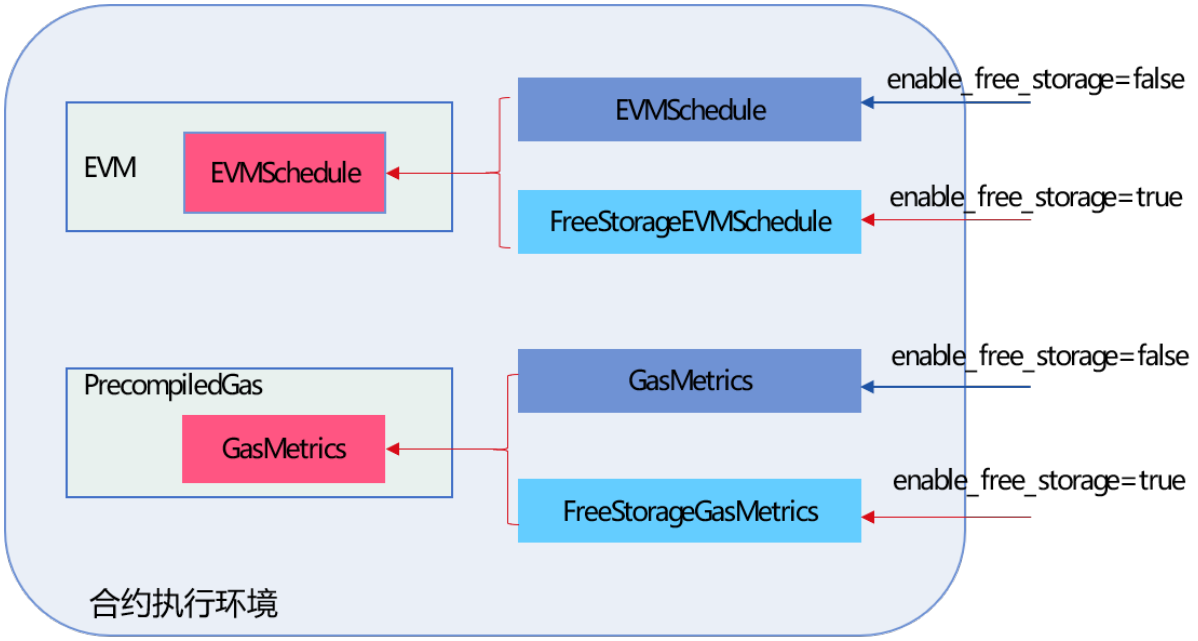
EVM Gas衡量标准插件化

如前面所述，针对部分场景衡量交易资源耗用时，更加关注CPU和Gas，FISCO BCOS v2.4.0引入了FreeStorage的Gas衡量模式，提升CPU和内存在交易Gas消耗中的占比。

注解： EVM Gas衡量标准支持插件化配置的特性从v2.4.0开始支持，当 supported_version 小于v2.4.0，或者旧链直接替换二进制升级时，不支持该特性

模块架构

为了支持Gas衡量标准插件化配置和FreeStorage的Gas衡量模式，FISCO BCOS v2.4.0在以太坊EVMSchedule引入FreeStorageEVMSchedule，在PrecopmiledGas的GasMetrics基础上引入了FreeStorageGasMetrics，并根据genesis文件的enable_free_storage配置项决定启用哪种Gas衡量模式，如下图所示：



为了提升CPU和内存在交易Gas消耗中的占比，`FreeStorageEVMSchedule`调整了创建合约、`SSTORE`、`SLOAD`等操作的Gas消耗；`FreeStorageGasMetrics`主要调整了`CreateTable`、`Insert`、`Remove`、`Update`等操作的Gas消耗。

Gas衡量标准

下面分别介绍非`FreeStorage`模式和`FreeStorage`模式下，EVM虚拟机和Precompiled合约Gas衡量标准：

EVM虚拟机Gas衡量标准

Precompiled合约Gas衡量标准

配置项

注解： EVM Gas衡量标准支持插件化配置项位于 `genesis` 文件中，详细可参考 这里

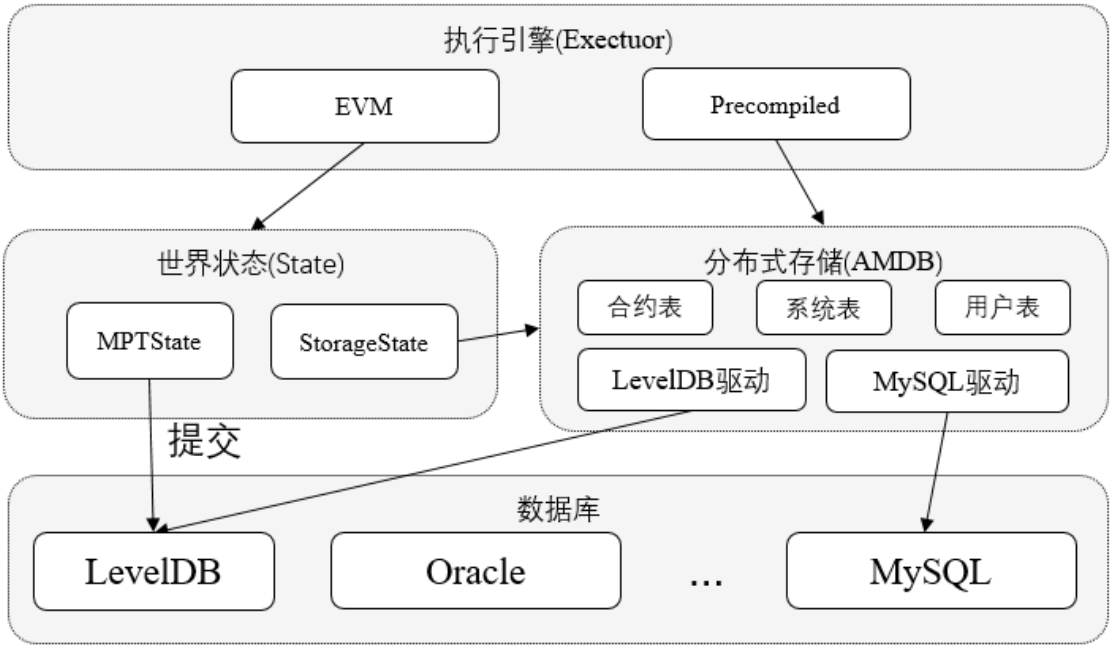
9.5 存储模块

FISCO BCOS继承以太坊存储的同时，引入了高扩展性、高吞吐量、高可用、高性能的分布式存储。存储模块主要包括两部分：

世界状态: 可进一步划分成 **MPTState** 和 **StorageState**

- **MPTState**: 使用MPT树存储账户的状态，与以太坊一致
- **StorageState**: 使用分布式存储的表结构存储账户状态，不存历史信息，去掉了对MPT树的依赖，性能更高

分布式存储(Advanced Mass Database，AMDB): 通过抽象表结构，实现了SQL和NOSQL的统一，通过实现对应的存储驱动，可以支持各类数据库，目前已经支持LevelDB和MySQL。



9.5.1 AMDB

分布式存储（Advanced Mass Database，AMDB）通过对表结构的设计，既可以对应到关系型数据库的表，又可以拆分使用KV数据库存储。通过实现对应于不同数据库的存储驱动，AMDB理论上可以支持所有关系型和KV的数据库。

- CRUD数据、区块数据默认情况下都保存在AMDB，无需配置，合约局部变量存储可根据需要配置为MPTState或StorageState，无论配置哪种State，合约代码都不需要变动。
- 当使用MPTState时，合约局部变量保存在MPT树中。当使用StorageState时，合约局部变量保存在AMDB表中。
- 尽管MPTState和AMDB最终数据都会写向RocksDB，但二者使用不同的RocksDB实例，没有事务性，因此当配置成使用MPTState时，提交数据时异常可能导致两个RocksDB数据不一致。

名词解释

Table

存储表中的所有数据。Table中存储AMDB主key到对应Entries的映射，可以基于AMDB主key进行增删改查，支持条件筛选。

Entries

Entries中存放主Key相同的Entry，数组。AMDB的主Key与Mysql中的主key不同，AMDB主key用于标示Entry属于哪个key，相同key的Entry会存放在同一个Entries中。

Entry

对应于表中的一行，每行以列名作为key，对应的值作为value，构成KV结构。每个Entry拥有自己的AMDB主key，不同Entry允许拥有相同的AMDB主key。

Condition

Table中的删改查接口支持传入条件，这三种接口会返回根据条件筛选后的结果。如果条件为空，则不做任何筛选。

数据更新或者插入过程中，需要根据主Key获取数据并将对数据更新或者append操作，然后再写回存储系统。因此，在一个主Key对应的的Entries中Entry个数很多的时候，执行效率会受到影响；同时，会加大内存的使用。所以，实际生产过程中主Key对应的的Entries中Entry个数不宜过多。

举例

以某公司员工领用物资登记表为例，解释上述名词。

解释如下：

- 表中**Name**是AMDB主key。
- 表中的每一行为一个Entry。一共有4个Entry，每个Entry以Map存储数据。4个Entry如下：
  - Entry1：`{Name:Alice，item_id:1001001,item_name:laptop}`
  - Entry2：`{Name:Alice，item_id:1001002,item_name:screen}`
  - Entry3：`{Name:Bob，item_id:1002001,item_name:macbook}`
  - Entry4：`{Name:Chris，item_id:1003001,item_name:PC}`
- Table中以**Name**为主key，存有3个Entries对象。第1个Entries中存有Alice的2条记录，第2个Entries中存有Bob的1条记录，第3个Entries中存有Chris的一条记录。
- 调用Table类的查询接口时，查接口需要指定AMDB主key和条件，设置查询的AMDB主key为Alice，条件为`item_id = 1001001`，会查询出Entry1。

AMDB表分类

表中的所有`entry`，都会有`_status_`,`_num_`,`_hash_`内置字段。

系统表

系统表默认存在，由存储驱动保证系统表的创建。

用户表

用户调用CRUD接口所创建的表，从2.2版本开始以`u_<TableName>`为表名，底层自动添加`u_`前缀。

StorageState账户表

从2.2版本开始以`c_`+`Address`作为表名。表中存储外部账户相关信息。表结构如下

9.5.2 StorageState

StorageState是一种使用AMDB实现的存储账户状态的方式。相比于MPTState主要有以下区别：

MPTState每个账户使用MPT树存储其数据，当历史数据逐渐增多时，会因为存储方式和磁盘IO导致性能问题。StorageState每个账户对应一个Table存储其相关数据，包括账户的`nonce`,`code`,`balance`等内容，而AMDB可以通过实现对应的存储驱动支持不同的数据库以提高性能，我们使用RocksDB测试发现，StorageState性能大约是MPTState的两倍。

9.5.3 MPT State

MPT State是以太坊上级经典的数据存储方式。通过MPT树的方式，将所有合约的数据组织起来，实现了对数据的查找和追溯。

重要： 推荐使用 **storage state**

MPT树

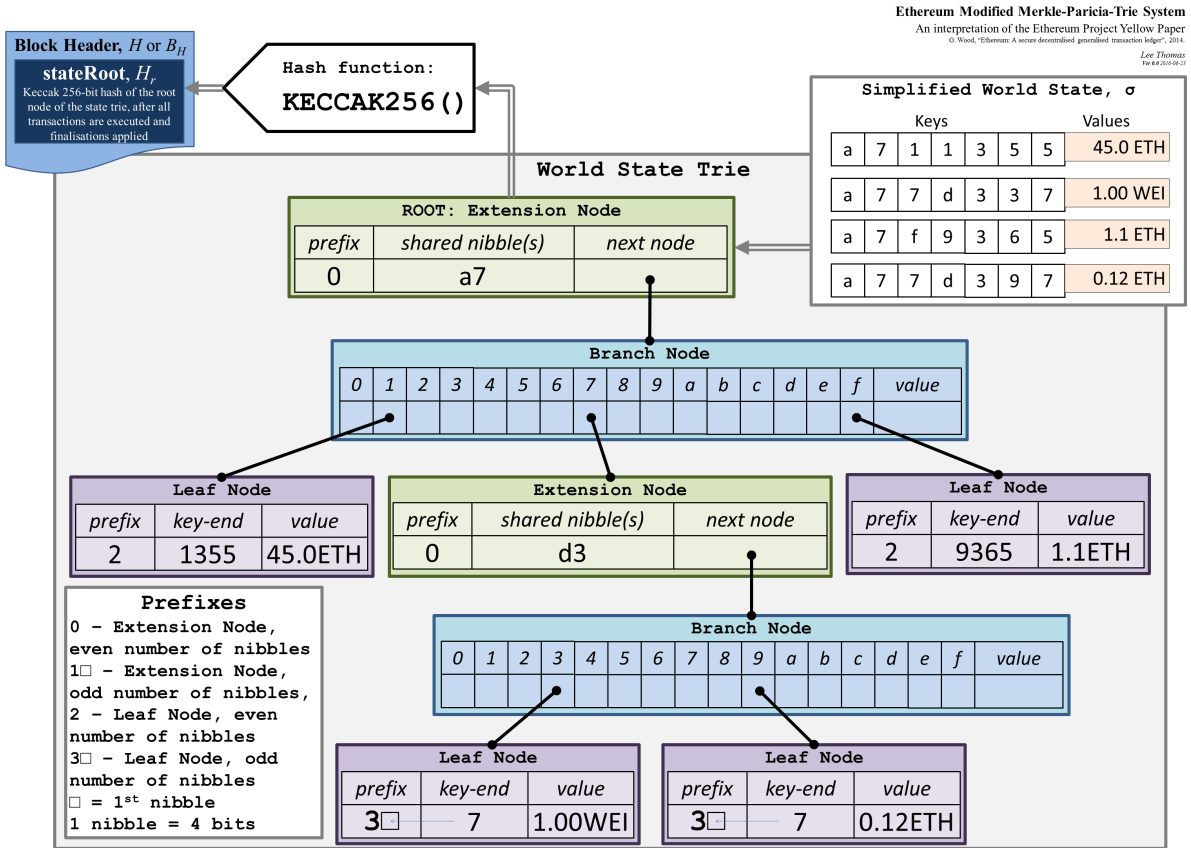
MPT(Merkle Patricia Trie)，是一种用hash索引数据的前缀树。

从宏观上来说，MPT树是一棵前缀树，用key查询value。通过key去查询value，就是用key去在MPT树上进行索引，在经过多个中间节点后，最终到达存储数据的叶子节点。

从细节上来说，MPT树，是一棵Merkle树，每个树上节点的索引，都是这个节点的hash值。在用key查找value的时候，是根据key在某节点内部，获取下一个需要跳转的节点的hash值，拿到下一个节点的hash值，才能从底层的数据库中取出下一个节点的数据，之后，再用key，去下一个节点中查询下下个节点的hash值，直至到达value所在的叶子节点。

当MPT树上某个叶子节点的数据更新后，此叶子节点的hash也会更新，随之而来的，是这个叶子节点回溯到根节点的所有中间节点的hash都会更新。最终，MPT根节点的hash也会更新。当要索引这个新的数据时，用MPT新的根节点hash，从底层数据库查出新的根节点，再往后一层层遍历，最终找到新的数据。而如果要查询历史数据，则可用老的树根hash，从底层数据库取出老的根节点，再往下遍历，就可查询到历史的数据。

MPT树的实现图（图片来自以太坊黄皮书）



状态 State

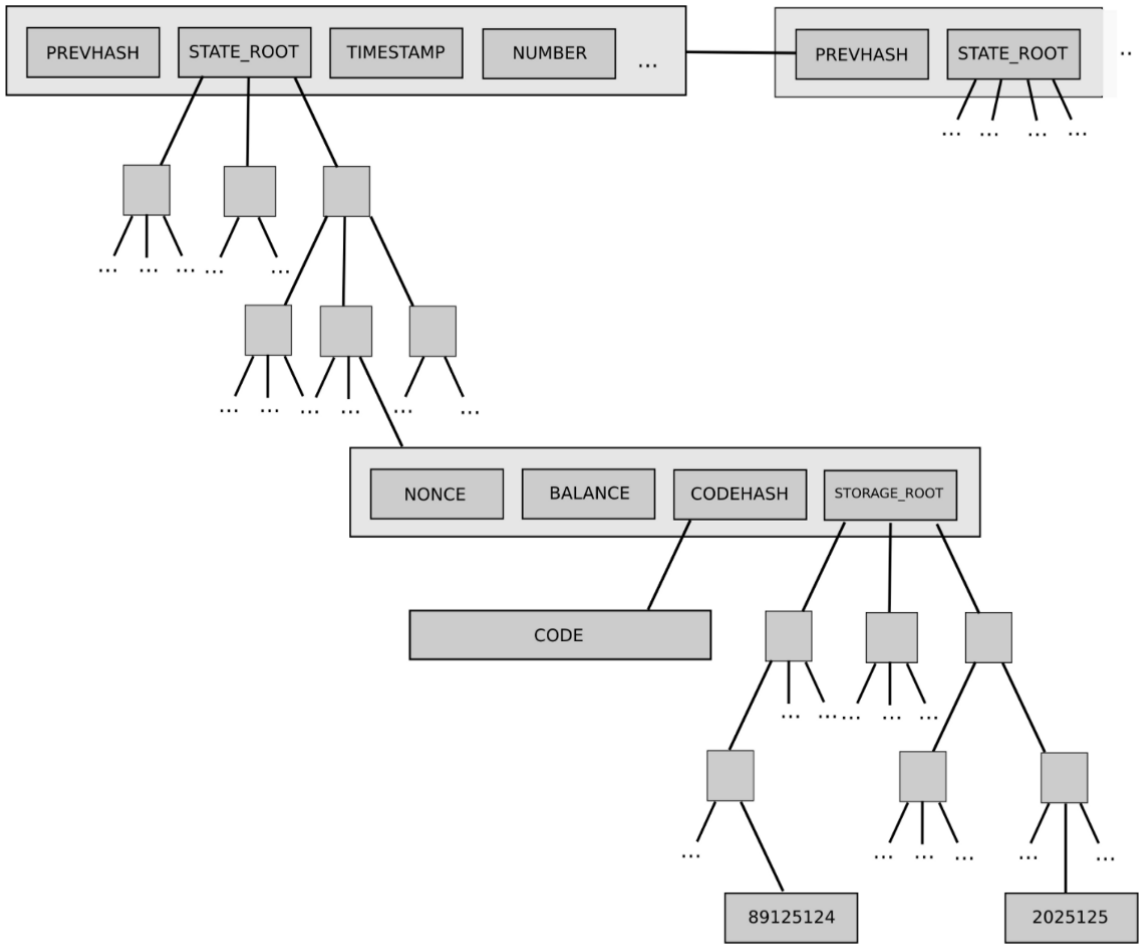
在以太坊上，数据是以account为单位存储的，每个account内，保存着这个合约(用户)的代码、参

数、nonce等数据。account的数据，通过account的地址（address）进行索引。以太坊上用MPT将这些address作为查询的key，实现了对account的查询。

随着account数据的改变，account的hash也进行改变。于此同时，MPT的根的hash也会改变。不同的时候，account的数据不同，对应的MPT的根就不同。此处，以太坊把这层含义进行了具体化，提出了"状态"的概念。把MPT根的hash，叫state root。不同的state root，对应着不同的"状态"，对应查询到不同的MPT根节点，再用account的address从不同的MPT根节点查询到此状态下的account数据。不同的state，拿到的MPT根节点不同，查询的account也许会有不同。

state root是区块中的一个字段，每个区块对应着不同的"状态"。区块中的交易会对account进行操作，进而改变account中的数据。不同的区块下，account的数据有所不同，即此区块的状态有所不同，具体的，是state root不同。从某个区块中取出这个区块的state root，查询到MPT的根节点，就能索引到这个区块当时account的数据历史。

(图片来自以太坊白皮书)



Trade Off

MPT State的引入，是为了实现对数据的追溯。根据不同区块下的state root，就能查询到当时区块下account的历史信息。而MPT State的引入，带来了大量hash的计算，同时也打散了底层数据的存储的连续性。在性能方面，MPT State存在着天然的劣势。可以说，MPT State是极致的追求可追溯性，而大大的忽略了性能。

在FISCO BCOS的业务场景中，性能与可追溯性相比，性能更为重要。FISCO BCOS对底层的存储进行了重新的设计，实现了Storage State。Storage State牺牲了部分的可追溯性，但带来了性能上的提升。

9.6 安全控制

为了保障节点间通信安全性，以及对节点数据访问的安全性，FISCO BCOS引入了节点准入机制、CA黑名单和权限控制三种机制，在网络和存储层面上做了严格的安全控制。

网络层面安全控制

- 节点使用 **SSL连接** ，保障了通信数据的机密性
- 引入 **网络准入机制** ，可将指定群组的作恶节点从共识节点列表或群组中删除，保障了系统安全性
- 通过 **群组白名单机制** ，保证每个群组仅可接收相应群组的消息，保证群组间通信数据的隔离性
- 引入 **CA黑名单机制** ，可及时与作恶节点断开网络连接
- 提出 **分布式存储权限控制** 机制，灵活、细粒度地控制外部账户部署合约和创建、插入、删除和更新用户表的权限。

存储层面安全控制

基于分布式存储，提出分布式存储权限控制的机制，以灵活、细粒度的方式进行有效的权限控制，设计并实现了权限控制机制限制外部账户(tx.origin)对存储的访问，权限控制范围包括合约部署、表的创建、表的写操作。

9.6.1 节点准入管理介绍

本文档对节点准入管理进行介绍性说明，实践方法参见《节点准入管理操作文档》。

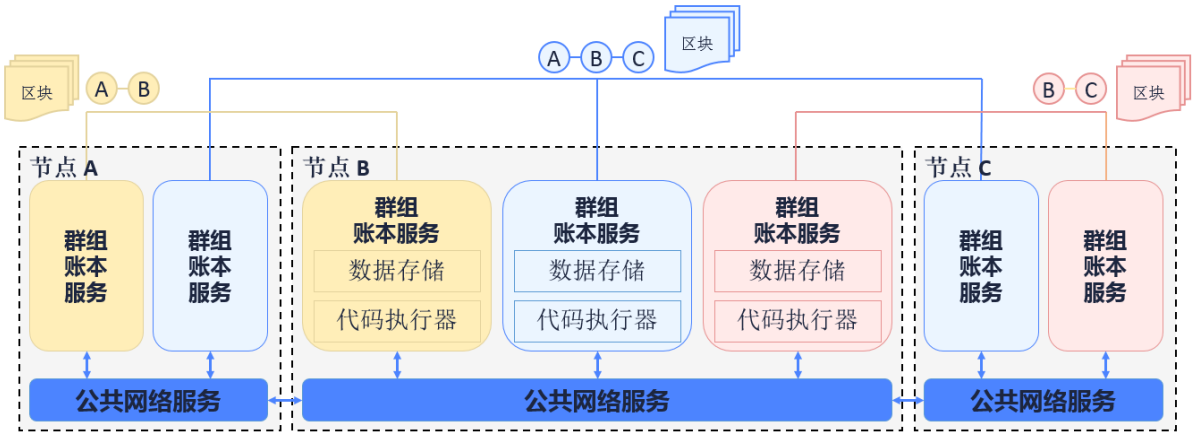
概述

单链多账本

区块链技术是一种去中心化、公开透明的分布式数据存储技术，能够降低信任成本，实现安全可靠的数据交互。然而区块链的交易数据面临着隐私泄露威胁：

- 对于公有链，一节点可任意加入网络，从全局账本中获得所有数据；
- 对于联盟链，虽有网络准入机制，但节点加入区块链后即可获取全局账本的数据。

作为联盟链的FISCO BCOS，对链上隐私这一问题，提出了**单链多账本**的解决方案。FISCO BCOS通过引入**群组**概念，使联盟链从原有一链一账本的存储/执行机制扩展为一链多账本的存储/执行机制，基于群组维度实现同一条链上的数据隔离和保密。



如上图所示，节点ABC加入蓝色群组，并共同维护蓝色账本; 节点B和C加入粉色群组并维护粉红色账本; 节点A和B加入黄色群组并维护黄色账本。三个群组间共享公共的网络服务，但各群组有各自独立的账本存储及交易执行环境。客户端将交易发到节点所属的某个群组上，该群组内部对交易及数据进行共识并存储，其他群组对该交易无感知不可见。

节点准入机制

基于群组概念的引入，节点准入管理可分为**网络准入机制**和**群组准入机制**。准入机制的规则记录在配置中，节点启动后将读取配置信息实现网络及群组的准入判断。

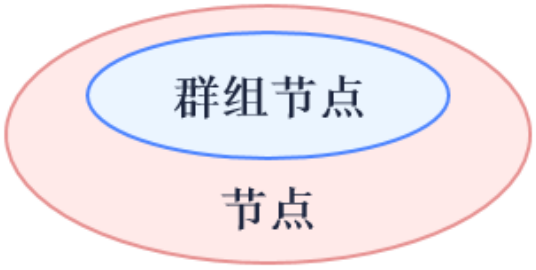
名词解释

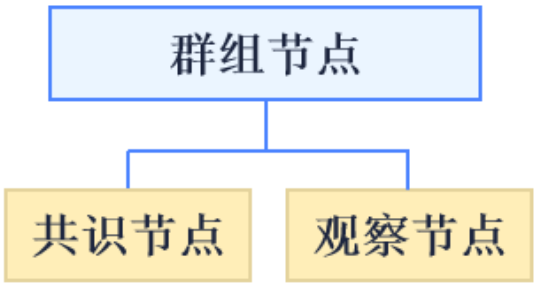
节点类型

本文档所讨论的节点为已完成网络准入可进行P2P通信的节点。**网络准入过程涉及P2P节点连接列表添加和证书验证。**

- **群组节点**：完成网络准入并加入群组的节点。群组节点只能是共识节点和观察节点两者之一。其中共识节点参与共识出块和交易/区块同步，观察节点只参与区块同步。**群组节点准入过程涉及动态增删节点的交易发送。**
- **游离节点**：完成网络准入但没有加入群组的节点。**游离节点尚未通过群组准入，不参与共识和同步。**

节点关系如下：



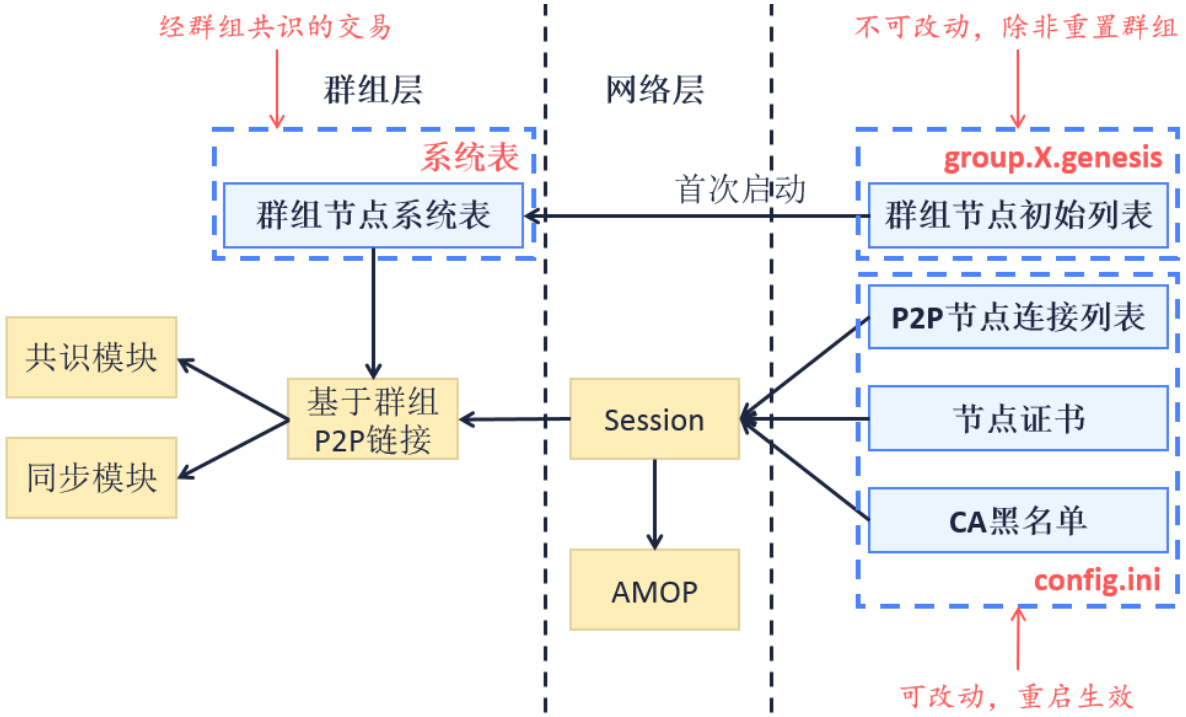


配置类型

节点准入配置项

涉及节点转入管理相关的配置项有：**P2P节点连接列表**，**节点证书**，**CA黑名单**，**群组节点初始列表**和**群组节点系统表**。

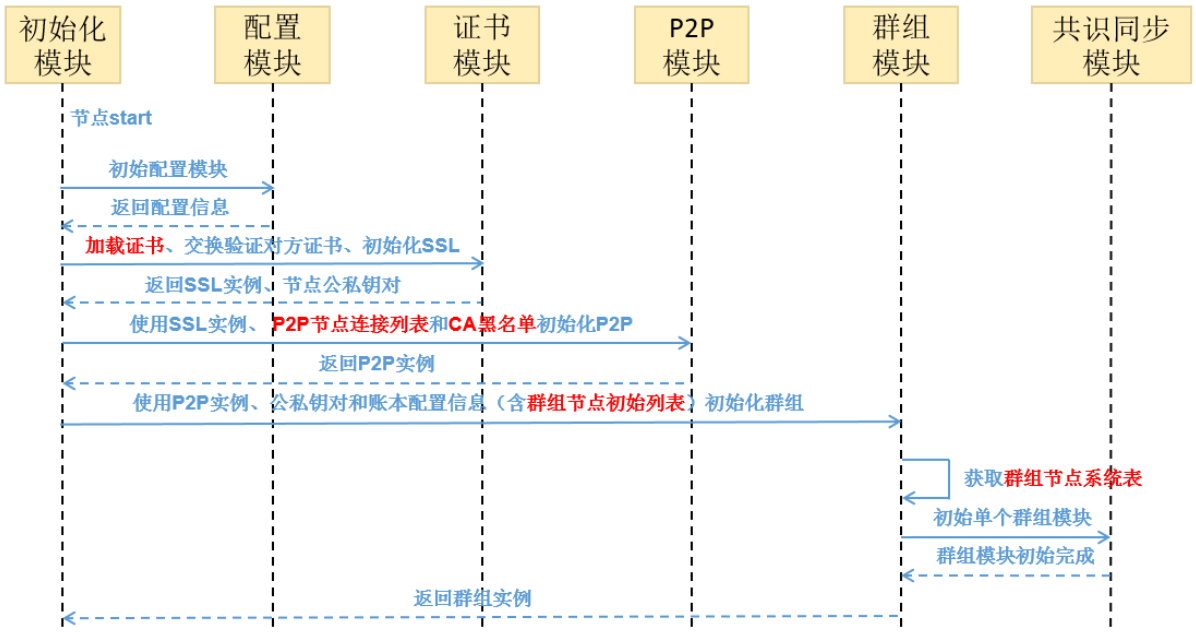
模块架构



配置项及系统模块关系图如上，箭头方向A->B表示B模块依赖A模块的数据，同时B模块晚于A模块初始化。

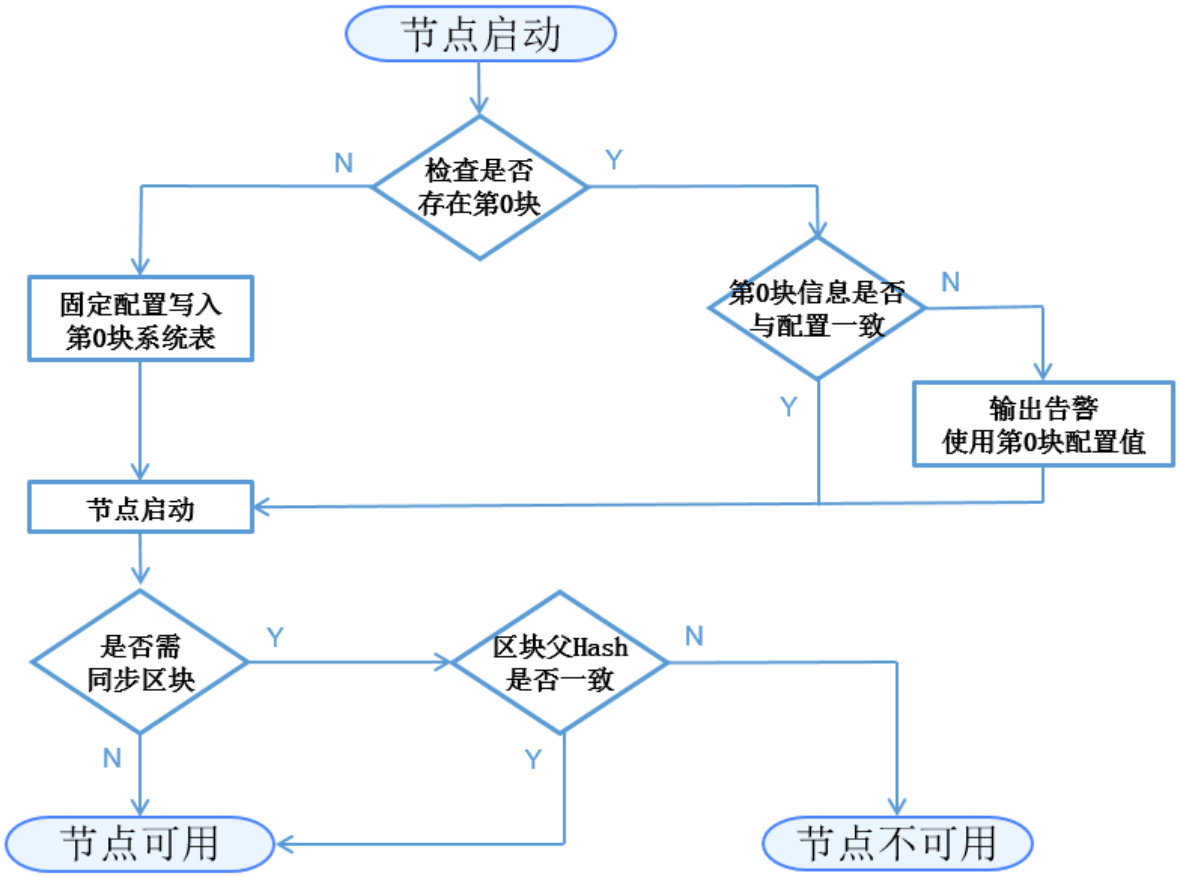
核心流程

一般初始化流程



首次初始化流程

节点在首次启动时，对其所属的各个群组，以群组为单位将固定配置文件的内容写入第0块并直接提交上链。初始化的具体逻辑为：



这一阶段需写入的与节点准入管理相关的配置内容有：**群组节点初始列表->群组节点系统表**。

说明：

- 同一账本的所有节点的第0块需一致，即**固定配置文件**均一致；
- 节点后续的每次启动均检查第0块信息是否与固定配置文件一致。如果固定配置文件被修改，节点再次启动将输出告警信息，但不会影响群组正常运作。

基于CA黑名单的节点建连流程

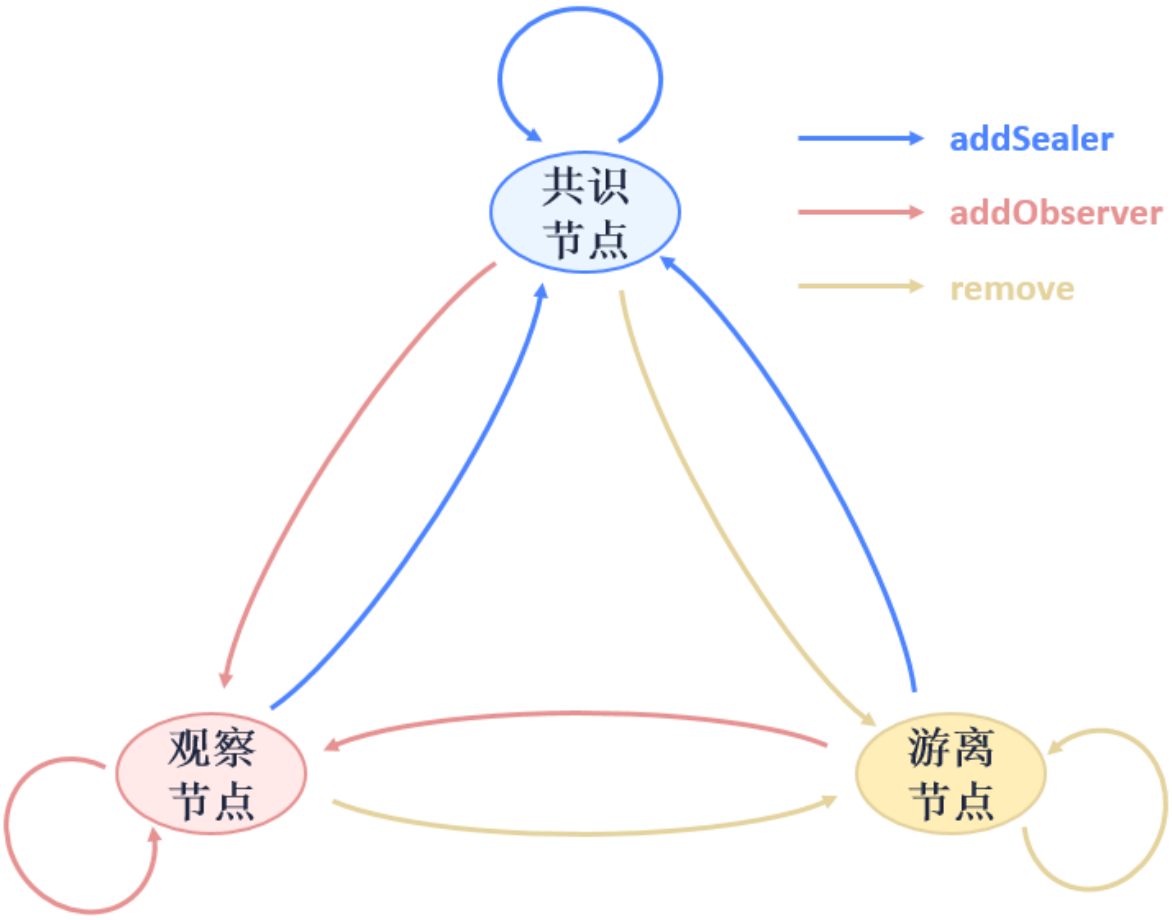
SSL认证用于确定节点之间是否许可加入某条链。一条链上的节点均信任可信的第三方（节点证书的颁发者）。

FISCO BCOS要求实现**SSL双向认证**。节点在handshake过程中，从对方节点提供的证书中获取对方节点的nodeID，检查该nodeID是否在自身的CA黑名单。如存在，关闭该connection，如不在，建立session。

CA黑名单机制也支持**SSL单向认证**的场景，作用时机是：节点在session建立后，可从session中获取对方节点的nodeID进行判断，如果nodeID在自身的CA黑名单中，将已建立的session断连。

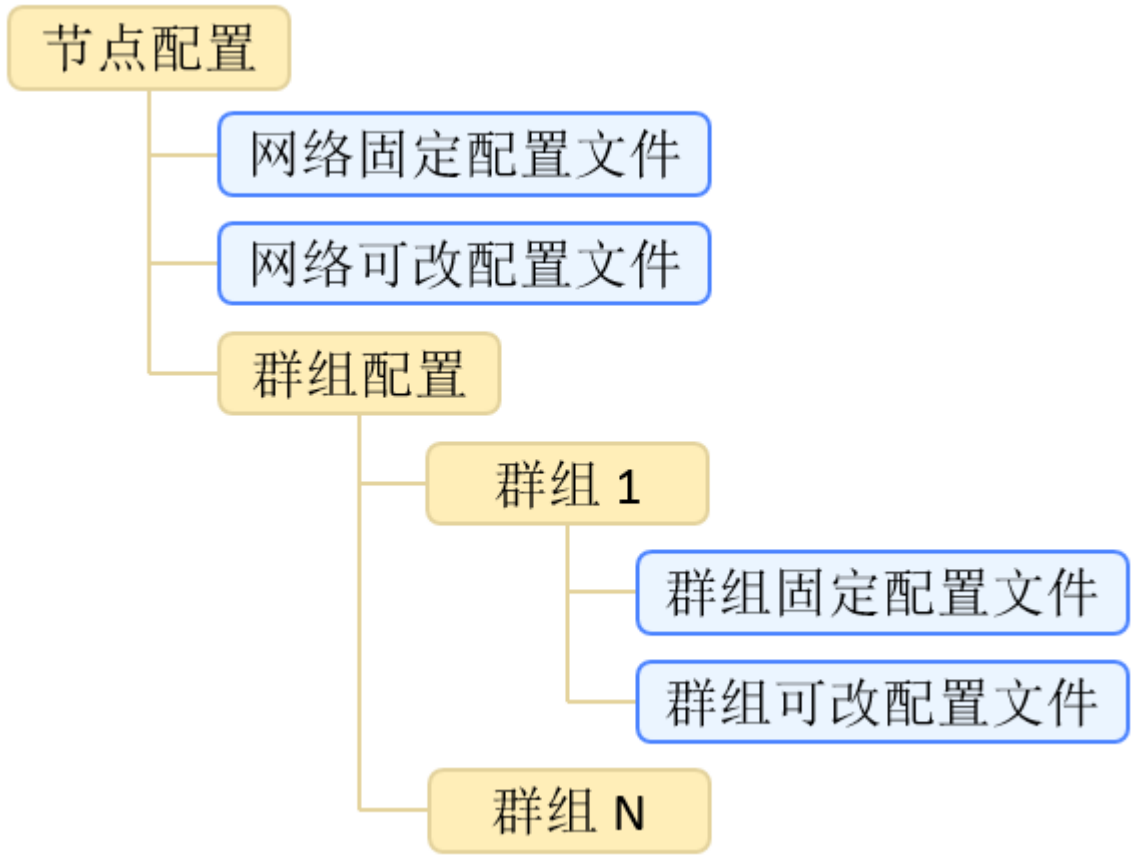
节点相关类型及其转换操作

三种节点类型（共识节点+观察节点+游离节点）可通过相关接口进行如下转换：



接口及配置描述

节点配置文件层级



配置文件的组织规则为：**各群组的配置独立**、**固定配置和可改配置相独立**。目前使用的文件有**网络可改配置文件**`config.ini`、**群组固定配置文件**`group.N.genesis`和**群组可改配置文件**`group.N.ini`，其中N为节点所在的群组号。对于**网络/群组可改配置文件**，如果文件中没有显式定义某配置项的值，程序将使用该配置项的默认值。

配置文件示例

对于**网络可改配置文件**`config.ini`，节点准入管理涉及**P2P节点连接列表[p2p]**、**节点证书[network_security]**、**CA黑名单[certificate_blacklist]**。`[certificate_blacklist]`可缺少。配置项举例如下：

注解： 为便于开发和体验，p2p模块默认监听IP是 *0.0.0.0* ，出于安全考虑，请根据实际业务网络情况，修改为安全的监听地址，如：内网IP或特定的外网IP

```
[p2p]
    ;p2p listen ip
    listen_ip=0.0.0.0
    ;p2p listen port
    listen_port=30300
    ;nodes to connect
    node.0=127.0.0.1:30300
    node.1=127.0.0.1:30301
    node.2=127.0.0.1:30302
    node.3=127.0.0.1:30303
```

(continues on next page)

(续上页)

```
;certificate blacklist
[certificate_blacklist]
    ;crl.0 should be nodeid, nodeid's length is 128
    ;crl.0=

;certificate configuration
[network_security]
    ;directory the certificates located in
    data_path=conf/
    ;the node private key file
    key=node.key
    ;the node certificate file
    cert=node.crt
    ;the ca certificate file
    ca_cert=ca.crt
```

对于**群组固定配置文件**`group.N.genesis`，节点准入管理涉及**群组节点初始列表[consensus]**。配置项举例如下：

```
;consensus configuration
[consensus]
    ;consensus algorithm type, now support PBFT(consensus_type=pbft) and␣
↪Raft(consensus_type=raft)
    consensus_type=pbft
    ;the max number of transactions of a block
    max_trans_num=1000
    ;the node id of leaders
    node.
↪0=79d3d4d78a747b1b9e59a3eb248281ee286d49614e3ca5b2ce3697be2da72cfa82dcd314c0f04e1f590da8db0b97d
    node.
↪1=da527a4b2aeae1d354102c6c3ffdfb54922a092cc9acbdd555858ef89032d7be1be499b6cf9a703e546462529ed9e
    node.
↪2=160ba08898e1e25b31e24c2c4e3c75eed996ec56bda96043aa8f27723889ab774b60e969d9bd25d70ea8bb8779b70
    node.
↪3=a968f1e148e4b51926c5354e424acf932d61f67419cf7c5c00c7cb926057c323bee839d27fe9ad6c75386df52ae2b
```

群组节点系统表定义

群组系统表接口定义

群组系统表实现群组层的白名单机制（对比CA黑名单实现网络的黑名单机制）。群组系统表提供的接口有：

```
contract ConsensusSystemTable
{
    // 修改一节点为共识节点
    function addSealer(string nodeID) public returns(int256);
    // 修改一节点为观察节点
    function addObserver(string nodeID) public returns(int256);
    // 把该节点从群组系统表中移除
    function remove(string nodeID) public returns(int256);
}
```

功能展望

- **可改配置**目前为修改后重启生效，后续可实现动态加载，修改实时生效；
- **CA黑名单**目前实现了基于节点的黑名单，后续可考虑基于机构的黑名单。

9.6.2 CA黑白名单介绍

本文档对黑、白名单进行介绍性说明，实践方法参见《CA黑白名单操作手册》。

名词解释

CA黑名单

- 别称**证书拒绝列表**（certificate blacklist，简称CBL）。CA黑名单基于`config.ini`文件中`[certificate_blacklist]`配置的NodeID进行判断，拒绝此NodeID节点发起的连接。

CA白名单

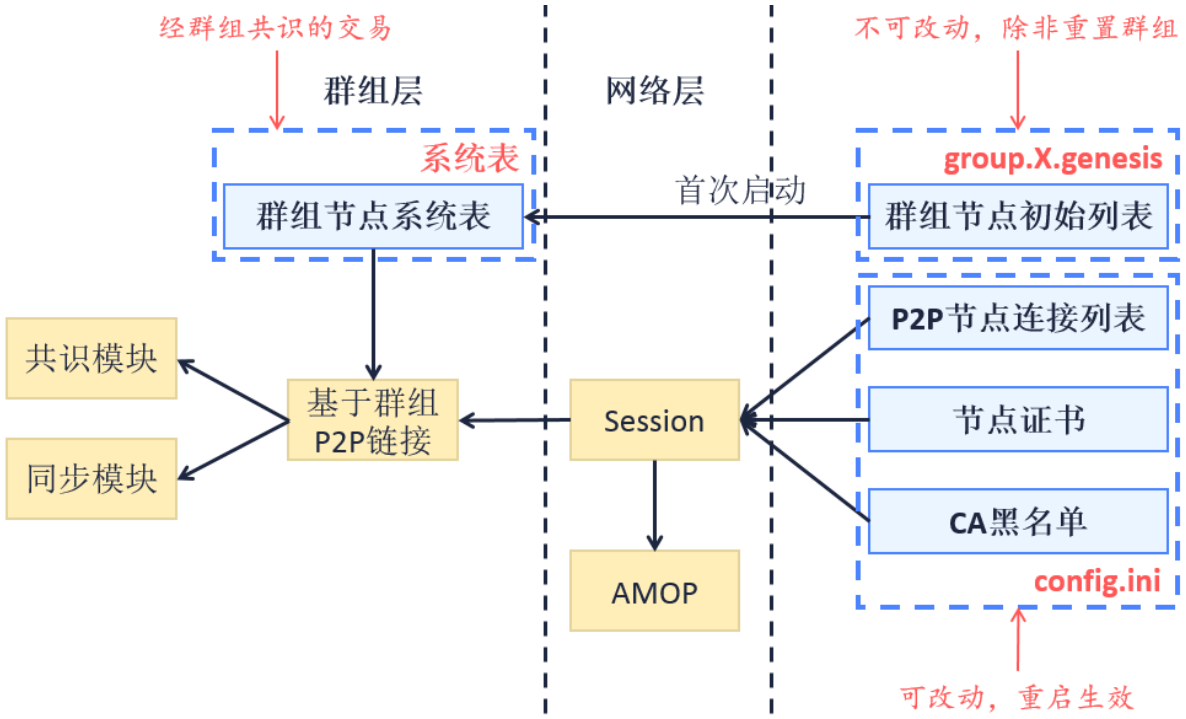
- 别称**证书接受列表**（certificate whitelist，简称CAL）。CA白名单基于`config.ini`文件中`[certificate_whitelist]`配置的NodeID进行判断，拒绝除白名单外所有节点发起的连接。

CA黑、白名单所属的配置类型

- 基于**作用范围**（网络配置/账本配置）维度可划分为**网络配置**，影响整个网络的节点连接建立过程；
- 基于**是否可改**（可改配置/固定配置）维度可划分为**可改配置**，内容可改，重启后生效；
- 基于**存放位置**（本地存储/链上存储）维度可划分为**本地存储**，内容记录在本地，不存于链上。

模块架构

下图表示CA黑名单所涉及的模块及其关系。图例A->B表示B模块依赖A模块的数据，同时B模块晚于A模块初始化。白名单的架构与黑名单相同。



底层实现SSL双向验证。节点在handshake过程中，通过对方提供的证书获取对方节点的nodeID，检查该nodeID与节点配置的黑、白名单是否有关系。如果根据黑、白名单的配置，拒绝该关闭的connection，继续后续流程。

拒绝逻辑

- 黑名单：拒绝写在黑名单中的节点连接

- 白名单：拒绝所有未配置在白名单中的节点连接。白名单为空表示不开启，接受任何连接。

优先级

黑名单的优先级高于白名单。例如，白名单里配置了A，B，C，会拒绝掉D的连接，若黑名单里同时配了A，则A也会被拒绝连接。

影响范围

- CA黑、白名单对网络层的P2P节点连接及AMOP功能有显著影响，**使之失效**；
- 对账本层的共识和同步功能有潜在影响，**影响共识及同步消息/数据的转发**。

配置格式

黑名单

节点`config.ini`配置中增加`[certificate_blacklist]`路径（`[certificate_blacklist]`在配置中可选）。CA黑名单内容为节点NodeID列表，node.X为本节点拒绝连接的对方节点NodeID。CA黑名单的配置格式示例如下。

```
[certificate_blacklist]
    crl.
↪0=4d9752efbb1de1253d1d463a934d34230398e787b3112805728525ed5b9d2ba29e4ad92c6fcde5156ede8baa5aca3
    crl.
↪1=af57c506be9ae60df8a4a16823fa948a68550a9b6a5624df44afcd3f75ce3afc6bb1416bcb7018e1a22c5ecbd016a
```

白名单

节点`config.ini`配置中增加`[certificate_whitelist]`路径（`[certificate_whitelist]`在配置中可选）。CA白名单内容为节点NodeID列表，node.X为本节点可接受连接的对方节点NodeID。CA白名单的配置格式示例如下。

```
[certificate_whitelist]
    cal.
↪0=4d9752efbb1de1253d1d463a934d34230398e787b3112805728525ed5b9d2ba29e4ad92c6fcde5156ede8baa5aca3
    cal.
↪1=af57c506be9ae60df8a4a16823fa948a68550a9b6a5624df44afcd3f75ce3afc6bb1416bcb7018e1a22c5ecbd016a
```

9.6.3 基于角色的权限控制

当前基于表的权限控制，由于涉及到许多系统表，要求用户对底层的逻辑有一定理解，使用门槛较高。从2.5.0版本开始，基于已有的表权限控制模型，新增了ChainGovernance预编译合约，用于实现基于角色的权限控制。

角色定义

分为治理方、运维方、监管方和业务方。**考虑到权责分离，治理方、运维方和开发方权责分离，角色互斥。**

- 治理方：拥有投票权，可以参与治理投票（AUTH_ASSIGN_AUTH），可以增删节点、修改链配置、添加撤销运维、冻结解冻合约、对用户表的写权限控制。链级别的可变配置的权限。
- 运维方：由治理方添加运维账号，运维账号可以部署合约、创建表、管理合约版本、冻结解冻本账号部署的合约。
- 业务方：业务方账号由运维添加到某个合约，可以调用该合约的写接口。
- 监管方：监管方监管链的运行，能够获取链运行中权限变更的记录、能够获取需要审计的数据

权限

以下简称治理账号为**委员**。

- 权限项命名符合动宾结构。
- 增删委员、修改委员权重、修改生效阈值三个操作，需要**有效投票权重/总权重>生效阈值**生效。其中**总权重=SUM(委员权重)**
- 治理账号可以添加运维账号，但治理账号不拥有运维的权限
- 运维账号可以为某个合约添加业务账号，但运维账号没有业务账号权限

权限项

计票与生效

- 所有治理操作需要**有效投票数/委员数>生效阈值**才能生效
- 每次投票操作，如果是委员投票，则记录操作内容和投票委员，不重复计票
- 每次投票操作，计票结束后，计算**有效投票数/委员数**，如果大于此操作的生效阈值，则对应操作生效，写入
- **投票设置过期时间，根据块高，blockLimit的10倍，固定不可改**

功能列表

1. 增删委员计票与生效
2. 修改委员权重计票与生效
3. 修改生效阈值计票与生效
4. 委员增删运维
5. 委员解冻冻结合约
6. 委员增删节点

7. 委员修改系统配置
8. 权限项默认阈值存储
9. 运维部署合约的权限
10. 运维管理合约版本的权限

委员相关

```
function grantCommitteeMember(address user) public returns (int256);
function updateCommitteeMemberWeight(address user, int256 weight) public returns␣
↪(int256);
function revokeCommitteeMember(address user) public returns (int256);
function listCommitteeMembers() public returns (string);
// threshold取值范围[0,100]
function updateThreshold(int256 threshold) public returns (int256);
```

运维相关

```
function grantOperator(address user) public returns (int256);
function revokeOperator(address user) public returns (int256);
function listOperator() public returns (string);
```

数据结构

表:_sys_committee_votes_

- key是账户，value记录grant/revoke，origin记录投票人
- key是账户_update_weight，value记录目标权重，origin记录投票人
- key是账户_weight，value记录权重
- key是auth_threshold，value记录当前阈值
- key是update_auth_threshold，value记录目标阈值，origin记录投票人

9.6.4 权限控制

权限控制介绍

与可自由加入退出、自由交易、自由检索的公有链相比，联盟链有准入许可、交易多样化、基于商业上隐私及安全考虑、高稳定性等要求。因此，联盟链在实践过程中需强调“权限”及“控制”的理念。

为体现“权限”及“控制”理念，FISCO BCOS平台基于分布式存储，提出分布式存储权限控制的机制，可以灵活，细粒度的方式进行有效的权限控制，为联盟链的治理提供重要的技术手段。分布式权限控制基于外部账户(tx.origin)的访问机制，对包括合约部署，表的创建，表的写操作（插入、更新和删除）进行权限控制，表的读操作不受权限控制。 在实际操作中，每个账户使用独立且唯一的公私钥对，发起交易时使用其私钥进行签名，接收方可通过公钥验签知道交易具体是由哪个账户发出，实现交易的可控及后续监管的追溯。

权限控制规则

权限控制规则如下：

1. 权限控制的最小粒度为表，基于外部账户进行控制。

2. 使用白名单机制，未配置权限的表，默认完全放开，即所有外部账户均有读写权限。
3. 权限设置利用权限表（_sys_table_access_）。权限表中设置表名和外部账户地址，则表明该账户对该表有读写权限，设置之外的账户对该表仅有读权限。

权限控制分类

分布式存储权限控制分为对用户表和系统表的权限控制。用户表指用户合约所创建的表，用户表均可以设置权限。系统表指FISCO BCOS区块链网络内置的表，系统表的设计详见存储文档。系统表的权限控制如下所示：

针对用户表和每个系统表，SDK分别实现三个接口进行权限相关操作：

- 用户表：
  - **public String grantUserTableManager(String tableName, String address)**： 根据用户表名和外部账户地址设置权限信息。
  - **public String revokeUserTableManager(String tableName, String address)**： 根据用户表名和外部账户地址去除权限信息。
  - **public List<PermissionInfo> listUserTableManager(String tableName)**： 根据用户表名查询设置的权限记录列表(每条记录包含外部账户地址和生效块高)。
- _sys_tables_表：
  - **public String grantDeployAndCreateManager(String address)**： 增加外部账户地址的部署合约和创建用户表权限。
  - **public String revokeDeployAndCreateManager(String address)**： 移除外部账户地址的部署合约和创建用户表权限。
  - **public List<PermissionInfo> listDeployAndCreateManager()**： 查询拥有部署合约和创建用户表权限的权限记录列表。
- _sys_table_access_表：
  - **public String grantPermissionManager(String address)**： 增加外部账户地址的管理权限的权限。
  - **public String revokePermissionManager(String address)**： 移除外部账户地址的管理权限的权限。
  - **public List<PermissionInfo> listPermissionManager()**： 查询拥有管理权限的权限记录列表。
- _sys_consensus_表：
  - **public String grantNodeManager(String address)**： 增加外部账户地址的节点管理权限。
  - **public String revokeNodeManager(String address)**： 移除外部账户地址的节点管理权限。
  - **public List<PermissionInfo> listNodeManager()**： 查询拥有节点管理的权限记录列表。
- _sys_cns_表：
  - **public String grantCNSManager(String address)**： 增加外部账户地址的使用CNS权限。
  - **public String revokeCNSManager(String address)**： 移除外部账户地址的使用CNS权限。
  - **public List<PermissionInfo> listCNSManager()**： 查询拥有使用CNS的权限记录列表。
- _sys_config_表：
  - **public String grantSysConfigManager(String address)**： 增加外部账户地址的系统参数管理权限。
  - **public String revokeSysConfigManager(String address)**： 移除外部账户地址的系统参数管理权限。

- **public List<PermissionInfo> listSysConfigManager()**：查询拥有系统参数管理的权限记录列表。

设置和移除权限接口返回json字符串，包含code和msg字段，当无权限操作时，其code定义-50000，msg定义为“permission denied”。当成功设置权限时，其code为0，msg为“success”。

数据定义

权限信息以系统表的方式进行存储，权限表表名为_sys_table_access_，其字段信息定义如下：

字段	类型	是否为空	主键	描述
table_name	string	No	PRI	表名称
address	string	No		外部账户地址
enable_num	string	No		权限设置生效区块高度
status	string	No		分布式存储通用字段，“0”表示可用，“1”表示移除

其中，对权限表的插入或更新，当前区块不生效，在当前区块的下一区块生效。状态字段为“0”时，表示权限记录处于正常生效状态，为“1”时表示已删除，即表示权限记录处于失效状态。

权限控制设计

权限控制功能设计

根据交易信息确定外部账户，待操作的表以及操作方式。待操作的表为用户表或系统表。系统表用于控制区块链的系统功能，用户表用于控制区块链的业务功能，如下图所示。外部账户通过查询权限表获取权限相关信息，确定权限后再操作相关的用户表和权限表，从而可以控制相关的系统功能和业务功能。

权限控制流程设计

权限控制的流程如下：首先由客户端发起交易请求，节点获取交易数据，从而确定外部账户和待操作的表以及操作表的方式。如果判断操作方式为写操作，则检查该外部账户针对操作的表的权限信息（权限信息从权限表中查询获取）。若检查有权限，则执行写操作，交易正常执行；若检查无权限，则拒绝写操作，返回无权限信息。如果判断操作方式为读操作，则不检查权限信息，正常执行读操作，返回查询数据。流程图如下。

权限控制工具

FISCO BCOS的分布式存储权限控制有如下使用方式：

- 针对普通用户，通过控制台命令使用权限功能，具体参考权限控制使用手册。
- 针对开发者，SDK根据权限控制的用户表和每个系统表均实现了三个接口，分别是授权，撤销和查询权限接口。可以调用SDK API的PermissionService接口使用权限功能。

9.7 P2P网络

9.7.1 设计目标

FISCO BCOS P2P模块提供高效、通用和安全的网络通信基础功能，支持区块链消息的单播、组播和广播，支持区块链节点状态同步，支持多种协议。

9.7.2 P2P主要功能

- 区块链节点标识

通过区块链节点标识唯一标识一个区块链节点，在区块链网络上通过区块链节点标识对区块链节点进行寻址

- 管理网络连接

维持区块链网络上区块链节点间的TCP长连接，自动断开异常连接，自动发起重连

- 消息收发

在区块链网络的区块链节点间，进行消息的单播、组播或广播

- 状态同步

在区块链节点间同步状态

9.7.3 区块链节点标识

区块链节点标识由ECC算法的公钥生成，每个区块链节点必须有唯一的ECC密钥对，区块链节点标识在区块链网络中唯一标识一个区块链节点

通常情况下，一个节点要加入区块链网络，至少要准备三个文件：

- node.key 节点密钥，ECC格式
- node.crt 节点证书，由CA颁发
- ca.crt CA证书，CA机构提供

区块链节点除了有唯一区块链节点标识，还能关注Topic，供寻址使用

区块链节点寻址：

- 区块链节点标识寻址

通过区块链节点标识，在区块链网络中定位唯一的区块链节点

- Topic寻址

通过Topic，在区块链网络中定位一组关注该Topic的节点

9.7.4 管理网络连接

区块链节点间，会自动发起和维持TCP长连接，在系统故障、网络异常时，主动发起重连

区块链节点间建立连接时，会使用CA证书进行认证

连接建立流程

9.7.5 消息收发

区块链节点间消息支持单播、组播和广播

- 单播，单个区块链节点向单个区块链节点发送消息，通过区块链节点标识寻址
- 组播，单个区块链节点向一组区块链节点发送消息，通过Topic寻址
- 广播，单个区块链节点向所有区块链节点发送消息

单播流程

组播流程

广播流程

9.7.6 状态同步

每个节点会维护自身的状态，并将状态的Seq在全网定时广播，与其它节点同步

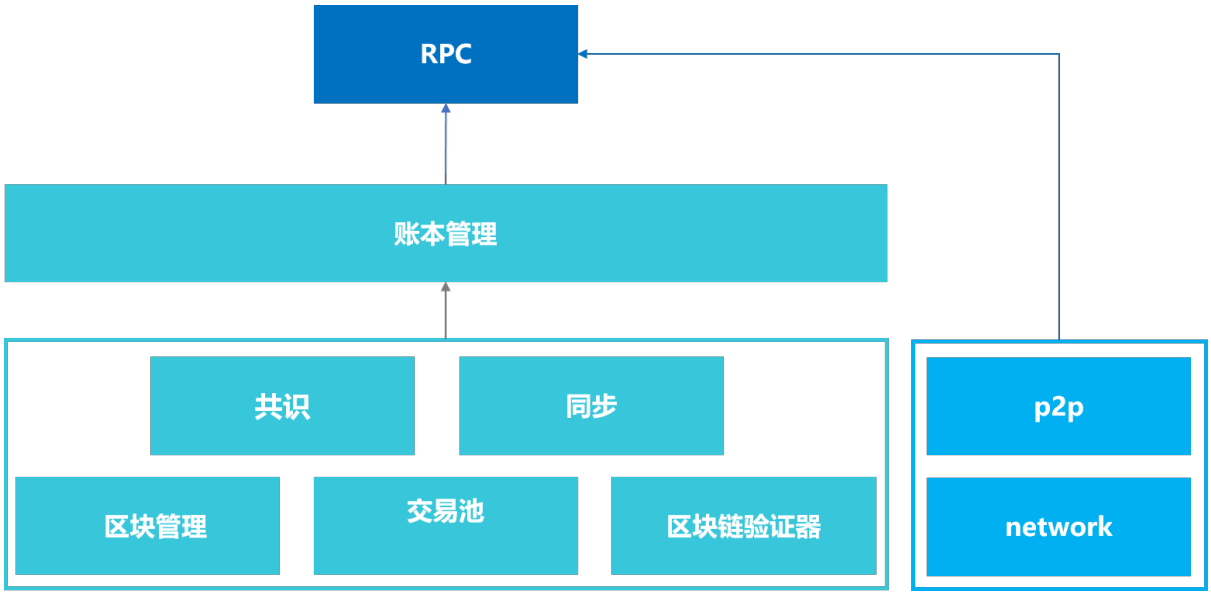
9.8 RPC

RPC(Remote Procedure Call，远程过程调用)是客户端与区块链系统交互的一套协议和接口。用户通过RPC接口可查询区块链相关信息（如块高、区块、节点连接等）和发送交易。

9.8.1 1 名词解释

- JSON(JavaScript Object Notation)：一种轻量级的数据交换格式。它可以表示数字、字符串、有序序列和键值对。
- JSON-RPC：一种无状态、轻量级的远程过程调用协议。 该规范主要定义了几个数据结构及其处理规则。它允许运行在基于socket，http等诸多不同消息传输环境的同一进程中。它使用JSON(RFC 4627)作为数据格式。FISCO BCOS采用JSON-RPC 2.0协议。

9.8.2 2 模块架构



RPC模块负责提供FISCO BCOS的外部接口，客户端通过RPC发送请求，RPC通过调用账本管理模块和p2p模块获取相关响应，并将响应返回给客户端。其中账本管理模块通过多账本机制管理区块链底层的相关模块，具体包括共识模块，同步模块，区块管理模块，交易池模块以及区块验证模块。

9.8.3 3 数据定义

3.1 客户端请求

客户端请求发送至区块链节点会触发RPC调用，客户端请求包括下列数据成员：

- jsonrpc: 指定JSON-RPC协议版本的字符串，必须准确写为“2.0”。
- method: 调用方法的名称。
- params: 调用方法所需要的参数，方法参数可选。由于FISCO BCOS 2.0启用了多账本机制，因此本规范要求传入的第一个参数必须为群组ID。
- id: 已建立客户端的唯一标识ID，ID必须是一个字符串、数值或NULL空值。如果不包含该成员则被认定为是一个通知。

RPC请求包格式示例:

```
{"jsonrpc": "2.0", "method": "getBlockNumber", "params": [1], "id": 1}
```

注：

- 在请求对象中不建议使用NULL作为id值，因为该规范将使用空值认定为未知id的请求。
- 在请求对象中不建议使用小数作为id值，因为具有不确定性。

3.2 服务端响应

当发起一个RPC调用时，除通知之外，区块链节点都必须回复响应。响应表示为一个JSON对象，使用以下成员：

- jsonrpc: 指定JSON-RPC协议版本的字符串。必须准确写为“2.0”。
- result: 正确结果字段。该成员在响应处理成功时必须包含，当调用方法引起错误时必须不包含该成员。
- error: 错误结果字段。该成员在失败是必须包含，当没有引起错误的时必须不包含该成员。该成员参数值必须为*3.3*节中定义的对象。
- id: 响应id。该成员必须包含，该成员值必须与对应客户端请求中的id值一致。若检查请求对象的id错误（例如参数错误或无效请求），则该值必须为空值。

RPC响应包格式示例:

```
{"jsonrpc": "2.0", "result": "0x1", "id": 1}
```

注： 服务端响应必须包含result或error成员，但两个成员不能同时包含。

3.3 错误对象

当一个RPC调用遇到错误时，返回的响应对象必须包含error错误结果字段，相关的描述和错误码，请参考：RPC 错误码

9.8.4 4 RPC接口的设计

FISCO BCOS提供丰富的RPC接口供客户端调用。其中分为3类：

- 以get开头命名的查询接口：例如`[getBlockNumber]`接口，查询最新的区块高度。
- `[sendRawTransaction]`接口: 执行一笔签名的交易，将等待区块链共识后才返回响应。
- `[call]`接口: 执行一个请求将不会创建一笔交易，不需要区块链共识，而是获取响应立刻返回。

9.8.5 5 RPC接口列表

参考RPC API文档

9.9 数据结构与编码协议

9.9.1 交易结构及其RLP编码描述

FISCO BCOS的交易结构在原以太坊的交易结构的基础上，有所增减字段。FISCO BCOS 2.0+的交易结构字段如下：

RC1的hashWith字段（也称交易hash/交易唯一标识）的生成流程如下：

RC2的生成流程也类似，只是在第一步`rlp+hash`的transaction结构体中增加chainId、groupId和extraData三个字段。

9.9.2 区块结构及其RLP编码描述

FISCO BCOS的区块由以下五部分组成

rc1版本

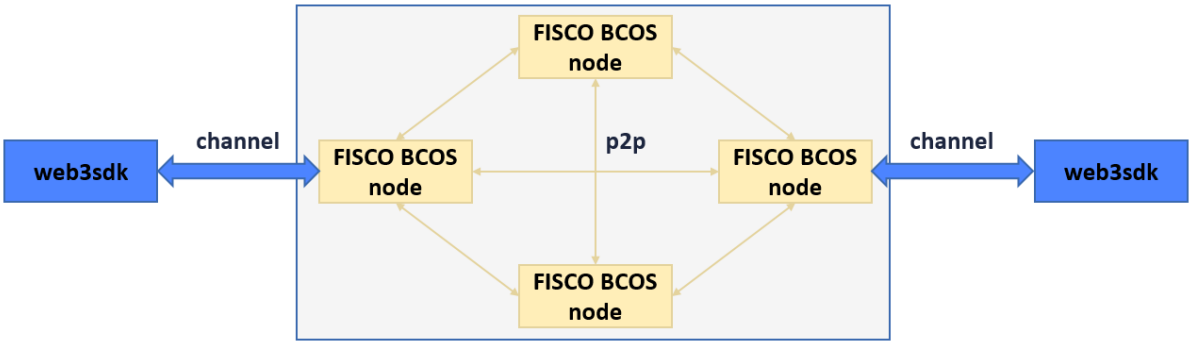
rc2、rc3、2.0及以上版本

FISCO BCOS的区块头中每个字段意义如下：

9.9.3 交易收据

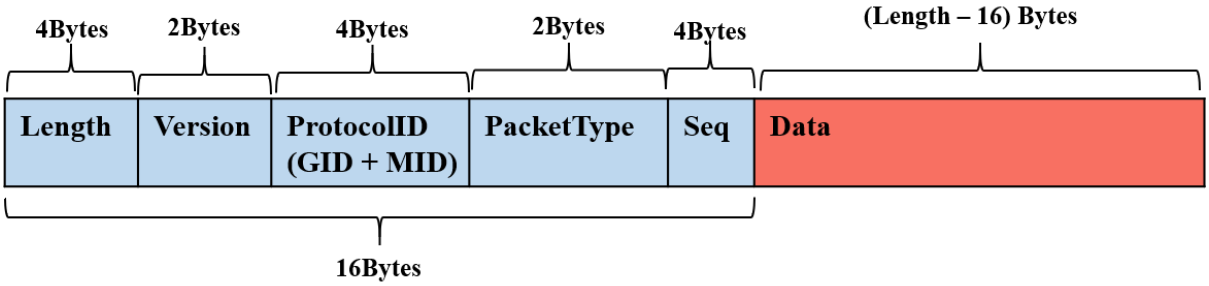
9.9.4 网络传输协议

FISCO BCOS 目前有两类数据包格式，节点与节点间通信的数据包为P2PMessage格式，节点与SDK间通信的数据包为ChannelMessage格式。



P2PMessage

v2.0.0-rc2扩展了**群组ID和模块ID**范围，**最多支持32767个群组**，且新增了**Version**字段来支持其他特性(如网络压缩)，包头大小为16字节，v2.0.0-rc2的网络数据包结构如下：



v2.0.0-rc2以前的P2PMessage定义请参考这里

补充

1. P2PMessage不限制包大小，由上层调用模块（共识/同步/AMOP等)进行包大小管理；
2. 群组ID和模块ID可唯一标识协议ID（protocolID），三者关系为`protocolID = (groupID << sizeof(groupID)*8) | ModuleID`;
3. 数据包通过protocolID所在的16位二进制数值来区分请求包和响应包，大于0为请求包，小于0为相应包。
4. 目前AMOP使用的packetType有`SendTopicSeq = 1`，`RequestTopics = 2`，`SendTopics = 3`。

ChannelMessage v2

ChannelMessage v1 请参考这里

AMOP消息包

AMOP消息包继承ChannelMessage包机构，在data字段添加了自定义内容。包括`0x30`,`0x31`,`0x35`,`0x1001`

消息包类型

数据包类型枚举值及其对应的含义如下：

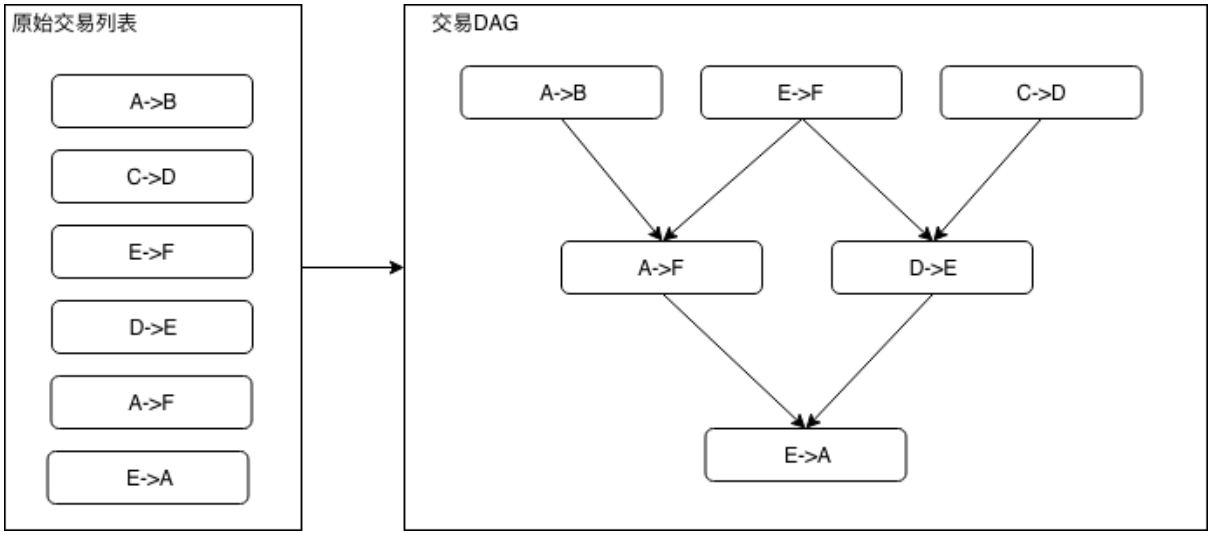
错误码

9.10 交易并行

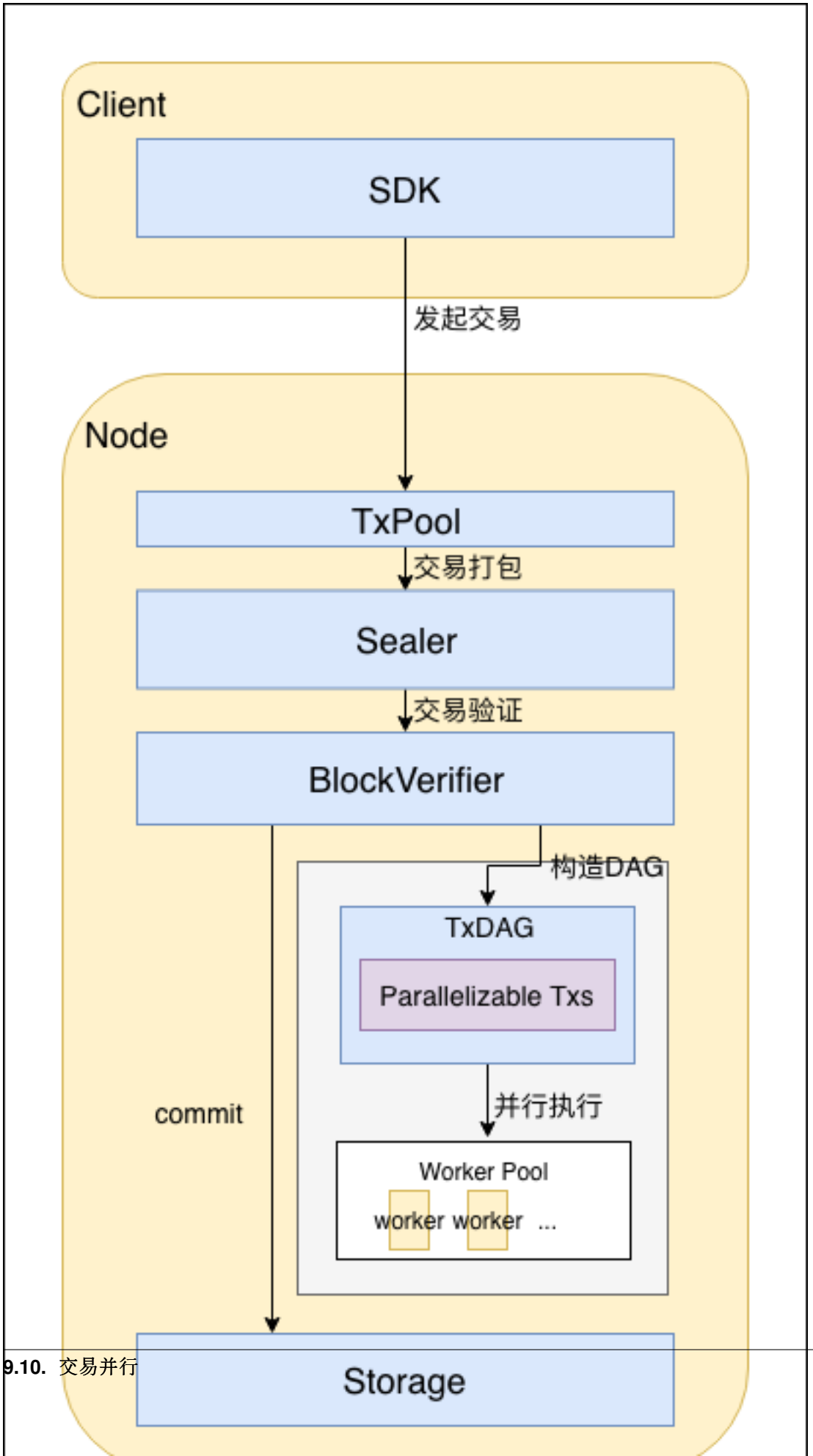
9.10.1 1 名词解释

1.1 DAG

一个无环的有向图称做有向无环图（**D**irected **A**cyclic **G**raph），简称DAG图。在一批交易中，可以通过一定方法识别出每笔交易需要占用的互斥资源，再根据交易在Block中的顺序及互斥资源的占用关系构造出一个交易依赖DAG图，如下图所示，凡是入度为0（无被依赖的前序任务）的交易均可以并行执行。如下图所示，基于左图的原始交易列表的顺序进行拓扑排序后，可以得到右图的交易DAG。



9.10.2 2 模块架构

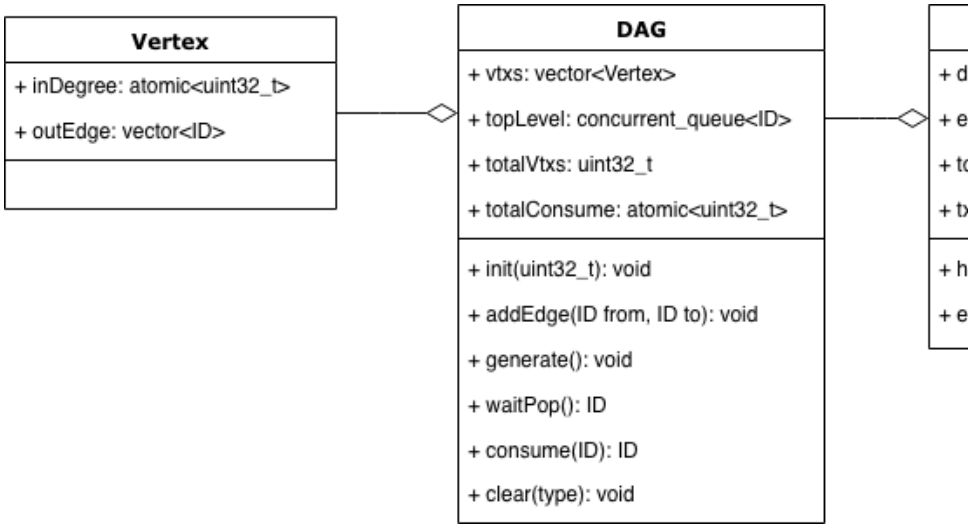

Client
SDK
发起交易
Node
TxPool
交易打包
Sealer
交易验证
BlockVerifier
构造DAG
TxDAG
Parallelizable Txs
并行执行
commit
Worker Pool
worker worker ...
Storage

- 用户直接或间接通过SDK发起交易。交易可以是能够并行执行的交易和不能并行执行的交易；
- 交易进入节点的交易池中，等待打包；
- 交易被Sealer打包为区块，经过共识后，发送至BlockVerifier进行验证；
- BlockVerifier根据区块中的交易列表生成交易DAG；
- BlockVerifier构造执行上下文，并行执行交易DAG；
- 区块验证通过后，区块上链。

9.10.3 3 重要流程

3.1 交易DAG构建

3.1.1 DAG数据结构



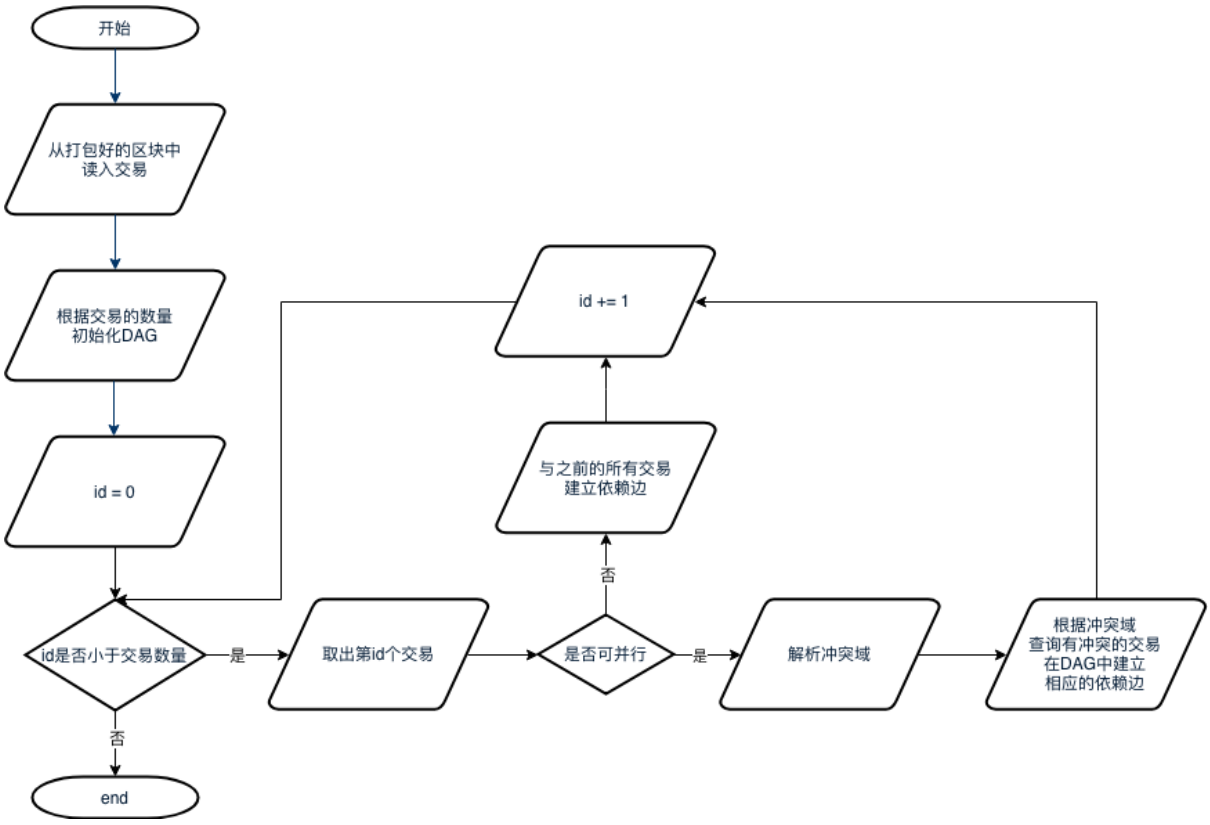
方案中所用到的DAG数据结构如下：
其中：

- 顶点（Vertex）
  - inDegree用于存储顶点当前的入度；
  - outEdge用于保存该顶点的出边信息，具体为所有出边所连顶点的ID列表。
- DAG：
  - vtxs是用于存储DAG中所有节点的列表；
  - topLevel是一个并发队列，用于存储当前入度为0的节点ID，执行时供多个线程并发访问；
  - totalVtxs：顶点总数
  - totalConsume：已经执行过的顶点总数；
  - void init(uint32_t _maxSize)：初始化一个最大顶点数为maxSize的DAG；
  - void addEdge(ID from, ID to)：在顶点from和to之间建立一条有向边；
  - void generate()：根据已有的边和顶点构造出一个DAG结构；
  - ID waitPop(bool needWait)：等待从topLevel中取出一个入度为0的节点；
  - void clear()：清除DAG中所有的节点与边信息。
- TxDAG:
  - dag：DAG实例

- exeCnt：已经执行过的交易计数；
- totalParaTxs：并行交易总数；
- txs：并行交易列表
- bool hasFinished()：若整个DAG已经执行完毕，返回true，否则返回false;
- void executeUnit()：取出一个没有上层依赖的交易并执行；

3.1.2 交易DAG构造流程

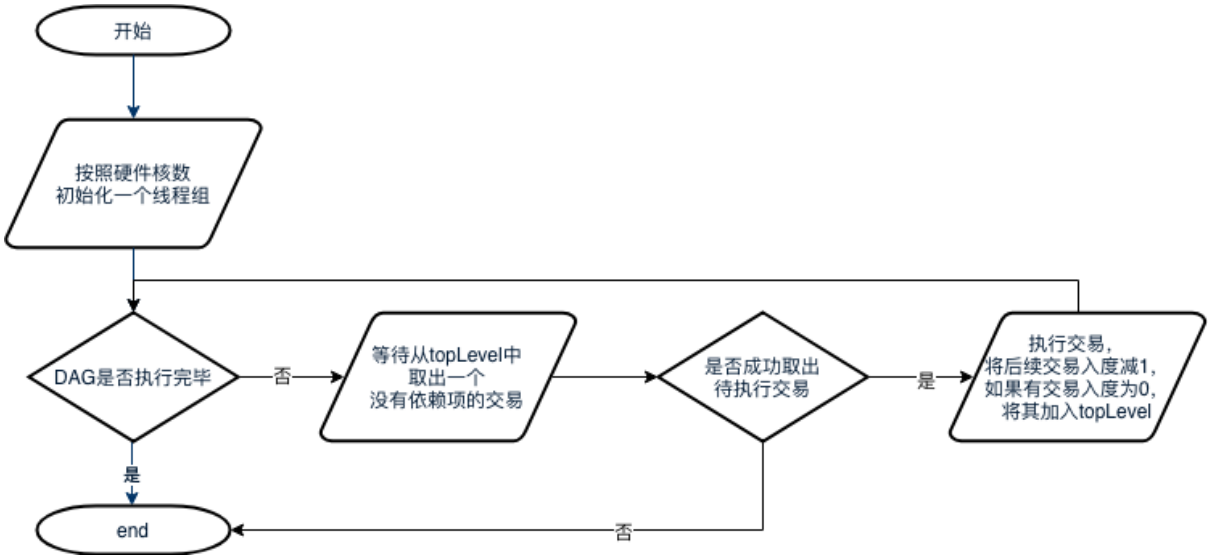
流程如下：



1. 从打包好的区块从取出区块中的所有交易；
2. 将交易数量作为最大顶点数量初始化一个DAG实例；
3. 按序读出所有交易，如果一笔交易是可并行交易，则解析其冲突域，并检查是否有之前的交易与该交易冲突，如果有，则在相应交易间构造依赖边；若该交易不可并行，则认为其必须在前序的所有交易都执行完后才能执行，因此在该交易与其所有前序交易间建立一条依赖边。

3.2 DAG执行流程

流程如下：



1. 主线程会首先根据硬件核数初始化一个相应大小的线程组，若获取硬件核数失败，则不创建其他线程；
2. 当DAG尚未执行完毕时，线程循环等待从DAG中pop出入度为0的交易。若成功取出待执行的交易，则执行该交易，执行完后将后续的依赖任务的入度减1，若有交易入度被减至0，则将该交易加入topLevel中；若失败，则表示DAG已经执行完毕，线程退出。

9.11 其他特性

为了提供更好的智能合约调用体验、支持更高的安全性，FISCO BCOS引入了合约命名服务(Contract Name Service, CNS)、国密算法和落盘加密特性。

- **合约命名服务(Contract Name Service, CNS)**

以太坊基于智能合约地址调用合约，存在如下问题：

- 合约abi为较长的JSON字符串，调用方无法直接感知
- 合约地址为20字节的魔数，不方便记忆，若丢失后将导致合约不可访问
- 约重新部署后，一个或多个调用方都需更新合约地址
- 不便于进行版本管理以及合约灰度升级

FISCO BCOS引入的合约命名服务CNS通过提供链上合约名称与合约地址映射关系的记录及相应的查询功能，方便调用者通过记忆简单的合约名来实现对链上合约的调用。

- **国密算法**

为了充分支持国产密码学算法，FISCO BCOS基于 国产密码学标准 ，实现了国密加解密、签名、验签、哈希算法、国密SSL通信协议，并将其集成到FISCO BCOS平台中，实现对 **国家密码局认定的商用密码** 的完全支持。

- **落盘加密特性**

考虑到联盟链的架构中，数据在联盟链的各个机构内是可见的，FISCO BCOS引入了落盘加密特性，对存储到节点数据库中的数据进行加密，并引入Key Manager保存加密密钥，保障了节点数据的机密性。

9.11.1 CNS方案

概述

调用以太坊智能合约的流程包括：

1. 编写合约；
2. 编译合约得到合约接口abi描述；
3. 部署合约得到合约地址address；
4. 封装合约的abi和地址，通过SDK等工具实现对合约的调用。

从合约调用流程可知，调用之前必须准备合约abi以及合约地址address。这种使用方式存在以下的问题：

1. 合约abi为较长的JSON字符串，调用方不需直接感知；
2. 合约地址为20字节的魔数，不方便记忆，若丢失后将导致合约不可访问；
3. 合约重新部署后，一个或多个调用方都需更新合约地址；
4. 不便于进行版本管理以及合约灰度升级。

为解决以上问题，给调用者提供良好的智能合约调用体验，FISCO BCOS提出**CNS合约命名服务**。

名词解释

- **CNS**（Contract Name Service）通过提供链上合约名称与合约地址映射关系的记录及相应的查询功能，方便调用者通过记忆简单的合约名来实现对链上合约的调用。
- **CNS信息**为合约名称、合约版本、合约地址和合约abi
- **CNS表**用于存储CNS信息

CNS对比以太坊原有调用方式的优势

- 简化调用合约方式；
- 合约升级对调用者透明，支持合约灰度升级。

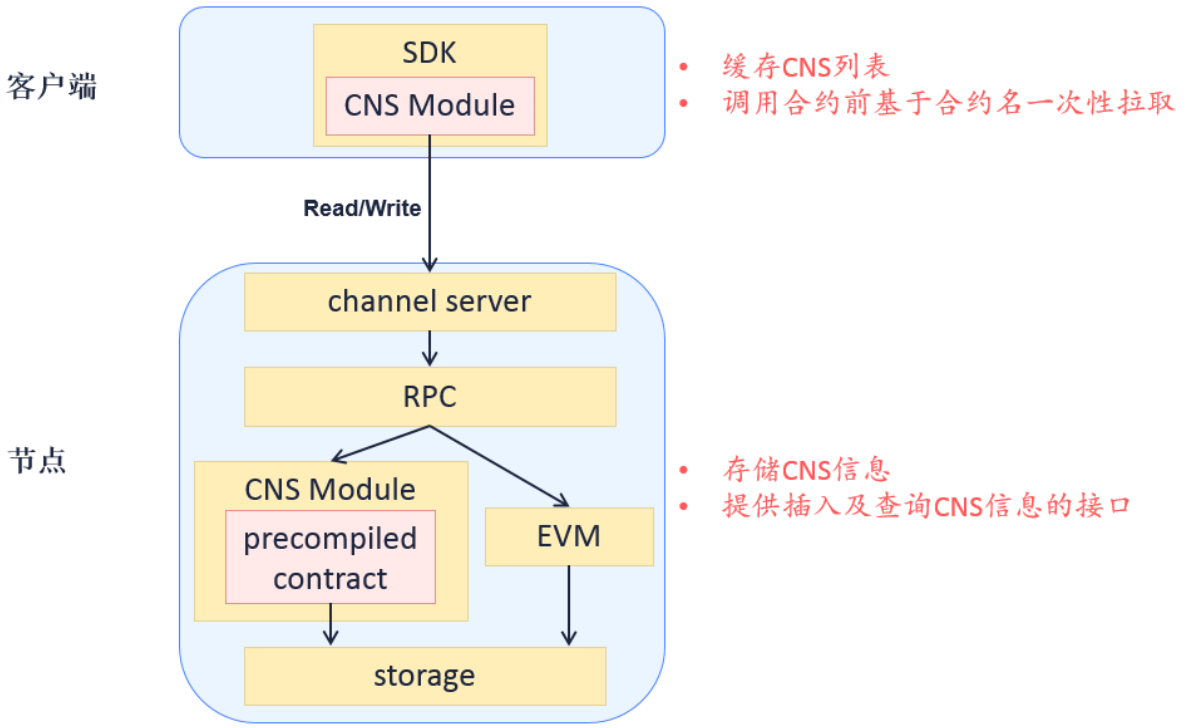
对标ENS

ENS (Ethereum Name Service)，以太坊名称服务。

ENS的功能类似我们较熟悉的DNS(Domain Name Service)域名系统，但提供的不是Internet网址，而是将以太坊(Ethereum)合约地址和钱包地址以xxxxxx.eth网址的方式表示，用于存取合约或转账。两者相比：

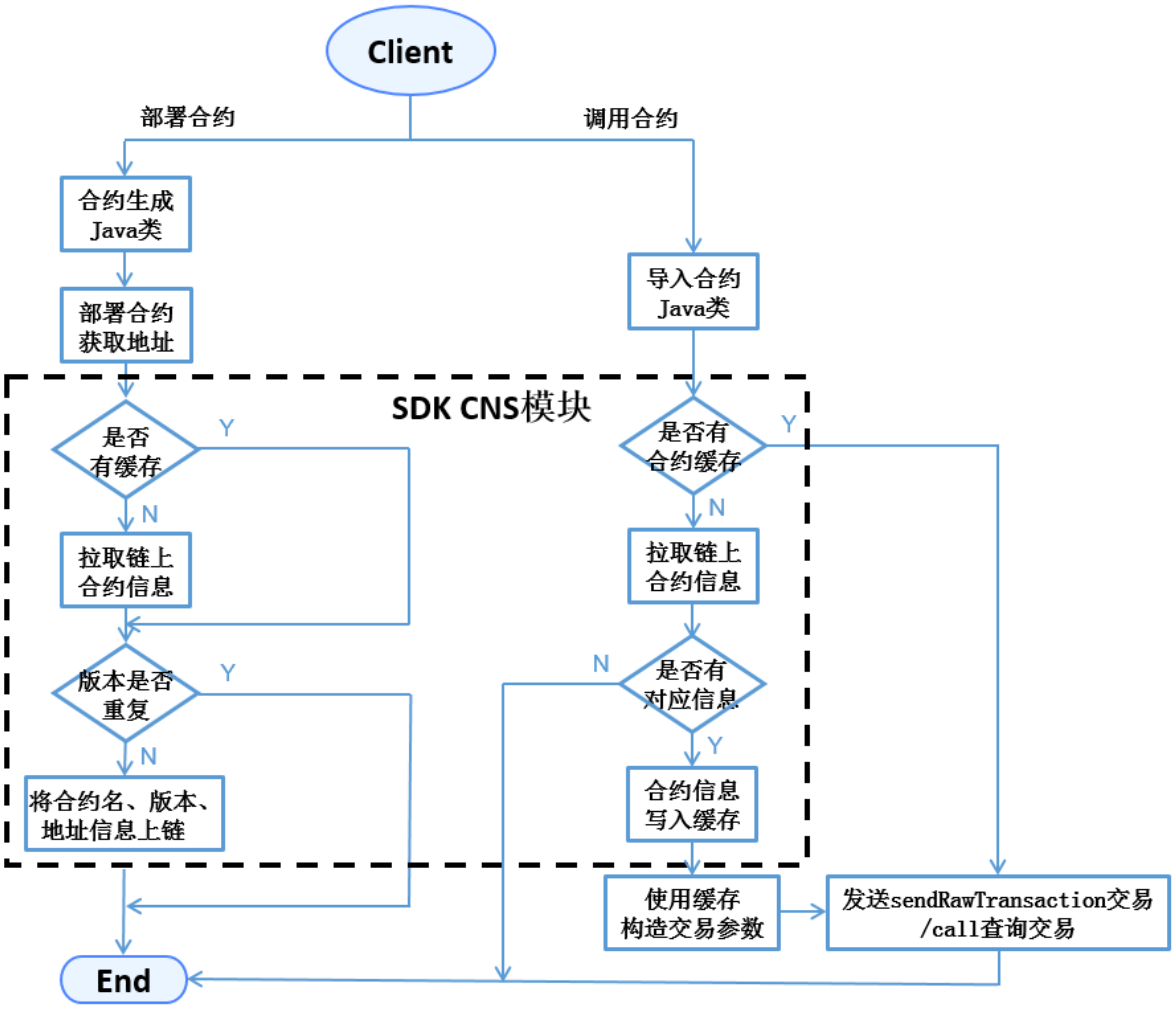
- ENS映射的地址类型包括合约地址及钱包地址，CNS可支持，当地址类型为钱包地址时合约abi为空。
- ENS有竞拍功能，CNS不需支持。
- ENS支持多级域名，CNS不需支持。

模块架构



核心流程

用户调用SDK部署合约及调用合约流程如下:



- 部署合约时，SDK生成合约对应的Java类，调用类的deploy接口发布合约获得合约地址，然后调用CNS合约insert接口上链CNS信息。
- 调用合约时，SDK引入合约的Java类，并加载实例化。load加载接口可传入合约地址（原有以太坊方式）或合约名称和合约版本的组合（CNS方式），SDK处理CNS方式时通过调用CNS模块查询链上信息来获取合约地址。
- 对于缺少版本号的合约调用，由SDK实现默认调用合约的最新版本。
- 上链的合约abi信息属于可选字段。

数据结构

CNS表结构

CNS信息以系统表的方式进行存储，各账本独立。CNS表定义如下：

合约接口

```
pragma solidity ^0.4.2;
contract CNS
{
    function insert(string name, string version, string addr, string abi) public␣
↪returns(uint256);
```

(continues on next page)

(续上页)

```
    function selectByName(string name) public constant returns(string);
    function selectByNameAndVersion(string name, string version) public constant␣
↪returns(string);
}
```

- CNS合约不暴露给用户，为SDK与底层CNS表的交互接口。
- insert接口提供CNS信息上链的功能，接口四个参数分别为合约名称name、合约版本version、合约地址addr和合约ABI信息abi。SDK调用接口需判断name和version的组合与数据库原有记录是否重复，在不重复的前提下才能发起上链交易。节点在执行交易时，precompiled逻辑会Double Check，发现数据重复就直接抛弃该交易。insert接口对CNS表的内容只增不改。
- selectByName接口参数为合约名称name，返回表中所有基于该合约的不同version记录。
- selectByNameAndVersion接口参数为合约名称name和合约版本version，返回表中该合约该版本的唯一地址。

更新CNS表方式

预编译合约是FISCO BCOS底层通过C++实现的一种高效智能合约，用于FISCO BCOS底层的系统信息配置与管理。引入precompiled逻辑后，FISCO BCOS节点执行交易的流程如下：

CNS合约属于预编译合约类型，节点将通过内置C++代码逻辑实现对CNS表的插入和查询操作，不经EVM执行，因此CNS合约只提供了函数接口描述而没有函数实现。**预置CNS合约的precompiled地址为0x1004。**

合约接口返回示例

selectByName和selectByNameAndVersion接口返回的string为Json格式，示例如下：

```
[
    {
        "name" : "Ok",
        "version" : "1.0",
        "address" : "0x420f853b49838bd3e9466c85a4cc3428c960dde2",
        "abi" : "[{\"constant\":false,\"inputs\":[{\"name\":\"num\",\"type\":\
↪"uint256\"}],\"name\":\"trans\",\"outputs\":[],\"payable\":false,\"type\":\
↪"function\"},{\"co
nstant\":true,\"inputs\":[],\"name\":\"get\",\"outputs\":[{\"name\":\"\",\"type\":\
↪"uint256\"}],\"payable\":false,\"type\":\"function\"},{\"inputs\":[],\"payable\
↪":false,\
"type\":\"constructor\"}]"
    },
    {
        "name" : "Ok",
        "version" : "2.0",
        "address" : "0x420f853b49838bd3e9466c85a4cc3428c960dde2",
        "abi" : "[{\"constant\":false,\"inputs\":[{\"name\":\"num\",\"type\":\
↪"uint256\"}],\"name\":\"trans\",\"outputs\":[],\"payable\":false,\"type\":\
↪"function\"},{\"co
nstant\":true,\"inputs\":[],\"name\":\"get\",\"outputs\":[{\"name\":\"\",\"type\":\
↪"uint256\"}],\"payable\":false,\"type\":\"function\"},{\"inputs\":[],\"payable\
↪":false,\
"type\":\"constructor\"}]"
    }
]
```

SDK_API

SDK开发者可使用`org.fisco.bcos.web3j.precompile.cns`中以下两接口实现CNS的注册及查询功能。

registerCns

- 描述：public TransactionReceipt registerCns(String name, String version, String addr, String abi)
- 功能：上链合约信息
- 参数：name——合约名，version——合约版本，addr——合约地址，abi——合约abi
- 返回：上链交易回执，回执中含上链结果信息及错误信息（如有）

resolve

- 描述：public String resolve(String contractNameAndVersion)
- 功能：基于合约名和合约版本查询合约地址
- 参数：contractNameAndVersion——合约名+合约版本信息
- 返回：合约地址，如无参数指定版本的合约信息，接口抛出异常
- 说明：contractNameAndVersion通过:来分割合约名和合约版本，当缺少合约版本时，SDK默认调用使用合约的最新版本进行查询

注意：

1. 在调用接口前，需将sol合约转换Java类，并将生成的Java类以及abi、bin文件置于正确的目录，详细使用方法请参考Web3SDK;
2. 两个接口的使用例子可参考ConsoleImpl.java中的deployByCNS和callByCNS接口实现。

操作工具

控制台可提供部署合约、调用合约、基于合约名查询链上已有合约的功能。控制台的详细使用方法请参考《控制台》。

控制台提供的命令包括：

- deployByCNS：通过CNS方式部署合约
- callByCNS：通过CNS方式调用合约
- queryCNS：根据合约名称和合约版本号（可选参数）查询CNS表信息

9.11.2 国密支持方案

设计目标

为了充分支持国产密码学算法，金链盟基于国产密码学标准，实现了国密加解密、签名、验签、哈希算法、国密SSL通信协议，并将其集成到FISCO BCOS平台中，实现了对**国家密码局认定的商用密码**的完全支持。

国密版FISCO BCOS将交易签名验签、p2p网络连接、节点连接、数据落盘加密等底层模块的密码学算法均替换为国密算法，国密版FISCO BCOS与标准版主要特性对比如下：

(注：国密算法SM2, SM3, SM4均基于国产密码学标准开发)

系统框架

系统整体框架如下图所示：

国密SSL 1.1 握手建立流程

国密版FISCO BCOS节点之间的认证选用国密SSL 1.1的ECDHE_SM4_SM3密码套件进行SSL链接的建立，差异如下表所示：

数据结构差异

国密版与标准版FISCO BCOS在数据结构上的差异如下：

9.11.3 落盘加密

背景介绍

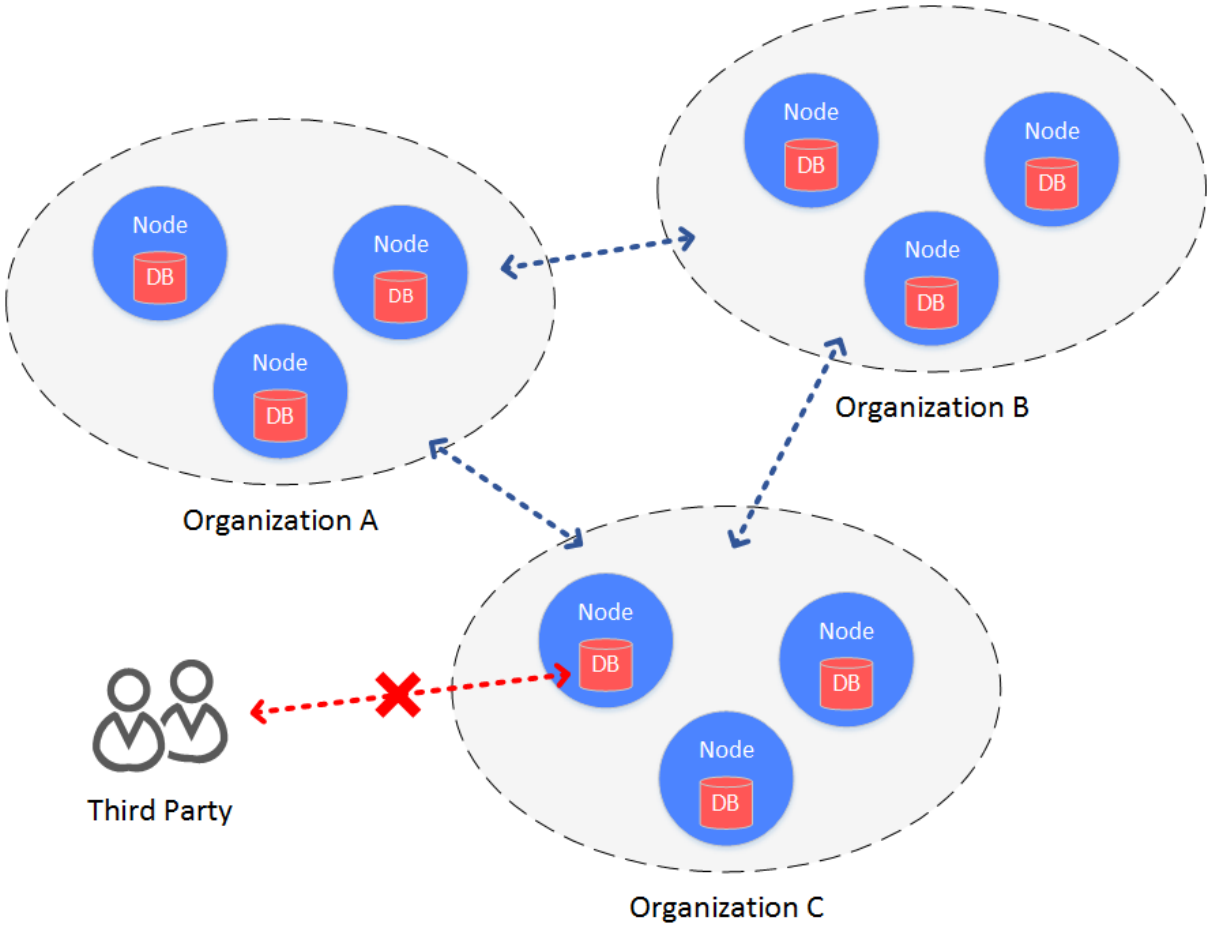
在联盟链的架构中，机构和机构之间搭建一条区块链，数据在联盟链的各个机构内是可见的。

在某些数据安全性要求较高的场景下，联盟内部的成员并不希望联盟之外的机构能够获取联盟链上的数据。此时，就需要对联盟链上的数据进行访问控制。

联盟链数据的访问控制，主要分为两个方面

- 链上通信数据的访问控制
- 节点存储数据的访问控制

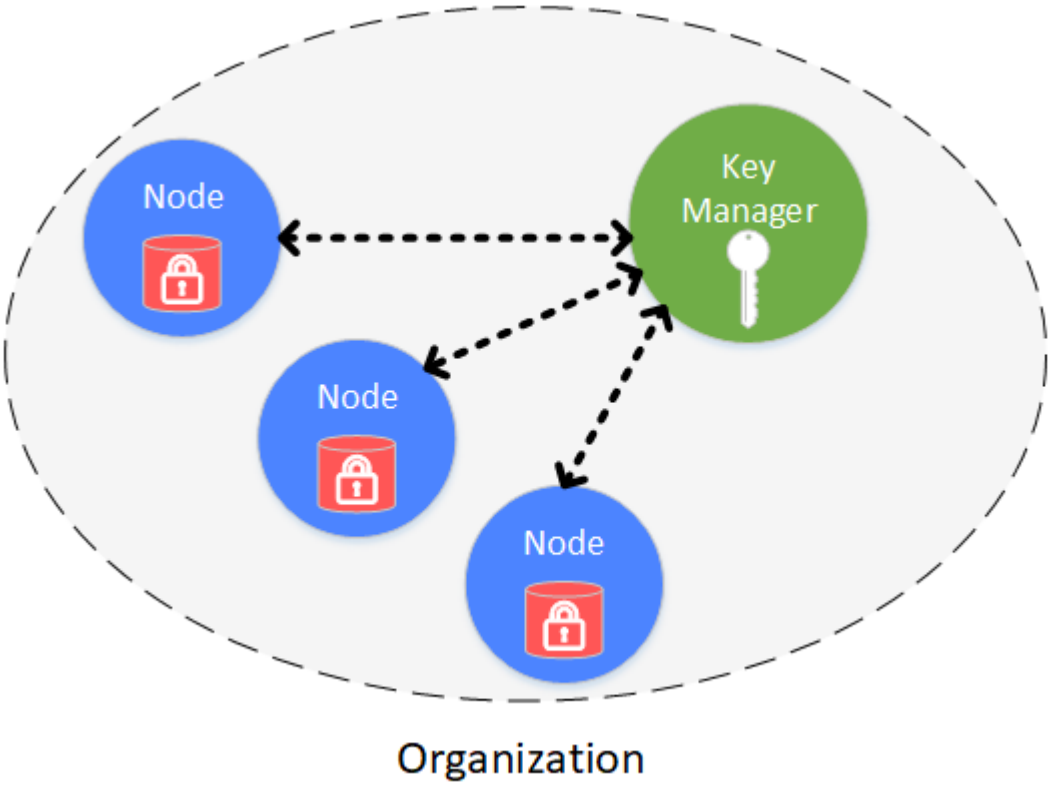
对于链上通信数据的访问控制，FISCO BCOS是通过节点证书和SSL来完成。此处主要介绍的是节点存储数据的访问控制，即落盘加密。



主要思想

落盘加密是在机构内部进行的。在机构的内网环境中，每个机构独立地对节点的硬盘数据进行加密。当节点所在机器的硬盘被带离机构，并让节点在机构内网之外的网络启动，硬盘数据将无法解密，节点无法启动。进而无法盗取联盟链上的数据。

方案架构



落盘加密是在机构内部进行的，每个机构独立管理自己硬盘数据的安全。内网中，每个节点的硬盘数据是被加密的。所有加密数据的访问权限，通过Key Manager来管理。Key Manager是部署在机构内网内，专门管理节点硬盘数据访问秘钥的服务，外网无法访问。当内网的节点启动时，从Key Manager处获取加密数据的访问秘钥，来对自身的加密数据进行访问。

加密保护的对象包括：

- 节点本地存储的数据库：leveldb
- 节点私钥：node.key，gmnode.key（国密）

具体实现

具体的实现过程，是通过节点自身持有的秘钥（dataKey）和Key Manager管理的全局秘钥（superKey）来完成的。

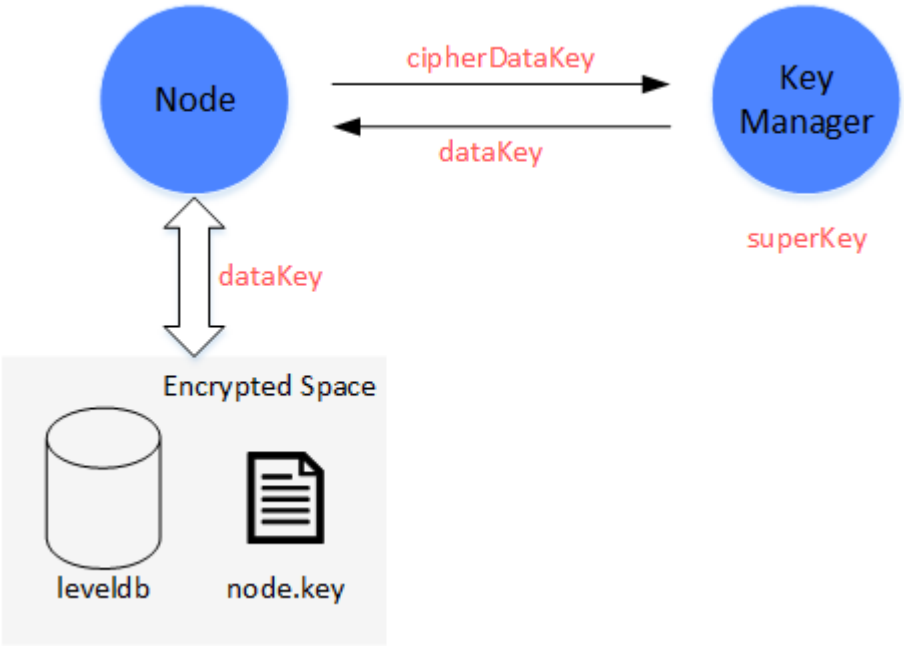
节点

- 节点用自己的dataKey，对自身加密的数据（Encrypted Space）进行加解密。
- 节点本身不会在本地磁盘中存储dataKey，而是存储dataKey被加密后的cipherDataKey。
- 节点启动时，拿cipherDataKey向Key Manager请求，获取dataKey。
- dataKey只在节点的内存中，当节点关闭后，dataKey自动丢弃。

Key Manager

持有全局的superKey，负责对所有节点启动时的授权请求进行响应，授权。

- Key Manager必须实时在线，响应节点的启动请求。
- 当节点启动时，发来cipherDataKey，Key Manager用superKey对cipherDataKey进行解密，若解密成功，就将节点的dataK返回给节点。
- Key Manager只能在内网访问，机构内的外网无法访问Key Manager.



方案流程

方案流程分为节点初始配置和节点安全运行。

节点初始配置

节点启动前，需要为节点配置dataKey

重要： 节点在生成后，启动前，必须决定好是否采用落盘加密，一旦节点配置成功，并正常启动，将无法切换状态。

（1）管理员定义好节点的的dataKey，并将dataKey发送给Key Manager，从Key Manager处获取cipherDataKey。

（2）将cipherDataKey配置到节点的配置文件中

（3）启动节点

节点安全运行

节点启动时，会通过Key Manager，获取本地数据访问的秘钥dataKey。

（1）节点启动，从配置文件中读取cipherDataKey，并发送给Key Manager。

（2）Key Manager收到cipherDataKey，用superKey解密cipherDataKey，若解密成功，则将解密后的dataKey返回给节点。

（3）节点拿到dataKey，用dataKey对本地的数据（Encrypted Space）进行交互。从Encrypted Space读取的数据，用dataKey解密获取真实数据。要写入Encrypted Space的数据，先用dataKey加密，再写入。

为什么可以保护数据?

当某节点的硬盘被意外的带到内网环境之外，数据是不会泄露的。

（1）当节点在内网之外启动时，无法连接Key Manager，虽然有cipherDataKey，也无法获取dataKey。

（2）不启动节点，直接对节点本地的数据进行操作，由于拿不到dataKey，无法解密Encrypted Space，拿不到敏感数据。

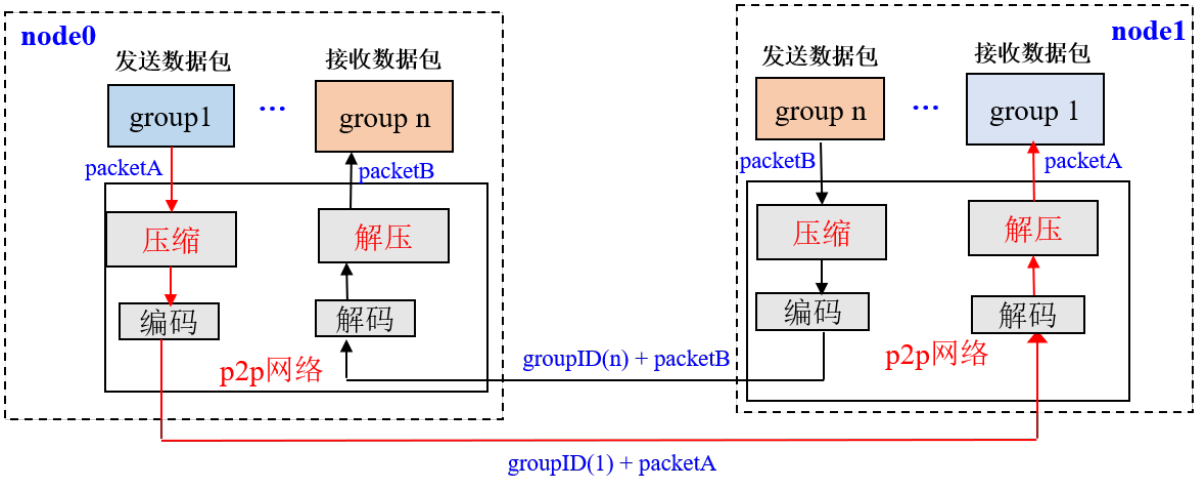
具体落盘加密的使用，可参考：落盘加密操作

9.11.4 网络压缩

外网环境下，区块链系统性能受限于网络带宽，为了尽量减少网络带宽对系统性能的影响，FISCO BCOS从`relase-2.0.0-rc2`开始支持网络压缩功能，该功能主要在发送端进行网络数据包压缩，在接收端将解包数据，并将解包后的数据传递给上层模块。

系统框架

网络压缩主要在P2P网络层实现，系统框架如下：



网络压缩主要包括两个过程：

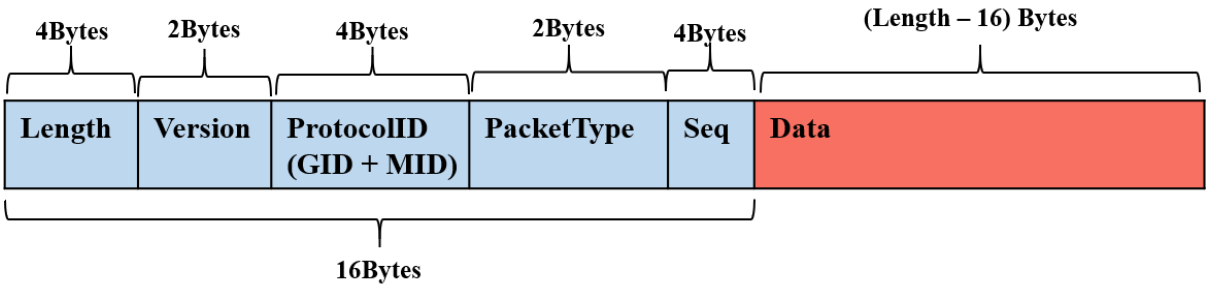
- **发送端压缩数据包**：群组层通过P2P层发送数据时，若数据包大小超过1KB，则压缩数据包后，将其发送到目标节点；
- **接收端解压数据包**：节点收到数据包后，首判断收到的数据包是否被压缩，若数据包是压缩后的数据包，则将其解压后传递给指定群组，否则直接将数据传递给对应群组。

核心实现

综合考虑性能、压缩效率等，我们选取了Snappy来实现数据包压缩和解压功能。本节主要介绍网络压缩的实现。

数据压缩标记位

FISCO BCOS的网络数据包结构如下图：

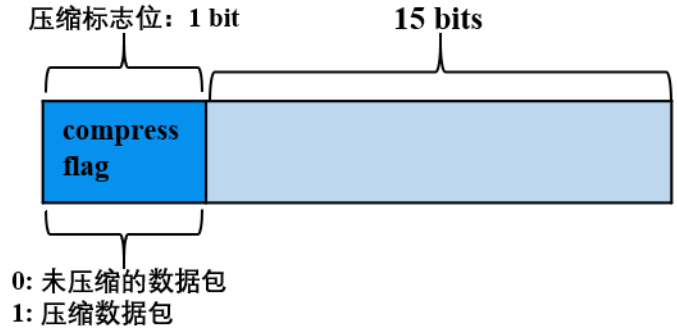


网络数据包主要包括包头和数据两部分，包头占了16个字节，各个字段含义如下：

- Length: 数据包长度
- Version: 扩展位，用于扩展网络模块功能
- ProtocolID: 存储了数据包目的群组ID和模块ID，用于多群组数据包路由，目前最多支持32767个群组
- PaketType: 标记了数据包类型
- Seq: 数据包序列号

网络压缩模块仅压缩网络数据，不压缩数据包头。

考虑到压缩、解压小数据包无法节省数据空间，而且浪费性能，在数据压缩过程中，不压缩过小的数据包，仅压缩数据包大于`c_compressThreshold`的数据包.`c_compressThreshold`默认是1024(1KB)。我们扩展了Version的最高位，作为数据包压缩标志：



- Version最高位为0，表明数据包对应的数据Data是未压缩的数据；
- Version最高位为1，表明数据包对应的数据Data是压缩后的数据。

处理流程

下面以群组1的一个节点向群组内其他节点发送网络消息包packetA为例（比如发送交易、区块、共识消息包等），详细说明网络压缩模块的关键处理流程。

发送端处理流程

- 群组1的群组模块将packetA传入到P2P层；
- P2P判断packetA的数据包大于`c_compressThreshold`，则调用压缩接口，对packetA进行压缩，否则直接将packetA传递给编码模块；
- 编码模块给packetA加上包头，附带上数据压缩信息，即：若packetA是压缩后的数据包，将包头Version的最高位置为1，否则置为0；
- P2P将编码后的数据包传送到目的节点。

接收端处理流程：

- 目标机器收到数据包后，解码模块分离出包头，通过包头Version字段的最高位是否为1，判断网络数据是否被压缩；
- 若网络数据包被压缩过，则调用解压接口，对Data部分数据进行解压，并根据数据包头附带的GID和PID，将解压后的数据传递给指定群组的指定模块；否则直接将数据包传递给上层模块。

兼容性说明

- **数据兼容**：不涉及存储数据的变更；
- **网络兼容rc1**：向前兼容，仅有relase-2.0.0-rc2及以上节点具有网络压缩功能。

9.11.5 合约管理

本文档描述合约生命周期管理中冻结/解冻操作（以下简称合约生命周期管理操作）及其操作权限的设计方案。

重要： 合约生命周期管理操作支持storagestate的存储模式，不支持mptstate的存储模式。这里提及的合约目前只限于Solidity合约，不包括预编译合约。

名词解释

合约管理的相关操作包括冻结、解冻、查询状态、授权、查询授权。

- 冻结合约：可逆操作，一合约冻结后读写接口都不能被调用
- 解冻合约：撤销冻结的操作，一合约解冻后读写接口都可调用
- 查询合约状态：查询合约状态，返回该合约可用/已冻结的状态
- 授权：已有权限的账号可以给其他账号授予合约管理权限
- 查询授权：查询合约的管理权限列表

重要： 冻结合约的操作不会对原有合约内容（代码逻辑+数据）进行修改，只会通过标志位进行记录。

合约状态（可用、已冻结）转换矩阵如下：

具体实现

合约状态存储

- 新增一字段frozen，用于记录该合约是否已冻结，该字段默认为false，表示可用，冻结时该值为true；
- 新增一字段authority，用于记录合约管理员账号，每个账号对应一行authority记录。

注意：

1. 对不存在字段frozen的合约表，查询该字段时将返回false；
2. 部署合约时，将部署账号tx.origin写入authority；
3. 调用合约A接口过程中创建新合约B时，对于新合约B，默认将tx.origin及合约A的权限信息写入合约B的authority。

合约状态判断

Executive中根据合约地址获取frozen字段值，进行判断后交易顺利执行，或者抛出异常，提示该合约已冻结。

管理权限判断

- 更新合约状态的操作需进行权限判断，只有authority列表中的账号才能设置该合约的状态；
- 授予权限的操作也需进行权限判断，只有authority列表中的账号才能授予其他账号管理该合约的权限；
- 查询合约状态及权限列表不需进行权限判断。

合约生命周期管理接口

新增一个合约生命周期管理的预编译合约ContractLifeCyclePrecompiled，地址为0x1007，用于给指定合约设置指定状态，并提供查询功能。

```
contract ContractLifeCyclePrecompiled {
    function freeze(address addr) public returns(int);
    function unfreeze(address addr) public returns(int);
    function grantManager(address contractAddr, address userAddr) public␣
↪returns(int);
    function getStatus(address addr) public constant returns(uint,string);
    function listManager(address addr) public constant returns(uint,address[]);
}
```

返回码描述

重要： 兼容性说明：合约管理相关操作只能在2.3及以上版本上进行。

9.11.6 账号管理

本文档描述账号的冻结/解冻操作及其操作权限的设计方案。

重要： 账号冻结/解冻操作支持storagestate的存储模式，不支持mptstate的存储模式。

名词解释

账号管理的相关操作包括冻结、解冻、查询状态、授权、查询授权。

- 冻结账号：可逆操作，一账号冻结后不能部署及调用合约
- 解冻账号：撤销冻结的操作，一账号解冻后可部署及调用合约
- 查询账号状态：查询账号状态，返回该账号可用/已冻结的状态

账号的冻结/解冻操作只能由链的委员进行。权限相关操作请参考基于角色的权限控制。

重要： 冻结账号的操作不会对原有账号内容进行修改，只会通过标志位进行记录。

账号状态（可用、已冻结）转换矩阵如下：

具体实现

账号状态存储

- 复用字段frozen，用于记录该账号是否已冻结，该字段默认为false，表示可用，冻结时该值为true；

注意：

1. 对不存在字段frozen的账号表，查询该字段时将返回false；

账号状态判断

Executive中根据账号地址获取frozen字段值，进行判断后部署或执行交易，或者抛出异常，提示该账号已冻结。

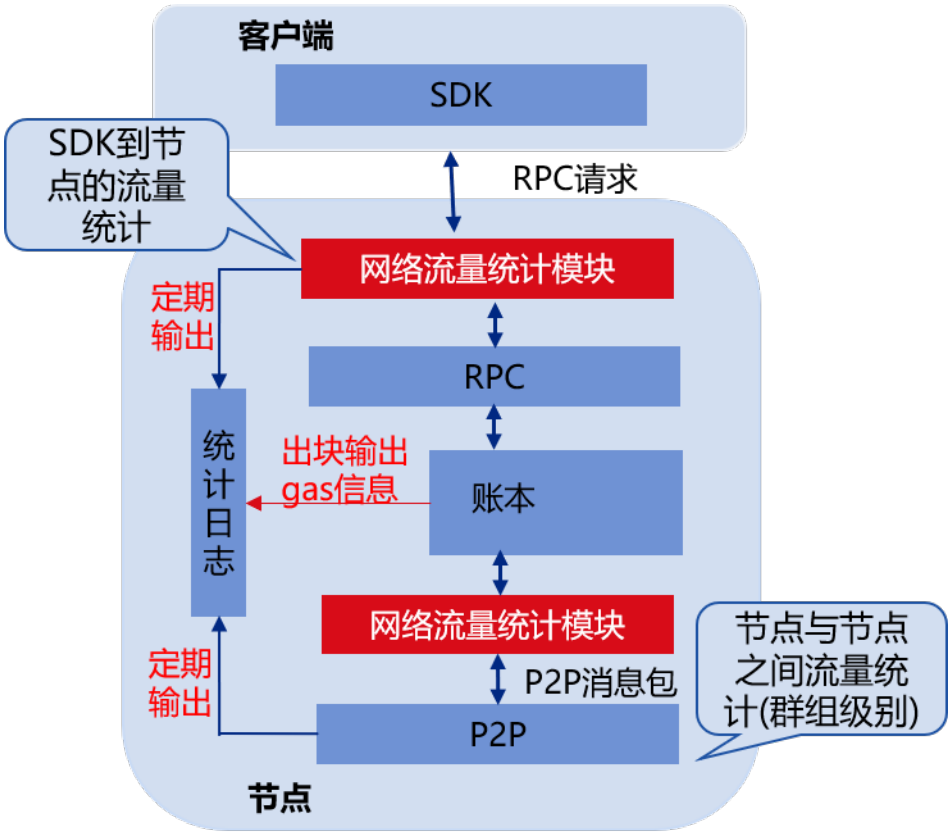
重要： 兼容性说明：账号管理相关操作只能在2.5及以上版本上进行。

9.11.7 网络流量和Gas统计

FISCO BCOS 2.0引入了多群组架构，允许一个节点启动若干个群组，这种架构可快速平行扩展、简化了运维复杂度、降低了管理成本，但由于一个节点进程运行了多个群组，增加了群组监控复杂度。考虑到实时监控系统资源使用情况在实际生产系统中非常重要，FISCO BCOS v2.4.0引入了统计日志，并实现了群组级别的网络流量统计以及交易级别的Gas消耗统计功能。

模块架构

下图网络流量和Gas统计的模块图，FISCO BCOS节点实时统计群组内网络、Gas消耗信息，并将其输出到统计日志中：



网络流量统计信息定期输出到统计日志中，目前统计的网络流量主要包括：

- SDK到节点的网络流量：统计每个群组RPC流量、交易推送流量以及事件推送流量
- P2P网络流量：统计群组间因同步、共识等模块进行P2P网络交互产生的网络流量
- 网络总流量：每个群组网络消耗总流量，包括SDK到节点的网络流量和P2P网络流量

Gas统计输出通过区块落盘触发，每落盘一个区块会将对应的Gas消耗信息输出到统计日志中，目前统计的信息包括：

- 每个区块Gas消耗信息
- 每笔交易Gas消耗信息

统计日志详细说明

注解:

- 统计日志的级别是info
- 统计日志每一小时分割一次

网络流量统计日志

为了方便开发者根据网络流量统计日志对系统进行诊断，FISCO BCOS v2.4.0统计网络流量时，根据流量的相关模块做了细分。这里详细介绍网络流量统计日志的格式和关键字含义。

注解:

- 输出到日志里的网络流量单位是字节Byte

- 网络统计功能仅统计某个时间段的网络消耗总流量，统计信息输出到日志后，计数器会清零，重新开始下一轮统计

SDK到节点的网络流量

SDK到节点的网络流量统计日志记录的模块关键字是SDK，不仅统计了每个群组与SDK之间的总的出流量和入流量，还细分了RPC请求产生的流量、事件推送流量以及交易推送流量。日志示例如下：

```
info|2020-04-24 12:58:41.173045|SDK|,g=1,SDK_RPCIn=10023,SDK_EventLogIn=500,SDK_
↪totalIn=10523,SDK_RPC_Out=0,SDK_Txs_Out=0,SDK_EventLog_Out=0,SDK_total_Out=0
```

关键字说明

P2P网络流量

群组间P2P流量统计日志记录的模块关键字是P2P，不仅统计了每个群组总的P2P出流量和入流量，还细分了共识流量、同步流量，日志示例如下：

```
info|2020-04-24 12:58:41.173077|P2P|,g=2,P2P_CONSIn=80505,P2P_SYNCIn=19008,P2P_
↪totalIn=99513,P2P_CONS_Out=211377,P2P_SYNC_Out=19008,P2P_total_Out=230385
```

关键字说明

群组总流量

为方便业务方对区块链系统的整体带宽消耗有所了解，网络流量统计模块还统计了每个群组的总出流量和入流量，模块关键字是Total，日志示例如下：

```
info|2020-04-24 12:58:41.173052|Total|,g=1,Total_In=74524,Total_Out=115434
```

关键字说明

Gas消耗统计日志

交易的Gas消耗衡量了交易消耗的物理资源，为了方便业务方监测交易资源消耗情况，FISCO BCOS v2.4.0统计了每个区块、每笔交易的Gas消耗情况。

每个区块Gas消耗信息

区块Gas消耗的统计日志记录模块关键字是BlockGasUsed，统计信息包括：区块高度、区块所属群组、区块内交易数目、区块gas消耗等，日志示例如下：

```
info|2020-04-24 12:46:31.974147|BlockGasUsed|,g=2,txNum=193,gasUsed=3860579,
↪blockNumber=1419,sealerIdx=2,
↪blockHash=b10bdcc5da9c9cd5399ca5821bed9ae6f3fecbe1ddf8ec723b44e6fa30c4bd05,
↪nodeID=0e23d6e237cfc5041d1754fa6682d71bef842b29ddfe3412b284aeac4b8b4794a51df409b667829750c2b4e9
```

关键字说明

每笔交易Gas消耗信息

每笔交易Gas消耗的统计日志记录模块关键字是`TxsGasUsed`，主要统计了每笔交易的Gas消耗，日志示例如下：

```
info|2020-04-24 12:46:31.976080|TxsGasUsed|,g=2,
↪txHash=a81ae1f60289cf7e8f6987b20c68ba9580a1c34d9252c5b4b9c097113309b9d7,
↪gasUsed=20003
```

关键字说明

配置选项

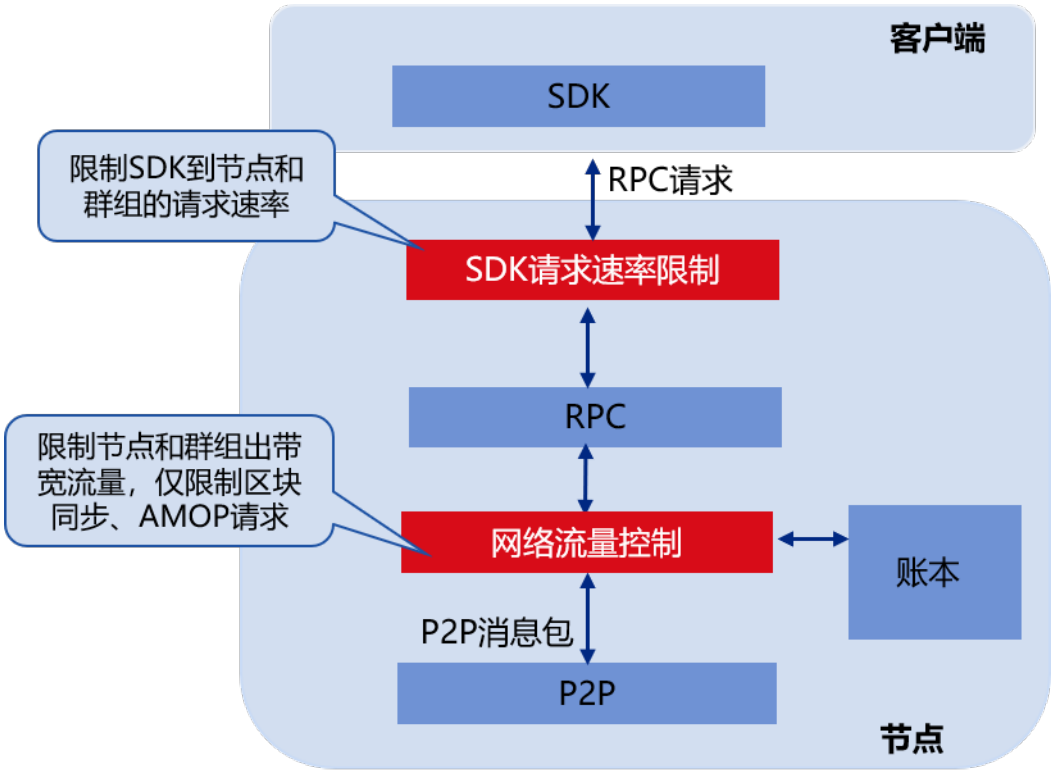
注解： 统计日志配置选项位于 `config.ini` 中，详细可参考 这里

9.11.8 流量控制

为实现区块链系统柔性服务，并防止多群组架构下，多个群组运行在相同进程中，某些群组占用资源过多干扰到其他群组，FISCO BCOS v2.5.0引入了流量控制功能。

模块架构

下图是流量控制的模块图，主要包括SDK请求速率限制以及网络流量限制，前者限制SDK到节点的请求速率，后者通过限制区块同步和AMOP请求流量，限制节点出带宽流量，防止区块同步、AMOP请求消息包过多影响共识模块性能。



SDK请求速率限制

SDK请求速率限制包括节点级别的请求速率限制和群组级别的请求速率限制：

- **节点级别请求速率限制**：限制SDK客户端到节点的总的请求速率，当SDK到节点的请求速率超过指定阈值后，节点会拒绝SDK的请求，达到QoS目标的同时，防止过多的SDK请求导致节点异常；
- **群组级别请求速率限制**：限制SDK客户端到群组的请求速率，当SDK发向指定群组的请求速率超过阈值后，群组会拒绝SDK请求。

注解：

节点和群组都开启请求速率限制时：

- 区块链节点收到SDK发送的请求包后，首先调用节点级别的请求速率限制模块判断是否应该接收该请求；
- 被接收的请求继续进入到群组级别的请求速率限制模块，通过群组级别请求速率限制模块检查的请求最终才会被转发到相应群组并被处理。

节点网络流量控制

类似于SDK请求速率限制，网络流量限制也包括节点级别的流量控制和群组级别的流量控制：

- **节点级别的流量控制**：限制节点的平均出带宽，当节点平均出带宽超过阈值后，节点收到区块同步请求后会暂缓发送区块，也会拒绝收到的AMOP请求，避免区块同步、AMOP消息包发送对节点共识的影响；
- **群组级别的流量控制**：限制每个群组的平均出带宽，当群组平均出带宽流量超过阈值后，该群组会暂停区块发送和AMOP请求包转发逻辑，优先将网络流量提供给共识模块使用。

注解：

节点和群组都开启流量控制功能时：

- 当节点收到客户端的AMOP请求时，调用节点级流量控制模块判断是否可以接收该AMOP请求；
- 当某个群组收到其他节点对应群组的区块请求后，群组向其回复区块前，(1) 调用节点级流量控制模块，判断节点平均出带宽是否超过设置的阈值；(2) 调用群组级流量控制模块，判断群组的出带宽是否超过设置的阈值，当且仅当节点级和群组级平均出带宽均没有超过设置阈值时，该群组才会回复区块

配置选项

注解：

- 节点级别的流量控制配置选项位于 `config.ini` ，具体可参考 这里
- 群组级别的流量控制选项位于 `group.{group_id}.ini` ，具体可参考 这里

CHAPTER 10

JSON-RPC API

下列接口的示例中采用curl命令，curl是一个利用url语法在命令行下运行的数据传输工具，通过curl命令发送http post请求，可以访问FISCO BCOS的JSON RPC接口。curl命令的url地址设置为节点配置文件[rpc]部分的[jsonrpc_listen_ip](若节点小于v2.3.0版本，查看配置项listen_ip)和[jsonrpc listen port]端口。为了格式化json，使用jq工具进行格式化显示。错误码参考RPC设计文档。交易回执状态列表参考这里。

10.1 getClientVersion

返回节点的版本信息

10.1.1 参数

无

10.1.2 返回值

- object - 版本信息，字段如下：
  - Build Time: string - 编译时间
  - Build Type: string - 编译机器环境
  - Chain Id: string - 链ID
  - FISCO-BCOS Version: string - 节点版本
  - Git Branch: string - 版本分支
  - Git Commit Hash: string - 版本最新commit哈希
  - Supported Version: string - 节点支持的版本
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getClientVersion","params":[],"id
↪":1}' http://127.0.0.1:8545 |jq

// Result
{
  "id": 83,
  "jsonrpc": "2.0",
  "result": {
    "Build Time": "20190106 20:49:10",
    "Build Type": "Linux/g++/RelWithDebInfo",
    "FISCO-BCOS Version": "2.0.0",
    "Git Branch": "master",
    "Git Commit Hash": "693a709ddab39965d9c39da0104836cfb4a72054"
  }
}
```

10.2 getBlockNumber

返回节点指定群组内的最新区块高度

10.2.1 参数

- groupID: unsigned int - 群组ID

10.2.2 返回值

- string - 最新区块高度(0x开头的十六进制字符串)
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockNumber","params":[1],"id
↪":1}' http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": "0x1"
}
```

10.3 getPbftView

返回节点所在指定群组内的最新PBFT视图

10.3.1 参数

- groupID: unsigned int - 群组ID

10.3.2 返回值

- `string` - 最新的PBFT视图
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getPbftView","params":[1],"id":1}'␣
↪http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": "0x1a0"
}
```

注： FISCO BCOS支持PBFT共识和Raft共识，当访问的区块链采用Raft共识时，该接口返回FISCO BCOS自定义错误响应如下:

```
{
  "error": {
    "code": 7,
    "data": null,
    "message": "Only pbft consensus supports the view property"
  },
  "id": 1,
  "jsonrpc": "2.0"
}
```

10.4 getSealerList

返回指定群组内的共识节点列表

10.4.1 参数

- `groupID: unsigned int` - 群组ID

10.4.2 返回值

- `array` - 共识节点ID列表
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getSealerList","params":[1],"id":1}
↪' http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [

↪"037c255c06161711b6234b8c0960a6979ef039374ccc8b723afea2107cba3432dbbc837a714b7da20111f74d5a24e9
↪",

↪"0c0bbd25152d40969d3d3cee3431fa28287e07cff2330df3258782d3008b876d146ddab97eab42796495bfbb281591
↪",
```

(continues on next page)

(续上页)

```
↪"622af37b2bd29c60ae8f15d467b67c0a7fe5eb3e5c63fdc27a0ee8066707a25afa3aa0eb5a3b802d3a8e5e26de9d5a
↪"
    ]
}
```

10.5 getObserverList

返回指定群组内的观察节点列表

10.5.1 参数

- groupID: unsigned int - 群组ID

10.5.2 返回值

- array - 观察节点ID列表
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getObserverList","params":[1],"id
↪":1}' http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [

↪"10b3a2d4b775ec7f3c2c9e8dc97fa52beb8caab9c34d026db9b95a72ac1d1c1ad551c67c2b7fdc34177857eada7583
↪"
    ]
}
```

10.6 getConsensusStatus

返回指定群组内的共识状态信息

10.6.1 参数

- groupID: unsigned int - 群组ID

10.6.2 返回值

- object - 共识状态信息。
- 当共识机制为PBFT时（PBFT详细设计参考PBFT设计文档），字段如下：
  - accountType: unsigned int - 节点类型，0表示观察节点，1表示共识节点
  - allowFutureBlocks: bool - 允许未来块标志，当前为true

  - cfgErr: bool - 表明节点是否出错，true表示节点已经异常
  - connectedNodes: unsigned int - 连接的节点数
  - consensusedBlockNumber: unsigned int - 当前正在共识的区块高度
  - currentView: unsigned int - 当前视图
  - groupId: unsigned int - 群组ID
  - highestblockHash: string - 最新块哈希
  - highestblockNumber: unsigned int - 最新区块高度
  - leaderFailed: bool - leader失败标志，若为false，节点可能正在处理超时
  - max_faulty_leader: unsigned int - 最大容错节点数
  - nodeNum: unsigned int - 节点的数
  - node_index: unsigned int - 共识节点索引
  - nodeId: string - 节点的ID
  - omitEmptyBlock: bool - 忽略空块标志位，为true
  - protocolId: unsigned int - 协议ID号
  - sealer.index: string - 指定索引index对应的共识节点nodeID
  - toView: unsigned int - 目前到达的view值
  - 与本节点相连的所有共识节点nodeID和视图view信息
- 当共识机制为Raft时（Raft详细设计参考Raft设计文档），字段如下：
  - accountType: unsigned int - 账户类型
  - allowFutureBlocks: bool - 允许未来块标志
  - cfgErr: bool - 配置错误标志
  - consensusedBlockNumber: unsigned int - 下一个共识的最新块高
  - groupId: unsigned int - 群组ID
  - highestblockHash: string - 最新块哈希
  - highestblockNumber: unsigned int - 最新区块高度
  - leaderId: string - leader的nodeId
  - leaderIdx: unsigned int - leader的序号
  - max_faulty_leader: unsigned int - 最大容错节点数
  - sealer.index: string - 节点序号为index的nodeId
  - node index: unsigned int - 节点的index
  - nodeId: string - 节点的ID
  - nodeNum: unsigned int - 节点的数
  - omitEmptyBlock: bool - 忽略空块标志位
  - protocolId: unsigned int - 协议ID号
- 示例

```
// Request PBFT
curl -X POST --data '{"jsonrpc":"2.0","method":"getConsensusStatus","params":[1],
↪"id":1}' http://127.0.0.1:8545 |jq

// Result
```

(continues on next page)

(续上页)


```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": [
    {
      "accountType": 1,
      "allowFutureBlocks": true,
      "cfgErr": false,
      "connectedNodes": 3,
      "consensusedBlockNumber": 38207,
      "currentView": 54477,
      "groupId": 1,
      "highestblockHash":
↪"0x19a16e8833e671aa11431de589c866a6442ca6c8548ba40a44f50889cd785069",
      "highestblockNumber": 38206,
      "leaderFailed": false,
      "max_faulty_leader": 1,
      "nodeId":
↪"f72648fe165da17a889bece08ca0e57862cb979c4e3661d6a77bcc2de85cb766af5d299fec8a4337eedd142dca026a
↪",
      "nodeNum": 4,
      "node_index": 3,
      "omitEmptyBlock": true,
      "protocolId": 65544,
      "sealer.0":
↪"6a99f357ecf8a001e03b68aba66f68398ee08f3ce0f0147e777ec77995369aac470b8c9f0f85f91ebb58a98475764b
↪",
      "sealer.1":
↪"8a453f1328c80b908b2d02ba25adca6341b16b16846d84f903c4f4912728c6aae1050ce4f24cd9c13e010ce922d339
↪",
      "sealer.2":
↪"ed483837e73ee1b56073b178f5ac0896fa328fc0ed418ae3e268d9e9109721421ec48d68f28d6525642868b40dd265
↪",
      "sealer.3":
↪"f72648fe165da17a889bece08ca0e57862cb979c4e3661d6a77bcc2de85cb766af5d299fec8a4337eedd142dca026a
↪",
      "toView": 54477
    },
    [
      {
        "nodeId":
↪"6a99f357ecf8a001e03b68aba66f68398ee08f3ce0f0147e777ec77995369aac470b8c9f0f85f91ebb58a98475764b
↪",
        "view": 54474
      },
      {
        "nodeId":
↪"8a453f1328c80b908b2d02ba25adca6341b16b16846d84f903c4f4912728c6aae1050ce4f24cd9c13e010ce922d339
↪",
        "view": 54475
      },
      {
        "nodeId":
↪"ed483837e73ee1b56073b178f5ac0896fa328fc0ed418ae3e268d9e9109721421ec48d68f28d6525642868b40dd265
↪",
        "view": 54476
      },
      {
        "nodeId":
↪"f72648fe165da17a889bece08ca0e57862cb979c4e3661d6a77bcc2de85cb766af5d299fec8a4337eedd142dca026a
↪",
```


(continues on next page)

(续上页)

```
        "view": 54477
      }
    ]
  ]
}


// Request Raft
curl -X POST --data '{"jsonrpc":"2.0","method":"getConsensusStatus","params":[1],
↪"id":1}' http://127.0.0.1:8545 |jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": [
    {
      "accountType": 1,
      "allowFutureBlocks": true,
      "cfgErr": false,
      "consensusedBlockNumber": 1,
      "groupId": 1,
      "highestblockHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
      "highestblockNumber": 0,
      "leaderId":
↪"d5b3a9782c6aca271c9642aea391415d8b258e3a6d92082e59cc5b813ca123745440792ae0b29f4962df568f8ad58b
↪",
      "leaderIdx": 3,
      "max_faulty_leader": 1,
      "sealer.0":
↪"29c34347a190c1ec0c4507c6eed6a5bcd4d7a8f9f54ef26da616e81185c0af11a8cea4eacb74cf6f61820292b24bc5
↪",
      "sealer.1":
↪"41285429582cbfe6eed501806391d2825894b3696f801e945176c7eb2379a1ecf03b36b027d72f480e89d15bacd434
↪",
      "sealer.2":
↪"87774114e4a496c68f2482b30d221fa2f7b5278876da72f3d0a75695b81e2591c1939fc0d3fadb15cc359c997bafc9
↪",
      "sealer.3":
↪"d5b3a9782c6aca271c9642aea391415d8b258e3a6d92082e59cc5b813ca123745440792ae0b29f4962df568f8ad58b
↪",
      "node index": 1,
      "nodeId":
↪"41285429582cbfe6eed501806391d2825894b3696f801e945176c7eb2379a1ecf03b36b027d72f480e89d15bacd434
↪",
      "nodeNum": 4,
      "omitEmptyBlock": true,
      "protocolId": 267
    }
  ]
}
```

10.7 getSyncStatus

返回指定群组内的同步状态信息

10.7.1 参数

- groupID: unsigned int - 群组ID

10.7.2 返回值

- object - 同步状态信息，字段如下：
  - blockNumber: unsigned int - 最新区块高度
  - genesisHash: string - 创世块哈希
  - isSyncing: bool - 正在同步标志
  - knownHighestNumber: unsigned int - 此节点已知的当前区块链最高块高
  - knownLatestHash: string - 此节点已知的当前区块链最高块哈希
  - latestHash: string - 最新区块哈希
  - nodeId: string - 节点的ID
  - protocolId: unsigned int - 协议ID号
  - txPoolSize: string - 交易池中交易的数量
  - peers: array - 已连接的指定群组内p2p节点，节点信息字段如下:
    * blockNumber: unsigned int - 最新区块高度
    * genesisHash: string - 创始区块哈希
    * latestHash: string - 最新块哈希
    * nodeId: string - 节点的ID
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getSyncStatus","params":[1],"id":1}
↪' http://127.0.0.1:8545 |jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "blockNumber": 0,
    "genesisHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
    "isSyncing": false,
    "knownHighestNumber":0,
    "knownLatestHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
    "latestHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
    "nodeId":
↪"41285429582cbfe6eed501806391d2825894b3696f801e945176c7eb2379a1ecf03b36b027d72f480e89d15bacd434
↪",
    "peers": [
      {
        "blockNumber": 0,
        "genesisHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
        "latestHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
```

(continues on next page)

(续上页)

```
        "nodeId":
↪"29c34347a190c1ec0c4507c6eed6a5bcd4d7a8f9f54ef26da616e81185c0af11a8cea4eacb74cf6f61820292b24bc5
↪"
      },
      {
        "blockNumber": 0,
        "genesisHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
        "latestHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
        "nodeId":
↪"87774114e4a496c68f2482b30d221fa2f7b5278876da72f3d0a75695b81e2591c1939fc0d3fadb15cc359c997bafc9
↪"
      },
      {
        "blockNumber": 0,
        "genesisHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
        "latestHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
        "nodeId":
↪"d5b3a9782c6aca271c9642aea391415d8b258e3a6d92082e59cc5b813ca123745440792ae0b29f4962df568f8ad58b
↪"
      }
    ],
    "protocolId": 265,
    "txPoolSize": "0"
  }
}
```

10.8 getPeers

返回已连接的p2p节点信息

10.8.1 参数

- `groupID`: `unsigned int` - 群组ID

10.8.2 返回值

- `array` - 已连接的p2p节点信息，字段如下：
  - `IPAndPort`: `string` - 节点连接的ip和端口
  - `nodeId`: `string` - 节点的ID
  - `Topic`: `array` - 节点关注的topic信息
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getPeers","params":[1],"id":1}'␣
↪http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
```

(continues on next page)

(续上页)

```
    "jsonrpc": "2.0",
    "result": [
        {
            "IPAndPort": "127.0.0.1:30308",
            "nodeId":
↪"0701cc9f05716690437b78db5b7c9c97c4f8f6dd05794ba4648b42b9267ae07cfcd589447ac36c491e760424214960
↪",
            "Topic": [ ]
        },
        {
            "IPAndPort": "127.0.0.1:58348",
            "nodeId":
↪"353ab5990997956f21b75ff5d2f11ab2c6971391c73585963e96fe2769891c4bc5d8b7c3d0d04f50ad6e04c4445c09
↪",
            "Topic": [ ]
        },
        {
            "IPAndPort": "127.0.0.1:30300",
            "nodeId":
↪"73aebaea2baa9640df416d0e879d6e0a6859a221dad7c2d34d345d5dc1fe9c4cda0ab79a7a3f921dfc9bdea4a49bb3
↪",
            "Topic": [ ]
        }
    ]
}
```

10.9 getGroupPeers

返回指定群组内的共识节点和观察节点列表

10.9.1 参数

- `groupID: unsigned int` - 群组ID

10.9.2 返回值

- `array` - 共识节点和观察节点的ID列表
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getGroupPeers","params":[1],"id":1}
↪' http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [

↪"0c0bbd25152d40969d3d3cee3431fa28287e07cff2330df3258782d3008b876d146ddab97eab42796495bfbb281591
↪",

↪"037c255c06161711b6234b8c0960a6979ef039374ccc8b723afea2107cba3432dbbc837a714b7da20111f74d5a24e9
↪",

↪"622af37b2bd29c60ae8f15d467b67c0a7fe5eb3e5c63fdc27a0ee8066707a25afa2aa0eb5a3b802d3a8e5e26de9d5a
↪",
```

(continues on next page)

(续上页)

```
↪"10b3a2d4b775ec7f3c2c9e8dc97fa52beb8caab9c34d026db9b95a72ac1d1c1ad551c67c2b7fdc34177857eada7583
↪"
    ]
}
```

10.10 getNodeIDList

返回节点本身和已连接的p2p节点列表

10.10.1 参数

- `groupID: unsigned int` - 群组ID

10.10.2 返回值

- `array` - 节点本身和已连接p2p节点的ID列表
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getNodeIDList","params":[1],"id":1}
↪' http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [

↪"0c0bbd25152d40969d3d3cee3431fa28287e07cff2330df3258782d3008b876d146ddab97eab42796495bfbb281591
↪",

↪"037c255c06161711b6234b8c0960a6979ef039374ccc8b723afea2107cba3432dbbc837a714b7da20111f74d5a24e9
↪",

↪"622af37b2bd29c60ae8f15d467b67c0a7fe5eb3e5c63fdc27a0ee8066707a25afa3aa0eb5a3b802d3a8e5e26de9d5a
↪",

↪"10b3a2d4b775ec7f3c2c9e8dc97fa52beb8caab9c34d026db9b95a72ac1d1c1ad551c67c2b7fdc34177857eada7583
↪"
    ]
}
```

10.11 getGroupList

返回节点所属群组的群组ID列表

10.11.1 参数

无

10.11.2 返回值

- array - 节点所属群组的群组ID列表
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getGroupList","params":[],"id":1}'␣
↪http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [1]
}
```

10.12 getBlockByHash

返回根据区块哈希查询的区块信息

10.12.1 参数

- groupID: unsigned int - 群组ID
- blockHash: string - 区块哈希
- includeTransactions: bool - 包含交易标志(true显示交易详细信息，false仅显示交易的hash)

10.12.2 返回值

- object - 区块信息，字段如下：
  - extraData: array - 附加数据
  - gasLimit: string - 区块中允许的gas最大值
  - gasUsed: string - 区块中所有交易消耗的gas
  - hash: string - 区块哈希
  - logsBloom: string - log的布隆过滤器值
  - number: string - 区块高度
  - parentHash: string - 父区块哈希
  - sealer: string - 共识节点序号
  - sealerList: array - 共识节点列表
  - stateRoot: string - 状态根哈希
  - timestamp: string - 时间戳
  - transactions: array - 交易列表，当includeTransactions为false时，显示交易的哈希。当includeTransactions为true时，显示交易详细信息（详细字段见getTransactionByHash）
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockByHash","params":[1,
↪"0x910ea44e2a83618c7cc98456678c9984d94977625e224939b24b3c904794b5ec",true],"id
↪":1}' http://127.0.0.1:8545 |jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "extraData": [],
    "gasLimit": "0x0",
    "gasUsed": "0x0",
    "hash": "0x910ea44e2a83618c7cc98456678c9984d94977625e224939b24b3c904794b5ec",
    "logsBloom":
↪"0x00000000000
```

(续上页)

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockByHash","params":[1,
↪"0x910ea44e2a83618c7cc98456678c9984d94977625e224939b24b3c904794b5ec",false],"id
↪":1}' http://127.0.0.1:8545 |jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "extraData": [],
    "gasLimit": "0x0",
    "gasUsed": "0x0",
    "hash": "0x910ea44e2a83618c7cc98456678c9984d94977625e224939b24b3c904794b5ec",
    "logsBloom":
↪"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "number": "0x1",
    "parentHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
    "sealer": "0x3",
    "sealerList":[

↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bca
↪",

↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966a
↪",

↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bc
↪",

↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54
↪"
    ],
    "stateRoot":
↪"0xfb7ca5a7a271c8ffb51bc689b78d0aeded23497c9c22e67dff8b1c7b4ec88a2a",
    "timestamp": "0x1687e801d99",
    "transactions": [
      "0x022dcb1ad2d940ce7b2131750f7458eb8ace879d129ee5b650b84467cb2184d7"
    ],
    "transactionsRoot":
↪"0x07506c27626365c4f0db788619a96df1e6f8f62c583f158192700e08c10fec6a"
  }
}
```

10.13 getBlockByNumber

返回根据区块高度查询的区块信息

10.13.1 参数

- `groupID`: `unsigned int` - 群组ID
- `blockNumber`: `string` - 区块高度(十进制字符串或0x开头的十六进制字符串)
- `includeTransactions`: `bool` - 包含交易标志(true显示交易详细信息，false仅显示交易的hash)

10.13.2 返回值

见getBlockByHash

- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockByNumber","params":[1,"0x0
↪",true],"id":1}' http://127.0.0.1:8545 |jq
```

Result见getBlockByHash

10.14 getBlockHashByNumber

返回根据区块高度查询的区块哈希

10.14.1 参数

- groupID: unsigned int - 群组ID
- blockNumber: string - 区块高度(十进制字符串或0x开头的十六进制字符串)

10.14.2 返回值

- blockHash: string - 区块哈希
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockHashByNumber","params":[1,
↪"0x1"],"id":1}' http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": "0x10bfdc1e97901ed22cc18a126d3ebb8125717c2438f61d84602f997959c631fa"
}
```

10.15 getTransactionByHash

返回根据交易哈希查询的交易信息

10.15.1 参数

- groupID: unsigned int - 群组ID
- transactionHash: string - 交易哈希

10.15.2 返回值

- object: - 交易信息，其字段如下：
  - blockHash: string - 包含该交易的区块哈希
  - blockNumber: string - 包含该交易的区块高度
  - from: string - 发送者的地址
  - gas: string - 发送者提供的gas
  - gasPrice: string - 发送者提供的gas的价格
  - hash: string - 交易哈希
  - input: string - 交易的输入
  - nonce: string - 交易的nonce值
  - to: string - 接收者的地址，创建合约交易的该值为0x0000000000000000000000000000000000000000
  - transactionIndex: string - 交易的序号
  - value: string - 转移的值
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getTransactionByHash","params":[1,
↪"0x7536cf1286b5ce6c110cd4fea5c891467884240c9af366d678eb4191e1c31c6f"],"id":1}'␣
↪http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "blockHash":
↪"0x10bfdc1e97901ed22cc18a126d3ebb8125717c2438f61d84602f997959c631fa",
        "blockNumber": "0x1",
        "from": "0x6bc952a2e4db9c0c86a368d83e9df0c6ab481102",
        "gas": "0x9184e729fff",
        "gasPrice": "0x174876e7ff",
        "hash": "0x7536cf1286b5ce6c110cd4fea5c891467884240c9af366d678eb4191e1c31c6f
↪",
        "input":
↪"0x48f85bce000000000000000000000000000000000000000000000000000000000000000001bf5bd8a9e7ba8b936ea70
↪",
        "nonce": "0x65f0d06e39dc3c08e32ac10a5070858962bc6c0f5760baca823f2d5582d03f
↪",
        "to": "0xd6f1a71052366dbae2f7ab2d5d5845e77965cf0d",
        "transactionIndex": "0x0",
        "value": "0x0"
    }
}
```

10.16 getTransactionByBlockHashAndIndex

返回根据区块哈希和交易序号查询的交易信息

10.16.1 参数

- groupID: unsigned int - 群组ID
- blockHash: string - 区块哈希
- transactionIndex: string - 交易序号

10.16.2 返回值

见getTransactionByHash

- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getTransactionByBlockHashAndIndex",
↪"params":[1,"0x10bfdc1e97901ed22cc18a126d3ebb8125717c2438f61d84602f997959c631fa",
↪"0x0"],"id":1}' http://127.0.0.1:8545 |jq
```

Result见getTransactionByHash

10.17 getTransactionByBlockNumberAndIndex

返回根据区块高度和交易序号查询的交易信息

10.17.1 参数

- groupID: unsigned int - 群组ID
- blockNumber: string - 区块高度(十进制字符串或0x开头的十六进制字符串)
- transactionIndex: string - 交易序号

10.17.2 返回值

见getTransactionByHash

- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getTransactionByBlockNumberAndIndex
↪","params":[1,"0x1","0x0"],"id":1}' http://127.0.0.1:8545 |jq
```

Result见getTransactionByHash

10.18 getTransactionReceipt

返回根据交易哈希查询的交易回执信息

10.18.1 参数

- groupID: unsigned int - 群组ID
- transactionHash: string - 交易哈希

10.18.2 返回值

- object: - 交易信息，其字段如下：
  - blockHash: string - 包含该交易的区块哈希
  - blockNumber: string - 包含该交易的区块高度
  - contractAddress: string - 合约地址，如果创建合约交易，则为合约部署地址，如果是调用合约，则为"0x0000000000000000000000000000000000000000"
  - from: string - 发送者的地址
  - gasUsed: string - 交易消耗的gas
  - input: string - 交易的输入
  - logs: array - 交易产生的log
  - logsBloom: string - log的布隆过滤器值
  - output: string - 交易的输出
  - root: string - 状态根（state root）
  - status: string - 交易的状态值，参考：交易回执状态
  - to: string - 接收者的地址，创建合约交易的该值为null
  - transactionHash: string - 交易哈希
  - transactionIndex: string - 交易序号
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getTransactionReceipt","params":[1,
↪"0x708b5781b62166bd86e543217be6cd954fd815fd192b9a124ee9327580df8f3f"],"id":1}'␣
↪http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "blockHash":
↪"0x977efec48c248ea4be87016446b40d7785d7b71b7d4e3aa0b103b9cf0f5fe19e",
        "blockNumber": "0xa",
        "contractAddress": "0x0000000000000000000000000000000000000000",
        "from": "0xcdcce60801c0a2e6bb534322c32ae528b9dec8d2",
        "gasUsed": "0x1fb8d",
        "input":
↪"0xb602109a0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
        "logs": [ ],
        "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
        "output":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        "root":"0x38723a2e5e8a17aa7950dc008209944e898f69a7bd10a23c839d341e935fd5ca
↪",
        "status": "0x0",
        "to": "0x15538acd403ac1b2ff09083c70d04856b8c0bdfd",
        "transactionHash":
↪"0x708b5781b62166bd86e543217be6cd954fd815fd192b9a124ee9327580df8f3f",
        "transactionIndex": "0x0"
```

(continues on next page)

(续上页)

```
    }
}
```

10.19 getPendingTransactions

返回待打包的交易信息

10.19.1 参数

- groupID: unsigned int - 群组ID

10.19.2 返回值

- object: - 带打包的交易信息，其字段如下：
  - from: string - 发送者的地址
  - gas: string - 发送者提供的gas
  - gasPrice: string - 发送者提供的gas的价格
  - hash: string - 交易哈希
  - input: string - 交易的输入
  - nonce: string - 交易的nonce值
  - to: string - 接收者的地址，创建合约交易的该值为null
  - value: string - 转移的值
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getPendingTransactions","params
↪":[1],"id":1}' http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        [
            {
                "from": "0x6bc952a2e4db9c0c86a368d83e9df0c6ab481102",
                "gas": "0x9184e729fff",
                "gasPrice": "0x174876e7ff",
                "hash":
↪"0x7536cf1286b5ce6c110cd4fea5c891467884240c9af366d678eb4191e1c31c6f",
                "input":
↪"0x48f85bce0000000000000000000000000000000000000000000000000000000000000001bf5bd8a9e7ba8b936ea70
↪",
                "nonce":
↪"0x65f0d06e39dc3c08e32ac10a5070858962bc6c0f5760baca823f2d5582d03f",
                "to": "0xd6f1a71052366dbae2f7ab2d5d5845e77965cf0d",
                "value": "0x0"
            }
        ]
```

(continues on next page)

(续上页)

```
    }
}
```

10.20 getPendingTxSize

返回待打包的交易数量

10.20.1 参数

- groupID: unsigned int - 群组ID

10.20.2 返回值

- string: - 待打包的交易数量
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getPendingTxSize","params":[1],"id
↪":1}' http://127.0.0.1:8545 |jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": "0x1"
}
```

10.21 getCode

返回根据合约地址查询的合约数据

10.21.1 参数

- groupID: unsigned int - 群组ID
- address: string - 合约地址

10.21.2 返回值

- string: - 合约数据
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getCode","params":[1,
↪"0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b"],"id":1}' http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
```

(continues on next page)

(续上页)

```
    "jsonrpc": "2.0",
    "result":
↪"0x60606040523415600b57fe5b5b60928061001a6000396000f300606060405260000357c0100000000000000000000000
↪"
}
```

10.22 getTotalTransactionCount

返回当前交易总数和区块高度

10.22.1 参数

- groupID: unsigned int - 群组ID

10.22.2 返回值

- object: - 当前交易总数和区块高度信息，其字段如下：
  - blockNumber: string - 区块高度
  - failedTxSum: string - 失败的交易总数
  - txSum: string - 交易总数
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getTotalTransactionCount","params
↪":[1],"id":1}' http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
      "blockNumber": "0x1",
      "failedTxSum": "0x0",
      "txSum": "0x1"
    }
}
```

10.23 getSystemConfigByKey

返回根据key值查询的value值

10.23.1 参数

- groupID: unsigned int - 群组ID
- key: string - 支持tx_count_limit和tx_gas_limit

10.23.2 返回值

- `string` - value值
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getSystemConfigByKey","params":[1,
↪"tx_count_limit"],"id":1}' http://127.0.0.1:8545 |jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": "1000"
}
```

10.24 call

执行一个可以立即获得结果的请求，无需区块链共识

10.24.1 参数

- `groupID`: `unsigned int` - 群组ID
- `object`: - 请求信息，其字段如下：
  - `from`: `string` - 发送者的地址
  - `to`: `string` - 接收者的地址
  - `value`: `string` - (可选)转移的值
  - `data`: `string` - (可选)编码的参数，编码规范参考Ethereum Contract ABI

10.24.2 返回值

- `object`: - 执行的结果
  - `currentBlockNumber`: `string` - 当前区块高度
  - `output`: `string` - 请求结果
  - `status`: `string` - 请求状态（与交易状态码一致）
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"call","params":[1,{"from":
↪"0x6bc952a2e4db9c0c86a368d83e9df0c6ab481102","to":
↪"0xd6f1a71052366dbae2f7ab2d5d5845e77965cf0d","value":"0x1","data":"0x3"}],"id":1}
↪' http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "currentBlockNumber": "0xb",
        "output": "0x",
```

(continues on next page)

(续上页)

```
        "status": "0x0"
    }
}
```

10.25 sendRawTransaction

执行一个签名的交易，需要区块链共识

10.25.1 参数

- groupID: unsigned int - 群组ID
- rlp: string - 签名的交易数据

10.25.2 返回值

- string - 交易哈希
- 示例

```
// RC1 Request
curl -X POST --data '{"jsonrpc":"2.0","method":"sendRawTransaction","params":[1,
↪"f8ef9f65f0d06e39dc3c08e32ac10a5070858962bc6c0f5760baca823f2d5582d03f85174876e7ff8609184e729fff
↪"],"id":1}' http://127.0.0.1:8545 |jq

// RC1 Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": "0x7536cf1286b5ce6c110cd4fea5c891467884240c9af366d678eb4191e1c31c6f"
}

// RC2 Request
curl -X POST --data '{"jsonrpc":"2.0","method":"sendRawTransaction","params":[1,
↪"f8d3a003922ee720bb7445e3a914d8ab8f507d1a647296d563100e49548d83fd98865c8411e1a3008411e1a3008201
↪"],"id":1}' http://127.0.0.1:8545 |jq

// RC2 Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": "0x0accad4228274b0d78939f48149767883a6e99c95941baa950156e926f1c96ba"
}

// FISCO BCOS支持国密算法，采用国密算法的区块链请求示例
// RC1 Request
curl -X POST --data '{"jsonrpc":"2.0","method":"sendRawTransaction","params":[1,
↪"f8ef9f65f0d06e39dc3c08e32ac10a5070858962bc6c0f5760baca823f2d5582d03f85174876e7ff8609184e729fff
↪"],"id":1}' http://127.0.0.1:8545 |jq
// RC2 Request
curl -X POST --data '{"jsonrpc":"2.0","method":"sendRawTransaction","params":[1,
↪"f90114a003eebc46c9c0e3b84799097c5a6ccd6657a9295c11270407707366d0750fcd598411e1a30084b2d05e0082
↪"],"id":1}' http://127.0.0.1:8545 |jq
```

10.26 sendRawTransactionAndGetProof

执行一个签名的交易，交易上链后，推送交易回执、交易Merkle证明、交易回执Merkle证明，Merkle证明可参考这里。

注解:

- `supported_version < 2.2.0`: 调用 `sendRawTransactionAndGetProof` 接口，交易上链后仅推送交易回执
- `supported_version >= 2.2.0`: 调用 `sendRawTransactionAndGetProof` 接口，交易上链后推送交易回执、交易Merkle证明、交易回执Merkle证明

10.26.1 参数

- `groupID`: `unsigned int` - 群组ID
- `rlp`: `string` - 签名的交易数据

10.26.2 返回值

- `string` - 交易哈希
- 示例：同`sendRawTransaction`，参考这里

10.27 getTransactionByHashWithProof

返回根据交易哈希查询的带证明的交易信息，本接口仅在兼容性版本为2.2.0及以后的版本有效，证明信息是为了验证交易的存在性，交易存在性证明请参考文档交易证明

10.27.1 参数

- `groupID`: `unsigned int` - 群组ID
- `transactionHash`: `string` - 交易哈希

10.27.2 返回值

- `object`: - 交易信息，其字段如下:
  - `blockHash`: `string` - 包含该交易的区块哈希
  - `blockNumber`: `string` - 包含该交易的区块高度
  - `from`: `string` - 发送者的地址
  - `gas`: `string` - 发送者提供的gas
  - `gasPrice`: `string` - 发送者提供的gas的价格
  - `hash`: `string` - 交易哈希
  - `input`: `string` - 交易的输入
  - `nonce`: `string` - 交易的nonce值

  - to: string - 接收者的地址，创建合约交易的该值为0x0000000000000000000000000000000000000000
  - transactionIndex: string - 交易的序号
  - value: string - 转移的值
- array - 交易证明，字段如下:
  - left: array - 左边的哈希列表
  - right: array - 右边的哈希列表
- 示例

```
curl -X POST --data '{"jsonrpc":"2.0","method":"getTransactionByHashWithProof",
↪"params":[1,"0xd2c12e211315ef09dbad53407bc820d062780232841534954f9c23ab11d8ab4c
↪"],"id":1}' http://127.0.0.1:8585 |jq
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "transaction": {
      "blockHash":
↪"0xcd31b05e466bce99460b1ed70d6069fdfbb15e6eef84e9b9e4534358edb3899a",
      "blockNumber": "0x5",
      "from": "0x148947262ec5e21739fe3a931c29e8b84ee34a0f",
      "gas": "0x1c9c380",
      "gasPrice": "0x1c9c380",
      "hash": "0xd2c12e211315ef09dbad53407bc820d062780232841534954f9c23ab11d8ab4c",
      "input":
↪"0x8a42ebe900000000000000000000000000000000000000000000000000000000000000400000000000000000000
↪",
      "nonce": "0x208f6fd78d48aad370df51c6fdf866f8ab022de765c2959841ff2e81bfd9af9",
      "to": "0xd6c8a04b8826b0a37c6d4aa0eaa8644d8e35b79f",
      "transactionIndex": "0x32",
      "value": "0x0"
    },
    "txProof": [
      {
        "left": [
          "30f0abfcf4ca152815548620e33d21fd0feaa7c78867791c751e57cb5aa38248c2",
          "31a864156ca9841da8176738bb981d5da9102d9703746039b3e5407fa987e5183e"
        ],
        "right": [
          "33d8078d7e71df3544f8845a9db35aa35b2638e8468a321423152e64b9004367b4",
          "34343a4bce325ec8f6cf48517588830cd15f69b60a05598b78b03c3656d1fbf2f5",
          "35ac231554047ce77c0b31cd1c469f1f39ebe23404fa8ff6cc7819ad83e2c029e7",
          "361f6c588e650323e03afe6460dd89a9c061583e0d62c117ba64729d2c9d79317c",
          "377606f79f3e08b1ba3759eceada7fde3584f01822467855aa6356652f2499c738",
          "386722fe270659232c5572ba54ce23b474c85d8b709e7c08e85230afb1c155fe18",
          "39a9441d668e5e09a5619c365577c8c31365f44a984bde04300d4dbba190330c0b",
          "3a78a8c288120cbe612c24a33cce2731dd3a8fe6927d9ee25cb2350dba08a541f5",
          "3bd9b67256e201b5736f6081f39f83bcb917261144384570bdbb8766586c3bb417",
          "3c3158e5a82a1ac1ed41c4fd78d5be06bf79327f60b094895b886e7aae57cff375",
          "3de9a4d98c5ae658ffe764fbfa81edfdd4774e01b35ccb42beacb67064a5457863",
          "3e525e60c0f7eb935125f1156a692eb455ab4038c6b16390ce30937b0d1b314298",
          "3f1600afe67dec2d21582b8c7b76a15e569371d736d7bfc7a96c0327d280b91dfc"
        ]
      },
      {
        "left": [
          "3577673b86ad4d594d86941d731f17d1515f4669483aed091d49f279d677cb19",
          "75603bfea5b44df4c41fbb9926836464189633 4f006af3a3f67edaa4b26477ca",
          "1339d43c526f0f34d8a0f4fb3bb47b716fdfde8d35697be5992e0888e4d794c9"
```

(continues on next page)

(续上页)

```
        ],
        "right": [
          "63c8e574fb2ef52e995427a8acaa72c27073dd8e37736add8dbf36be4f609ecb",
          "e65353d911d6cc8ead3fad53ab24cab69a1e31df8397517b124f578ba908558d"
        ]
      },
      {
        "left": [],
        "right": []
      }
    ]
  }
}
```

10.28 getTransactionReceiptByHashWithProof

返回根据交易哈希查询的带证明的交易回执信息，本接口仅在兼容性版本为2.2.0及以后的版本有效，证明信息是为了验证回执的存在性，回执存在性证明请参考文档交易证明

- `groupID`: `unsigned int` - 群组ID
- `transactionHash`: `string` - 交易哈希

10.28.1 返回值

- `array` - 回执证明，字段如下:
  - `left`: `array` - 左边的哈希列表
  - `right`: `array` - 右边的哈希列表
- `object`: - 交易信息，其字段如下:
  - `blockHash`: `string` - 包含该交易的区块哈希
  - `blockNumber`: `string` - 包含该交易的区块高度
  - `contractAddress`: `string` - 合约地址，如果创建合约交易，则为合约部署地址，如果是调用合约，则为"0x0000000000000000000000000000000000000000"
  - `from`: `string` - 发送者的地址
  - `gasUsed`: `string` - 交易消耗的gas
  - `input`: `string` - 交易的输入
  - `logs`: `array` - 交易产生的log
  - `logsBloom`: `string` - log的布隆过滤器值
  - `output`: `string` - 交易的输出
  - `status`: `string` - 交易的状态值，参考：交易回执状态
  - `to`: `string` - 接收者的地址，创建合约交易的该值为null
  - `transactionHash`: `string` - 交易哈希
  - `transactionIndex`: `string` - 交易序号
- 示例

```
curl -X POST --data '{"jsonrpc":"2.0","method":
↪"getTransactionReceiptByHashWithProof","params":[1,
↪"0xd2c12e211315ef09dbad53407bc820d062780232841534954f9c23ab11d8ab4c"],"id":1}'␣
↪http://127.0.0.1:8585 |jq

{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "receiptProof": [
      {
        "left": [
          "3088b5c8f9d92a3411a911f35ff0119a02e8f8f04852cf2fdfaa659843eac6a3ad",
          "31170ac8fd555dc50e59050841da0d96e4c4bc7e6266e1c6865c08c3b2391801dd"
        ],
        "right": [
          "33c572c8f961e0c56689d641fcf274916857819769a74e6424c58659bf530e90e3",
          "341233933ea3d357b4fdd6b3d1ed732dcff15cfd54e527c93c15a4e0238585ed11",
          "351e7ba09965cce1cfb820aced1d37204b06d96a21c5c2cf36850ffc62cf1fc84c",
          "361f65633d9ae843d4d3679b255fd448546a7b531c0056e8161ea0adbf1af12c0f",
          "37744f6e0d320314536b230d28b2fd6ac90b0111fb1e3bf4a750689abc282d8589",
          "386e60d9daa0be9825019fcf3d08cdaf51a90dc62a22a6e11371f94a8e516679cc",
          "391ef2f2cee81f3561a9900d5333af18f59aa3cd14e70241b5e86305ba697bf5f2",
          "3ac9999d4f36d76c95c61761879eb9ec60b964a489527f5af844398ffaa8617f0d",
          "3b0039ce903e275170640f3a464ce2e1adc2a7caee41267c195469365074032401",
          "3ca53017502028a0cb5bbf6c47c4779f365138da6910ffcfebf9591b45b89abd48",
          "3de04fc8766a344bb73d3fe6360c61d036e2eeedfd9ecdb86a0498d7849ed591f0",
          "3e2fc73ee22c4986111423dd20e8db317a313c9df29fa5aa3090f27097ecc4e1a9",
          "3fa7d31ad5c6e7bba3f99f9efc03ed8dd97cb1504003c34ad6bde5a662481f00a0"
        ]
      },
      {
        "left": [
          "cd46118c0e99be585ffcf50423630348dbc486e54e9d9293a6a8754020a68a92",
          "3be78209b3e3c83af3668ec3192b5bf232531323ef66b66de80a11f386270132",
          "bd3a11d74a3fd79b1e1ea17e45b76eda4d25f6a5ec7fc5f067ea0d086b1ce70f"
        ],
        "right": [
          "6a6cefef8b48e455287a8c8694b06f4f7cb7950017ab048d6e6bdd8029f9f8c9",
          "0a27c5ee02e618d919d228e6a754dc201d299c91c9e4420a48783bb6fcd09be5"
        ]
      },
      {
        "left": [],
        "right": []
      }
    ],
    "transactionReceipt": {
      "blockHash":
↪"0xcd31b05e466bce99460b1ed70d6069fdfbb15e6eef84e9b9e4534358edb3899a",
      "blockNumber": "0x5",
      "contractAddress": "0x0000000000000000000000000000000000000000",
      "from": "0x148947262ec5e21739fe3a931c29e8b84ee34a0f",
      "gasUsed": "0x21dc1b",
      "input":
↪"0x8a42ebe900000000000000000000000000000000000000000000000000000000000000400000000000000000000000
↪",
      "logs": [],
      "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
      "output": "0x",
```

(continues on next page)

(续上页)

```
      "root": "0xc3b4185963c78a4ca8eb90240e5cd95371d7217a9ce2bfa1149d53f79c73afbb",
      "status": "0x0",
      "to": "0xd6c8a04b8826b0a37c6d4aa0eaa8644d8e35b79f",
      "transactionHash":
↪"0xd2c12e211315ef09dbad53407bc820d062780232841534954f9c23ab11d8ab4c",
      "transactionIndex": "0x32"
    }
  }
}
```

10.29 generateGroup

根据群组ID及创世块参数创建新的群组，本接口仅在兼容性版本为2.2.0及以后的版本有效

10.29.1 参数

- groupID: unsigned int - 群组ID
- params: object - 创世块参数，其字段如下：
  - timestamp: unsigned int - 创世块时间戳
  - sealers: array - 共识节点列表，要求所有所列共识节点间存在有效的P2P连接
  - enable_free_storage: bool - 可选，是否启用"free storage"模式，启用后节点将减少STORAGE相关指令的gas耗费

10.29.2 返回值

- object: - 接口调用结果，其字段如下：
  - code: - 接口调用状态码，状态码的释义请参见动态群组管理 *API* 状态码
  - message: - 接口消息
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"generateGroup","params":[2, {
↪"timestamp":"1585214879000","sealers":[
↪"70f18c055d366615e86df99f91b6d3f16f07d66293b203b73498442c0366d2c8ff7a21bb56923d9d81b1c291625188
↪",
↪"dde37f534885f08db914566efeb03183d59363a4be972bbcdde25c37f0b350e1980a7de4fdc4aaf956b931aab00b73
↪",
↪"d41672b29b3b1bfe6cad563d0f0b2a2735865b27918307b85085f892043a63f681ac8799243e920f7bb144b111d854
↪",
↪"7ba2717f81f38e7371ccdbe173751f051b86819f709e940957664dbde028698fd31ba3042f7dd9accd73741ba42afc
↪"],"enable_free_storage":true}],"id":1}' http://127.0.0.1:8545 | jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "code": "0x0",
    "message": "Group 2 generated successfully"
  }
}
```

10.30 startGroup

根据群组ID启动相应的群组，本接口仅在兼容性版本为2.2.0及以后的版本有效

10.30.1 参数

- groupID: unsigned int - 群组ID

10.30.2 返回值

- object: 接口调用结果，其字段如下：
  - code: - 接口调用状态码，状态码的释义请参见动态群组管理 *API* 状态码
  - message: - 接口消息
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"startGroup","params":[2],"id":1}'␣
↪http://127.0.0.1:8545 | jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "code": "0x0",
    "message": "Group 2 started successfully"
  }
}
```

10.31 stopGroup

根据群组ID停止相应的群组，本接口仅在兼容性版本为2.2.0及以后的版本有效

10.31.1 参数

- groupID: unsigned int - 群组ID

10.31.2 返回值

- object: 接口调用结果，其字段如下：
  - code: - 接口调用状态码，状态码的释义请参见动态群组管理 *API* 状态码
  - message: - 接口消息
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"stopGroup","params":[2],"id":1}'␣
↪http://127.0.0.1:8545 | jq

// Result
```

(continues on next page)

(续上页)

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "code": "0x0",
    "message": "Group 2 stopped successfully"
  }
}
```

10.32 removeGroup

根据群组ID删除相应群组，群组数据会被保留以供将来恢复群组，本接口仅在兼容性版本为2.2.0及以后的版本有效

10.32.1 参数

- `groupID: unsigned int` - 群组ID

10.32.2 返回值

- `object`: 接口调用结果，其字段如下：
  - `code`: - 接口调用状态码，状态码的释义请参见动态群组管理 *API* 状态码
  - `message`: - 接口消息
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"removeGroup","params":[2],"id":1}'␣
↪http://127.0.0.1:8545 | jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "code": "0x0",
    "message": "Group 2 deleted successfully"
  }
}
```

10.33 recoverGroup

根据群组ID恢复相应群组，本接口仅在兼容性版本为2.2.0及以后的版本有效

10.33.1 参数

- `groupID: unsigned int` - 群组ID

10.33.2 返回值

- object: 接口调用结果，其字段如下：
  - code: - 接口调用状态码，状态码的释义请参见动态群组管理 *API* 状态码
  - message: - 接口消息
- 示例

```
// Request
curl -Ss -X POST --data '{"jsonrpc":"2.0","method":"recoverGroup","params":[2],"id
↪":1}' http://127.0.0.1:8545 | jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "code": "0x0",
    "message": "Group 2 recovered successfully"
  }
}
```

10.34 queryGroupStatus

根据群组ID查询相应群组的状态

10.34.1 参数

- groupID: unsigned int - 群组ID

10.34.2 返回值

- object: 接口调用结果，其字段如下：
  - code: - 接口调用状态码，状态码的释义请参见动态群组管理 *API* 状态码
  - message: - 接口消息
  - status: - 群组状态标识，为下列值之一：
    - INEXISTENT: 群组不存在
    - STOPPING: 群组正在停止
    - RUNNING: 群组正在运行
    - STOPPED: 群组已停止
    - DELETED: 群组已删除
- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"queryGroupStatus","params":[2],"id
↪":1}' http://127.0.0.1:8545 | jq

// Result
{
  "id": 1,
```

(continues on next page)

(续上页)

```
  "jsonrpc": "2.0",
  "result": {
    "code": "0x0",
    "message": "",
    "status": "STOPPED"
  }
}
```

10.35 错误码描述

10.35.1 RPC 错误码

当一个RPC调用遇到错误时，返回的响应对象必须包含error错误结果字段，该字段有下列成员参数：

- code: 使用数值表示该异常的错误类型，必须为整数。
- message: 对该错误的简单描述字符串。
- data: 包含关于错误附加信息的基本类型或结构化类型，该成员可选。

错误对象包含两类错误码，分别是JSON-RPC标准错误码和FISCO BCOS RPC错误码。

JSON-RPC标准错误码

标准错误码及其对应的含义如下：

FISCO BCOS RPC错误码

FISCO BCOS RPC接口错误码及其对应的含义如下：

10.35.2 交易回执状态

10.35.3 Precompiled Service API 错误码

10.35.4 动态群组管理 API 状态码

CHAPTER 11

常见问题解答

11.1 版本相关

问: FISCO BCOS 2.0版本与之前版本有哪些变化？ 答: 请 参考这里。

问: 开发者如何与FISCO BCOS平台交互？ 答: FISCO BCOS提供多种开发者与平台交互的方式，参考如下:

- FISCO BCOS 2.0版本提供JSON-RPC接口，具体请 参考这里。
- FISCO BCOS 2.0版本提供Web3SDK帮助开发者快速实现应用，具体请 参考这里。
- FISCO BCOS 2.0版本提供控制台帮助用户快速了解使用FISCO BCOS，具体请 参考这里。

问: FISCO BCOS 2.0版本如何搭建？ 答: FISCO BCOS支持多种搭建方式，常用方式有:

- 开发部署工具 build_chain.sh：适合开发者体验、测试FISCO BCOS联盟链，具体请 参考这里。
- 运维部署工具 generator：适用于企业用户部署、维护FISCO BCOS联盟链，具体请 参考这里。

问: FISCO BCOS 2.0版本的智能合约与之前版本合约有什么不同，兼容性如何？ 答: FISCO BCOS 2.0版本支持最新的Solidity合约，同时增加了precompile合约，具体请 参考这里。

问: 国密和普通版本的区别有哪些？ 答: 国密版FISCO BCOS将交易签名验签、p2p网络连接、节点连接、数据落盘加密等底层模块的密码学算法均替换为国密算法。同时在编译版本，证书，落盘加密，solidity编译java，Web3SDK使用国密版本和普通版本都有区别，具体请 参考这里。

问: 是否支持从1.3或1.5升级到2.0版本? 答: 不支持。

11.2 控制台

问: 控制台指令区分大小写吗？ 答: 区分大小写，命令是完全匹配，但是可以采用`tab`补全命令。

问: 加入共识列表或观察者列表报错，nodeID is not in network，为什么？ 答: 节点加入共识列表和观察者列表的节点必须是连接peer的nodeID列表里面的成员。

问: 删除节点操作报错，nodeID is not in group peers，为什么？ 答: 节点删除操作中的节点必须是getGroupPeers里面展示的group的peers。

问: 游离节点（非群组节点）是否可以同步group数据？ 答: 游离节点不参与group内的共识、同步和出块，游离节点可以通过控制台`addSealer/addObserver`命令可以将退出的节点添加为共识/观察节点。

问: 某节点属于不同的group，是否可以支持查询多group的信息。 答: 可以，在进入控制台时，输入要查看的groupID: ./start [groupID]

11.3 FISCO BCOS使用

问: 2.0版本证书在哪里使用? 答: 请参考证书说明文档

问: 2.0版本交易结构包括哪些字段? 答: 请参考这里

问: 系统配置、群组配置、节点配置分别指什么？ 答: 系统配置是指节点配置中一些影响账本功能，并需账本节点共识的配置项。群组配置指节点所属的群组的相关配置，节点的每个群组都有独立的配置。节点配置指所有可配置项。

问: 群组配置都是可改的吗？ 答: 从配置项是否可改的维度，分为

- 节点首次启动生成创世块后不能再修改。这类配置放置于group.x.genesis文件，其中x表示组编号，全链唯一。
- 通过发交易修改配置项实现账本内一致。
- 修改自身配置文件后，节点重启生效。这类配置放置于`group.x.ini`文件。群组配置改后重启可改项就是本地配置，nodeX/conf下的`group.*.ini`文件，更改重启生效。涉及配置项为[tx_pool].limit（交易池容量），[consensus].ttl(节点转发数)。

问: 群组配置用户可以改的涉及哪些配置？ 答: 群组可修改配置分为共识可改配置和手工可改配置

- 共识可改配置：全组所有节点相同，共识后生效。[consensus].max_trans_num,[consensus].node.X,[tx].gas_limit。
- 手工可改配置：`group.x.ini`文件中，修改后重启生效，只影响节点。配置项有[tx_pool].limit。

问: 群组共识可改配置如何更改、查询？ 答: 共识可改配置可以通过控制台修改。共识可改配置项查询除了控制台外，还可以通过RPC接口查询，具体请 参考这里。

- [consensus].max_trans_num，[tx].gas_limit使用接口setSystemConfigByKey更改，对于的配置项为tx_count_limit，tx_gas_limit。具体参见setSystemConfigByKey -h 。
- [consensus].node.X的更改涉及到节点管理，控制台接口涉及到addSealer，addObserver，removeNode，具体参考《节点管理》。

问: 群组观察节点和共识节点有什么区别？ 答: 观察节点能同步群组数据，但不能参与共识。共识节点除了具有观察者权限，还参与共识。

问: 如何将合约纳入CNS管理？ 答: 在部署合约时，调用CNS合约接口，将合约name、version、address信息写入CNS表中。

问: 如何查询合约CNS表？ 答: 通过Web3SDK控制台指令查询，查询指令根据合约name查询。

问: 为什么本地SDK无法连接云服务器上的FISCO BCOS节点？ 答:

1. 检查云服务器上的节点配置，channel是否监听外网IP，而不是`127.0.0.1`。端口介绍参考这里
2. 检查通过云服务器提厂商提供的控制台，检查是否配置了安全组，需要在安全组中开放FISCO BCOS节点所使用的channel端口。
3. 检查生成的证书是否正确，参考这里

问: 节点启动后，为什么无法连接其他节点且节点日志中出现『错误的文件描述符』等网络异常信息？ 答:

1. 请检查节点证书配置是否正确
2. 请检查节点类型（国密、非国密）是否与链中其他节点一致

问： 日志中为何出现形如"invalid group status"的错误提示?

答：可能由于文件系统错误等原因导致节点在本地记录的群组状态不合法，可以检查群组数据目录下的.group_status文件，将其内容改为下列值之一：

- STOPPED
- DELETED
- RUNNING

11.4 Web3SDK

问: Web3SDK对Java版本有要求吗? 答: 参考Java环境要求

问: Web3SDK配置完成，启动失败的原因是什么? 答: 参考JavaSDK异常场景

11.5 运维部署工具

问： 运维部署工具使用时出现找不到pip

答： 运维部署工具依赖python pip，使用以下命令安装：

```
$ bash ./scripts/install.sh
```

问: 运维部署工具使用时出现

```
Traceback (most recent call last):
   File "./generator", line 19, in <module>
    from pys.build import config
   File "/data/asherli/generator/pys/build/config.py", line 25, in <module>
     import configparse
```

答: 系统缺少python configparser模块，请按照以下命令安装：

```
  $ pip install configparser
```

问: 节点或SDK使用的OpenSSL证书过期了，如何续期?

答: 证书续期操作可以参考证书续期操作

问： 使用下载命令提示certificate verify failed 答： 在 ./pys/tool/utils.py 这个文件的开头中加入如下两行

```
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

CHAPTER 12

社区

FISCO BCOS是国内企业主导研发、对外开源、安全可控的企业级金融联盟链底层平台。由金融区块链合作联盟（深圳）（简称：金链盟）成立的开源工作组协作打造，工作组成员包括博彦科技、华为、深证通、神州信息、四方精创、腾讯、微众银行、亦笔科技和越秀金科等金链盟成员机构。

12.1 FISCO BCOS资源列表

- Github主页
- 技术文档
- 深度解析系列文章
- 贡献代码
- 反馈问题
- 应用案例集

12.2 加入FISCO BCOS社区

来Meetup畅聊技术

走出去拓展区块链人脉 | 打破技术认知边界

- 全国巡回进行时 -

深圳站

区块链如何砍掉公益黑手

深圳站

券商区块链的“钱途”

成都站

解开物联网的安全桎梏

北京站

撬动供应链金融万亿蓝海市场

上海站

区块链让每一颗爱心都不落空

北京站

区块链技术消除版权纠纷

深圳站

利用区块链找靠谱月嫂

精彩继续

广州 | 北京 | 上海 | 西安

活动日程安排及报名信息
请关注公众号

成为贡献者

希望以后你可以拿这个项目给自己加分：
“FISCO BCOS是我一手搞起来的！”

★ Star	于你是收藏，于我是鼓励
New issue	反馈bug | 问题交流
New PR	文档修改 | bug修复 | 提交新功能特性